JHU/APL
TG 1386
SEPTEMBER 1994

Technical Memorandum

PROCEEDINGS OF THE FOURTH SYSTEMS REENGINEERING TECHNOLOGY WORKSHOP

BRUCE I. BLUM, editor

THE JOHNS HOPKINS UNIVERSITY ■ APPLIED PHYSICS LABORATORY

9

JHU/APL
TG 1386
SEPTEMBER 1994

Technical Memorandum

PROCEEDINGS OF THE FOURTH SYSTEMS REENGINEERING TECHNOLOGY WORKSHOP

Monterey Marriot Hotel
Monterey, California
February 8–10, 1994

BRUCE I. BLUM, editor

Sponsored by
NAVAL SURFACE WARFARE CENTER
DAHLGREN DIVISION—WHITE OAK DETACHMENT

Silver Spring, MD 20903-5640

With the Cooperation of
JOHNS HOPKINS UNIVERSITY APPLIED PHYSICS LABORATORY

Laurel, MD 20723-6099

Approved for public release; distribution is unlimited.

The Systems Reengineering Technology Workshop is sponsored by the Navy Surface Warfare Center, Dahlgren Division, as part of the Complex Systems Engineering Block Program. The organizers of this workshop wish to express their appreciation to CDR Grace Thompson and Dr. Harry Crisp for their continuing guidance and assistance. Mr. Blum is supported by ONR tasks under contract N00039-91-C-0001 with SPAWAR.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) JHU/APL TG-1386	5. MONITORING ORGANIZATION REPORT NUMBER(S) JHU/APL TG-1386

6a. NAME OF PERFORMING ORGANIZATION The Johns Hopkins University Applied Physics Laboratory	6b. OFFICE SYMBOL (*If applicable*) RCO	7a. NAME OF MONITORING ORGANIZATION NAVTECHREP Laurel, Maryland
6c. ADDRESS (*City, State, and ZIP Code*) Johns Hopkins Road Laurel, Md. 20723-6099		7b. ADDRESS (*City, State, and ZIP Code*) Johns Hopkins Road Laurel, Md. 20723-6099
8a. NAME OF FUNDING/SPONSORING ORGANIZATION Naval Surface Warfare Center	8b. OFFICE SYMBOL (*If applicable*)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER N00039-91-C-0001

8c. ADDRESS (*City, State, and ZIP Code*)	10. SOURCE OF FUNDING NUMBERS			
Dahlgren Division—White Oak Detachment Silver Spring, MD 20903-5640	PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT ACCESSION NO.

11. TITLE (*Include Security Classification*)

Proceedings of the Fourth Systems Reengineering Technology Workshop (U)

12. PERSONAL AUTHOR(S)

Bruce I. Blum, editor

13a. TYPE OF REPORT Technical Memorandum	13b. TIME COVERED FROM______ TO______	14. DATE OF REPORT (*Year, Month, Day*) September 1994	15. PAGE COUNT 410

16. SUPPLEMENTARY NOTATION

Presented at Fourth Systems Reengineering Technology Workshop, Monterey, Calif., 8-10 Feb. 1994

17. COSATI CODES			18. SUBJECT TERMS (*Continue on reverse if necessary and identify by block number*)
FIELD	GROUP	SUB-GROUP	Reengineering Systems reengineering Software reengineering

19. ABSTRACT (*Continue on reverse if necessary and identify by block number*)

The Navy has invested significant resources in the development of large and complex systems that must be modified and extended to respond to changing requirements. However, many of these systems are based on archaic automation technologies that do not support modern hardware and software engineering methodologies or maintenance strategies. Consequently, system modification has become increasingly complex. To reduce the complexity, developers can employ reengineering techniques to create new systems. This report contains the papers presented at the Fourth Systems Reengineering Technology Workshop sponsored by the Naval Surface Warfare Center. The papers discuss theoretical anc applied techniques that can be used to facilitate systems reengineering efforts. Specific topics include design issues in systems reengineering, reuse in reengineering and forward engineering, experience reports, reengineering translation and transformation, evaluating a reengineering project, tools for reengineering, engineering science and reengineering, the impact of object orientation, and approaches to reengineering.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED	
22a. NAME OF RESPONSIBLE INDIVIDUAL NAVTECHREP Security Officer	22b. TELEPHONE (*Include Area Code*) (301) 953-5403	22c. OFFICE SYMBOL NAVTECHREP

83 APR edition may be used until exhausted
All other editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE

UNCLASSIFIED

ABSTRACT

The Navy has invested significant resources in the development of large and complex systems that must be modified and extended to respond to changing requirements. However, many of these systems are based on archaic automation technologies that do not support modern hardware and software engineering methodologies or maintenance strategies. Consequently, system modification has become increasingly complex. To reduce the complexity, developers can employ reengineering techniques to create new systems. This report contains the papers presented at the Fourth Systems Reengineering Technology Workshop sponsored by the Naval Surface Warfare Center. The papers discuss theoretical and applied techniques that can be used to facilitate systems reengineering efforts. Specific topics include design issues in systems reengineering, reuse in reengineering and forward engineering, experience reports, reengineering translation and transformation, evaluating a reengineering project, tools for reengineering, engineering science and reengineering, the impact of object orientation, and approaches to reengineering.

Foreword

This is the fourth of a series annual workshops sponsored by the Naval Surface Warfare Center (NSWC). These Systems Reengineering Technology Workshops are motivated by the fact that the Navy has invested billions of dollars in the development of systems that may be modified and extended to respond to changing requirements. Systems reengineering technology is necessary if the Navy (as well as the other users of large-scale, complex systems) are to benefit from their extensive investments.

The workshop brings together representatives of government, industry, and academia to address the issues confronting this technology. Although the principal interest of the sponsors is the reegineering of embedded, real-time systems that include hardware, software, and human-computer interaction, the workshop encourages the participation of all individuals and organizations concerned with the reengineering of large-scale, complex systems.

These participant's proceedings contain the papers accepted for presentation at the workshop. They have been organized to according to the order of the program. The workshop begins with a keynote address by Andrew P. Sage. His paper, "Systems Engineering and Management for Reengineering," provides an introduction to the issues to be discussed during the workshop. The paper is abstracted from *Systems Management for Information Technology and Software Engineering*, which will be published by Wiley later in 1994. This opening paper is complemented by Robert S. Arnold, "A Road Map Guide to Software Reengineering Technology," which is included as an appendix. It is the introduction to R. Arnold (ed.), *Software Reengineering*, IEEE Computer Science Press, 1993. Thus, this volume begins and ends with two very important surveys that present a conceptual context for reengineering and this workshop.

The papers in the volume have been grouped into units with common themes. The commonality, of course, often is more apparent than real. With the decision not to have parallel sessions and a commitment to accept all the relevant, quality papers that could be accommodated in the time available, the workshop organizers were not always able to focus a session on a central theme. We overcame this difficulty, in part, by structuring some of the workshop sessions as panels rather than paper presentations. In those cases, the papers in these proceedings constitute an extended discussion of concepts that may have been presented only briefly in the workshop.

By the standards of most academic conferences, these proceedings are long. Because the literature on systems (as opposed to software or organizational) reengineering is sparse, we elected not to place any page limits on the papers. Thus, the papers are as long (or short) as they needed to be to convey what the authors believe is important. As the organizers of this workshop, we hope that those who read these proceedings will concur with this decision.

Mark Wilson, Workshop Chair
Bruce Blum, Program Co-Chair
Gilbert Myers, Program Co-Chair

The Johns Hopkins University
Applied Physics Laboratory
Laurel Maryland 20723-6099

Table of Contents

Keynote Address

Design Issues in Systems Reengineering

Reuse in Reengineering and Forward Engineering

Experience Reports and Discussion (Panel)

Reengineering Translation and Transformation (Panel)

Evaluating a Reengineering Project

Tools for Reengineering

Engineering Science and Reengineering

The Impact of an Object Orientation

Approaches to Reengineering (Panel)

Appendix

Systems Engineering and Management for Reengineering

Andrew P. Sage

School of Information Technology and Engineering
George Mason University
Fairfax, VA 22030-4444

Abstract **This paper presents and overview and perspective on systems engineering and systems management for reengineering and related approaches towards organizational and technology revitalization. As we will see, there are at least three types of reengineering that can be considered: reengineering at the levels of product, process, and systems management. We claim that all three are generally needed and an approach at one level only may not be satisfactory.**

I. INTRODUCTION

Responsiveness is very clearly a critical need today. By this, we mean of course, organizational responsiveness in providing products and services of demonstrable value to customers, and thereby to the organizations own stakeholders. This must be accomplished by efficiently and effectively employing leadership and empowered people, such that systems management strategies, organizational processes, human resources, and appropriate technologies are brought to bear on the production of high quality and trustworthy goods and services. We also mean responsiveness in supplying appropriate technologies and systems to meet these objectives. Figure 1 illustrates these ingredients and some of their linkages in the production of products and services. It is a composite representations that indicates some of the many ingredients responsible for trustworthy and high quality products.

Many recent papers [1] have indicated the need for continual revitalization in the way in which we do things, such that they are always done better. This is the case, even if the external environment were static and unchanging. However, when we are in a period of high velocity environments, then continual organizational change and associated change in processes and product must be considered as a fundamental rule of the game for progress.

In many ways, past progress can act to impede future progress. This is especially the case when we become very accustomed to a particular way of doing things, and have allowed a very large overhead situation to accumulate around what were once highly successful efforts at the production of quality products and services. It is especially difficult to change when what we are doing now is done very well. Yet, it is entirely possible that a competitor may be able to do it better, in any of a number of ways. Also, what we do well now may well not be what we will need to be doing in the future.

In many if not most cases, improvement needs come about not because of human inattention to the tasks they perform. Rather, and more often than not, it suggests that the tasks themselves are in need of restudy and renovation. These tasks may be strategic in nature, or they may be tactical, or they may be purely operational. Most often, however, attempts at improvement through attention at only operational levels will yield very modest improvements for the effort invested. As many have indicated, ***it is the*** (strategic and tactical, and not the operational) ***system that is at fault***.

Figure 2 indicates some improvement approaches, each of which relate to reengineering. They are interrelated and our listing is not complete. One of our objectives in this paper is to provide a perspective and overview of some of the many reengineering

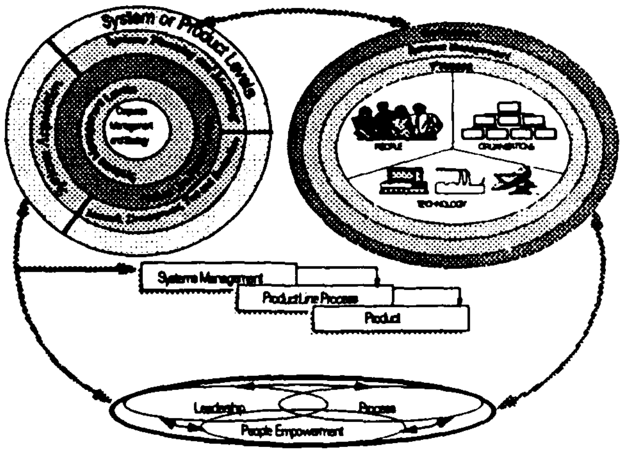


Fig. 1 Some of the Many Major Ingredients in Reengineering

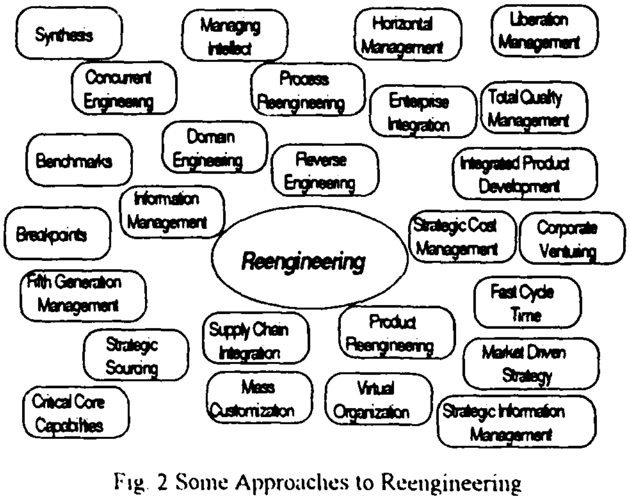


Fig. 2 Some Approaches to Reengineering

approaches and methodologies that have been suggested. A much more complete dissuasion is contained in [2], and this paper is a summary of Chapter 8 which discusses reengineering.

Figure 3 represents a generic view of reengineering. The entity to be reengineered can be either systems management, process, or product, or some appropriate combination of these. We will expand on this illustration and its interpretation in our discussions to follow.

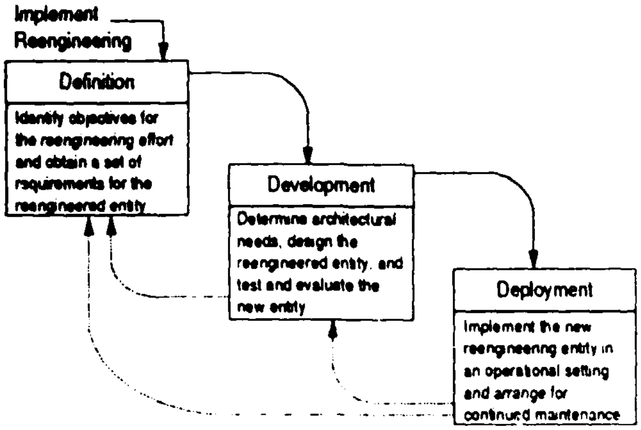

Implement Reengineering
Definition
Identify objectives for the reengineering effort and obtain a set of requirements for the reengineered entity
Development
Determine architectural needs, design the reengineered entity, and test and evaluate the new entity
Deployment
Implement the new reengineering entity in an operational setting and arrange for continued maintenance

Fig. 3 A Three Phase Approach to Generic Reengineering

We can approach a discussion of reengineering from several perspectives. First, we can discuss the

- structural,
- functional, and
- purposeful

aspects of reengineering. Alternately, or in addition, we can examine reengineering at the level of

- systems management,
- process, or
- product.

We may examine reengineering issues at any, or all, of the three fundamental systems engineering lifecycles:

- research, development, test, and evaluation (RDT&E);
- systems acquisition, procurement, or production; or
- systems planning and marketing.

Within each of these lifecycles, we could consider reengineering at any or all of the three generic phases of definition, development, or deployment. At the level of systems management, we reengineer each of these phases and potentially all other processes within the company for integrated improvement. At the level of process reengineering only, as we define it, only a single process is redesigned, and with no fundamental changes in the structure or purpose of the organization as a whole. Changes, when they occur, may be radical and revolutionary, or incremental and evolutionary at the levels of systems management, processes, or products.

One fundamental notion of reengineering is, however, the reality that it must be top down directed if it is to achieve the significant and long-lasting effects that are possible. Thus, there should be a strong purposeful and systems management orientation to reengineering, even though it may have well major implications for such lower level concerns as structural facets of a particular product.

Our paper is organized as follows. We first provide some definitions of reengineering. Then, we discuss some five of the many perspectives that have been taken relative to reengineering. Then, we provide some summary comments.

II. PERSPECTIVES ON REENGINEERING

In this section, we provide definitions and perspectives on what we consider to be three related but different types of systems reengineering: reengineering at the levels of

- product,
- process or product line, and
- systems management.

There have been a number of definition, formal and informal, of reengineering. The word is occasionally spelled as re-engineering. We choose the former spelling here; both are correct.

A. Product Reengineering

The term reengineering could mean some sort of reworking or retrofit of an already engineered product. This could well be interpreted as maintenance or refurbishment. As we have noted previously [3], maintenance can be viewed from reactive or corrective, interactive or adaptive, and proactive or perfective perspectives. Or, reengineering could be interpreted as reverse engineering, in which the characteristics of an already engineered product are identified, such that the product can perhaps be modified or reused. Inherent in these notions are two major facets of reengineering.

1. It improves the product or system delivered to the user for enhanced reliability, maintainability, or for an evolving user need.
2. It increases understanding of the system or product itself.

Thus, this interpretation of reengineering is almost totally product focused. We will call it product reengineering.

Thus, we might say that

> ***Product reengineering is the examination, study, capture, and modification of the internal mechanisms or functionality of an existing system or product in order to reconstitute it in a new form and with new features, often to take advantage of newly emerged technologies, but without major change to the inherent functionality and purpose of the system.***

This definition indicates that product reengineering is basically structural reengineering with, at most, minor changes in purpose and functionality of the product that is reengineered. This reengineered product could be integrated with other products having rather different functionality than was the case in the initial deployment. Thus, reengineered products could be used, together with this augmentation, to provide new functionality and serve new purposes. A number of synonyms for product reengineering easily come to mind. Among these are: renewal, refurbishing, rework, repair, maintenance, modernization, reuse, redevelopment, and retrofit.

A specific example of a product reengineering effort might be that of taking a legacy system written in Cobol or Fortran, reverse engineering it to determine the system definition, and then reengineering it in C^{++} or Ada. Depending upon whether or not any modified user requirements are to be incorporated into the reengineered product, we would either reengineer the product, through a forward engineering effort, just after reverse engineering had determined either the initial development (technical) system specifications, or after reverse engineering far enough to determine user requirements and user specifications, and then updating these. This reverse engineering concept [4], in which salient aspect of user requirements or technological specifications are recovered from examination of characteristics of the product, predates product reengineering.

Figure 4 illustrates product reengineering conceptually. An IEEE software standards reference [5] states that "*reengineering is a complete process that encompasses an analysis of existing applications, restructuring, reverse, and forward engineering.*" The IEEE standard for software maintenance [6] suggests that reengineering is a subset of software engineering that is comprised of reverse engineering and forward engineering. We have no disagreement with the sort of definition at all, we prefer to call it product reengineering for the reasons just stated. It is also necessary to consider reengineering at the levels of processes and systems management if we are to take full advantages of the major opportunities offered. Thus, the qualifier "product" appears appropriate and desirable in the context used here.

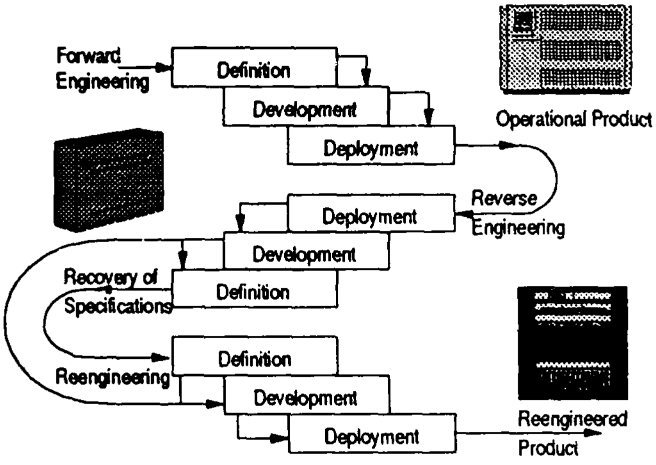


Fig. 4 Representation of Product Reengineering

B. Process Reengineering

Reengineering can also be considered at the levels of processes and systems management. At the level of processes only, the effort would be almost totally internal. It would consist of modifications to whatever standard lifecycle processes are in use in a given organization in order to better accommodate new and emerging technologies or new customer requirements for a system. For example, an explicit risk management capability might be incorporated at several different phases of a given lifecycle and accommodated by a revised configuration management process. This could be implemented into the processes for RDT&E, acquisition, and systems planning and marketing. Basically, reengineering at the level of processes would consist of the determination, or synthesis, of an efficacious process for ultimately fielding a product on the basis of a knowledge of generic customer requirements, and the objectives and critical capabilities of the systems engineering organization. Our Figure 5 illustrates, conceptually, some of the facets of process reengineering.

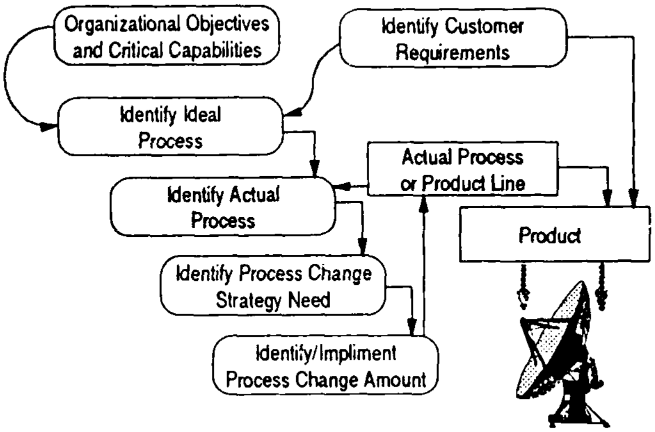

Organizational Objectives and Critical Capabilities
Identify Customer Requirements
Identify Ideal Process
Actual Process or Product Line
Identify Actual Process
Product
Identify Process Change Strategy Need
Identify/Impliment Process Change Amount

Fig. 5 Conceptual Illustration of Process Reengineering

In accordance with this discussion, we offer the following definition.

Process reengineering is the examination, study, capture, and modification of the internal mechanisms or functionality of an existing process, or systems engineering lifecycle, in order to reconstitute it in a new form and with new features, often to take advantage of newly emerged capabilities, but without changing the inherent functionality and purpose of the process itself that is being reengineered.

We could reengineer either the process for RDT&E, system acquisition or production, or systems planning and marketing. Among the first discussions of this sort of effort at business process reengineering, although the word redesign was used rather than reengineering, is in a contemporary paper by Davenport and Short [7]. This was greatly expanded upon in a recent and seminal text by Davenport [8] which does make use of the term reengineering. We will provide an overview of this major work in a later section.

These and other authors recognize, of course, that redesign of processes only and without attention to reengineering at a higher level than processes only may, in many instances, represent an incomplete and not fully satisfactory way to improve organizational capabilities. Thus, the process considered as candidates for reengineering are high level managerial as well as operational processes. Information technology is considered to be a major enabling catalyst for process reengineering. Continuous process improvement and institutionalization is advocated.

Reengineering at the process level, and the resulting improvement in product that results from improvement in the product line, is essentially what the AT&T Bell Laboratories has used to identify a common taylorable systems acquisition process for Federal Systems Advanced Technology (FSAT) use as a deployment methodology for all Process Management Teams (PMTs) at this organization [9]. The indicated benefits to using a common and understood lifecycle include:

- shorter development cycles,
- fewer engineering change orders,
- products that fulfill customer expectations, and
- reduced program and product development costs throughout the lifecycle.

Thus, the process improvement results ultimately in an increase in effectiveness of product for the same cost, or a reduction in cost for the same effectiveness, or some blend of these two.

Essentially what we call process reengineering here is termed "*domain engineering*" by the Software Productivity Consortium [10]. The combination of domain engineering with the "*application engineering*," or lifecycle, effort needed to product the actual product is termed "*synthesis*." It is intended for use in the systems acquisition, procurement, or production lifecycle. Figure 6 illustrates the synthesis concept which is intended, in part, to facilitate the incorporation of reusable software products in new software systems. The synthesis process proceeds as follows.

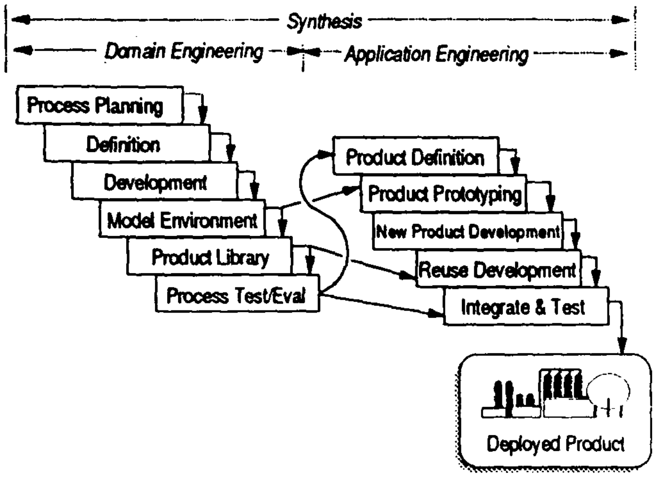


Fig. 6 The Software Productivity Consortium Synthesis Process

1. A set of 6 domain engineering phases is used to develop the product line. In the first phase, process, or product line, planning is accomplished. This is presumably based on knowledge of the organization and its critical core capabilities, and customer needs, as illustrated in Figure 5.
2. This is followed by a process definitional phase, called product line analysis by SPC, in which the requirements for the process or product line are specified.
3. This leads to product-line, or lifecycle process, development.
4. A modeling and simulation environment is next constructed such that it will be possible to accomplish prototyping in the actual lifecycle for production of the product.
5. A product library is next constructed. This is comprised of reusable software modules and code generators which can produce executable code from a set of input technical specifications. A code generator can be viewed as a special type of reusable software product.
6. A process, or product line, test and evaluation facility is constructed next. This has the capacity for test and evaluation of products from the product line. Following this phase, the applications engineering lifecycle begins. This involves actual definition, development, and deployment on the basis of the just engineered product line, or process, lifecycle.
7. Product definition is the first phase in actual production of the software product. This is achieved by identifying the user requirements for a software product and translating them into a set of specifications.
8. Product prototyping occurs in the second phase of the applications engineering lifecycle. Here, a prototype is built and, with user interaction presumably, used to refine the technical specifications for the software product. The modeling and simulation environment, built earlier in domain engineering, is used for this purpose.
9. New product development occurs next. In this particular instance, this refers to the production of custom built executable code for those portions of the software product that are not to be comprised of reusable code.
10. Reuse development occurs next. In this phase, the reengineered code that has become reusable code and code produced by applications generators in phase 5 of domain engineering is integrated in with the customized code to result in a complete functioning product.
11. In the final phase, integration and test of this functional software occurs. This generally involves use of the test suites produced by the product line test and evaluation facility.
12. The deployed product results from this effort.

As indicated in Figure 6, there are several entry points from domain engineering to application engineering as several of the results of intermediate phases of domain engineering (at phases 5. 5, and 8) are used in the actual lifecycle that is developed as a result of domain engineering.

C. Reengineering at the Level of Systems Management

At the level of systems management, reengineering is directed at potential change in all business or organizational processes, including the systems acquisition process lifecycle itself. Many authors have discussed reengineering the corporation. Arguably, the earliest use of the term *business reengineering* was by Hammer [11], in 1990, and more fully documented in a more recent work on *Reengineering the Corporation* [12]. There are a small plethora of related works, as we will soon discuss.

Hammer's definition of reengineering "*Reengineering is the fundamental rethinking and radical redesign of business processes to achieve dramatic improvements in critical, contemporary measures of performance, such as cost, quality, service and speed*" is a definition of what we will call reengineering at the level of systems management. There are four major terms in this definition.

- *Fundamental* refers to a large scale and broad scope examination of virtually everything about an organization and how it operates. The purpose is to identify potential weakness that are in need of diagnosis and correction.

- ***Radical redesign*** suggests disregarding existing organizational processes and structures, and inventing totally new ways of accomplishing work.
- ***Dramatic improvements*** suggests that, in Hammer's view, reengineering is not about making marginal and incremental improvements in the status quo. It is about making "quantum" leaps in organizational performance.
- ***Processes*** represent the collection of activities that are used to take input materials, including intellectual inputs, and transform them into outputs and services that have value to the customer.

Hammer suggests that reengineering and revolution are almost synonymous terms. His identification of the three types of firms that attempt reengineering - those in trouble, those who see trouble coming, and those who are ambitious and seek to avoid impending troubles.

He indicates that one major catalyst for reengineering is the creative use of information technology. Reengineering, is not just automation however, it is the ambitious and rule breaking study of everything about the organization to enable more effective and efficient organizational processes to be designed.

We essentially share this view of reengineering at the level of systems management. Our definition is similar.

> ***Systems management reengineering is the examination, study, capture, and modification of the internal mechanisms or functionality of existing system management processes and practices in an organization in order to reconstitute them in a new form and with new features, often to take advantage of newly emerged organizational competitiveness requirements, but without changing the inherent functionality and purpose of the organization itself.***

We make no representation that this definition, or the other two for that matter, of reengineering is at all the same across the many works that we discuss. Figure 7 represents this conception of reengineering at the level of systems management.

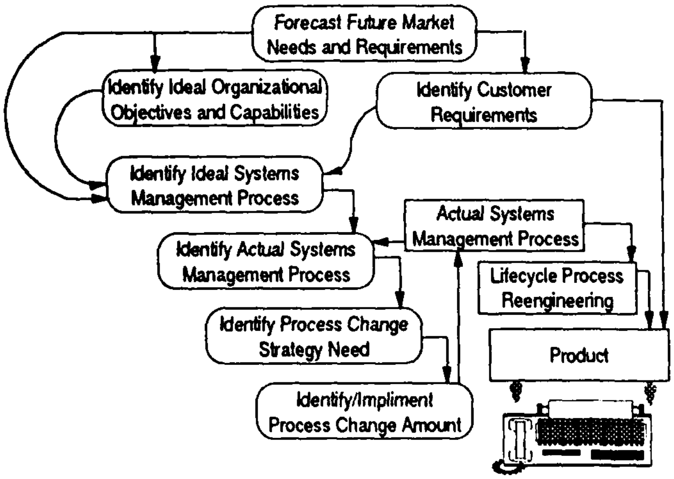

Forecast Future Market Needs and Requirements
Identify Ideal Organizational Objectives and Capabilities
Identify Customer Requirements
Identify Ideal Systems Management Process
Actual Systems Management Process
Identify Actual Systems Management Process
Lifecycle Process Reengineering
Identify Process Change Strategy Need
Product
Identify/Impliment Process Change Amount

Fig. 7 Reengineering at the Level of Systems Management

Lifecycle process reengineering occurs as a natural byproduct of reengineering at the level of systems management. This may or may not result in the reengineering of already existing products. Generally, it will as new products and competitive strategies are a major underlying objective of reengineering at the level of systems management, or organizational reengineering as it is more commonly called.

D. Perspectives of Reengineering

This very brief discussion of reengineering suggests that we can consider reengineering at three levels: systems management, lifecycle processes, and product. The major purpose of reengineering, regardless of whether it is at the product level or the process level or the level of systems management, is to enable us to product a better product for the same cost, or a lower cost product at a cost comparable to that for the initial product. Thus it improves competitiveness of the organization in coping with changing external situations and environments. We may approach reengineering, at any or all of these levels, from either of three perspectives:

- *reactive*, because we realize that we are in trouble and perhaps in a crisis situation, and reengineering is one way to bring about needed change;
- *interactive*, because we wish to stay abreast of current changes as they evolve; or
- *proactive*, because we wish to position our organization now for changes that we believe will occur in the future, and to emerge in the changed situation as a market leader.

In our next section, we examine some of the many contemporary ideas that have been expressed about the subject of reengineering.

II. AN OVERVIEW OF REENGINEERING APPROACHES

As we have noted, it is possible to consider reengineering at the levels of strategy and systems management, process or product line, or product. In this section, we provide an expanded overview of each by means of an overview of contemporary literature on reengineering.

Without question, more has been written about reengineering at the levels of strategy or systems management that at the other levels. This is not unreasonable since reengineering efforts at the level of organizational strategy have direct implications for management at the levels of management control, and thence at the levels of process to implement management controls, and product.

It is difficult to trace the origins of reengineering in any unique manner. The notions of reengineering are very interrelated with those of systems engineering, information technology, strategic planning, and many other subjects. The term systems management reengineering is not at all common. Others have used such related terms as systems reengineering, organizational reengineering, corporate reengineering, and business process improvement. These terms can, however, be interpreted to mean reengineering at the level of system or product, reengineering at the level of lifecycle process, and of the structure and functionality of the organization.

A. Business Process Improvement

In 1992, Harrington published a seminal work on business process improvement [13]. The major thesis of the work is that it is business and manufacturing processes that are the key to error-free performance. His view, that the process is the problem and not the employees, is essentially that of Deming and others in the TQM areas. Harrington defines a process as a group of activities that take inputs, adds value to them, and produces an output that is in support of an organization's objectives. Two generic types of processes are identified. Production processes are directly concerned with yielding the output product or service. Business processes support production processes. These include design of production processes, payroll processes, and engineering change processes.

Many works on reengineering recognize a dichotomy between organizational processes and organizational functionality. Most organizations are structured into vertically functioning groups, or hierarchies, and most processes are organized into horizontal phases for work flow. The many waterfall lifecycle models we have illustrated in our effort surely demonstrate this. Three desirable attributes of *Business Process Improvement* (BPI) are:

- ***efficiency***, in terms of minimizing the cost of the resources used;
- ***effectiveness***, in terms of producing desired results; and
- ***adaptability***, in terms of flexibility in accommodating changing customer and organizational needs.

To do this requires processes with the following well defined characteristics: ownership and accountability, boundaries and scope, interfaces and responsibilities, work tasks and training requirements, measurement and feedback controls, customer related measurements and targets, cycle times, and formalized change procedures. Figure 8 illustrates the BPI process.

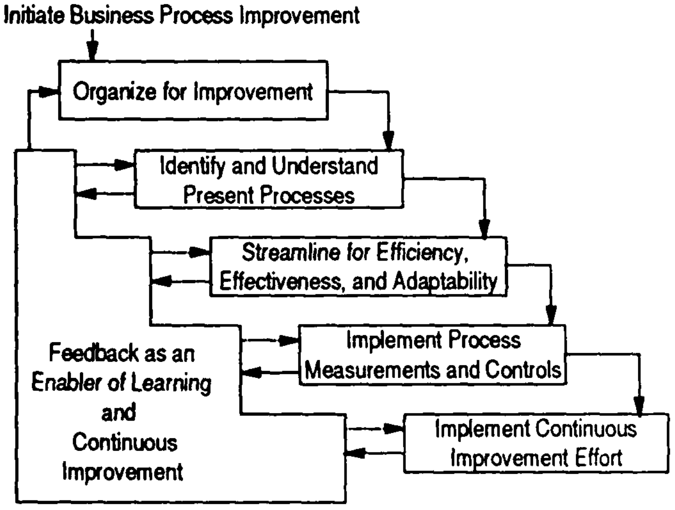

Initiate Business Process Improvement
Organize for Improvement
Identify and Understand Present Processes
Streamline for Efficiency, Effectiveness, and Adaptability
Implement Process Measurements and Controls
Implement Continuous Improvement Effort
Feedback as an Enabler of Learning and Continuous Improvement

Fig. 8 Interpretation of Business Process Reengineering

Harrington suggests five phases for business process improvement (BPI). These, represented in Figure 8, are as follows.

1. *Organize for Improvement* - This phase involves several steps. First, it is necessary to establish an *Executive Improvement Team* (EIT). Then a BPI champion is appointed and executive training is provided. An normative improvement model is identified and BPI objectives are communicated to employees. Next, it is necessary to review business strategy and anticipated customer requirements. This enables the organization to identify and select critical processes for improvement and to appoint process owners. *Performance Improvement Team* members are selected.
2. ***Develop an Understanding of the Various Processes Currently in Use*** - This phase includes defining the scope and objectives of current processes and the boundaries within which they are functional. This includes a variety of analysis functions, including structuring current processes and detection and diagnosis of areas for potential improvement.
3. ***Streamline Organizational Processes for Enhanced Efficiency, Effectiveness, and Adaptability*** - Current processes are corrected through various streamlining approaches that are responsive to the diagnosis performed in the last phase. This includes eliminating bureaucracy, reducing the opportunity for errors, reducing non-value-added activities, and otherwise simplifying processes such as to reduce cost and increase effectiveness.
4. ***Implement a Program of Systematic Measurement and Controls*** - In this phase, a program of systematic measurements is used as a quality control and monitoring system in order to both maintain the new process productivity and to keep from regressing into poorer process implementations only because they have been used in the past. An extensive program of measurements, feedback and action is suggested. This includes audits of *what are denoted as Poor Quality Cost* (PQC) facets.
5. ***Continue the Evolutionary Improvement*** - Periodic reviews of the effort are used to enable detection, diagnosis, and correction of difficulties as improvement continues. A formal program of business process qualification is suggested through use of a six level PBI scale for process maturity status. The levels are unknown, understood, effective, efficient, error-free, and world class.

Harrington's effort has much in common with those in strategic quality management [1] [14]. The work has much relevance to benchmarking, a topic we will soon discuss, and to systematic measurements. The author has had much experience in efforts of this sort, and this is much in evidence in his high caliber writing.

B. Intelligent Enterprise

The efforts of James Brian Quinn are focused on knowledge based services as a necessary compliment to manufacturing efforts. His view of an organization is basically that of a collection of service activities and that both service and product oriented organizations will obtain their major competitive advantages not from superior physical facilities and materials alone, but from knowledge and service based capabilities. In three recent works [15] [16] [17] concerning technologies in services, the claim is set forth that it is services and not manufacturing activities that provide the major course of value to consumers. This does not suggest that services replace manufacturing, but rather that if it were not for the value added by the services that are associated with a manufactured product, there will be far diminished value to the product itself. Several characteristics of new organizations are cited:

- "infinitely flat," or horizontal;
- "spider's web" like, or non-hierarchical and with highly networked interconnections;
- "hollow corporations," in which outsourcing of both products and services becomes an increasing reality;
- demolished bureaucracies and vertical integration; and
- "intellectual holding companies," in which intellectual technologies are critical.

It is represented that this will lead to precise and swift strategy execution, the leveraging and retention of key people, and "creative management for profits."

A 1992 text by Quinn [18] represents a definitive integration and synthesis of earlier efforts on this subject. It is much concerned with concentrating organizational strategy on core intellectual competencies and core service competencies. He suggests four key rules to follow in order to generate success in this regard.

1. Focus internal organizational resources on those relatively few basic sources of intellectual strength and service strength that will create and sustain a real and *meaningful* distinctiveness to the customer over the long term.
2. Approach the remaining capabilities as a non critical set of services activities which may be supplied internally or outsourced from external suppliers who compete well in functional activities related to these capabilities.
3. Sustain success by building entry barriers around those selected critical core capabilities to prevent a competitor from assuming a substantial market position.
4. Plan and control outsourcing such as never to become either dependent upon, or dominated by, external suppliers.

Strategic sourcing, including outsourcing, is a major ingredient in these maxims. we discuss it later. Appropriate interfaces between production efforts and service efforts are also stressed as is the management of knowledge based intellect and professional intellect.

Seven types of innovative organizations are identified in this work.

1. ***Basic research organizations*** support large RDT&E units. They select products for development on the basis of careful and conservative tradeoffs among risk and potential profit.
2. ***Large system producers*** develop large-scale systems that generally cost a great deal and which must perform in a reliable manner for a very long time.
3. ***Dominant market share oriented companies*** are often not the first to introduce an emerging technology into the marketplace. They often support large research units and plan market entry and product evolution to obtain maximum penetration, decreased risk, great product reliability, and lower overall costs.
4. ***State of the art technology development companies*** are preeminent in their knowledge of a specific technology. Often they operate in markets in which technical performance criteria are the major drivers of demand and are therefore able to self-define the characteristics of next generation technology.
5. ***Discrete freestanding product line companies*** form strong research divisions and act as entrepreneurial units. They depend more on technology push for their initiatives than demand pull, and will often introduce new products on a small scale and obtain real time market test rather than pay for very expensive marketing studies that may not be as effective as small scale product introduction and interactive modification of the product to meet consumer needs.
6. ***Limited volume*** or ***fashion companies*** provide [illegible] scale and limited quantity products to a specialized market niche.
7. ***Job shop*** or ***custom design*** companies provide one of a kind products that meet an individual customer's requirements. These may often involve flexible manufacturing or mass customization approaches to enable highly specialized design to met an individual customer requirements.

Clearly, this is not a mutually exclusive listing. Nor is the listing collectively exhaustive. Each of the industry types may be associated with an organizational strategy and configuration that is tuned to providing maximum success opportunities. Each type will have a different propensity for organizational growth, maturity, decline, and rebirth as we discussed in our last chapter. In his effort, Quinn discusses the typical product lifecycle for each organizational characteristics, the mix of attributes that describe typical innovations, and recommended organizational structures for RDT&E, acquisition or production, planning and marketing, and interfaces between marketing and customers. Approaches for managing the intelligent enterprise are also suggested. These involve efforts that also encompass core capabilities, outsourcing, total quality management, and benchmarking, as well as some of the other ingredients illustrated in Figure 8.2.

C. Process Innovation

In the aforenoted text by Davenport [8], a careful distinction is made between process improvement and process innovation. His fundamental distinctions are that improvement is continuous and incremental in nature, deals with the existing process, can be accomplished in a relatively short time, is a bottom-up activity, and is a narrow scope effort with relatively moderate attendant risks. On the other hand, innovation is generally a discrete phenomenon that is revolutionary and radical in nature. It starts with a zero base as contrasted with the existing process, can only be accomplished over a long time, is a top-down activity, is a broad scope effort that cuts across all of the functional areas and processes in the organization, and is usually characterized by high associated risks.

Of course, there may exist questions of whether an innovation is a major improvement and whether a set of incremental improvements does not add up to an innovation. Rather than a binary scale to separate these two, perhaps it would be best to consider a continuous scale. This would enable us to consider incremental improvement at one end and radical innovation at the other. Obviously, either can be appropriate or inappropriate in specific situations. Change increments may be so small that centuries would be required to accomplish any change with appreciable value added. On the other hand radical innovation may be so dramatic that the organization is culturally and otherwise unable to adapt. Culture shock and customer revolt is often the result.

Information technology is suggested as a major enabler of process innovation, together with the organizational and human enabler, and an enabler based on measurements associated with process *information and management of the information* environment. Essentially these three were suggested as enablers in our Figure 1. The process innovation process itself is comprised of 5 phases, and a number of activity steps within each phase, as connoted by Figure 9. We note that there are opportunities both for traditional incremental improvement and the more extreme innovative improvement. Presumably, the overall process is iterative and this leads to continual innovation, as suggested by the feedback from phase V to phase I in the illustration of Figure 9.

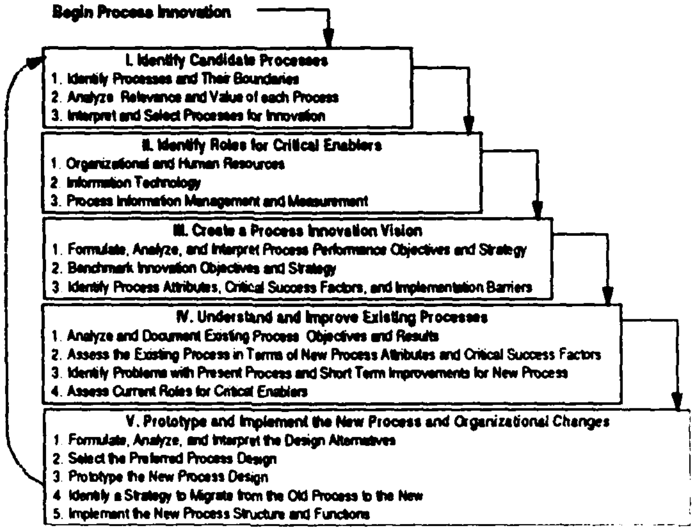

Begin Process Innovation
I. Identify Candidate Processes
1. Identify Processes and Their Boundaries
2. Analyze Relevance and Value of each Process
3. Interpret and Select Processes for Innovation
II. Identify Roles for Critical Enablers
1. Organizational and Human Resources
2. Information Technology
3. Process Information Management and Measurement
III. Create a Process Innovation Vision
1. Formulate, Analyze, and Interpret Process Performance Objectives and Strategy
2. Benchmark Innovation Objectives and Strategy
3. Identify Process Attributes, Critical Success Factors, and Implementation Barriers
IV. Understand and Improve Existing Processes
1. Analyze and Document Existing Process Objectives and Results
2. Assess the Existing Process in Terms of New Process Attributes and Critical Success Factors
3. Identify Problems with Present Process and Short Term Improvements for New Process
4. Assess Current Roles for Critical Enablers
V. Prototype and Implement the New Process and Organizational Changes
1. Formulate, Analyze, and Interpret the Design Alternatives
2. Select the Preferred Process Design
3. Prototype the New Process Design
4. Identify a Strategy to Migrate from the Old Process to the New
5. Implement the New Process Structure and Functions

Fig. 9 The Process of Process Innovation

Not explicitly shown in this figure is the organizational communications that occur at each phase of the effort and the commitment building that is also needed. Davenport examines a number of enablers of specific processes including: planning, research, development, design, production, marketing, and sales.

Also considered briefly are the role of organizational culture in shaping desirable innovation strategies. In a related work [19], this is considered in greater detail. Five models of information culture, denoted as information politics in the paper, are defined.

1. *Technocratic Utopianism* is a formal analytical approach to information management. It stresses highly quantitative approaches, major reliance on emerging technologies, and full information assets. There is an underlying assumption that technology will resolve all problems. Organizational and political issues are generally ignored, perhaps because they are considered unmanageable. In reality, the major focus may be on the technologies used to process data rather than on the information content in the data. The major objective is modeling and use of organizational data. Data engineering, and some aspects of information engineering, is considered king. All that is really needed for organizational paradise is the selection of the best hardware and software, and use of the most appropriate CASE tools to assist in constructing the most appropriate entity relationship diagrams and data flow models.
2. *Feudalism* was the model of information politics most often encountered in this study. In this model, a number of people in feudal departments individually control information acquisition, representation, storage, analysis, transmission, and use. The language used for information representation is often different across the various departments. Critical information that affects the organization as a whole is often not collected. When a subset of this is collected by an individual department, it may not be passed on outside the boundaries of the department. Making organizationally informed decisions for the common good is often very difficult, as a result of the information poor environment. Sometimes, however, strategic alliances across departments are possible and innovation within these units may be possible.
3. *Monarchy* is a pragmatic solution to the difficulties inherent in the feudal model. Information management is centralized and there is little autonomy concerning information policies. A benign monarch who is enlightened concerning information technology and organizational needs for information may set up, potentially through a Chief Information Officer (CIO), a very effective and efficient system. A constitutional monarchy may well accomplish this; a despotic monarchy seldom will perform satisfactorily. An Executive Information System (EIS) or Executive Support System (ESS) [20] may well be a beneficial product of a constitutional monarchy. One problem with a constitutional information monarchy, however, is the problem of succession when the monarch dies, retires, or is overthrown. This can lead to information anarchy.
4. *Anarchy* is the result of complete absence of any information policy. Usually this is not a willfully imposed model but a result of a breakdown of one of the centralized approaches, such as a monarchy. This results in individuals and units in an organization managing their own information resources and developing information reports that serve their own needs rather than the organization as a whole. The major shortcoming of an information anarchy is not the wasted effort involved in the redundant information processing and storage effort across units, but the tower of Babel effect that results in terms of providing information that is useful for the entire organization. There will seldom exist any interoperability across units, at the level of either data or

information. As a consequence of this, the various units may well present entirely different results from using the same data on the generally very different alternative model management systems, including intuitive model manaqement systems, that will be in use.

5. ***Federalism*** involves the use of negotiations [21] [22] to bring competing and noncooperative individuals and units into consensus on information related issues. Strong central leadership and an organizational culture that encourages learning, cooperation and consensus are required in order for this model to work. Understanding by top organizational management of both information technologies and the value of information is a need if a shared information vision is to be created.

It seems quite clear that these models are neither mutually exclusive nor collectively exhaustive. A given organization may well function with a hybrid model, and the model that is in use at a given time may evolve from one form to the other with changes in organizational dynamics. Some of the models of organizational cultures discussed in [1] and [2] seem also very plausible as information cultures as well.

In a study of the information culture at 25 companies, Davenport and his colleagues found that the feudal model was the most common model in 12 companies. Federalist, monarchist, and technocratic utopianist models were found with approximately equal frequency in that 8, 7, and 9 companies were found predominantly following the prescriptions of these models. Only 4 organizations followed the anarchist model. They also ranked these five alternative models of information cultures according to their scoring on four attributes of information value:

- commonality of vocabulary,
- access to information,
- information quality, and
- information management efficiency.

Each of the four attributes had equal weight in this study. Their conclusion, that the federalist and monarchist models are most appropriate, in most situations, seems inescapable. The feudalist and anarchist models, which actually seem to have a lot in common, were the poorest performers.

In order to determine a "best" model for a given organization, we need to know the current model that is in use, and the evolutionary or revolutionary model for the information culture to which the organization should be moving. Four suggestions are given; our interpretation of these is a s follows.

1. Match information management strategies to the organizational culture extant and the organizational culture the organization is desirous of adopting.
2. Practice technological realism, both in terms of information technologies themselves, interoperability across potentially different platforms, and the value of information that is not always easily captured in electronic form.
3. Select appropriate information managers, both from the standpoint of technical skills and broad-scope organizational understanding skills.
4. Avoid building information empires, such as through creation of despotic monarchies and information czars.

Explicit recognition of information management cultures and managing them constructively is suggested as the bottom line. In conformance to our discussions in [1] and [2], we would prefer to call these information management styles, and information cultures or information management cultures, rather than "information politics" term used by these authors. As we have noted before, there are semantic differences in use by the many workers in these areas. In terms of the vocabulary we use here, the organizational political cultures identified by the authors are the anarchist and the feudalist models. These are not to be encouraged. On this, of course, we agree strongly with the authors of this excellent work.

D. Benchmarking

A benchmark is much like a breakpoint [23]. Actually, the concept of a benchmark predates that of a breakpoint. Much further discussion of benchmarking is available in [24] [25] [26]. A recent and definitive work by Watson [27] provides an excellent overview of many current efforts concerning benchmarking. He identifies four types of benchmarking efforts.

1. *Internal benchmarking* involves observations taken entirely within the same organization, generally of the best practices that have resulted in one segment of the organization that are desired to be transferred into anther segment of the organization. Internal benchmarking efforts at Hewlett-Packard are described.
2. *Competitive benchmarking* involves targeted best practices in an external organization. These can be at the level of systems management, processes, or product. Often, much secondary research is used for competitive benchmarking. The competitive benchmarking studies of the Ford Motor Company, and their use of these in developing Ford Tarus, are described.
3. *Functional benchmarking* is concerned with performance investigation within a specific functional area for an industry-wide function. The results of functional benchmarking are especially suited to identifying process improvement related benchmarks. The efforts of the General Motors Corporation in September 1994, which led to major GM strides to enhance product quality and reliability, are described.
4. *Generic benchmarking* is concerned with studying the best processes in actual use in practice in a given organization such as to enable analogous development of enhanced processes by the organization(s) sponsoring the benchmarking study. A generic benchmarking study by Xerox, which ultimately led to understanding and analogous modification and adoption of facets of the shipping processes activities of the catalog casual-clothing sales organization L. L. Bean, are described. This effort was so successful that it led to establishment of the Benchmarking Effectiveness Strategy Team (BEST) network to transfer lessons learned throughout the Xerox Corporation.

In our terminology, we would use the term functional product benchmarking to refer to benchmarking of the directly measurable aspects of a product. These serve to describe the operational capabilities that are supplied to the user of the product, or service. The nonfunctional attributes of a product are those constraints, alterables, or limitations that relate to the structural or purposeful properties of the product that are not a part of the functional properties. The nonfunctional attributes of a product relate to such important characteristics of a product as reliability, maintainability, quality, and availability. Thus what was accomplished here would not be described, using our terminology, as functional product benchmarking. It is really nonfunctional product benchmarking. The study conducted by GM was not restricted to automotive products. It was a study of quality processes at 11 cooperating organizations in a variety of product areas. They had established 10 hypotheses for empirical evaluation and these relate very closely to quality processes. It would, therefore, be very appropriate to denote this type of benchmarking as *functional process benchmarking.*

Like a breakpoint, a benchmark involves a baseline comparison of existing practices of an organization with those best practices as used by one organization, or perhaps many other organizations. These practices may be at the level of product, process, or systems management. Breakpoints and benchmarks have the same ultimate purpose, that of enabling and enhancing organizational improvements for customer satisfaction. The use of a benchmark can be reactive, in which case we desire to emulate the best practices of others that have already been established, or perhaps excel in these. In can be interactive or anticipative, in which case, we desire to foresee efforts that are now occurring and adjust organizational practices, more or less in real time, to keep abreast of the competition. We can use benchmarks for forecasting purposes such that we evolve entirely new forms of organizational results in the form of system management strategies, processes, and products that are each world class in quality and trustworthiness. In all cases, benchmarks are based on systematic measurements, both internal and external measurements, and serve as a catalyst for action and results. In this regard, benchmarks are also very much related to Critical Success Factors (CSF) [28] [29].

We see that a benchmark is a standard of excellence or achievement, or critical success factor that provides a baseline against which to measure or evaluate, or otherwise judge, similar entities. While the benchmarking notion may seem quite simple at first, there are really a number of benchmarking features that need to be examined. There are at least seven important and interrelated questions that need to be asked before we benchmark.

1. ***Why*** do we benchmark?
2. ***What*** do we benchmark?
3. ***Which*** benchmark measures do we need?
4. ***Who*** do we benchmark?
5. ***Where*** do we benchmark?
6. ***When*** do we benchmark?
7. ***How*** do we benchmark?

We can and should ask these questions of our organization, we can and should ask these questions of those organizations that represent competitors. We can and should ask these questions in the search for structural, functional, and purposeful answers. A similar set needs to be asked after we benchmark. Benchmarking efforts can be conducted relative to issues at the level of systems management, process, or product. Answers to these questions need to be obtained in terms of implications for each of these levels.

There can be some obvious ethical, moral, and legal concerns relative to benchmarking. Some aspects of benchmarking may seem to amount to spying, or espionage. Clearly, successful benchmarking should and can not involve behavior that is either immoral, unethical, or illegal. One of the major activities of the International Benchmarking Clearinghouse Committee of the American Productivity and Quality Center (APQC) have resulted in a set of guidelines, known as a Code of Conduct, for benchmarking that have been subscribed to by a considerable number of companies and which are generally felt to be above reproach [30]. There are seven principles in the Benchmarking Code of Conduct. They may be summarized as follows.

1. *Legality* - The acquisition of trade secrets is proscribed, as is doing benchmarking without first requesting approval by the benchmarkee. All actions and intents should be legal ones.
2. *Exchange* - The same type and level of information should be provided both by the benchmarked organization and the organization doing the benchmarking in an openly communicated exchange.
3. *Confidentiality* - Benchmarking communications should be considered as confidential to the organizations involved and obtained information concerning benchmarking should not be divulged outside the concerned organizations without prior consent of all parties.
4. *Use* - Benchmarking information should only be used for formulation of improvement options for processes or products within organizations participating in the study. Attribution of benchmarking partner names requires prior permission, and benchmarking information must not be used for marketing or sales purposes.
5. *First-Party Contact* - Benchmarking information contacts with a partner organization should be made through the point of contact established by the benchmarking partner. Mutual agreement must be reached for delegation of responsibility in this regard to other parties.
6. *Third-Party Contact* - Prior permission must be obtained before divulging the name of the benchmarking point of contact to a third party or in an open forum.
7. *Preparation* - A benchmarking contact with the point of contact at another organization should only be made after proper planning and preparation of an interview guide, in order to make the encounter efficient and effective.
8. *Completion* - Each benchmarking study must be completed to the satisfaction of all benchmarking partners in a timely manner, as agreed to prior to the study.
9. *Understanding and Action* - All partners should be treated with mutual respect and understanding, and information should be used as mutually agreed.

It seems quite clear that these conduct codes are also applicable as ethical codes of conduct, as well as being very useful standards for benchmarking practice. We can obtain benchmarks at the levels of systems management, process, and product.

E. Integrated Product Development

In many ways, integrated product development is an extension of concurrent engineering as discussed in [31] [32] [33]. It is also closely related to the other reengineering approaches we describe here. The following definition seems appropriate.

> ***Integrated product development is a systems management philosophy and approach that uses functional teams to produce a efficient and effective process for the ultimate deployment of a product that satisfies customer needs through concurrent application and integration of all necessary lifecycle processes.***

Integrated Product Development, or IPD, involves systems management, leadership, systems engineering processes, the products of the process, concurrent engineering and integration of all necessary functions and processes throughout the organization, to result in a cost-effective product that provides total quality and customer need satisfaction, generally in a just-in-time fashion.

Thus, IPD is an organization's product-development strategy. It is focused on results. It also addresses the organizational need for continual enhancement of efficiency and effectiveness in all of its processes that lead to a product, or service. There are many focal points for IPD. Twelve are particularly important.

1. A ***customer satisfaction focus*** is needed as a part of competitive strategy and ***is the result of a successful*** competitive strategy.
2. A ***results focus*** and a ***product focus*** is needed in order to bring about total customer satisfaction.
3. A ***process focus*** is needed as high quality competitive products that satisfy customers and result in organizational success come from efficient and effective processes. This necessarily requires process understanding.
4. A ***strategic planning and marketing focus*** is needed to insure that product and process lifecycles are fully integrated throughout all organizational functions, external suppliers, and customers.
5. A ***concurrent engineering focus*** is needed to insure that all functions and structures associated with fulfilling customer requirements are applied throughout the lifecycle of the product to insure correct people, correct place, correct product, and correct time deployment.
6. An ***integration engineering focus*** is needed to insure that relevant processes and the resulting processes fit together in a seamless manner.
7. A ***teamwork and communications focus*** is needed to insure that all of the functional, or multifunctional, teams function synergistically for the good of the customer and organization.
8. A ***people empowerment focus*** is needed such that all decisions are made by qualified people at the lowest possible level that is consistent with authority and responsibility. Empowerment is a responsibility and not just an entitlement and entails commitment and appropriate resource allocation to support this commitment.
9. A ***systems management reengineering focus*** is needed, both at the levels of radical and revolutionary change, as well as for evolutionary change, such as to also result in radical, revolutionary, or evolutionary changes in processes and product.
10. A ***organizational culture and leadership focus*** is needed in order to successfully accommodate changed perspectives relative to customers, total quality, results and products, processes, employees, and organizational structures.
11. A ***methods, tools, and techniques focus*** is needed as they are needed throughout all aspects of the IPD process even through they alone will not bring about success.
12. A ***systematic measurements focus***, primarily on proactive measurements but also on interactive and reactive measurements, is needed as we need to know where to go and where we are now, in order to make progress towards getting there.

All of this should bring about high quality, continual and evolutionary, and perhaps even radical and revolutionary, improvement for customer satisfaction. Each of these could be expanded into a series of questions, or a checklist, and used to evaluate the potential effectiveness of a proposed integrated product development process and team. While our discussion of IPD may make it seem as an approach uniquely suitable for system acquisition, production, or procurement; it is equally applicable to the products of the RDT&E and marketing lifecycle.

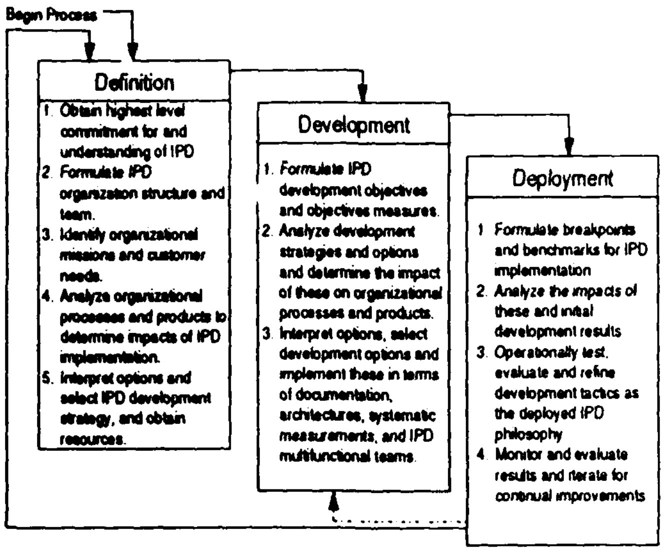
We see that IPD is a people, organizations, and technology focuses effort as linked together through a number of lifecycle processes by systems management. These are major ingredients for all our efforts here, as we suggested in Figure 8.1. The major result of IPD is the ability to make optimum decisions within available resources and to execute them efficiently and effectively in order to achieve three causally linked objectives:

- to integrate people, organizations, and technology into a set of multifunctional and networked product development teams,
- to increase the quality and timeliness of decisions through centrally controlled, decentralized and networked operations, and thereby
- to completely satisfy customers through quality products and services that fulfill their expectations and meet their needs.

The bottom line is clearly customer satisfaction through quality, short product delivery time, reduced cost and improved performance and functionality. Equally supported by IPD are organizational objectives for enhanced profit, well-being of management, and a decisive and clear focus on risk and risk amelioration.

Figure 10 illustrates a suggested sequences of steps and phases to establish an integrated product development process. As with other efforts, this embodies the definition, development, and deployment triage we have used so often in this book. As the implementation of the detailed steps in these phases is relatively standard, we will not describe them further here.

Appropriate references to IPD include [34] and [35]. At this point, IPD is a relatively new concept and there are few available references on the subject.



Definition

1. Obtain highest level commitment for and understanding of IPD
2. Formulate IPD organization structure and team.
3. Identify organizational missions and customer needs.
4. Analyze organizational processes and products to determine impacts of IPD implementation.
5. Interpret options and select IPD development strategy, and obtain resources.

Development

1. Formulate IPD development objectives and objectives measures.
2. Analyze development strategies and options and determine the impact of these on organizational processes and products.
3. Interpret options, select development options and implement these in terms of documentation, architectures, systematic measurements, and IPD multifunctional teams.

Deployment

1 Formulate breakpoints and benchmarks for IPD implementation
2. Analyze the impacts of these and initial development results
3. Operationally test, evaluate and refine development tactics as the deployed IPD philosophy
4. Monitor and evaluate results and iterate for continual improvements

Fig. 10 Phases in Integrated Product Development Process

F. Product Reengineering

Reengineering at the level of product has received much attention in recent times, especially in information technology and software engineering areas. This is not a subject that is truly independent of reengineering at the levels of either systems management or of a single lifecycle process.

As we noted earlier, much of product *reengineering* is very closely associated with reverse engineering to recover either design specifications or user requirements. This is then followed by refinement of these requirements and/or specifications and the forward engineering to result in an improved product. The term reverse engineering, rather than reengineering, we used in one of the early seminal papers in this area [36]. In this work, as well as in a more recent chapter on the subject [37], the following efforts represent both the taxonomy of and phases for what we denote here as *product reengineering*.

1. ***Forward engineering*** is the original process of defining, developing, and deployment of a product, or realizing a system concept as a product.

2. ***Reverse engineering***, sometimes called inverse engineering, is the process though which a given system or product is examined in order to identify or specify the definition of the product either at the level of technological design specifications or system or user level requirements.

   2.1 ***Redocumentation*** is a subset of reverse engineering in which a representation of the subject system or product is recreated for the purpose of generating functional explanations of original system behavior and, perhaps more importantly, to aid the reverse engineering team in better understanding the system both at a functional and structural level. There are number of redocumentation tools for software available and some of these are cited in [37]. One of the major purposes of redocumentation is producing new documentation for an existing product where the existing documentation is faulty, and perhaps virtually absent.

   2.2 ***Design Recovery*** is a subset of reverse engineering in which the redocumentation knowledge is combined with other efforts, often involving the personal experiences and knowledge of others about the system, that lead to functional abstractions and enhanced product or system understanding at the level of function, structure, and even purpose. We would prefer to call this deployment, development (which would include design) recovery, and definition recovery, depending upon the phase in the reverse engineering lifecycle at which the recovery knowledge is obtained.

3. ***Restructuring*** involves transformation of the reverse engineering information concerning the original system structure into another representation form. This generally preserves the initial functionality of the original system, or modifies it slightly in a purposefully manner that is in accord with the user requirements for the reengineered system and the way in which they differ from the requirements for the initial system. For our purposes, the terms deployment restructuring, development restructuring, and definition restructuring seem to be appropriate disaggregations of the restructuring notion.

4. ***Reengineering*** is, as defined in these efforts, equivalent to redevelopment engineering, renovation engineering, and reclamation engineering. Thus, it is more related to maintenance and reuse than the other forms of systems management and process reengineering that we have discussed in this chapter. Reengineering is the recreation of essentially the original system in a new form that has improved structure but generally not much altered purpose and function. The nonfunctional aspects of the new system may be considerably different from those of the original system, especially with respect to quality and reliability.

Figure 4, which illustrates product reengineering, involves essentially these six activities.

We can recast this by considering a single phase for definition, for development, and for deployment that is exercised three times. We then see that there is a need for recovery, redocumentation, and restructuring as a result of the reverse engineering product obtained at each of the three basis phases. This leads us to suggest Figure 11 as an alternative way to represent Figure 4 and an interpretation of the representations used in [36] and [37]. Their discussions utilized a three phase generic lifecycle of requirements, design and implementation. In this representation, implementation contains some of the detailed design and production efforts of our development phase and potentially less of the maintenance efforts that follow initial fielding of the system. The restructuring effort, based on recovery and redocumentation knowledge obtained in reverse engineering, is used to effect deployment restructuring, development restructuring, and definition restructuring. To these restructured products, which might well be considered as reusable products, we augment the knowledge and results obtained by detailed consideration of potentially augmented requirements.

These augmented requirements are translated, together with the results of the restructuring efforts, into the outputs of the reengineering effort at the various phases to ultimately result in the reengineered product.

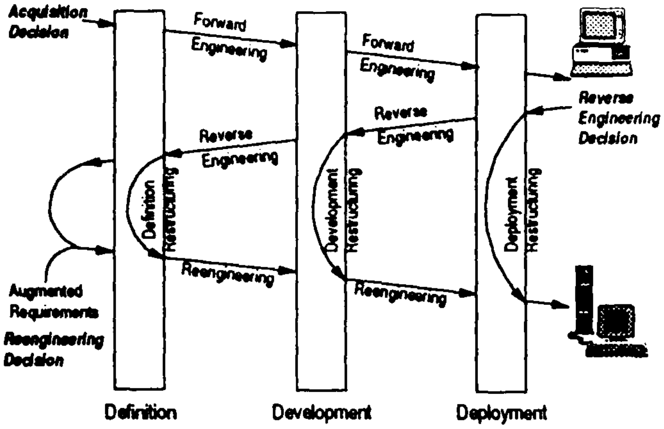


Fig. 11 Expanded Notion of Product Reengineering

For the most part, this is also the perspective taken on reengineering in a recent definitive reprint book on software reengineering [38], especially in the lead article by the editor of this work [39] that takes an inherently transformational view of product reengineering. This reprint book is much concerned with the major ingredients needed for software product reengineering as summarized in Figure 12.

Major Product Reengineering Facets
Reengineering Process
Reengineering Cost Effectiveness
Reengineering Risks
Reengineering to Reduce Software Maintenance
Technology and Tools for Reengineering
Data Reengineering
Source Code Analysis for Reengineering
Software Restructuring and Translation
Reverse Engineering and Design Recovery
Reengineering for Reuse
Reengineering to Object Oriented Architectures
Reengineering through Knowledge Based Program Analysis and Understanding

Fig. 12 Major Product Reengineering Issues

Arnold indicates that there are three basic classes of transformational views.

1. *Nonprocedural views* are meta-level views, such as decision tables, event trees, attribute trees, data schemas, user requirements, and system specifications. There do not represent views of the actual entity but a view of a view of the entity or, in other words, salient characteristics of the entity. A non procedural view is a purposeful view.
2. *Procedural views* contain direct information about procedures or representations, or information intimately associated with this information. Source code, and the objects and entities of object oriented languages are procedural views. A procedural view is a functional view.
3. **Psuedoprocedural views**, or architecturally oriented views, contain perspectives both of procedural and nonprocedural views. Hierarchy charts, structural models, data flow diagrams, entity-relationship diagrams, and Petri nets are examples of psuedoprocedural views. A psuedoprocedural view could also be called a structural view.

We can also have views that are derived from analysis of one, of in some other way derived from one, of the three basic view categories. Arnold denotes these as analysis views. For the most part, purposeful views or nonprocedural views are associated with the definitional phases of the lifecycle for product acquisition. They concern user requirements and technological specifications. Functional or procedural views tend to be associated with the very end of the development phase of the lifecycle and the deployment phase when systems may be thought of in terms of their input output characteristics. Psuedoprocedural, or architectural or structural, views tend to be associated with the earlier phases of system development. One of the major purposes of both forward and reverse engineering, and tools that support these, is to enable transformation from one view to another such as to ultimately obtain a functionally useful product.

Arnold [39] indicates several potential uses and characteristics of product reengineering. These, which are neither mutually exclusive nor collectively exhaustive, include the following.

1. Reengineering may help reduce an organization's risk of product evolution through what effectively amounts to reuse of proven subproducts.
2. Reengineering may help and organization recoup its product development expenses through constructing new products that are based on existing products.
3. Reengineering may make products easier to modify for purposes of accommodating evolving customer needs.
4. Reengineering may be a catalyst for automating product maintenance.
5. Reengineering may be a catalyst for application of new technologies, such as CASE tools and artificial intelligence, to system acquisition.
6. Reengineering is big business, especially considering the major investment in legacy systems that need to be updated and maintained.

In short, reengineering provides a mechanism that enables us to understand systems better, such that we are capable of extending this knowledge to new and better systems. Thus, it enhances both understanding and improvement abilities. Reengineering is accompanied with a variety of risks that are associated with processes, people, tools, strategies, and the application area for reengineering. These risks can be managed using the methodologies and the metrics discussed [2] and [14].

A number of authors have suggested specific lifecycles that will lead to determination of a decision to, or not to, reengineer a product and, in support of a positive decision, enable a product reengineering lifecycle [40] [41]. There are a number of needed accomplishments. These include the following.

1. Initially, there exists a need for formulation, assessment, and implementation of definitional issues associated with the technical and organizational environment. These issues include organizational needs relative to the area under consideration, and the extent to which technology and the product or system under reengineering consideration supports these organizational needs.
2. Identification and evaluation of options for continued development and maintenance of the product(s) under consideration is a need, including an option for outsourcing this activity.
3. Formulation and evaluation of options for composition of the reengineering team, including insourcing and outsourcing possibilities is a need.
4. Identification and selection of a program of systematic measurements that will enable demonstration of cost efficiency of the identified reengineering options and selection of a chosen set of options is required.
5. The existing legacy systems in the organizations need to be examined in order to determine the extent to which these existing systems are functionally useless at present and in need of total replacement, functionally useful but with functional and nonfunctional defects that could potentially be remedied using product reengineering to create renovated systems, or systems that are fully appropriate for the current and intended future uses.
6. A suite of tools and methods to enable reengineering needs to be established. Method and tool analysis and integration is a need in order to provide for multiple perspective views across the various abstraction levels (procedural, psuedoprocedural, and nonprocedural) that will be encountered in reengineering is needed.
7. A reengineering process for product reengineering needs to be created on the basis of the results of these earlier steps that will provide for the reengineering of complete products, or reengineering of systems, and for incremental reengineering efforts that are phased in over time.
8. There must be major provisions for education and training such that it becomes possible to implement whatever reengineering process eventuates through education and training.

This is more of a checklist of needed accomplishments for a reengineering process than it is a specification of a lifecycle for the process itself. Through pursual of this checklist, we should be able to establish an appropriate process for product reengineering in the form of Figures 4 or 11.

There are several needs that must be considered if a product reengineering process is to yield appropriate and useful results.

1. There is a need to consider organizational and technological issues to develop useful product reengineering strategy.
2. There is a need to consider human, leadership, and cultural issues, and how these will be impacted by the development and deployment of a reengineered product as a part of the definition of the specifications for the reengineered product.
3. It must be possible to demonstrate that the reengineering process and product are, or will be, each cost effective and of high quality, and that they support continued evolution of future capabilities.
4. Reengineered products must be considered within a larger framework that also considers the potential need for reengineering at the levels of systems management and organizational processes as it will generally be a mistake to assume that technological fixes only will resolve organizational difficulties at these levels.
5. Product reengineering for improved post deployment maintainability must consider maintainability at the level of process rather than at the level of product only, such as would result in the case of software through rewriting source code statements. Use of model based management systems or code generators should yield much greater productivity, in this connection, than rewriting code at the level of source code.
6. Product reengineering must consider the need for reintegration of the reengineered product in with existing legacy systems that have not been reengineered.
7. Product reengineering should be such that increased conformance to standards is a result of the reengineering process.
8. Product reengineering must consider legal issues associated with reverse engineering.

The importance of most of these issues is relatively self evident. Issues surrounding legality are in a state of flux in much the same way as for benchmarking.

It is clearly legal for an organization to reverse engineer a product that it owns. Also, there exists little debate at this time on whether inferring purpose from the analysis of existing functionality of a product and without any attempt to examine the architectural structure or detailed components that comprise the existing product, and then recapturing the functionality in terms of a new development effort (the so called black box approach), is legal. Doubtlessly, it is legal. Major questions surround the legality of "white box" reverse engineering in which the detailed architectural structure and components of a system, including code for software, are examined in order to reverse engineer and reengineer it. The major difficulty appears to surround the fair use provisions in copyright law, and the fact that it is the use of trade secrets for illicit gain. Copyrighted material cannot be trade secret since the copyright law requires disclosure of the material copyrighted. In particular, software is copyright and not patented. So, trade-secret restrictions do not apply. There is a pragmatist group that says even black box reengineering is legal and a constructionist group that says it is illegal [42] [43]. These issues will be the subject of much debate over the near term.

Arnold [44] has identified many of these product reengineering needs in the form of risks that must be managed during the reengineering effort. Thus, we see that there are risks associated with a variety of factors for product reengineering, as suggested in Figure 12.

- ***Integration risk*** is the risk associated with having a reengineered product that cannot be satisfactorily integrate with, or interface to, existing legacy systems.
- ***Maintenance improvement risk*** is the risk that the reengineered product will exacerbate, rather than ameliorate, maintenance difficulties.
- ***Systems management risk*** is the risk that the reengineered product attempts to impose a technological fix on a situation where the major difficulties are not need for greater support, but for organizational reengineering at the level of systems management.
- ***Process risk*** is that associated with having a reengineered product that might well represent an improvement in a situation where the specific organizational process in which the reengineered process is to be used is defective and in need of reengineering.
- ***Cost risk*** is associated with having major cost overruns in order to obtain a deployed reengineered product that meets specifications.
- ***Schedule risk*** is associated with having schedule delays in order to obtain a deployed reengineered product that meets specifications.
- ***Human acceptance risk*** is the risk associated with obtaining a reengineered product that is not suitable for human interaction, or one that is unacceptable to the user organization for other reasons.
- ***Application supportability risk*** is that risk associated with having a reengineered product that does not really support the application or purpose it was intended to support.
- ***Tool and method availability risk*** is associated with proceeding with reengineering a product based upon promises for a method or tool, needed to complete the effort, which doe not become available or which is faulty.
- ***Leadership, strategy, and culture risk*** is that associated with imposing a technological fix in the form of a reengineered product, in an organizational environment that cannot adapt to the reengineered product.

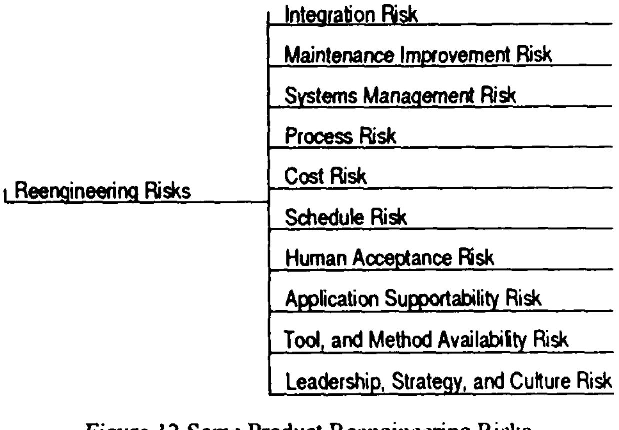

Reengineering Risks
Integration Risk
Maintenance Improvement Risk
Systems Management Risk
Process Risk
Cost Risk
Schedule Risk
Human Acceptance Risk
Application Supportability Risk
Tool, and Method Availability Risk
Leadership, Strategy, and Culture Risk

Figure 12 Some Product Reengineering Risks

Clearly, these risks are not mutually exclusive, the risk attributes are not independent, and the listing is incomplete. For example, we could surely include legal risks. We can use this as the basis for a multi attribute type assessment or for a decision support system [45] that supports risk management for product reusability.

G. Other Approaches and Considerations

There are a number of related approaches. Many, as noted, are discussed in [2]. Two of these are very worthy of further comment here: sourcing and integration.

The principles supporting strategic sourcing decisions are conceptually simple and nicely stated by Venkatesan [46].

1. The organization should focus on those components and subsystems that are crucial to the product itself and where the organization has critical core capabilities [47] [48] that support the efforts required and which the organization desires to sustain. This enables an organization to exercise judgment concerning subsystems that are strategic and those that are nonstrategic. It potentially eliminates difficulties that result from conflicting priorities and sourcing decisions.
2. Components and subsystems should be outsourced where there exist potential suppliers with a distinct competitive advantage at producing these. These competitive advantages could be either those of lower cost producers or higher subsystem differentiation.
3. Outsourcing should always be used in such a manner that it supports continuing employee commitment and empowerment. It is necessary to outsource this in such a manner that there is minimum opportunity for exploitation and hollowing of the organization, including its people, by the external supplier.

These principles lead to a process for strategic sourcing. Lacity and Hirschheim have published two recent works on information systems outsourcing [49] [50] in which they identify three generic types of outsourcing.

- *Body shop outsourcing* is a way to meet short term demands that cannot be met by people internal to the organization even though the decision would otherwise be favorable to insourcing.
- *Project management outsourcing* involves the use of external suppliers to furnish a subsystem or service activity, such as training. This would seem closely equivalent to product line or subsystem outsourcing.
- *Total outsourcing* exists whenever an external supplier is responsible for all, or a very major portion of a complete turn-key like information system function.

The bottom line summary message of these authors is that one really cannot outsource the management of information systems. There seems to be much agreement with this, especially as concerns information support for the highest level managerial decisions [51] [52]. Particularly when there is considerable outsourcing to external suppliers, there will be a major mandate for very careful integration of all aspects of the supply chain and an understanding of the relationships between the supply chain and the value chain.

There are also a number of interesting discussions that relate to integration at the level of methods and tools [53] [54] [55]. While of interest primarily for product reengineering, there is much relevance to process and systems management reengineering as well.

IV. Summary

In this paper, we have considered a number of issues relative to systems reengineering. We indicated that reengineering can take place at either, or all, of the levels of

- product,
- process, or
- systems management.

Reengineering at any of these levels is related to reengineering at the other two levels. Reengineering can be viewed from the perspective of the organization fielding a product as well as from the perspective of the customer, individual or organizational receiving the product. From the perspective of either of these, it may well turn out to be the case that reengineering at the level of product only may not be fully meaningful if this is not also associated, and generally driven by, reengineering at the levels of process and systems management. For an organization to reengineer a product when it is in need of reengineering at the levels of systems management and/or process is almost a guarantee of a reengineered product that will not be fully trustworthy and cost efficient. An organization that contracts for product reengineering when it is in need of reengineering at the levels of systems management and/or process is asking for a technological fix and a symptomatic cure for difficulties that are institutionally and value related. Such solutions are not really solutions at all.

Figure 13 is a hypothetical representation of potential need for reengineering at the levels of product, process, and systems management. While it may well be the case, as suggested in the figure, that product reengineering may well occupy much resources, the combined total of resources needed for systems management and process reengineering may be not insubstantial. What the figure does not show is the fact that resources expended upon product reengineering only, and with no investigation of needs at the systems management and process levels, may well not be wise expenditures - from either the perspective of the organization producing the product or the one consuming it.

In an insightful study [56], it is indicated that organizations often squander resources that look very promising but which fail to produce long-lasting results of value for the organization. Four major ways to fail are identified.

1. Assigning average performers to the reengineering effort, often because the more valuable people are needed for other more important efforts, will guarantee mediocre performance of the product of the reengineering effort.
2. Measuring the reengineering plan and activities only, and not the results, will often product deceptive measurement results.
3. Allowing new and innovative ideas for reengineering to be squelched through opportunistic politics and extreme risk aversion will preserve the status quo rather than encourage implementation of beneficial activities in the form or results.
4. Failing to communicate wisely and widely during implementation will almost always frustrate success.

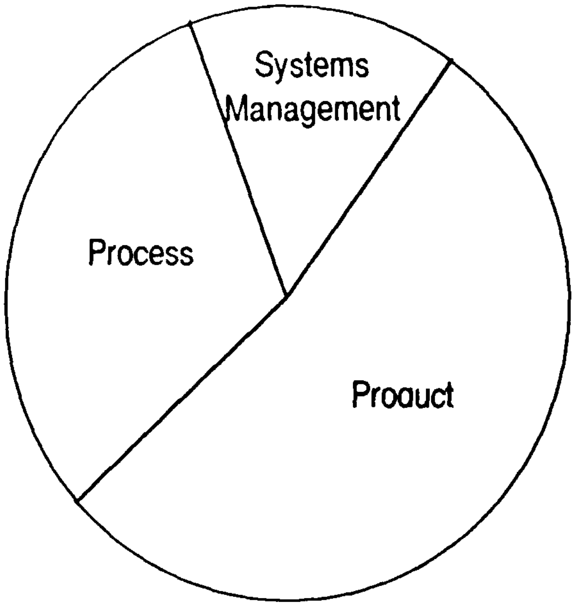


Fig 13 Resource Distribution to Implement Systems Reengineering

To this list, we might add failure to obtain real commitment from the highest levels of the organization for the reengineering effort. It might be argues, of course, that this leads to such things as assignment of average and mediocre performers to the reengineering effort. These authors also offer five factors said to enhance success at reengineering.

1. Set aggressive reengineering performance targets in terms of results.
2. Commit a significant portion of the CEOs time to the reengineering effort, especially during deployment.
3. Assign a very senior executive to head the reengineering effort, especially during deployment.
4. Perform a comprehensive review and analysis of customer needs, organizational realities, strategic economic issues, and market trends as a prelude to reengineering.
5. Conduct a pilot study and prototype the reengineering effort in order to obtain results useful both to refine the reengineering process and the enhance communications and build enthusiasm.

while the study was based primarily on organizational reengineering efforts, there are clear implications in these suggestions for all three types of reengineering efforts

- systems management,
- process, and
- product.

And, as we have indicated, there is every reason why due consideration needs to be given to all three of these efforts in an integrated fashion for the betterment of each effort.

V. REFERENCES

[1] Sage, A. P., "Systems Engineering and Information Technology: Catalysts for Total Quality in Industry and Education," *IEEE Transactions on Systems, Man, and Cybernetics*, Vol. 22, No. 5, September 1992, pp. 833-864.

[2] Sage, A. P., *Systems Management for Information Technology and Software Engineering*, John Wiley and Sons, 1994 (in press).

[3] Sage, A. P., and Palmer, J. D., *Software Systems Engineering*, John Wiley and Sons, New York, 1990.

[4] Rekoff, Jr., M. G., "On Reverse Engineering," *IEEE Transactions on Systems, Man, and Cybernetics*, Vol. SMC 15, No. 2, March 1985, pp. 244-252.

[5] "Software Engineering Glossary," *IEEE Software Engineering Standards*, IEEE Press, New York, 1991.

[6] *IEEE Standard for Software Maintenance*, P1219/D14, IEEE Standards Department, New York, 1992.

[7] Davenport, T. H. and Short, J. E., "The New Industrial Engineering: Information Technology and Business Process Redesign," *Sloan Management Review*, Vol. 31, No. 4, Summer 1990, pp. 11-27.

[8] Davenport, T. H., *Process Innovation: Reengineering Work through Information Technology*, Harvard Business School Press, Boston MA, 1993.

[9] Hudak, G. J., "Reengineering the Systems Engineering Process," *Proceedings of the National Council on Systems Engineering Annual Meeting*, Alexandria VA, August 1993, pp. 105-112.

[10]Brackett, J. W., and Pyster, A. B., *"High-Level Software Synthesis," Proceedings of the National Council on Systems Engineering Annual Meeting*, Alexandria VA, August 1993. pp. 207-214.

[11]Hammer, M., Reengineering Work: Dont Automate, Obliterate," *Harvard Business Review*, Vol. 68, No. 4, July 1990, pp. 104-112.

[12]Hammer, M., and Champy, J., *Reengineering the Corporation: A Manifesto for Business Revolution*, Harper Business, New York, 1993.

[13]Harrington, H. J., *Business Process Improvement: The Breakthrough Strategy for Total Quality, Productivity, and Competitiveness*, McGraw Hill Cook Co., New York, 1991.

[14]Sage, A. P., *Systems Engineering*, John Wiley and Sons, New York, 1992.

[15]Quinn, J. B., Paquette, P. C., and Doorley, T., "Technology in Services: Rethinking Strategic Focus," *Sloan Management Review*, Winter 1990.

[16]Quinn, J. B., Paquette, P. C., and Doorley, T., "Technology in Services: Creating Organizational Revolutions," *Sloan Management Review*, Winter 1990.

[17]Quinn, J. B., Paquette, P. C., and Doorley, T., "Beyond Products: Service Based Strategies," *Harvard Business Review*, 68, No. 3, March 1990.

[18]Quinn, J. B., *Intelligent Enterprise: A Knowledge and Service Based Paradigm for Industry*, Free Press, New York, 1992.

[19]Davenport, T. H., Eccles, R. G., and Prusak, L., "Information Politics," *Sloan Management Review*, Vol. 34, No. 1, Fall 1992, pp. 53-65.

[20]Rockart, J. F., and DeLong, D. W., *Executive Support Systems: The Emergence of Top Management Computer Use*, Dow Jones-Irwin, Homewood IL, 1988.

[21]Raiffa, H., *The Art and Science of Negotiation*, Belknap, Cambridge MA, 1982.

[22]Neale, M. A., and Bazerman, M. H., *Cognition and Rationality in Negotiation*, Free Press, New York, 1991.

[23]Strebel, P., *Breakpoints: How Managers Exploit Radical Business Change*, Harvard Business School Press, Boston MA, 1992.

[24]Camp, R. C., *Benchmarking: The Search for Industry Best Practices That Lead to Superior Performance*, Quality Press, American Society for Quality Control, Milwaukee WI, 1989.

[25]Liebfried, K. H. J., and McNair, C. J., *Benchmarking: A Tool for Continuous Improvement*, HarperCollins Publishers, New York, 1992.

[26]Watson, G. H., *The Benchmarking Workbook: Adapting Best Practices for Performance Improvement*, Productivity Press, Cambridge MA, 1992.

[27]Watson, G. H., *Strategic Benchmarking: How to Rate Your Company's Performance Against the World's Best*, John Wiley and Sons, New York, 1993.

[28]Rockart, J. F., "Chief Executives Define Their Own Data Needs," *Harvard Business Review*, Vol. 57, No. 2, 1979, pp. 81-93.

[29]Rockart, J. F., and Bullen, C. V. (Eds.), *The Rise of Managerial Computing*, Dow Jones Irwin, Homewood Il, 1986.

[30]Watson, G. H., Bookhart, S., et. al., "Applying Moral and Legal Considerations to Benchmarking Protocols," Appendix 2 in *Planning, Organizing, and Managing Benchmarking: A User's Guide*, American Productivity and Quality Center, Houston TX, 1992.

[31]Nevins, J. L., and Whitney, D. E. (Eds.), *Concurrent Design of Products and Processes: A Strategy for the Next Generation in Manufacturing*, McGraw Hill Book Co., New York, 1989.

[32]Shina, S. G., *Concurrent Engineering and Design for Manufacture of Electronics Products*, Van Nostrand Reinhold, New York, 1991.

[33]Carter, D. E., and Baker, B. S., *Concurrent Engineering: The Product Development Environment for the 1990s*, Addison Wesley, Reading MA, 1992.

[34]Hunt, V. D., *Reengineering: Leveraging the Power of Integrated Product Development*, Oliver Wright Publications, Essex Junction VT, 1993.

[35]*Air Force Material Command Guide on Integrated Product Development*, May 25, 1993.

[36]Chikofsky, E., and Cross, J. H., "Reverse Engineering and Design Recovery," A Taxonomy, *IEEE Software*, Vol. 7, No. 1, January 1990, 99. pp. 13-17.

[37]Cross, J. H. II, Chikofsky, E. J., and May, C. H. Jr., "Reverse Engineering," in Yovitz, M. C. (Ed.), *Advances in Computers*, Vol. 35, Academic Press, San Diego CA, 1992, pp. 199-254.

[38]Arnold, R. S. (Ed.), *Software Reengineering*, IEEE Computer Society Press, Los Altos CA, 1993.

[39]Arnold, R. S., "A Road Map Guide to Software Reengineering Technology, in Arnold, R. S. (Ed.), *Software Reengineering*, IEEE Computer Society Press, Los Altos CA, 1993, pp. 3-22.

[40]Ulrich, W. M., "Re-engineering: Defining an Integrated Migration Framework," in Arnold, R. S. (Ed.), *Software Reengineering*, IEEE Computer Society Press, Los Altos CA, 1993, pp. 108-118.

[41]Olsem, M. R., "Preparing to Reengineer," *IEEE Computer Society Reverse Engineering Newsletter,* December 1993, pp. 1-3.

[42]Samuelson, P., "Reverse Engineering Someone Else's Software: Is it Legal?," *IEEE Software*, Vol. 7, No. 1, January 1990, pp. 90-96.

[43]Sibor, V., "Interpreting Reverse Engineering Law," *IEEE Software*, Vol. 7, No. 4, July 1990, pp. 4-10.

[44]Arnold, R. S., "Common Risks of Reengineering," *IEEE Computer Society Reverse Engineering Newsletter,* April 1992, pp. 1-2. Also in [38], pp. 119-120.

[45]Sage, A. P., *Decision Support Systems Engineering*, John Wiley and Sons, New York, 1991.

[46]Venkatesan, R., "Strategic Sourcing: To Make or Not to Make," *Harvard Business Review*, Vol. 70, No. 6, November 1992, pp. 98-107.

[47]Prahalad, C. K., and Hamel, G., "The Core Competence of the Corporation," *Harvard Business Review*, Vol. 68, No. 3, May 1990, pp. 60-74.

[48]Stalk, G., Evans, P., and Shulman, L. E., "Competing on Capabilities: The New Rules of Corporate Strategy," *Harvard Business Review*, Vol. 70, No. 2, March 1992, pp. 57- 63.

[49]Lacity, M. C., and Hirschheim, R., "The Information Systems Outsourcing Bandwagon," *Sloan Management Review*, Vol. 35, No. 1, Fall 1993, pp. 73-86

[50]Lacity, M. C., and Hirschheim, R., *Information Systems Outsourcing: Myths, Metaphors, and Realities,* John Wiley and Sons, Chichester UK, 1993.

[51]Benjamin, R. J., and Blunt, J., "Critical IT Issues: The Next Ten Years, *Sloan Management Review*, Vol. 33, No. 4, Summer 1992, pp. 7-19.

[52]Boynton, A. C., Jacobs, G. C., and Zmud, R. W., Whose Responsibility is IT Management," *Sloan Management Review*, Vol. 33, No. 4, Summer 1992, pp. 32-38.

[53]Kronlof, K (Ed.), *Method Integration: Concepts and Case Studies*, John Wiley and Sons, Chichester UK, 1993.

[54]Schefsttom, D., and van den Broek, G., *Tool Integtration: Environments and Frameworks,* John Wiley and Sons, Chichester UK, 1993.

[55]Andrews, D. C., and Leventhal, N. S., *FUSION - Integrating IE, CASE, and JAD: A Handbook for Reorganizing the Systems Organization*, Prentice Hall, Englewood Cliffs NJ, 1993.

[56]Hall, G., Rosenthal, J., and Wade, J., "How to Make Reengineering Really Work," *Harvard Business Review*, Vol. 71, No. 6, November 1993, pp. 119-131.

Design Capture and Optimization Issues for System-Level Reengineering

Steven Howell, NgocDung Hoang, Cuong Nguyen
Naval Surface Warfare Center, Dahlgren Division
Nicholas Karangelen
Trident Systems Inc.

ABSTRACT

Given the increasing maturity of computer-based systems, more and more systems are evolving from past systems ratherthan being developed from scratch. However, large and complex systems have put dramatic burdens on the systems engineers in designing these applications. The situation has become even more complicated by the increased introduction of commercial off the shelf (COTS) products into these systems. Additionally, many reengineering efforts have only focused on software reengineering,providing little insight into the other system functionality not embodied in the software. Today reengineeringtechnology must be integrated into a system-level forward engineeringframework to effectively meet the challenges of the future systems. Four critical issues that reengineeringtechnology must address in order to effectively facilitate the system-level design of evolution systems include: (1) the ability to provide means to recapture not only functional and control (behavioral) descriptions of the system, but all necessary information at the system level; (2) the ability to merge reengineered information with forward engineeringinformation at the system level; (3) the ability to separate "required" functionality from "legacy" functionality which was the result of design and architecture decisions; (4) and the ability to maintain and reengineer system level designs. Without these capabilities, future systems engineer will face two critical shortcomings: (1) an exceedingly restricted design space during system-level optimization which willproduce ineffective systems, and (2) fragile implementations which are difficult to modify in response to either requirements changes or technology advancements. This paper describes a framework to support both reengineeringand forward engineeringinformation and provides a means to manage the information. The framework provides for separation of concerns, systematic capture of non-functional attributes, and integration of system information. This framework can also be used to assess the ability of current reengineeringtechnology to meet the needs of system level design.

INTRODUCTION

For computer-based systems, optimizing a reengineered static or semi-static software architecture to fit a static or semi-static distributed computer hardware implementation late in the application's implementation provides marginal performance enhancements. Vastly increased performance can be gained through proper system engineering. The computer-based system engineering of large and distributed applications involves identifying the proper definition and structure of the system functionality and allocating these functions to proper resource architectures. Engineers must be able to effectively trade-off between different types of resources (i.e., hardware, software, and humanware) as well as alternative resource architectures, to meet requirements for real-time, cost, safety, etc.

A key issue in designing a real-time, large-sized, complex system is to optimize and assess the design, based on multiple competing requirements, early and throughout the design process. However, "optimal" solutions cannot be generated by addressing various aspects of the system singularly. Non-functional issues, including financial costs (development, production, maintenance, logistics), physical constraints, timing, security, dependability (reliability, safety), maintainability, etc., will drive the system design as much as functionality for most large, complex systems.

Many times, effective trade-offs cannot be performed due to the lack of design descriptions. Complete information about an application cannot be specified in a single design model even for moderately sized systems. At different stages of the design process, the designer must decide which information is important to capture and at what level of detail. Existing reengineering technologies only focusses on certain types of design information (i.e., the design of software algorithms can be easily recapture). Reengineering at other levels in the design process is usually a labor-intensive and manual process. Existing design reflects a specific implementation (i.e., a particular hardware platform and/or a unique software language) that is selected to performed the desired functions. In capturing an existing system design, system engineers do not have the capability to distinguish which part of the design is driven by the original requirements and which part is dependent on a particular design decision. Also, reverse-engineering from code tends to lack non-functional information.

For most computer-based evolutionary systems the amount of the former system reengineered or reused may vary tremendously. For instance, the reengineering may facilitate a system upgrade where only a small percent (on the order of 5-30%) of the operational code is modified. In this case the system architecture and infrastructure is not likely to change significantly. Therefore, the reengineering efforts must integrate into the existing system/software design.

On the other hand, if the evolution involves the next generation of a system, some of the original system level design might be reused; however, the system architecture and infrastructure would likely differ vastly from the original system. This appears to be the case for the Next Attack Submarine (the follow-on to the AN/BSY-2 Seawolf) and Combat System 2003 (the follow-on to Aegis).

For the development of large and complex systems, requirements derived from user needs are defined. In turn, these requirements are captured and an initial design is produced [Hoa91]. Analysis is executed to assure that the initial design is complete and consistent [BlF90]. This design is optimized iteratively until a feasible or an optimal design is achieved [HNH91], [HNH92]. Collected results are then passed through for rapid prototyping, assessment, evaluation, test and refinement to yield the final design [BoB85], [CYH91], [JeY91], [Kam91], [SvL76]. The design components are then implemented, integrated and tested. This description follows a waterfall approach. Other models used, such as modified waterfall, rapid prototyping or spiral model, follow the same steps in at different levels of details and different orders. In actual system developments, a hybrid of these approaches is typically used.

This paper describes a framework to support both reengineering and forward engineering information and provides a means to manage the information. The framework provides for separation of concerns, systematic capture of non-functional attributes and integration of system information. This framework can also be used to assess the ability of current reengineering technology to meet the needs of system level design. First, a capture framework is described. Next, additional annotation capabilities are described which provide a basis for engineering decisions. Optimization of system designs using the annotation and capture methods in an evolutionary environment are also addressed.

SPECIFICATION AND INTEGRATION ISSUES

Regardless of the process used to develop the system, a lack of a cohesive and coherent engineering discipline starts with a shortage of formal and cohesive techniques for specifying the system under development. This is especially true for system level design capture and design analysis. Most large systems are reactive systems that must respond to external stimuli; therefore, the systems engineering must be able to predict the system's behavior in all scenarios and environments under which the system will operate including planned mode of operation and action, and reaction to failure to internal system failure (fault tolerance) or external inflicted damage (damage tolerance). In order to effectively design these systems, definitions of the multiple design domains or views which address the principal system design perspectives are needed. We propose five Design Capture Views (DCV) [Hoa91] which as follows: (1) Environmental, (2) Informational, (3) Functional, (4) Behavioral, and (5) Implementation. The capture approach for each design domain or capture view share a common hierarchial structure which supports management of the magnitude and complexity associated with a large system design. Flat representations of complex system designs rapidly become unwieldy as the design detail unfolds. A hierarchial structure allows the system capture views to be represented at various levels of detail from a broad top level, which encompasses the breadth of the system and its external interfaces, to very low levels, which describe the details of a particular segment of the system design.

The five system capture views partition the system design into logical segments which correspond to key perspectives of the system design. These five capture views are distinct but related representations of key aspects of the system. They are summarized in Table 1-1. The design element portion of the table describes current generalized methods or techniques used to specify the information. Specific methods exist for capturing the Informational, Functional and Behavioral capture views [DeM79], [HaP87], [WaM85], [ShM88] [Har86]. Implementation and Environmental capture views lack mature methods for computer intensive real-time systems. The following paragraphs describe each capture view objective in more detail.

The *Informational Capture View* captures a conceptual representation of the system under design in abstract terms. This view captures all components that make up the system and the interaction between these components. This allows for a description of the intended system concept of operations (use analysis) without implying or constraining the physical implementation of the system under design.

The *Functional Capture View* establishes the functional structure of the system. It specifies how the functions are decomposed and how the information is transformed through these functions. This provides a better understanding of the system's functional decomposition.

The *Behavioral Capture View* describes the system's attributes over time and describes the event or time-driven aspects of the system. This view allows for specification of the system's behavior at different times and under various conditions and situations. This provides a mechanism for specific real-time and time-critical aspects of the system. This also may include behavioral descriptions of dependability, safety, and security.

The *Implementation Capture View* defines the physical hardware, software and human resources which comprise the system and its connectivity to external systems [HoK92]. This is where alternative physical system architectures are defined.

The *Environmental Capture View* captures the system under design from an external viewpoint [Kar92]. This view describes the situation(s), environment(s), expected events and other factors that make up the conditions under which the system is operating. This

TABLE 1. SYSTEM CAPTURE VIEWS

DESIGN CAPTURE VIEW	VIEW OBJECTIVES	DESIGN METHODS
Informational Capture View	▪ Characterizes system concept of operations ▪ Represents system components in abstract terms	▪ Entity-relationship diagrams ▪ Attribute/method descriptions
Functional Capture View	▪ Defines system functions and decompositions ▪ Specifies data flow requirements	▪ Function/data flow diagrams ▪ Process specifications ▪ Data dictionary
Behavioral Capture View	▪ Defines system states and trigger events ▪ Specifies system behavior characteristics	▪ Control flow diagrams ▪ State transition diagrams ▪ Control specifications
Implementation Capture View	▪ Defines the physical hardware, software, and human resources which make up the system ▪ Specifies system physical interconnectivity	▪ Hardware, software, and human resource descriptions ▪ Performance parameters and resource characteristics ▪ Function-resource mapping
Environmental Capture View	▪ Establishes conditions and events constraining system operations ▪ Specifies performance Measure Of Effectiveness (MOEs) and conditions of measurement	▪ Environmental conditions and event descriptions ▪ External system descriptions ▪ System initial conditions ▪ MOEs

includes the following: initial state of the system under design; environmental conditions including acoustic, electromagnetic, and meteorological conditions; threat types and locations (in a military application); operational constraints; likely strategic and tactical considerations and other pertinent items; and concept of operations. The MOEs which characterize system performance and establish the "success criteria" for the system are also specified together with the conditions under which the MOEs are measured. The Environmental Capture View also specifies the guidance and constraint of the environment which the design itself must face.

An attempt to address all of the issues associated with these capture views simultaneously or without a structured methodology is a multi-dimensional problem of a magnitude which exceeds the capacity of most, if not all, systems engineers. Each of these capture views provides key information concerning particular aspects of the system under design. Taken individually the capture views allow the systems engineer to partition the design of a proposed or existing system into manageable parts.

As systems increase in complexity and size, the capturing design methods have to scale so that they are able to capture different aspects of the system in a complete and systematic manner. Deciding what aspect of the system needs to be captured and to what level of detail is a very difficult task. The group of descriptions used to capture one type of system may be irrelevant or incomplete for another system. Therefore, the ideal situation is to define the capturing method for all aspects of the system and, depending on the system requirements or on the design phases, emphasize or deemphasize certain system capture views.

SYSTEM DESIGN ANNOTATION

One key to designing a real-time, large, complex system is to optimize the design to meet the requirements and desired MOEs. In order to achieve this, the system engineer/analyst must have the capability to annotate the system design with design goals/criteria that relate to the requirements and MOEs. Whether the system design emphasizes real-time, reliability, safety, security, cost, physical constraints, or any specific criteria, a set of design goals is required to describe the desired characteristics of the system. This set of design goals provides a framework for the specification of critical information from which system's qualities and performances can be measured. The design goals also provide a basis for the trade-off between design criteria and design alternatives. One mechanism that allows these design goals to be specified is *System Design Factor* (SDF) [NHL93]. SDFs are "attributes" associated with any design element in the design. A *Design Element (DE)* is a set of one or more design components such as functions, dataflows, states, objects, or relationships. Regardless of the employment of any system development process, the SDFs can have major influence in various design activities within the process (i.e., design capture, design structure, design allocation, and design trade-off). The SDFs not only provide a mechanism to capture the system attributes but also allow them to be related.

SDFs can also be viewed as a communication path between customers and systems engineers. In general, system engineers must be able to express and prioritize the customers' criteria. These criteria are, in turn, annotated through the design factors and are used as a guideline for the design team. By considering these factors early and throughout the development process, the design team can avoid both bad designs and design that does not meet the requirement to reduce costs, and to optimize productivity [HHN90a], [HHN90b].

Each SDF has one or more metrics defined. In addition, when associated with a design element, an SDF may have more than one measurement. The metric describes manners in which the factor might be measured. Metrics can be derived from (1) past experience, (2) results of simulation/analysis/prototyping, (3) actual system measurements, (4) other SDF measurements, or (5) other SDF metrics. By using other SDF metrics or measurements to define a SDF metric, SDFs are defined in a hierarchical manner. For example, a higher level Real-Time Performance (RTP) SDF can be derived from two other SDFs: Deadline Success Rate (DSR), and Deadline Criticality (DC). The metric for RTP might be a mathematical formula related DSR and DC; e.g., the product. Metrics may also be related to the hierarchical specification. For example, in a functional decomposition, the Total Real-Time Performance of some Design Element, TRTP(de) may be the product of decomposition components of de as shown in EQ 1.

EQ 1: $TRTP(de) = \Pi RTP(DDE)$
such that DDE is the set of Design Elements which are a decomposition of Design Element de.

It has been previously shown that based on the design goal and design parameter, the engineer can tailor the single criteria or multi-criteria objective function for optimization [NaF91].

SYSTEM LEVEL OPTIMIZATION ISSUES

A template of SDF is presented in Figure 1. This is a super class definition of SDF. The purpose of this template is to provide a general format to guide the systems engineer or the customer in the application of the SDF. It assists the engineer/customer in specifying the goals/criteria to be measured. As a system design matures, SDF class types are defined, a class hierarchy of SDF is generated and instantiated SDF "objects" are used to annotate particular design elements.

Currently, there are twelve items in the template. (1) *Name* is an unambiguous name of the SDF. (2) *Type* is a classification of the SDF. (3) *Range* is either the minimum and maximum values or the cardinality of the SDF. (4) *Units* is the unit of measurement of the SDF. (5) *Methods/Principle* lists approaches or techniques that the designer/customer considers to affect changes in the factor's value. (6) *Rationale* lists the reasons that this factor applies to design elements and specifies which design element types are appropriate. (7) *Relationship* lists other SDFs that are closely associated. The *Relational Expression* field in this item lists types of associations for each Relationship. (8) *Quantification* is divided into Type and Metrics fields. The Type field describes value types of quantifications such as integer, float, double, short, or long. The Metrics field describes how measures are determined. Once instantiated, the Metrics field also holds values measured. A description of this aspect of the SDF is described in more detail in an earlier section. (9) *Consistency Rules* list rules to determine consistency of the SDF in various design specifications. Rule types include By-Aggregation, By-Type, By-Design Factors, By-View, and By-Component rules. For example, the By-Aggregation field provides a slot that holds the rule for governing this factor consistency throughout the hierarchy (e.g., Use Rule X and Rule Y). (10) *Reference* is the source or reference of factor which may be a publication or simply the name of the designer that formulated this SDF. (11) *Definition* provides a text book style definition of the SDF. (12) *Annotation* provides areas for free form comments relevant to the SDF. The Annotation may include further SDF background information or provide warnings related to the SDF.

1.	Name.	*Character Text Description (e.g., Reliability)*	
2.	Type:	*SDF Type (e.g., Probability)*	
3.	Range:	*Range of Values of Quantification (e.g., 0.0 to 1.0)*	
4.	Units:	*Units of Type (e.g., Units of Probability)*	
5.	Methods/Principle:	*One or More of Methods That Might be Used to Support Factor (e.g., Highly Reliable Component, Fault Tolerance)*	
6.	Rationale:	*Character Text Describing Need*	
7.	Relationship:	*One or More of other SDFs Types(e.g., Availability), ...*	
	a. Relational Expression	*One or More of relationship (e.g., Positive Correlation, Negative Correlation)*	
8.	Quantification		
	a. Type	(.e.g., fixed)	
	b. Metrics	*ACTUAL:*	*Metric Type Specification (e.g., R(t) = 1 - F(t))* *Source: One of Source Type (e.g., Calculated)* *Value: Measured Value of Type "Type"*
		REQUIRED:	*Metric Type Specification (e.g., Scalar Value .989) entered* *Source: One of Source Type (e.g. Entered)* *Value: Measured Value of Type "Type"*
		BUDGETED:	Metric: Type Specification (e.g. 1.01 * SDF (Required Value)) Source: One of Source Type (e.g., Calculated) *Value: Measured Value of Type "Type"*
9.	Consistency Rules		
	a. By Aggregation	*Use Rule X and Y;* *Rule X: The probability of the component in series is the product of its probabilities. Rule Y: The probability of the component in parallel use one of rating, voting, scheme.*	
	b. By Type		
	c. By Design Factor		
	d. By View		
	e. By component		
10.	Reference:	*Character Type of Complete Reference*	
11.	Definition:	*Character Type Definition from Reference*	
12.	Annotation:	*Character Type of Comments*	

Figure 1 SYSTEM DESIGN FACTORS TEMPLATE

THE FRAMEWORK AND REENGINEERING

The framework proposed addresses many of the critical issues described in the abstract. It provides the ability to capture and maintain more complete system information. Since the key issue in reengineering is the ability to understand the functionality, the behavior, and the implementation of the existing system, the design capture views framework naturally supports this process. Depending on the level of reengineering, information about the existing system can be specified in the appropriate capture views. Supporting various levels of detailed descriptions as well as multi-levels of design abstraction, the capture views provide a framework for the whole spectrum of reengineering. For example, for the reuse of existing software, the functionality of the existing code can be captured in the software architecture of the implementation view. This software architecture provides a better understanding for the software function of the existing code which can then be assessed for its use in new required applications. For the reengineering at system or subsystem level, the functionality, the behavior, and the implementation of the existing subsystem (or system) can be captured by the functional, behavioral and implementation capture views.

With the support of the SDFs in describing various DEs of the capture views, the design of an existing systems can be evaluated for reuse and integrated into new and evolutionary designs. Existing designs can be evaluated independently or as part of a new design. The capability to integrate old and new designs and analyze them based on different criteria allows the employment of reengineering to be properly evaluated and justified.

With these framework advantages, the design space considered by system designers could be broadened. In addition, systems will be more easily evolve from the design, with modification alternatives quickly determined and assessed. This should be true whether the changes occur due to a top-level requirements change or an implementation-level technology insertion.

SUMMARY OF CURRENT STATUS

Definitions and examples of usage of the multi-domain system design capture methodology have been documented in several reports [HKH91], [Hoa91], [HoK92]. Much work still needs to be done in the development of specific representations within each of the capture views especially within the Behavior, Implementation and Environmental Capture Views. Currently, the study of the relationship and transition between these capture views is only focused on the link between the Functional/Behavioral and Implementation Views [Hoa91]. Near-term goals also include the study of the transition between design capture and design evaluation. Transformation techniques must be able to preserve all information from design capture such that analysis results will reflect a correct system.

Automation support for the generation of the capture views, transition between them, and transition from design capture to design evaluation is a critical issue for the success of the employment of this method. The evaluation of current CASE tools (i.e., Teamwork, Software Through Pictures, Statemate, RDD-100) and simulation tools (i.e., ADAS, SES Workbench, Bones) shows that they do not provide complete representation and evaluation capabilities. One immediate solution is the integration of various tools. However, more research is needed to provide a seamless support environment for system design.

The relationship between SDFs is not well understood at the present time, but there are attempts to correlate these factors as this effort progresses. SDFs for security and dependability aspects of the system are being developed by related efforts and the use of SDFs within a multi-view specification of the system is ongoing. These factors are intended to be used throughout and are critical to the entire system engineering process. For instance, they are used to specify in the requirements phase, encapsulate in the capturing phase, quantify and evaluate in the analysis phase, characterize in the optimization phase and, justify in the design trade-off phase.

A software tool called DESTINATION (Design Structuring and Allocation Optimization) has been developed to explore issues described in this paper. DESTINATION has been linked to commercial specification tools and is being linked to simulation and scheduleability assessment tools. In the future, DESTINATION will be linked to reengineering tools. DESTINATION provides SDF annotation and resource allocation optimization. Design guidance capabilities are in development.

The future plans include refining, restructuring and streamlining (if necessary) the optimization of designs using Multi-View Capture and SDFs. A dedicated research effort is considering a small but widely used set of design factors. Given formalisms are currently lacking and single point solutions are being used for the Environmental and Implementation Capture Views. In addition, research into providing robust capture methods for these areas will be explored. The formulation will be incorporated into a sonar [Hoa91] and other applicable examples.

Together, the multi-view capture and optimization based on SDFs provide a framework for the results of real-time system simulation/analysis to view and manage a total system engineering process.

REFERENCES

[BIF90] Blanchard and Fabrycky, Systems Engineering and Analysis, 1990.

[BoB85] Bowen, B., and A. Brown, W. R., System Design: Volume II of System Design for Digital Signal Processing, Prentice-Hall, Inc., 1985

[CYH91] Choi, D., Youngblood, J., and Hwang, P., "Modeling Technology for Dynamic Systems", Proc. 1991 Systems Evaluation and Assessment Technology

Workshop, Aug 1991.

[DeM79] DeMarco, T., Structured Analysis and System Specification, Prentice-Hall, Inc. Yourdon Press, Englewood Cliffs, NJ, 1979.

[HaP87] Hatley, D. and Pirbhai, I., Strategies for Real-Time System Specification, Dorset Publishing, New York, NY, 1987.

[Har86] Harel, D., Statecharts: A Visual Formalism for Complex Systems, The Weizmann Institute of Sci. Tech. Report, Israel, Jul 1986, (also in Science of Programming 8, 1987).

[HHN90a] Howell, S., Hwang, P., and Nguyen, C., "Expert Design Advisor," Proc. 5th Jerusalem Conference on Information Technology (JCIT), IEEE Computer Society Press, Los Alamistos, CA, Oct 1990, pp 743-756.

[HHN90b] Howell, S., Hwang, P., and Nguyen, C., "Expert Design Advisor," Naval Surface Warfare Center Technical Report, TR-90-46, Oct 1990.

[HKH91] Hoang, N., Karangelen, N., and Howell, S., Mission Critical System Development: Design Views and Their Integration, Technical Report, NAVSWC TR 91-586

[HNH92] Howell, S., Nguyen, C., and Hwang, P., "Design Structuring and Allocation Optimization (DeStinAtiOn): A Front-end Methodology for Prototyping Large, Complex, Real-Time Systems," Proc. Hawaii International Conference on System Sciences, IEEE Computer Society Press, Los Alamistos, CA, Jan 1992, Vol. II, pp 517-528.

[HNH91] Howell, S., Nguyen, C., and Hwang, P., "System Design Structuring and Allocation Optimization (DeStinAtiOn), " Proc. 1991 Systems Design Synthesis Technology Workshop, Sep 1991.

[Hoa91] Hoang, N., "Essential Views of Systems Development, " Proc. 1991 Systems Design Synthesis Technology Workshop, Sep 1991.

[HoK92] Hoang, N. and Karangelen N., "A View to an Implementation, " Proc. Complex Systems Engineering Synthesis and Assessment Technology Workshop, Silver Spring, MD, July 1992, pp 223-233.

[JeY91] Jenkins, M. and Yeh, C., "An Approach to Design of Processor Networks Based On Massively Interconnected Models," Proc. 1991 Systems Design Synthesis Technology Workshop, Sep 1991.

[Kam91] Kamat, V., "Computer System Evaluation: Paths and Pitfalls," Proc. 1991 System s Evaluation and Assessment Technology Workshop, Aug 1991.

[Kar92] Karangelen, N., "The Environmental Capture View: Addressing External Factors in Capture and Analysis of Large Scale Complex System Design," Proc. Complex Systems Engineering Synthesis and Assessment Technology Workshop, Silver Spring, MD, July 1992, pp 235-247.

[NaF91] Mansour, N. and Fox, G., "Physical Optimization Methods for Allocating Data to Multicomputer Nodes," Proc. 1991 Systems Design Synthesis Technology Workshop, Sep 1991.

[NHL93] Nguyen, C., Howell, S., Lock, E., and Prasad B., "Employing System Design Factors for Optimization and Trade-Off Analysis in Distributed Real-Time System Software Design Structuring," Proc. of the Workshop on Parallel and Distributed Real-Time Systems, Apr. 1993.

[ShM88] Shlaar, S. and Mellor, S., Object-Oriented Systems Analysis: Modeling the World in Data, Prentice-Hall, Inc. Yourdon Press, Englewood Cliffs, NJ, 1988.

[SvL76] Svobodova and Liba, Computer Performance Measurement and Evaluation Methods: Analysis and Applications, 1976

[WaM85] Ward, P. and Mellor, S., Structured Development of Real-Time Systems, Prentice-Hall, Inc, Yourdan Press, Englewood Cliffs, NJ, 1988.

Information Architecture

An Architectural Basis for Evolution of Large Scale Software Systems

John R. Leary [1]
Software Engineering Institute
(SEI Washington Office)
801 N. Randolph St, Suite 405
Arlington VA, 22203
Tel: 703-908-8206; Email: jrl@sei.cmu.edu

Abstract

The complexity and the volatility of requirements for large scale software systems, and the vast in-place investments, make evolution a necessity. With evolution there is risk that critical user objectives will not be met within schedule and funds budgeted for new capabilities. To assure that mission needs are satisfied, evolutionary process must be strongly influenced by users, but also must be carefully controlled by buyers and efficiently carried out by builders. This paper [2] discusses how architectural approaches add value by providing, from multiple perspectives, a vision of objectives that is understandable to users, buyers, and builders alike. It then describes how these same approaches also offer a means to organize the evolutionary engineering activity around information needs for the target system. The paper illustrates this and shows that use of an "information architecture" helps to assure that results meet ultimate mission needs by focusing engineering activities on the needs of the end user.

1 Introduction

Continuing evolution of large-scale software-intensive systems provides the context for this paper. Even when precedents can be used to foster common understanding of objectives and methods, large-scale systems pose exponentially greater difficulty in communication than is the case in smaller systems. We introduce the notion of perspective as a means to mediate this difficulty. In art, this allows appreciation of an artist's viewpoint. Perspective in engineering also helps create common viewpoints on which to optimize communication among users, buyers, and builders.

Object-oriented views offer users of software systems a means to deal with complex software abstractions in familiar terms. In addition, object-orientation provides builders a means to reuse artifacts efficiently. Realizing Cook's [16] first principle of object technology (i.e.: hiding the data behind the processing) also requires an information structure within which useful and feasibly produced implementation objects are identified and elaborated. These differing aspects of object orientation suggest the levels of architectural information that are needed to successfully evolve systems.

Evolutionary reengineering is iterative and incremental. The continuing challenge is to determine how to know, and when it is true, that the interim results are proceeding most rapidly toward the ultimate mission needs of the user. Insight from consideration of the object-oriented and architectural bases for software evolution offers a model that provides the leverage needed to answer these questions.

Software architectures are conceptual, intangible, and abstract. Yet if they are to guide software engineers, they must be concrete, visible, and as obvious as possible. To achieve this goal, we apply the notion that one learns best while doing. From this comes the conclusion that architecture is most effective in communicating vision when it is thoroughly integrated into the evolutionary process itself. Architectural insight clearly provides a framework for that process. By inverting the title of Best's article ("If They Built Buildings the Way They Build

[1] This work is sponsored in part by the U. S. Department of Defense. The views and conclusions contained in this document are solely those of the author and should not be interpreted as representing official policies, either expressed or implied, of the Software Engineering Institute, Carnegie Mellon University, the U.S. Air Force, the U.S. Department of Defense, or the U.S. Government.

[2] The concepts presented in this paper are based partly upon past experience in maintaining large scale software systems for satellite ground data processing, and based partly upon recent work at the SEI related to a mission critical evolutionary development program.

Software") we suggest a way to define a practicable, and pragmatic paradigm for applying architectural concepts toward reengineering large scale software systems [11].

Evolutionary reengineering process mediates problems in dealing with large-scale systems but fails (1) without user-oriented direction and control, (2) without a process framework that assures that mission needs are met, and (3) without consistent engineering methods and evaluation paradigms. Gelernter's insight that a different way of viewing large scale systems (i.e.: that software "shadow systems" are needed to track the evolution of actual systems) is the beacon followed in this paper for identifying technology needed to enable evolutionary reengineering [23].

An architectural framework for evolution should offer (1) direction based on vision clearly understood by both users and builders; (2) controls that reflect knowledge of both legacy systems and problem domain; and (3) correlated process framework, engineering methods, and evaluation paradigms.

The thesis of this paper is that an information architecture provides this framework. To develop this position, the use of architectural concepts is discussed, and hypotheses are offered regards applying them for evolution. Architectural views, combined with object-oriented concepts, prevent the failure modes identified for evolutionary reengineering.

Terms of reference are offered to frame the discussion. Architectural concepts that add value to the evolutionary process are introduced in the context of object oriented techniques. A set of five architectural propositions for evolving systems suggest how architectural approaches contribute to software engineering activities.

Discussion of engineering activities then illustrates how object-oriented considerations and levels of architectural information help assure engineering results that meet long-term mission needs. The value of information architecture in guiding this activity is summarized in a five-point conclusion.

2 Concepts and Definitions

2.1 Architecture

The term "architecture" is used in this paper in the context of abstract systems. Architecture reflects many kinds of structure and is a basis for several kinds of reasoning. Ideas presented are influenced heavily by tutorials and presentations from M. Shaw [53], and J. Zachman [64,65] and by papers from D. Perry [44] and B. Gaines [21]. Saunders [49, 50] and Horowitz [28] offer views of the nature of software architecture for use in support of software acquisition. Anderson's ideas [5] that architectural concepts are the structuring paradigms of software systems, and that envisioning a product family involves creating an appropriate architecture, are also helpful in obtaining an operational understanding of what the term architecture means to software engineering. Jones' [31] definition of architecture offers the following key points:

- a structure of elements of known properties,
- which have rules of interaction and relations
- and provide a basis for reasoning

2.2 Evolution

The notion of "evolution" is used in a literal sense. As a "series of changes", evolution must have a starting point (legacy). Having "a certain direction", evolution differs from an unconstrained series of modifications that may have little direction over the long-term. It also differs from development, which has both direction and precisely specified results. Webster defines evolution as

- a series of related changes in a certain direction

2.3 Object

In this paper, the literal definition of "object" is broadened by the notion of "cognitive apprehension", versus the more literal "visible", and extended by including software engineering notions re object capabilities. This produces the definition that follows:

- a tangible or cognitively apprehensible thing, having
- discrete boundaries, and having
- intrinsic information and state, and
- capabilities including communications

2.4 System

The idea of "large scale software system" is central to this paper. In the literal definition of "system", the key idea used is "aggregation of objects". An extract from Webster brings this out.

- a complex unity of diverse parts
- serving a common purpose

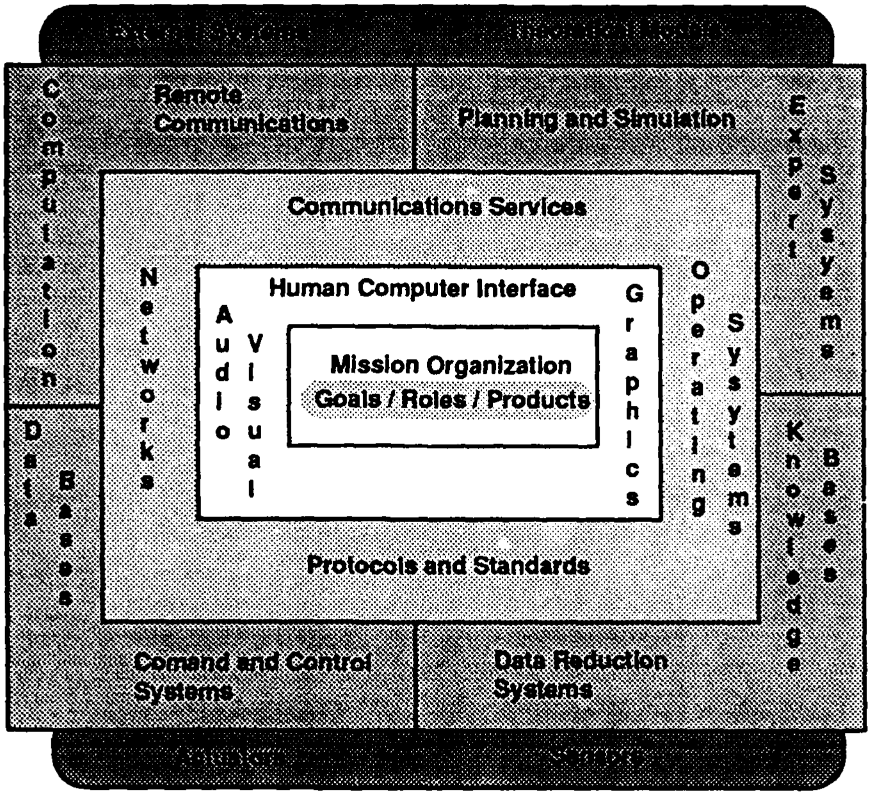
- an aggregation of objects
- joined in regular interaction or interdependence

3 Leverage for Reengineering

Using an object-oriented approach to identify, elaborate, and incrementally realize a software architecture provides leverage for several aspects of rereengineering large software systems. Both the object-oriented and the software architectural approaches improve communications about the system by reducing technical and mission-oriented goals to familar aggregates of computer processing, user functionality, and mission data. Both approaches support evolution of new capabilities versus specialized development of integral functions. The combination of the two approaches facilitates efficient integration of the improved system capabilities which satisfy ultimate user needs.

3.1 Object-Orientation

The Farringdon Group report on systems productivity with an object-oriented paradigm cites disintegration of manual user functions, inability to modify newly automated functions, and complications in accessing and sharing data as the intransigent problems of classical computer systems development. The object-oriented approach to systems development is neither object-oriented programming, nor object-oriented design, nor object-oriented analysis. This approach applies the concepts underlying these object-oriented disciplines so as to realize systems that are designed for modification, structured around real world objects, and composed of self-organizing components. The underlying principles of object-orientation, which are encapsulation, information hiding, classification, inheritance, and polymorphism are applied in this object-oriented systems approach to build readily evolvable parcels of software whose functions and data are localized; which intrinsically reflect real world objects; which avoid artificial division of enterprise activities into "automation functions"; which rely on "intelligent data packages"; and which emphasize systems development as a series of iteratively refined modeling activities. This latter emphasis is inherently evolutionary in that development of new applications is an implicit result of continuing modeling of the attributes and operations of new and refined objects of the mission or enterprise. In the Farringdon Group's object-oriented systems development paradigm, the associated development environments and applications architectures



adapted from B. Gaines

Figure 1 - A Layered Architectural View of Generic System Capabilities

are themselves similarly integrated elements (e.g.: class browsers and debuggers, subsystem prototypes, and federated platforms and information infrastructures) of the large scale object oriented system [19].

Capretz examined methodological aspects of the object-oriented paradigm. He cites instances of jeopardized traceabilty of requirements when object-oriented and structured methods are combined, and strongly recommends use of object-oriented design and analysis methods so as to assure realization of object-oriented systems. However, he found need for more experimentation before a large scale software system can be developed without risk with use of an object-oriented approach, and he strongly advocates the pursuit of improved object-oriented analysis and design methodologies [12].

Shelton reports the criticality of using an iterative development approach to guide implementation of an object-oriented methodology. He cites distinct needs for enterprise (mission-oriented) models, for operational models that have application independent components, and for implementation models (blueprints for service layer classes of objects), as well as for integration models that are class level physical designs for object implementation. Where legacy systems are available for integration, these latter integration models deviate from implementation models and become the means for integrating legacy components and features [54].

Analysis techniques link the object-oriented paradigm to architecture-based development approaches. Coad details procedure for object-oriented analysis. It is based upon recognition of three intuitive analytical approaches: differentiation of experience into objects and attributes; distinguishing between whole objects and their component parts; and formation of and distinguishing among classes of objects [15].

In the manufacturing domain, Coad's object-oriented modeling paradigms have been applied successfully to develop process planning and assembly control software applications. The generality of these object-oriented applications has led to practical definition of subsystem software architectures [4].

3.2 Architectural Leverage

The architectural approach for software engineering of large scale systems is being researched by the Advanced Research Project Agency (ARPA) program in Domain Specific Software Architectures (DSSA). Different approaches have been examined. One of these is oriented toward exploiting domain expertise. This is represented in a domain model that is independent of any implementation. Another approach is based upon representation of a particular architectural style associated with a hierarchy of control devices. A third approach relies upon formal engineering models of domain dependent computations. However, all of these approaches provide a basis for a software architecture that supports both focus on and resolution of design decisions, and which becomes a framework for development support tools. As a result of this research, common languages for evolution of domain-specific applications, and frameworks for software reuse are becoming available [41].

Perry emphasizes that the three chief values added by the architectural approach are significantly greater support for software reuse, increased support for generational reuse, and insight into the nature of principles for composition of software systems [45]. Hayes-Roth et al. report that their development of architectural approaches for software systems development has enhanced their ability to provide knowledge-based, artificially intelligent tool support for the process of developing controller applications [26]. Agrawala et al. report that their research relies extensively on formal models with which an open toolset and an layered architecture are used to increase applications' reliablity, real-time performance, and fault-tolerance [3].

In advancing the understandabilty of evolving systems, architectural representations are of particular value. Certain capabilities are common to nearly all systems. These can be viewed as generic attributes of typical system functions via a layered structure of related generic capabilities. Figure 1 presents a set of layered slices of system capability as concentric rings of increasing relevance to user operations from the outer to the innermost. These layers themselves are arbitrary; the aim of the diagramming technique is to facilitate understanding of the system [21]. The layering approach to modeling evolving systems is also applied to formal representations that are well suited for development of tools which can automatically provide a capability for program restructuring [25].

3.3 Enabling Evolution

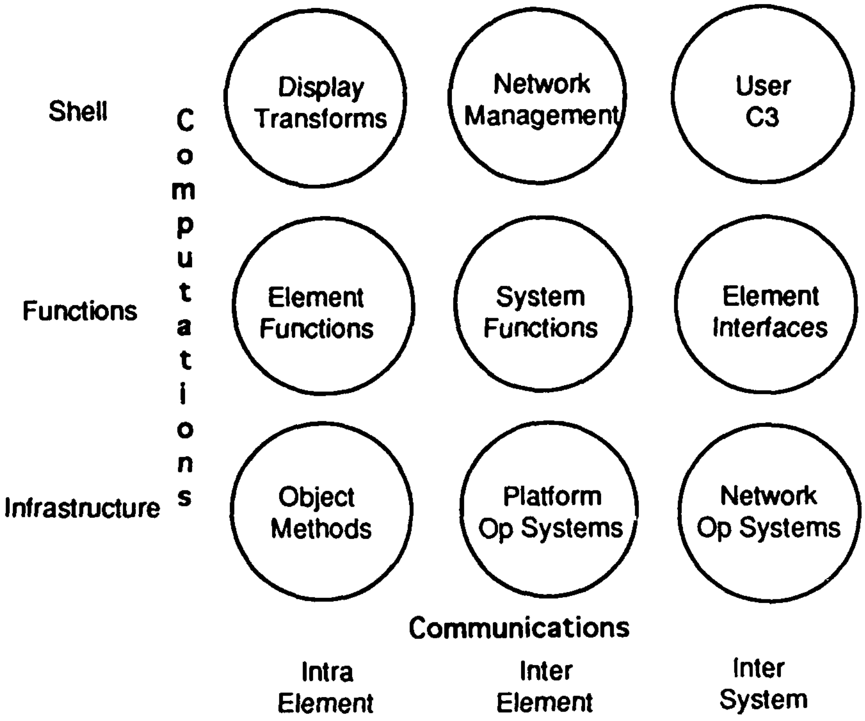
The layout of the concentric rings in Figure 1 also suggests relationships within the system that will allow isolation of locales in which technological upgrades can proceed independently from changes in mission or functionality. These locales are sites for technology growth. Analyzing them identifies paths for technology transition that are essential for successful evolution.

Evolution of system capabilities is further advanced by definition of intra-object schemas. Mittermeir proposes a semantic roadmap to an object's subcomponents, and a service channel into an object. Through these paths, high level modification operations on aspects of the structure of the object can be performed [42]. Limited implementations of this kind of capability currently exist in commercial software architectural frameworks which enable development and operation of object-oriented workstation applications [18].

Using life cycle products of previous developments is the essence of evolution. Basili and others have defined a reference architecture for a software factory which enables the derivation of specific architectural instances [7]. The potential of this software factory for supporting evolution of software system artifacts is presented as a case study of a Toshiba Corporation development organization which produces application programs for manufacturing process control systems. A high degree of productivity is reported to have resulted from reuse of over 50% of code artifacts. The lack of models of experience functions, such as result from the object-oriented modeling paradigms described above, is seen as the limiting factor in the productivity of this factory. Another aspect of enabling evolution involves support for prototyping. Maxim et al. describe an object-oriented design tool which permits non-programmers to construct high performance configuration design systems graphically from a library of reusable mechanisms. His experimental system also includes tools to support analysis of the performance of the resulting code [40].

The utility of an architecture for enabling evolution of an acceptable systems design is greatly enhanced if its perspectives completely span the range of systems of interest to the user. The smallest set of such perspectives has members that are completely uncorrelated with one another.

For software systems, basis architectural perspectives should address topology, behavior, function, and information. In the list that follows, these perspectives are represented by models of varying fineness. This fineness reflects composition relationships among system elements, objects, data entities and operations, and similar relations among system components, processes, tasks, messages and calls. Shaw [52] addresses the composition of systems from subsystems in which codification of architectural features becomes a similar basis for developing formal system specifications.



adapted from Uenohara

Figure 2 -- A Sample Decomposition of System Components

Organizing, rationalizing, and rectifying systems models from architectural perspectives helps assure complete and consistent reflection of all key system properties for an evolutionary software engineering process. The following list of diagrams illustrates several different perspectives from which a software architecture can be modeled.

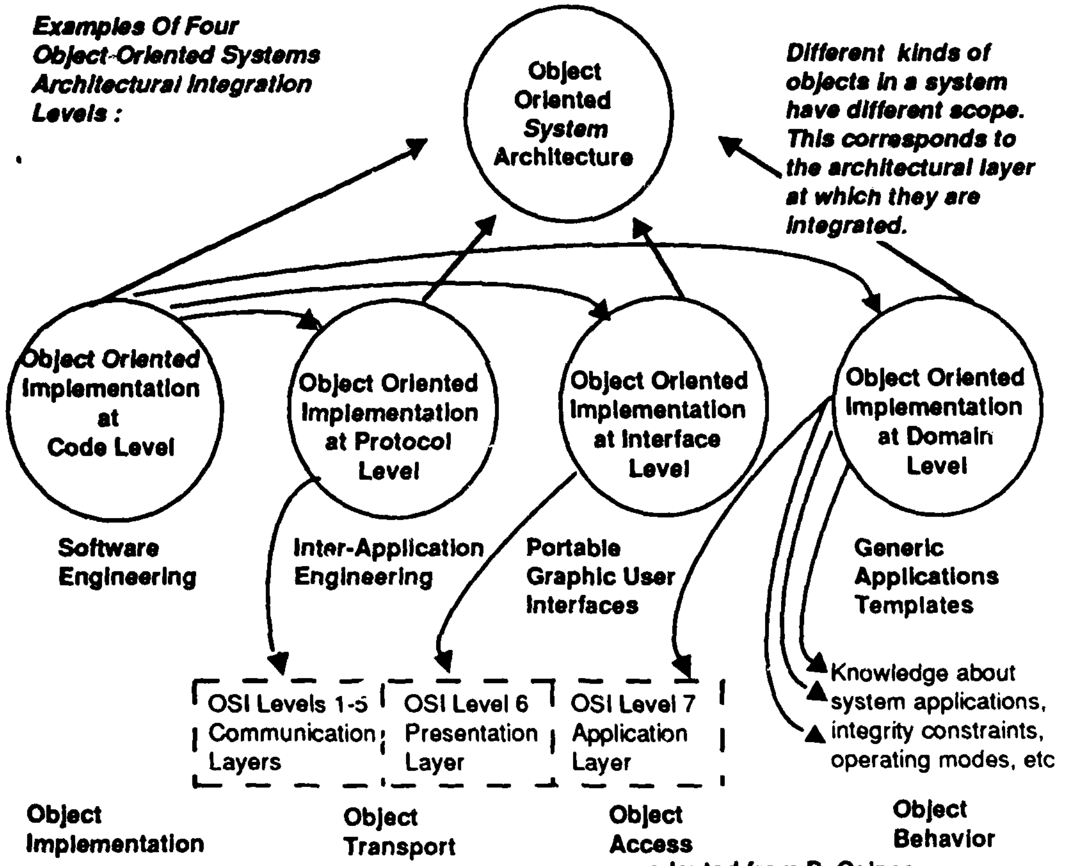
- Generic Framework Model
  (topology of elements)
- Control Flow Model
  (operations or activities)
- Data Flow Model
  (data transformations)
- Dynamic Behavioral Model
  (state transitions)
- Object Model
  (element attributes; relations)
- Object Interaction Model
  (messages, calls, callbacks, flags)
- Functional Flow Model
  (components, processes, tasks)

3.4 Understanding System Properties

The rules of interaction between elements of an architectural model, and the properties of those elements, are themselves captured in the above models of static and dynamic aspects of a system. Software systems usually have at least two kinds of properties: communications properties and computational properties. In terms of these two, a general decomposition of the component structure of a system can be readily obtained with another layering approach.

Computational layers may include a shell that relates computations to the user, a functional layer, and a supporting or infrastructural layer through which theory, mechanisms, and operating environment are linked and engaged. Communications layers may address interchange within system elements, among those elements and outside the system.

Figure 2 illustrates the decomposition of an arbitrary system into categories of components that are indicated by pairs of system properties from within the above layers [60]. These categories of components are


Examples Of Four Object-Oriented Systems Architectural Integration Levels :
Object Oriented System Architecture
Different kinds of objects in a system have different scope. This corresponds to the architectural layer at which they are integrated.
Object Oriented Implementation at Code Level
Object Oriented Implementation at Protocol Level
Object Oriented Implementation at Interface Level
Object Oriented Implementation at Domain Level
Software Engineering
Inter-Application Engineering
Portable Graphic User Interfaces
Generic Applications Templates
OSI Levels 1-5 Communication Layers
OSI Level 6 Presentation Layer
OSI Level 7 Application Layer
Knowledge about system applications, integrity constraints, operating modes, etc
Object Implementation
Object Transport
Object Access
Object Behavior

adapted from B. Gaines

Figure 3 -- A View of Integration Levels of Object-Oriented Systems

inherently understandable to both builders and users. Lane [35] identifies three other categories of components (applications-specific, shared user-interface, and device-dependent) which are useful in describing structural alternatives. In his work [34] with architectures for user interface so^ftware, he identifies a design space of 25 functional and 19 structural dimensions for which alternative component designs can be selected to form a user interface software architecture.

Characterization of systems components via this sort of layered decomposition can help users and builders uncover and prioritize required features and performance factors in terms that are mutually intelligble.

An object-oriented system architecture uses several different kinds of objects to convey system properties, to describe the fine and coarse structure of design, and to enable reasoning about various system properties.

3.5 Integration Layers

Four kinds of objects are typically found in software systems. Gaines [21] identifies these objects as belonging to the four levels of integration depicted in Figure 3. The most elementary or fine-grained objects deal with typical code-level implementations such as for device drivers. At the next higher level, protocol standards may be reflected in objects that provide generic processing interfaces for communications. Presentation of systems outputs requires different kinds of protocols for data bases and graphical user interfaces. Finally, functional applications can be composed of objects whose apprehensibility derives directly from the problem domain.

The Object Management Architecture [56] represents one object-oriented means to facilitate efficient integration of software objects. It defines means for inter object communication and identifies operations that all classes must support, and common objects that are useful in wide ranges of applications.

Evolutionary integration of an object-oriented system proceeds from the bottom up, providing an integrated base of protocol and implementation objects that can be used for testing higher level constructs. Lorin [39] notes the necessity of design method that supports instances of bottom up effort where super classes are extruded from sets of smaller base objects. This is effectively a way to discover the structure of a solution from its basic components. At the same time a top down integration provides system level capabilities for prototyping and user evaluation.

3.6 Potential Value Added

From the foregoing discussion, several

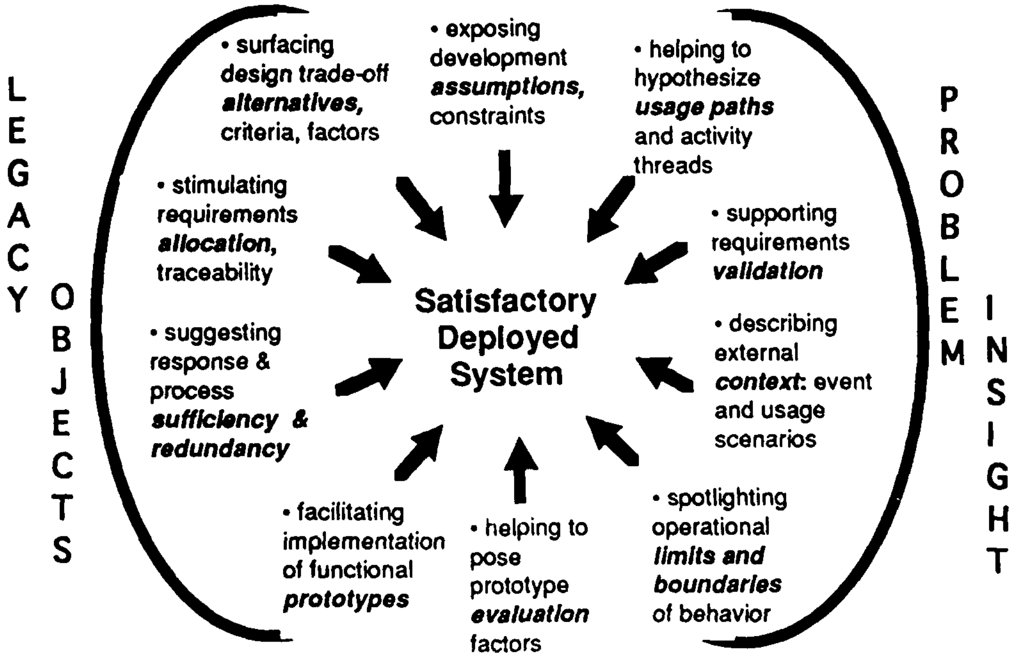

LEGACY
OBJECTS
• surfacing design trade-off ***alternatives,*** criteria, factors
• exposing development ***assumptions,*** constraints
• helping to hypothesize ***usage paths*** and activity threads
PROBLEM
INSIGHT
• stimulating requirements ***allocation,*** traceability
• supporting requirements ***validation***
Satisfactory Deployed System
• suggesting response & process ***sufficiency & redundancy***
• describing external ***context:*** event and usage scenarios
• facilitating implementation of functional ***prototypes***
• helping to pose prototype ***evaluation*** factors
• spotlighting operational ***limits and boundaries*** of behavior

Figure 4 -- Challenges to Achieving Evolutionary Success

observations can be made about the potential value to be added to reengineering of large scale systems by using an object-oriented and architecture-based approach.

Architectural views of generic system capabilities improve user understanding of systems, and help builders plan better technology growth paths. Garlan's experience with a course in software architecture testifies that architecture

- increases the shared understanding of high level relathionships in systems,
- facilitates development of new variations of previous systems, and
- allows the software engineer to make principled choices among design alternatives. [22]

Zachman recognized the value of multiple views for improving information system development [65]. His model suggested a perspective-based architectural approach for re-engineering of information systems [36].

Generalization and specialization of systems are easier when systems knowledge is categorized and organized in an architectural fashion.

Characterization of systems components via layered decomposition helps uncover and prioritize required features and performance factors.

Different kinds of objects in a system will have different scope. This corresponds to the architectural layer at which objects are integrated.

Organizing, rationalizing, and rectifying system models from architectural perspectives helps assure complete and consistent reflection of key system properties for an evolutionary software engineering process.

The above observations regarding the value added by architectural approaches for evolving systems can be summarized in the following propositions:

- Evolutionary direction is guided by architectural vision and facilitated by robust legacy.
- Representations of structure promote understanding of generic capabilities.
- Architectural views frame both system specific and system generic characteristics.
- Layering techniques help architectural models to surface properties of system components.
- Object-oriented architecture aids in prototyping user capabilities from the top down, and aids in integrating systems platforms from the bottom up.

4 Challenges to Successful Evolution

Evolutionary reengineering must satisfy user needs within limits that are posed by

(a) physical constraints
(b) allowable budgets for time and money,
(c) unknown problem domains
(d) volatile and poorly understood requirements
(e) current technology

Misjudgments of any of the above factors can misdirect the course of evolutionary reengineering such that resulting operational capabilities

(a) perform outside required tolerances,
(b) consume more budget (of time or dollars) than is available,
(c) provide insufficient flexibility for planned future enhancements,
(d) fail to include all required features or mis-estimate priorities or other parameters of operations,
(e) constrain future adaptability or present performance by adopting unswapable architectural elements or underwhelming technologies.

Evolutionary reengineering needs to provide for integrating future technology, and to expedite resolution of physical and logical engineering decisions by building on a base of objects synthesized from legacy systems. It also needs to leverage a base of architectural and problem domain knowledge. This must yield sufficient insight to enable efficient model-based interaction with users to validate operational requirements. It must also facilitate evaluation of prototypes to derive system functions that are acceptable to users.

In guiding the path of evolution, pre-resolution of certain kinds of questions is essential. Especially in Command, Control, and Communitations (C3) systems, the chief need for evolutionary development is the fact that not all requirements can be understood or known explicitly at the outset of new development or reengineering effort [10]. Salasin and Waugh [47] observe that in order to effectively reengineer systems, information is needed about alternative processes, decompositions and data structures individually and in terms of their relationships. Both decisions to use a particular process and dependence upon data and other processes must be explicitly visible if they are to be used effectively in evolving systems.

These needs are the basis for the challenges that are illustrated in Figure 4.

5 A Model for Systems Evolution

Systems evolution differs from systems development and from systems modification. Small modifications, over time, can create large system changes that may not have had a single direction or unifying intent. Systems development can start from scratch. However, systems evolution is directed towards long term user needs, and it operates on the legacy of existing systems. This legacy includes the target system itself, and an existing depth of knowledge about the problem domain.

To satisfy users, a systems evolution process must assure that changes create enhanced operational capabilities that meet long-term needs. Architecture provides guidelines for the direction of evolution, and enables development of constraints that organize the evolutionary process.

The consensus fostered by architectural representations enables early identification of critical issues regarding the use of existing technology and of both hardware and software mechanisms and artifacts. This consensus similarly helps to surface details of the problem domain that require specialized analysis or that stress system engineering or performance capacities.

Successful evolution results when concerns for synthesis of mechanisms are separated from concerns for analysis of problems, and when those concerns are handled in parallel activities. Legacy mechanisms are inherited from legacy systems and from general purpose design activities that fill needs of multiple products. Problem knowledge acquired from domain analysis offers specific insights that permit the tailoring of applications solutions that uniquely meet user needs.

Figure 5 offers a model for evolving software systems. It shows the stages of the evolutionary process from conceptualization of the system through evaluation of tentative implementations. It differentiates two distinct aspects of the evolutionary process and the artifacts that

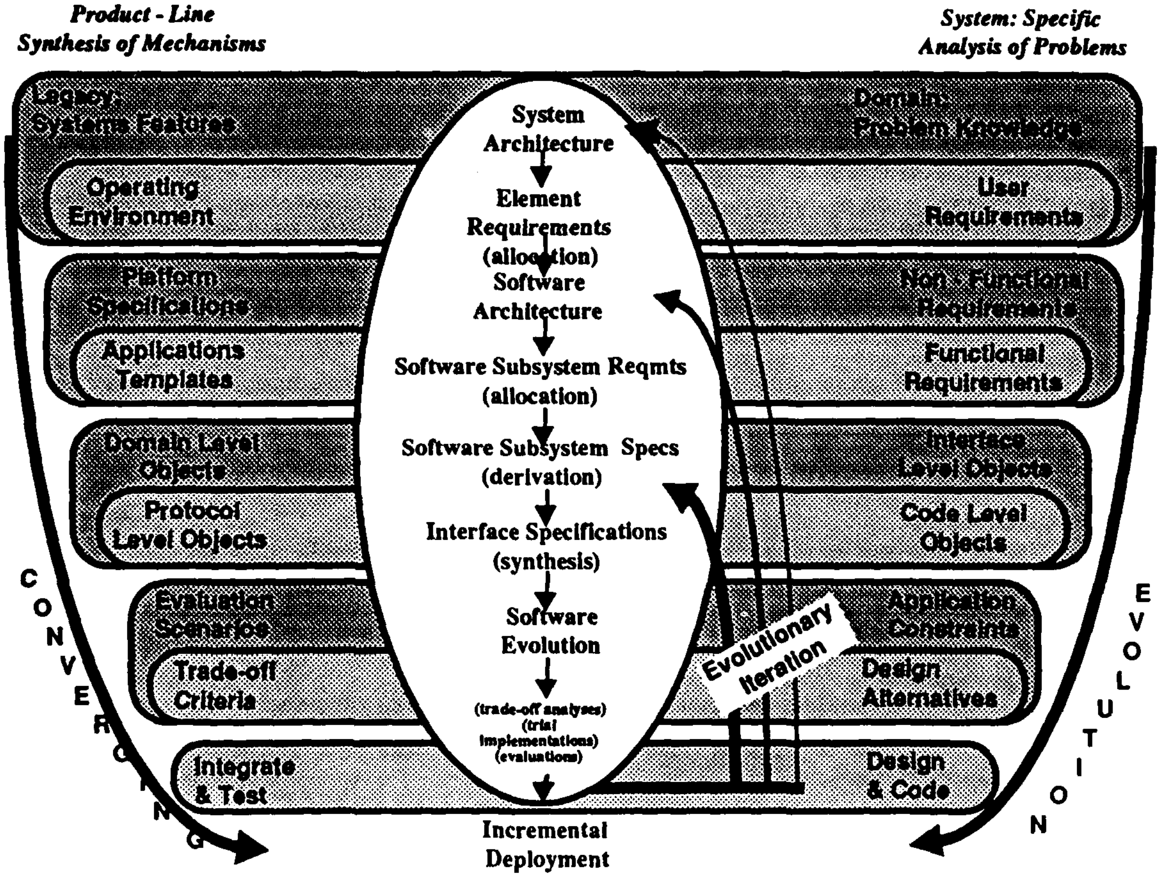


Figure 5 -- Evolution of software systems

comprise the evolving system. One aspect reflects *artifacts that are derived from the legacy of prior systems so as to provide features of new or changed systems.* The other aspect reflects the sequence of analytical activity through with fresh knowledge of a problem is iteratively teased out of increasingly detailed problem specifics.

Rules of component composition and ways to standardize component interfaces based upon common units of avionics components were investigated by Batory and others [8]. Properly staged composition and flexible harnessing of artifacts from both legacy sources and domain knowledge are the essence of this model for systems evolution.

6 Information Architecture Concepts

System evolution using an information architecture to preserve a focus on user needs is "converging evolution". This use of an information architecture leads to a refinement of system functions that satisfy user needs. It also facilitates the identification of generic interfaces between subsystems. These permit reuse of software components and enable continuing refinement of software mechanisms via a process of successive replacement.

The ongoing evolutionary development of Command, Control and Communications capabilities for the Ballsistic Missile Defense Organization (BMDO) is an evolutionary development that must consider the reengineering of many existing systems in order to integrate the overall BMC3 capability. Figure 6 summarizies concepts for an Information Architecture as they are viewed by contractor and government engineers involved in this activity. Urban and others have elaborated the underlying concepts for the BMC3 Information Architecture [61]. Their draft paper identifies this information architecture as "an information model for evolving the system."

Information architecture content provides terms of reference, guidance for defining and supporting engineering process and organized abstractions in model form that represent structural, process and behavioral aspects of the system.

An essential starting point for developing an information architecture is a representation of system operating context in terms of user activities and the external events that stimulate the system. Hufnagel and Harbison propose a methodology which emphasizes these user views of a system in their seamless, scenario-driven object-oriented approach [29,30]. The key notion in their work is that a meta-linguistic object-oriented approach can be used efficiently to organize system objects. They propose a domain independent and virtual specification of systems based upon conceptual analysis of a systems' requirements, specifications, and designs. Artifacts of such scenario-based approaches are essential elements of an information architecture.

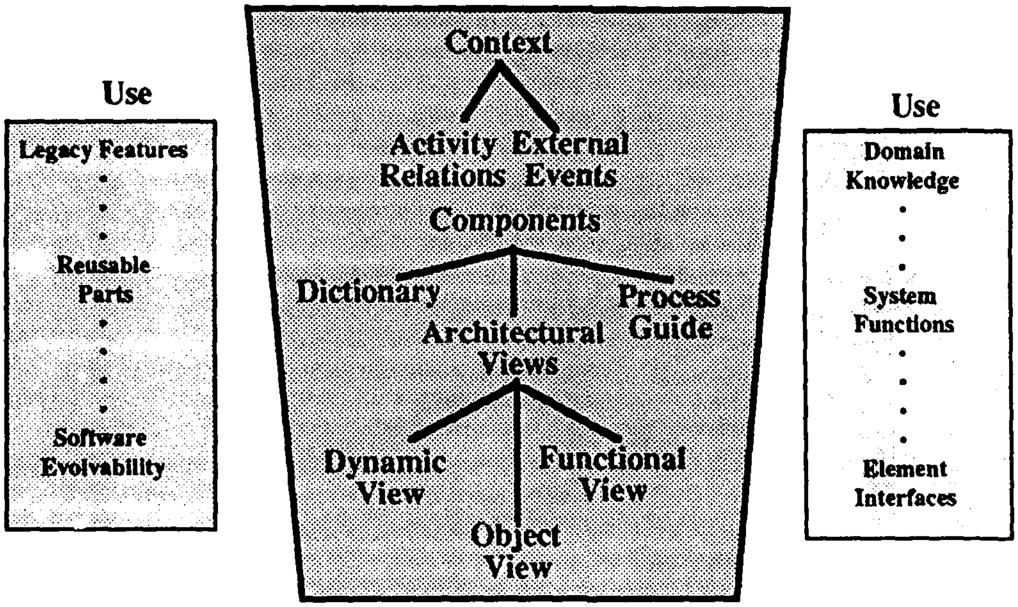


Figure 6 -- Information Architecture

Abstractions of attributes of system properties and component features are essential when allocations are made and derivations are worked out at each level of legacy synthesis and at each level of problem analysis. Abstractions of properties and features help surface the constraints, assumptions, factors, and alternatives needed for thorough analysis, specification, implementation and evaluation. They thereby facilitate evolution that satisfies long-term user needs.

7 Information Architecture Methods

Methods for applying information architectural concepts address management, strategy, evolutionary process and tools, modeling, representation and analysis, and use of information architectures.

7.1 Management

The Air Force Science Advisory Board (SAB) sponsored a 1993 summer study on Information Architecture [17]. It recommended that the Air Force develop of an enterprise wide information architecture. This was characterized as an enterprise-wide building code that is layered, open, and driven by commercial off-the-shelf (COTS) considerations. A focus on common data element definitions and on applications interface standards and conventions was also recommended. A process for managing architecture development was advocated as a means to apply an information architecture to both administrative corporate information management (CIM) applications and to tactical warfare (mission critical) applications. Four facets of this process are appropos for implementation of any information architecture. These facets are

(a) establishing a continuous process for evolving the "building code" to meet changing needs including compatibility with external organizations.

(b) involving users and developers in assessing and evolving this building code.

(c) applying with accountability the concept of "central direction and decentralized execution" to the architecture development process.

(d) placing a priority focus on developing

- enterprise architecture and process,
- interoperability,
- use of COTS technology,
- continuing utility assessments of standards,
- tools for defining / analyzing architectures,
- tools for migrating legacy software

7.2 Strategy

Commercial tool vendors recommend technical strategies for developing information architectures. Many current commercial strategies address needs identified by the Air Force SAB. Three of these were advocated by Lock [37] at the 1993 CASE World Conference:

(a) aim the architectural building code at mission functions and not at implementation technology by

- establishing internal /external exchange protocols
- connecting to existing applications and data
- migrating first data, then functions
- adopting broad based external standards

(b) allow the architecture to guide the building environment with layered designs, and equivalent treatment of hardware and software entities, by requiring

- generalized application tool kits,
- widely usable application services,
- easily accessible production services, and
- pervasive infrastructure elements.

(c) developing tools for design engineering and performance evaluation which support reverse engineering, and integrated modeling of mission, network, and data functions.

7.3 Evolutionary Process and Tools

Lockman and Salasin describe an object-oriented approach to implementing a four phase evolutionary reengineering process [38]. With these phases they "appeal to intuition" in their advocacy for creating and operating upon an object-oriented depiction of the current system to realize new or extended features of a similarly depicted target system. Their work identifies specific steps are for each of these phases, and examines the needs and opportunities for tool support. The intuition regards both a transformation-oriented reengineering process and the use of an object-oriented representation for an information architecture, which they offer in their 1989 paper, is validated by three current architecture-based software engineering environments, all of which support an evolutionary reengineering process. These are SNAP [18], ANSA [1], and DISCUS [20].

SNAP is targeted at rapid implementation of client-server applications subsystems which operate upon workstations and networks of distributed processors. It relies upon a standard architectural template to organize and support the process of evolving new objects within generic class libraries. These class libraries address graphic user interfaces, communications, data base access, external applications, internal storage, and knowledge based support functions. SNAP has been applied successfully with high productivity to developments of decision, analysis support, and command and control systems in such diverse applications as anti-submarine warfare and air traffic control.

ANSA is a programming support environment based upon an architecture that has been represented with enterprise, information, computation, engineering, and technology models of generic distributed systems. This architecture makes the fact of distribution transparent to application builders and users, and produces distributed applications subsystems which can be managed and evolved as a coordinated whole, rather than as separate black boxes with specialized, and potentially incompatible development paths.

DISCUS is a generic reusable software architecture which provides high levels of reusability between tools and data sources. Its aim is realization of seamless interoperability, particularly for the class of workstation applications which involve significant image manipulation. It represents one of several MITRE Corporation sponsored efforts in the area of evolvable systems development. Other similar systems / architecture-based software engineering environments include EXCITE (coordinated sharing of information for intelligence analysis), DOMIS (distributed object management system for integrating legacy data bases), and systems for support of collaborative computing and for support of distributed simulations [9].

The evolutionary reengineering process has an intrinsic focus on design and on redesign. Few generic tools have yet been developed to support generic design activities. However, the nature of these activities is becoming better understood to correlate well with the use of architectural information. Studies of the design process also suggest several information attributes that are needed in architectural representations to support typical designer behavior. Adelson and Solloway report that, in general, designers exhibit six characteristic behaviors [2]. These behaviors are as follows

(a) formulation of mental models
(b) simulation of mechanism behavior
(c) systematic expansion of level of detail
(d) representation of constraints
(e) recollection of prior plans and strategies
(f) annotation of intermediate artifacts

7.4 Modeling, Representation and Analysis

Computer Integrated Manufacturing (CIM) Open Systems Architecture (OSA) documents [33] provide a succinct explanation of concepts for modeling software architectures and applying specific architectural concepts. The CIMOSA modeling concept shows how to develop enterprise models in an evolutionary mode, and illustrates the impact of information architecture on the evolution of information intensive systems. Requirements for modeling languages which support integration of layered models have been identified by Gielingh [24], whose work also addressed evolutionary reengineering for CIM applications. He argues that to support layered modeling, information modeling languages must support definition of modeling dimensions for specialization, discrimination, and orthogonalization. Specialization expresses a hierarchy of concepts, discrimination separates concepts, and orthogonalization identifies concepts which are independent of one another. Application of these definitions helps realize abstract and layered architectures which are truly evolvable.

Representation of information architectures inherently involves both legacy and problem domain information. Models can provide top-down representations of architectural features. From a bottom-up perspective, component attributes also must be represented. Tracz reports on the structure of a design record for Avionics Domain Application Generation Environments (ADAGE) [59]. He details 18 distinct elements of a design record for legacy avionics software. These include the following dynamically changing aspects of any potential software component:

(a) name / type
(b) description
(c) requirement specification fragment
(d) design structure
(e) design rationale
(f) interface specifications and dependencies
(g) program design language text
(h) implementation
(i) configuration and version data
(j) test cases
(k) metric data
(l) access rights
(m) search points
(n) catalog information
(o) library and architecture links

(p) hypertext paths
(q) models
(r) constraints

Analysis of problem domain attributes is essential for understanding and refining solutions which realize new capacities, improve non-functional qualities, refine existing functions, create interfaces to new or changed external contexts, etc. The nature of problem domains has been investigated using several techniques, many of which are characterized as "domain analysis". Wartik and Prieto-Diaz catalog and compare five differing approaches to domain analysis [62]. Brief synopses and references are provided below. Each domain analysis method offers a slightly different perspective on uncovering user needs and understanding the legacy upon which evolutionary reengineering must be based.

Prieto-Diaz' own analysis method is a hybrid of problem and solution oriented approaches. He supports bottom-up analysis with a classification approach and top-down activities with systems analysis, and aims to provide artifacts which can be reused. These range from problem features to solution mechanisms.

The FODA method of domain analysis and the Synthesis method follow a top down, problem oriented approach to analysis of a domain [43,14]. Both of these domain analysis methods focus on invariable, unique, and commonly used features of elements of the domain. The Synthesis technique is aimed at definition of a process model for application development, while the FODA method is focused on user decisions and views which can become attributes of product architecture.

The KAPTUR method [6] for domain analysis and Lubars's method (62] both provide domain models which can be more readily transformed by domain engineers into implementations. Both of these methods apply to the stepwise process of assessing legacy as well as to the analysis of problem domain attributes.

Another domain analysis approach was developed for the Joint Integrated Avionics Working Group, and applies an analysis technique which is based upon Coad and Yourdon techniques [27,15]. This

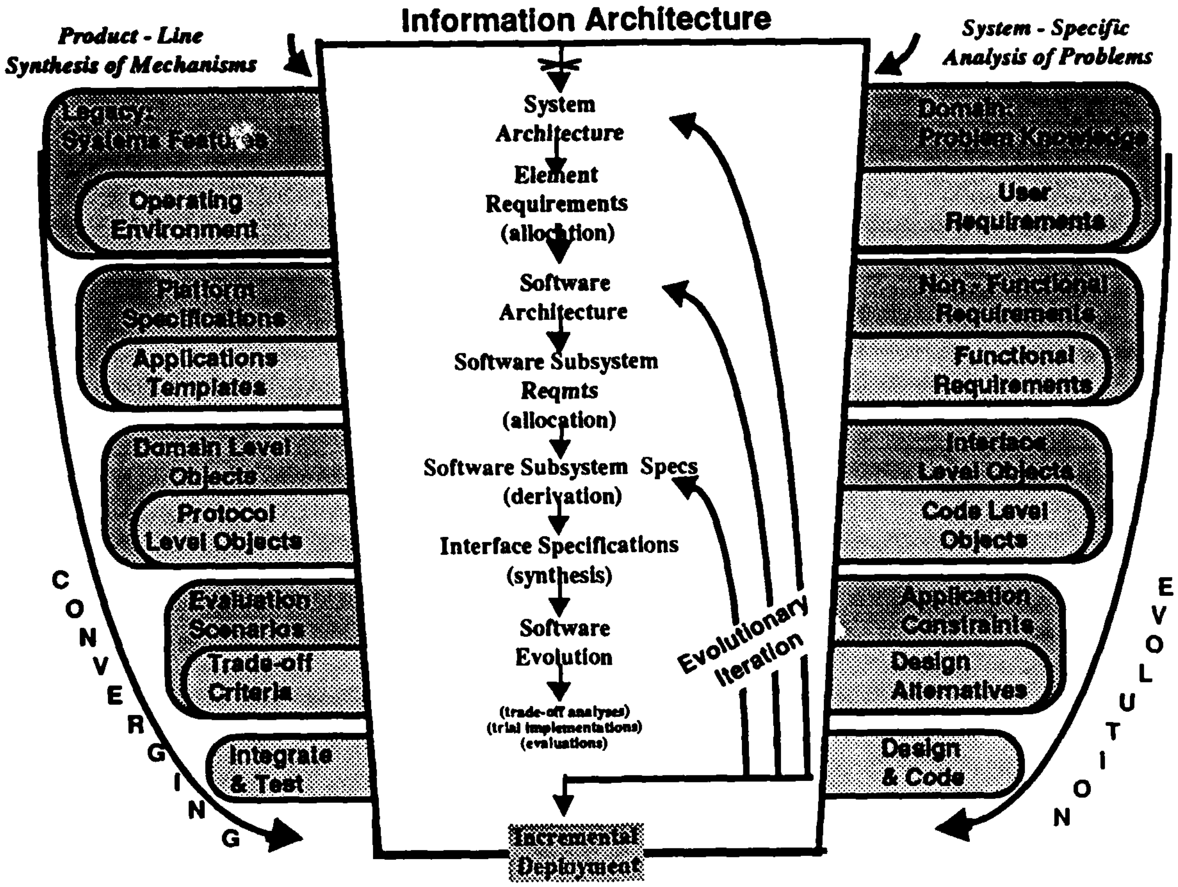


Figure 7 -- Information Architecture Guides Domain and Applications Engineering

technique is also implementation oriented, and produces detail which facilitates generation of reusable artifacts.

7.5 Use of Information Architectures

Gelernter's vision ("shadow programs" which mirror the operation of existing systems so as to facilitate their evolution [23]) points to the goals of recent research reported by Shaw [51] in which he investigates formal methods and mechanisms for executable, universal, formal, and scalable specifications. In specialized instances [55], some success has been reported with formal software development tools for automated transformational development. However, in the mainstream, information architectures must be developed by careful analysis of legacy systems and problem features. A process ("Implementing Model Based Software Engineering (MBSE)") is described by Withey which covers this more general range of applications [63]. MBSE consists of two parallel processes. One applies to domain engineering - the process for creating software models and other core assets; the other applies to application engineering - the process for using models in the construction of software systems. When software architectures are sufficiently mature in terms of their formal representations and in terms of the generality of their abstractions, then it will be possible to automatically generate components of applications. Presently, the MBSE process is largely manual. Its effectiveness is dependent upon careful use of modeling and representational formalisms and upon an information architecture to guide its convergence on long term user needs.

An information framework, within which design, behavioral, and engineering attributes of a system are collected and stored, should be structured to facilitate each level of synthesis and analysis that is needed to meet reengineering objectives. It must preserve views of legacy mechanisms and problem domain features which are significant to successful reengineering. Provision for defining design objects is needed within each of these views.

Figure 7 ı.epicts an information architecture as such a framework, and separates the related but distinctly different concepts of systems and software architecture.

In an evolutionary reengineering process, abstractions of system properties and component relations provide a basis for reasoning about the system. From this reasoning, both legacy details and domain knowledge can be used to create a system architecture. In the subsequent increments of the process for evolving new capabilities, allocations of requirements and derivations of specifications are revisited after evaluation of trial implementations that prototype significant new parts.

When tradeoff analyses are posed for design alternatives, independent technical factors must be scored to make balanced decisions. On the other hand, when quality is assessed, correlated assessment perspectives lead to quality generalizations. The domain insight and legacy detail within an information architecture lead to this general balance.

When boundary conditions and limits are known, evaluators can be assured of covering all cases of interest completely. Similarly the precedence of required features and the priority for behavioral options are derived from system context detail. These in turn derive from user consensus on domain features, which is facilitated by elaboration of an information architecture.

In the object-oriented community, a kind of meta-information architecture exists in the Object Management Architecture (OMA) [57]. The OMA Guide provides a general and an abstract framework for object-oriented systems that outlines a single terminology, technical goals (engineering process) and architectural goals (product feature), and provides a reference model for integrating distributed applications using object-oriented techniques.

8 Impact of An Information Architecture

Impact of an information architecture is seen in the realization of systems performance and capability by means of an efficient and cost-effective engineering process, which provides required levels of functional and non-functional quality.

An information architecture adds value to the evolutionary reengineering process by :

- Illuminating with abstraction, e.g.: functions, entity relationships, objects, object classes, processing states, data events, operational modes;
- Decorrelating relationships to enable balanced tradeoffs, e.g.: operational flexibility versus efficiency, or modifiability versus seamless integration;
- Correlating perspectives to enable qualitative assessment, e.g.: processing accessibility with availability, or data persistence with redundancy;
- Framing boundaries to enable thorough functional evaluation, e.g.: event scenarios, process threads, file schema;

- Providing context for setting precedence and prioritization, e.g.: operator scenarios, metadata, theoretical models.

Information architecture has a positive impact on acceptability, optimization, efficiency, and return on investment (ROI) of evolutionary reengineering efforts. It adds value to several aspects of an evolutionary system: These are its:

- capability
- performance
- engineering process
- overall quality

The object-oriented aspects of an Information Architecture offer the opportunity to assess the quality of a proposed reengineering effort according to the techniques proposed by Chidamber and Kemerer [13]. While these techniques are aimed at the evaluation of object-oriented design, they clearly apply to the envelope of object oriented design alternatives posed within an Information Architecture. Implications from the use of these metrics can include

- indications of design tradeoff opportunities (e.g.: between inheritance and related reusability on the one hand and simplicity and ease of understanding on the other);
- identifications of opportunity for optimizing the allocation of testing resources (e.g.: by identifying classes of error-prone interfaces); and
- assessments of the viability of the revised class structure proposed for the system (e.g.: in terms of numbers of object classes at the root level, and interconnections between various parts of an application).

To forecast the impact of an evolutionary reengineering effort, qualitative and quantitative metrics are necessary. Kazman, Bass and others have experimentally applied methods for evaluation of architectures [32]. Their approach applies a life cycle perspective (which considers engineering process aspects of an information architecture). It also relies upon a common representational form to surface a common understanding (which permits comparison of product attributes contained in an information architecture). Although present work has been limited to user interface architectures, it permits comparison and ranking of software architectures, and the approach promises to extend to other software architectures.

In their analysis of non-functional factors in the quality of large systems, Salasin and Waugh construct a

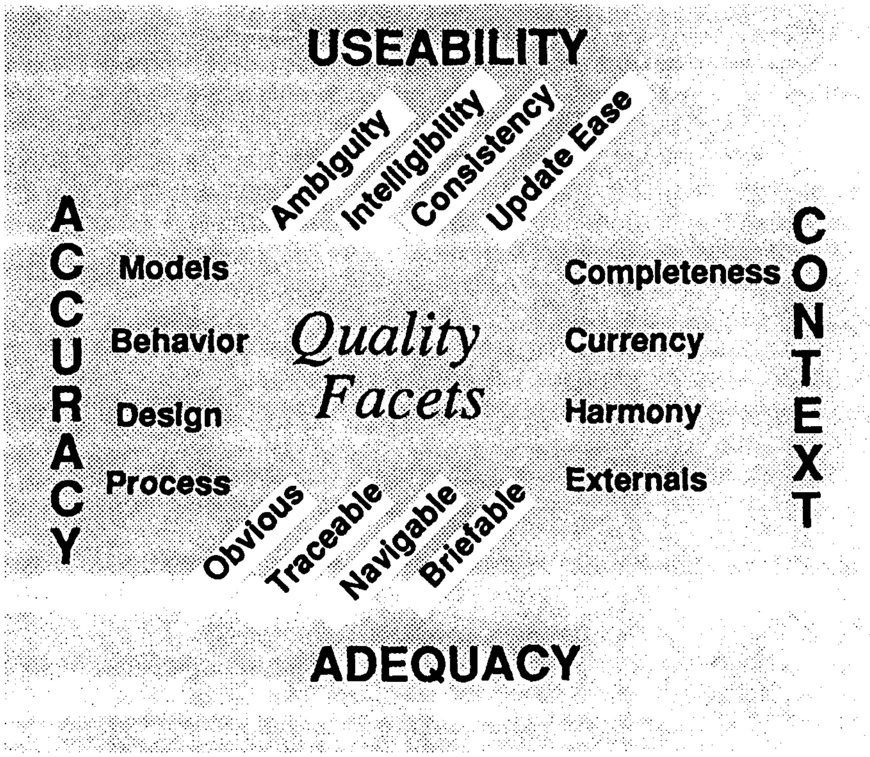


Figure 8 Facets of Quality of Information Architecture

chain of interlocking commitments and obligations to map system level quality factors into indicators of non-functional quality [46,48]. Their technique permits assessment of information architectural attributes so as to forecast such system qualities as testability and survivability. It enables an information architecture to become a source of primary evidence for on-going examination of non-functional quality characteristics of an evolving system. The approach uses several categories of reengineering scenario to stimulate analysis. The categories are platform changes (e.g.: processors, displays, software subsystems); performance improvements (e.g.: response, latency, speed, capacity, throughput, accuracy, precision); extensions of functionality (e.g.: updated constraints and operating parameters); changes to capabilities (e.g.: new missions); quality improvements (e.g.: modifiability); and new external interfaces (e.g.: swapped out functionality).

In general two kinds of factors indicate the quality of an information architecture: its intrinsic leverage is a function of content and context factors; its extrinsic leverage flows from its adequacy, usefulness and usability. Figure 8 relates several facets of an information architecture to four aspects of its quality.

Information architecture qualifies both behavior and design characteristics, and identifies necessary characteristics of the engineering process environment. As a framework for evolutionary reengineering, the value of an information architecture is indicated in its

(a) high-level, abstract models of the target system from all perspectives that add to developer and user insight (at least dynamic, functional, and structural).

(b) boundaries and definitions for both operating and engineering environments.

9 Conclusion

Likelihood of satisfactory operations of large scale systems increases with evolution by virtue of increased iteration to refine implementation aspects that meet long-term user needs.

Objects whose scope ranges from infrastructural mechanisms to problem domain artifacts guide builders and users toward efficient realization of systems components. Architectures provide structure and form for mediating this organization and understanding so as to meet constraints of builders and needs of users.

User-functional capabilities that aptly typify an evolving system are uncovered by comprehensive top-down analysis of its problem domain. This analysis also helps decide which models are the best platforms for evaluating prototypes of these capabilities.

Robust objects whose value has been proven in legacy systems are the result of continuing, bottom-up synthesis of previously successful products and systems. Layered organizations of proven objects enables rapid realization of platforms for evolving and evaluating prototypes of subsystems elements that have potential operational value.

Several conclusions about evolutionary reengineering of large scale software systems follow:

• Converging evolution assures operational success by realizing user goals incrementally and reducing risk of cost or schedule overruns and technical shortfalls.

• Layered object organization facilitates evolution by enabling efficient problem-oriented exploration and product-domain tailoring.

• Information architecture leads to converging evolution. Systems and software architectures frame the target for evolution, and information architecture frames the engineering activity.

• Analysis of problem domains is essential to uncover typical functions for evolutionary prototypes, and best models for evolving and evaluating prototype capabilities.

• Synthesis from legacy systems speeds definition of robust implementation objects from products that have had proven success, and realistic platforms for evaluating prototypes / initial operations.

10 Summary Observation

An analogy highlights the above conclusions. Evolutionary reengineering using an information architecture can be viewed as a problem of stabilizing the dynamic behavior of a software system. Linear systems models are used to approximate control solutions for such problems. Solutions are obtained via an iterative process of integration in which vector representations and matrix organization of the known information about the problems enable generation of parameters of the new stabilized system state. In evolutionary reengineering using an information architecture, the iterative process of integration can be viewed as a series of incremental changes to functionality and infrastructure through which new states of behavior and capability are evolved for the software system.

In this analogy, the system behavior of a software system and the nature of its evolution relate to the behavior and nature of a dynamic system. Information representing the resulting state of software capability is analogous to the state vector that represents the stabilized system state in the dynamics model. The driving force of user needs is comparable to the random noise vector that perturbs a dynamic system to cause a change in its state. Controls provided via an architecture-based evolutionary process derive from problem domain analysis and synthesis of artifacts from legacy systems. These are analogous to the control vector which constrains the stabilization of the linear dynamics system.

Stabilization of a dynamics model is dependent upon integrating the information content of these fundamental vector representations with a matrix of information that represents the changing system state. This matrix is analogous to an information architecture. Just as determining the precise form and content of a system state matrix is essential for developing a solution to stabilize a mechanical linear dynamic system, elaborating the content and understanding the necessary form of a matrix of information architectural information are essential to transforming a large-scale software system via an evolutionary process into a system with significantly enhanced, extended, or adapted capabilities. Controlling the convergence (stabilizing stages) of evolutionary reengineering requires that both the guiding architectural vision and the evolutionary process be represented in the same context and form.

An information architecture-based approach to reengineering provides a pattern for representing the evolutionary process itself, the control inputs provided by domain analysis and synthesis of legacy, and the stimuli provided by new user requirements.

In different terms, this analogy expresses the insight presented by Srinivas and Smith in their recent short paper on property preserving transformation of programs [58]. When combined with Gelernter's notion of "shadow programs", a path is clearly evident for realizing system features which support self-sustained evolution .

References

[1] -, "Advanced Network Systems Architecture (ANSA) Manual", Architecture Projects Management Ltd, Poseidon House, Cambridge, United Kingdom, 1989

[2] Adelson, B, and Solloway, E., "The Role of Domain Experience in Software Design", IEEE Transactions on Software Engineering, 1985, Volume SE-11, pages 1351-1360

[3] Agrawala, A., Krause, J., and Vestal , S., "Domain-Specific Software Architectures for Intelligent Guidance, Navigation, & Control", in Special Report CMU/SEI-92-SR-9, pages 63-71. Software Engineering Institute (SEI), Carnegie Mellon University, Pittsburgh, Pennsylvania, June 1992

[4] Almgren, R., and Hasson, A., , "Classification and Object Modeling of Assembly System Architectures", Proceedings of the International Conference on Object-Oriented Manufacturing Systems, pages 320-325, University of Calgary, Calgary, Alberta, Canada, May 1992

[5] Anderson, B., "Towards an Architecture Handbook" in Proceedings of the ARPA Domain Specific Software Architectures VII Workshop, Key West Florida, July 1993

[6] Bailin, S., "KAPTUR: Knowledge Acquisition for Preservation of Tradeoffs and Underlying Rationales", unpublished paper, CTA Incorporated, Rockville MD, May 1992

[7] Basili, V. Caldiera, G, and Cantone, G., "A Reference Architecture for the Software Factory", pages 53-80 ACM Transactions on Software Engineering and Methodology, Vol. 1, No. 1, January 1992

[8] Batory, D., "A Process and Retrospection on Creating a Domain Model for Avionics Software", ADAGE-UT-93-04, in Proceedings of the ARPA Domain Specific Software Architectures VII Workshop, Key West Florida, July 1993

[9] Bayard, H, and Prelle, M.,,, "Evolvable Systems Initiatives" in Proceedings of AFCEA Symposium, on New Directions in Software Acquisition, MITRE Corporation, Bedford Massachusetts, November 1993

[10] Bersoff, E., et al., "A New Look at the C3I Software Lifecycle", Signal Magazine, pages 85-93, AFCEA, Fairfax Virginia, April 1987

[11] Best, L., "If They Built Buildings the Way They Build Software", white paper, in AMS Special Topics, American Management Systems, Fairfax Virginia, January 1991

[12] Capretz, L., and Lee, P., "Classification of Object-Oriented Development Methodologies", in Proceedings of the Sixth Brazilian Symposium on Software, Engineering, Granado/RS, Brazil, November 1992

[13] Chidamber, S., and Kemerer, C., "A Metrics Suite for Object- Oriented Design", MIT Center for Information Systems Research Working Paper #249,,MIT Sloan School, Cambridge Massachusetts, July 1993

[14] Cohen, S., et al., "Application of Feature Oriented Domain Analysis (FODA) to the Army Movement Control Domain", technical report, CMU/SEI TR-91-TR-28, Carnegie Mellon University, Software Engineering Institute, Pittsburgh Pennsylvania, June 1992

[15] Coad, P., "Analysis and Design: New Advances in Object-Oriented Analysis", in The International OOP Directory, SIGS Publications Inc, 1992

[16] Cook, S. "The Three Ages of Objects", in FIRST CLASS, Vol 3, No 3, Object Management Group, Boulder Colorado, September 1993

[17] Druffel, L., et al., , "Air Force 1993 Science Advisory Board Summer Study on Information Architecture" in Proceedings of AFCEA Symposium, on New Directions in Software Acquisition, MITRE Corporation, Bedford Massachusetts, November 1993

[18] Fox, J., "System Management with SNAP - Architecture Overview", in System Management Template internal technical paper, TEMPLATE Software, Herndon, Virginia, January 1993

[19] Feltham, P, and Dachuk, J., "Systems Productivity: The Impact of Object Orientation", The Farringdon Forum Club, London, 1992

[20] Fleisher, J., Mowbray, T., "Integrating Tools and Data Sources with the DISCUS Framework", technical report for DISCUS Working Group, Programmers Tutorial, MITRE Corporation, McLean Virginia, August 1993

[21] Gaines, B.R., "Manufacturing in the Knowledge Economy", Proceedings of the International Conference on Object-Oriented Manufacturing Systems, pages 19-36, University of Calgary, Calgary, Alberta, Canada, May 1993

[22] Garlan, D., Shaw, M., Okasaki, C., Scott C., Swonger, R., "Experience with a Course on Architectures for Software Systems", Technical Report CMU/SEI-92-TR-17, SEI, Carnegie Mellon University, Pittsburgh Pennsylvania, August 1992

[23] Gelernter, D. "The Metamorphosis of Information Management", pages 66-73, Scientific American, August 1989

[24] Gielingh, W., "Requirements for the Development of Layered Information Models", Proceedings of the First International Conference on Enterprise Intrgration Modeling, pages 269-277, MIT Press Cambridge Massachusetts, 1992

[25] Griswold, W., "An Architecture and Models for a Meaning-Preserving Program Restructuring Tool ", in Proceedings of the ARO/AFOSR/ONR Workshop on Increasing the Practical Impact of Formal Methods for Computer-Aided Software Development, pages pages 25-27, US Naval Post Graduate School, Monterey California, October 1993

[26] Hayes-Roth, F, et al., "Domain Specific Software Architectures for Distributed Intelligent Control and Communications", in Special Report CMU/SEI-92-SR-9, pages 27-62. Software Engineering Institute (SEI), Carnegie Mellon University, Pittsburgh, Pennsylvania, June 1992

[27] Holibaugh, R., "Joint Integrated Avionics Working Group (JIAWG) Object-Oriented Domain Analysis Method (JODA), in Special Report CMU/SEI-92-SR-3, Software Engineering Institute (SEI), Carnegie Mellon University, Pittsburgh, Pennsylvania, June 1992

[28] Horowitz, B., "The Importance of Architecture in DoD Software", Technical Paper, M91-35, The MITRE Corporation, Bedford, Massachusetts, July 1991

[29] Hufnagel, S, Harbison, K, "Scenario -Based Engineering Process: Computer-Aided Software Engineering(CASE) Tool Specification", draft paper in Proceedings of the ARPA Domain Specific Software Architectures VII Workshop, Key West Florida, July 1993

[30] Hufnagel, S, Harbison, K, and Hammons, C, "Seamless Scenario Driven Object-Oriented Approach: Methodology Notation and CASE Tool Integration for OOPSLA-93", draft paper in Proceedings of the ARPA Domain Specific Software Architectures VII Workshop, Key West Florida, July 1993

[31] Jones, A., "The Maturing of Software Architecture", keynote presentation at 1993 Software Engineering Symposium, Software Engineering Institute, (SEI) Carnegie Mellon University, Pittsburgh Pennsylvania, August 1993

[32] Kazman, R., Bass, L., Abowd, G., Webb, M., "Analyzing the Properties of User Interface Software Architetures", draft paper, SEI, Carnegie Mellon University, Pittsburgh Pennsylvania, August 1993

[33] Kosanke, K., "Computer Integrated Manufacturing Open System Architecture (CIM-OSA)" in "Enterprise Integration Modeling", Proceedings of the First International Conference, pages 179-188, MIT Press Cambridge Massachusetts, 1992

[34] Lane, T., "A Design Space and Design Rules for User Interface Software Architecture", Technical Report CMU/SEI-90-TR-22, SEI, Carnegie Mellon University, Pittsburgh Pennsylvania, November 1990

[35] Lane, T., "Studying Software Architecture Through Design Spaces and Rules", Technical Report CMU/SEI-90-TR-18, SEI, Carnegie Mellon University, Pittsburgh Pennsylvania, Nov 1990

[36] Leary, J., "Six Views of Any Information Architecture", technical presentation, Martin Marietta Corporation, Information Systems Group (ISG), Chantilly, Virginia, May 1990

[37] Lock, E., and Sherr, D., "Reengineering Principles for Information System Evolution", presentation at CASE WORLD, Boston Massachusetts, October 1993

[38] Lockman, A. and Salasin J., "A Procedure and Tools for Transition Engineering", ACM SIGSOFT 90, Fourth Annual Symposium on Software Development Environments, Irvine California, December 1990

[39] Lorin, H., "Objects, I-CASE, and Architectures", in FIRST CLASS, Vol 3, No 3, Object Management Group(OMG), Framingham Massachusets, September 1993

[40] Maxim, B., et al., "Prototyping Knowledge-based Design Systems in an Object-Oriented Environment", Proceedings of International Conference on Object-Oriented Manufacturing Systems, pages 55-59, University of Calgary, Calgary, Alberta, Canada, May 1993

[41] Mettala, E., and Graham, M., "The Domain Specific Software Architecture (DSSA) Program", in Special Report CMU/SEI-92-SR-9, pages 1-7. Software Engineering Institute (SEI), Carnegie Mellon University (CMU), Pittsburgh, Pennsylvania, June 1992

[42] Mittermeir, R., and Kinzl, K., "Intra-Object Schemas to Enhance the Evolution of Software Objects", in Proceedings of the ARO/AFOSR/ONR Workshop on Increasing the Practical Impact of Formal Methods for Computer-Aided Software Development", pages 12-14, US Naval Post Graduate School, Monterey California, October 1993

[43] O'Connor, J., "Introducing Systematic Reuse to the Command and Control Systems Division of Rockwell International", technical report, SPC-92020, (DTIC: AD-A252-271) Software Productivity Consortium (SPC), May 1992

[44] Perry, D., and Wolf, A., "Software Architecture", technical paper, ATT Bell Labs, Murray Hill New Jersey,, January 1991

[45] Perry, J., and Shaw, M., "The Role of Domain Independence in Promoting Software Reuse: Architectural Analysis of Systems", in Position Papers of the Reuse in Practice Workshop, SEI, Carnegie Mellon University, Pittsburgh PA, July 1989

[46] Salasin, J. and Waugh, D. , "An Approach to Analyzing Non-Functional Aspects During System Definition", in Proceedings of the ARPA Domain Specific Software Architectures VII Workshop, Key West Florida, July 1993

[47] Salasin, J., and Waugh, D. "The Design Record: Keystone of Software Engineering", annotated presentation in Proceedings of the 3rd Reverse Engineering Forum, section 14, Northeastern University, Burlington Massachussets, September 1992

[48] Salasin, J., and Waugh, D., "Analysis of Critical Non-Functional Factors of Systems", Proceedings of the ARO/AFOSR/ONR Workshop on Increasing the Practical Impact of Formal Methods for Computer-Aided Software Development", pages 64-66, US Naval Post Graduate School, Monterey California, October 1993

[49] Saunders, T., "Architectures and Standards" in Proceedings of AFCEA Symposium, on New Directions in Software Acquisition, MITRE Corporation, Bedford Massachusetts, November 1993

[50] Saunders, T., Horowitz, B., and Mleziva, M., "A New Process for Acquiring Software Architecture", Technical Paper M92B0000126, The MITRE Corporation, Bedford, Massachusetts, November 1992

[51] Shaw, A., "State-Based Specifications In-the-Large", Proceedings of the ARO/AFOSR/ONR Workshop on Increasing the Practical Impact of Formal Methods for Computer-Aided Software Development", pages 47-50, US Naval Post Graduate School, Monterey California, October 1993

[52] Shaw, M., "Larger Scale Systems Require Higher Level Abstractions", Proceedings of the Fifth International Workshop on Software Specification and Design, ACM, New York, NY, May 1989

[53] Shaw, M., "Software Architecture", tutorial notes at 5th International Conference on Software Engineering (ICSE-15), IEEE, Baltimore, Maryland, May 1993

[54] Shelton, R., "Object-Oriented Business Engineering", in FIRST CLASS, Vol 3, No 3, Object Management Group, Boulder Colorado, September 1993

[55] Smith, D., "Toward Practical Applications of Software Synthesis", Proceedings of the ARO/AFOSR/ONR Workshop on Increasing the Practical Impact of Formal Methods for Computer-Aided Software Development", pages 67-69, US Naval Post Graduate School, Monterey California, October 1993

[56] Soley, R., (ed), "Object Management Architecture Guide", Rev 2.0, Object Management Group(OMG), Framingham Massachusets, September 1992

[57] Soley, R., "Using Object Technology to Integrate Distributed Applications" in "Enterprise Integration Modeling", Proceedings of the First International Conference, pages 445-454, MIT Press Cambridge Massachusetts, 1992

[58] Srinivas, Y., and Smith, D.R., "A Theoretical Basis for Software Evolution", in Proceedings of the ARO/AFOSR/ONR Workshop on Increasing the Practical Impact of Formal Methods for Computer-Aided Software Development", pages 15-17, US Naval Post Graduate School, Monterey California, October 1993

[59] Tracz, W., Shafer, S., and Coglianese, L. "DSSA_Avionics Domain Generation Environment (ADAGE)", ADAGE-IBM-93-05, in Proceedings of the ARPA Domain Specific Software Architectures VII Workshop, Key West Florida, July 1993

[60] Uenohara, M., , "Flexible Automation in Japanese Electronics Industry", Proceedings of Japan - USA Symposium on Flexible Automation, pages 5-12, Osaka, Japan, 1986

[61] Urban, M., et al., "Command and Control (C2) Information Architecture Concept Overview", draft paper, for ADPA Symposium on Ballistic Missile Defense Command, Control, and Communications, Colorado Springs, Colorado, March 1994

[62] Wartik, S., and Prieto-Diaz, R., "Criteria for Comparing Reuse-Oriented Domain Analysis Approaches", pages 403-431 in International Journal for Software Engineering and Knowledge Engineering, Vol. 2, No. 3., 1992

[63] Withey, J., "Implementing Model Based Software Engineering in your Organization: An Approach to Domain Engineering", draft technical report, pages 23-29, Software Engineering Institute, Carnegie Mellon University, Pittsburgh Pennsylvania, November 1993

[64] Zachman, J., and Pyryamybida, S., "IDEF Framework", presentation, version 1.2, Zachman & Associates, May 1990

[65] Zachman, J. "Information Architecture", pages 87-113, IBM Systems Journal, Vol 26, No 1, IBM Corporation, Armonk New York, 1987

A Framework for Automated Reengineering of Complex Computer Systems

Lonnie R. Welch [*,†]
Antonio L. Samuel [‡]
Michael W. Masters
Robert D. Harrison
Alexander D. Stoyenko
Joe Caruso [§]

Abstract

The financial pressure to meet the need for change in complex systems through evolution rather than through revolution has spawned the discipline of reengineering. One driving factor of reengineering is that it is increasingly becoming the case that enhanced requirements placed on complex systems are overstressing the processing resources of the systems. Thus, the distribution of processing load over highly parallel and distributed hardware architectures is being explored as part of the reengineering process. Existing complex systems were developed originally to exploit small scale concurrency in programming paradigms that support little or no expression of concurrent execution. Therefore, several difficult tasks must be accomplished to reverse engineer, transform and restructure systems so that they exploit significantly increased amounts of concurrency. In this paper we present metrics for capturing features of complex systems needed for a transformation approach for enhancing concurrency. The metrics not only capture systems' features necessary for concurrency analysis, but also are independent of any programming language, operating system or hardware architecture. Using the metrics, we define an approach for transforming complex systems that exploit modest amounts of concurrency into systems that utilize large scale concurrency. The approach includes the aggregation of concurrency information across levels of abstraction hierarchies to enable concurrency analysis at varying degrees of granularity, ranging from the statement level, to the package instance level, to the level of federated systems. In conjunction with concurrency enhancement for the accommodation of enhanced requirements, our reengineering approach also employs software component layering and reuse to reduce the costs of design, implementation, testing, verification, and maintenance.

1 Introduction

A complex system has many characteristics, including performance, timeliness, availability, dependability, safety and security. Furthermore, such a system typically performs many related functions concurrently, interacts with the environment and many human operators and/or clients simultaneously, consists of many interconnected processing elements, contains many millions or tens of millions of lines of code, takes years to develop from first concept formulation to final deployment, and has development costs of many tens or hundreds of millions of dollars.

Complex systems generally address nontransient requirements that simply cannot be addressed with simpler solutions. Thus they tend to be characterized by long life cycles, often spanning decades. During such extended life cycles, change is inevitable in many dimensions: operational environment, system requirements, technology base, etc. Because of the time and cost of development of complex systems, and because of the infrastructure needed for their development and continued support once deployed, infrastructure which includes highly trained personnel, hardware and support tools, documentation, test procedures, and many other components, there is enormous financial pressure to meet the need for change through evolution rather than revolution.

This need has spawned the discipline of reengineering, the systematic application of methodology and tools to managing the evolutionary transformation of existing complex systems to encompass new or altered requirements and to transport such systems into new environments and onto new technology bases.

[*] Welch and Stoyenko are with The Real-Time Computing Laboratory, Department of Computer and Information Science, Institute for Integrated Systems Research, New Jersey Institute of Technology, Newark, NJ 07102, e-mail: welch@vienna.njit.edu, phone: 201-596-5683, fax: 201-596-5777.

[†] This work is supported in part by The U.S. NSWC (N60921-93-M-1912), by the U.S. ONR (N00014-92-J-1367), by AT&T (UEDP-91-134), and by the State of New Jersey (SBR-421290).

[‡] Samuel, Masters and Harrison are with The Naval Surface Warfare Center, Dahlgren Division, Dahlgren, VA.

[§] Caruso is with Computer Sciences Corporation, Dahlgren, VA.

While reengineering holds the promise of increased efficiency in dealing with, and managing change in, large, complex systems, it is not, and probably can never be, a panacea. The act of changing (rather than rebuilding) a complex system inevitably introduces something akin to entropy into a well ordered system. The accumulation of disorder can be minimized to some degree by careful choice of original design and by advancing and fully exploiting reengineering technology. In particular, the current trend toward so called open system designs is a de facto recognition of the inevitability and cost of change.

It should be added that because of the multicomponent nature of complex systems, it is often the case that old components must coexist harmoniously with new components in a mature complex system. Thus, forward engineering of complex system components may occur simultaneously with reengineering of other components. Reengineering, along with open system design, may properly be viewed as available techniques for optimizing the resources required to extend the life cycles of complex systems.

It is increasingly becoming the case that the increased requirements placed on complex systems are overstressing the processing resources of the systems. Thus, implementations that exploit highly parallel and distributed hardware are being developed. While complex systems of the previous generation employed some parallel processing, they typically used on the order of twenty-five processors. Functionality enhancements in modern complex systems may need more than one-thousand processors Since complex systems were developed for small scale parallelism in programming paradigms that supported little or no expression of parallelism, the exploitation of the tremendously increased parallelism is a challenge that must be addressed.

In conjunction with concurrency enhancement for the accommodation of enhanced requirements, reengineering should also consider the use of modern software engineering principles to reduce the costs of design, implementation, testing, verification, and maintenance. Layering of software components is one technique that addresses these concerns. When a system is constructed by layering, the benefits include encapsulation and information hiding (i.e., loose coupling of software components) [6], abstraction (highly cohesive modules), ease of understandability and simplification of analyses for concurrency [12, 14, 9], timing properties [12, 10], dependability [2] and security. The reuse of software components during system implementation and reimplementation is another technique that addresses the aforementioned concerns. When previously engineered and validated components are reused, the elapsed time from the initial phases of reengineering until system deployment can be reduced significantly. Fortunately, modern programming languages provide constructs which enable software to be implemented by layering components and by reuse. For example, Ada provides the generic package, which can be used to implement abstract data type modules which are parameterized by types and operations. Similarly, C++ allows the definition of generic class templates, which can be instantiated with type and operation parameters to obtain abstract data objects. The effective use of such language constructs should be considered during reengineering.

We have developed an automated reengineering framework that considers concurrency as well as layering and reuse of software (see Figure 1). To reengineer a system so that the resulting system has enhanced concurrency and is implemented using modern software engineering principles, it is necessary first to reverse engineer the existing system. That is, it is necessary to capture in an intermediate representation (IR) the syntactic and semantic attributes and interrelationships of the current system's software, hardware, and humanware (e.g., radar operator or other tactical operators, manual entry of key to perform secure operations). Using the IR, concurrency metrics are extracted. After capturing the syntactic and semantic aspects, as well as the concurrency metrics of the system to be reengineered, redesign (transformation) and reimplementation (translation) of the system are performed. Redesign (or design transformation) performs a restructuring of the system by formulating abstractions and exposing concurrency, while considering serializability and deadlock. The new system design is translated into programs in one or more target languages. The generation of code from a design involves the definition of packages/templates/classes to implement the design abstractions (employing genericity, reuse, encapsulation, and information hiding), the definition of data structures (using types exported by, and/or objects encapsulated within, package/template/class instances), and the retrieval and reuse of previously implemented components that are stored in a software repository. Given the complete collection of software components, they are clustered/partitioned and assigned to a parallel and/or distributed hardware platform in a manner that allows effective utilization of concurrency and that also allows compliance with timing, dependability, security and other constraints.

The work described in this manuscript is inspired by both the AEGIS and the HIPER-D projects, which provided the impetus to consider the problems addressed and which are among the potential recipients of the research results. AEGIS [4, 5] is a complex system engineered to protect a fleet of ships from subsonic and supersonic threats such as manned aircraft, air-to-surface missiles, surface-to-surface missiles, and undersea missiles. The requirements of the system include fast reaction (instant response to targets in specific sectors that match particular patterns with respect to speed, course and altitude), accuracy, resilience to faults and to overloads, and security. For example, AEGIS may detect a sea-skimming missile at a distance of 15 miles and traveling at Mach 1. To avoid loss of lives and equipment due to impact of the missile, the detection, classification, tracking, assignment of threat priority, and firing of a counter-missile

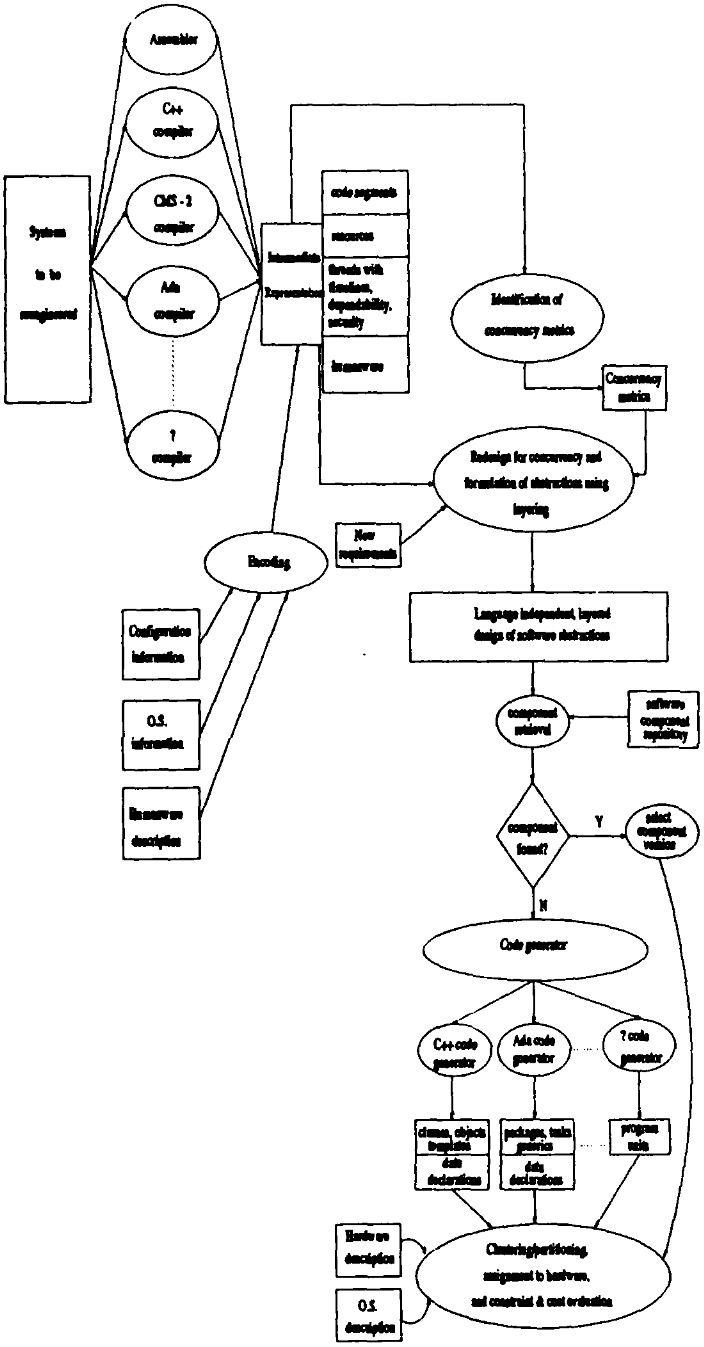

Assembler
C++ compiler
CMS-2 compiler
Ada compiler
? compiler
System to be reengineered
Intermediate Representation
code segments
resources
threads with timeliness, dependability, security
hardware
Identification of concurrency metrics
Concurrency metrics
Redesign for concurrency and formulation of abstractions using layering
New requirements
Encoding
Configuration information
O.S. information
Hardware description
Language independent, layered design of software abstractions
component retrieval
software component repository
component found?
Y
N
select component version
Code generator
C++ code generator
Ada code generator
? code generator
classes, objects templates
data declarations
packages, tasks generics
data declarations
program units
Hardware description
O.S. description
Clustering/partitioning, assignment to hardware, and constraint & cost evaluation

Figure 1: Reengineering framework for concurrency enhancement, software component layering and reuse.

must occur within a very stringent deadline [3]. The other motivator of this work, the HIPER-D project, is an ARPA sponsored project that is exploring the use of high-performance distributed (and parallel) computer systems as platforms for complex systems.

The remainder of the paper is organized as follows. Section 2 illustrates typical software, hardware and operating system characteristics of Navy systems for which reengineering is being considered presently. In Section 3, it is shown how contemporary programming languages, operating systems and hardware platforms can be used as targets for a reengineering process that enhances concurrency, produces layered software components and applies software reuse. In Section 4 we discuss an intermediate representation and metrics that capture the relevant attributes of complex systems in a manner that is independent of any programming language, hardware architecture, or operating system. An overview of our methodology for using our IR and metrics to transform complex systems that have minimal software layering and small-scale parallelism into systems with extensive layering of software components and large-scale parallelism is also presented in Section 4. The methodology is based in part on a model of concurrent execution that we have defined called asynchronous remote procedure call (ARPC) [15], which allows concurrency in amounts proportional to the amount of layering in application software. Since parallelism is not the only concern in complex systems, our methodology also considers timeliness, dependability and security.

2 Typical Characteristics of Systems to be Reengineered

Complex Navy systems built more than one decade ago exhibit certain trends, reflecting the previous state-of-the-art in the areas of software, hardware, and operating system technology. In this Section we examine the past trends in each of these areas, with an emphasis on characterizing aspects of such systems in paradigm-independent fashions.

A typical software paradigm observed during reverse engineering is one in which procedures are combined to form a module (see Figure 2). A module may contain exported operations (callable from without the module) and internal operations (callable only from within the module). In addition to containing operations, each module may contain data accessible only by its operations. Each exported operation of a module is termed a module entry, and serves as a means of manipulating the module's internal state. A module may contain at most one of each of several kinds of module entries, of which the following are representative. An initialization entry is scheduled when the tactical system is being initialized or when a disabled module is enabled. A message entry is scheduled to accept and process messages from other modules. An error entrance is scheduled when an error condition is detected. A successor entrance is used for any general purpose, nonperiodic task. Both a buffer complete entrance and a channel complete entrance are scheduled in response to user-controlled directions associated with a completed I/O process. A periodic entrance executes functions that must be performed at regular intervals. An important consideration during reengineering is that assembly language appears frequently in the code bodies of entries, since module development languages (such as CMS-2) provide no easy way to control hardware device accesses. In addition to the user-defined modules, a system may contain global data (tables) that are accessible by the operations of any module. There is also a set of common service routines (CSRs) and executive service requests (ESRs), callable from any operation.

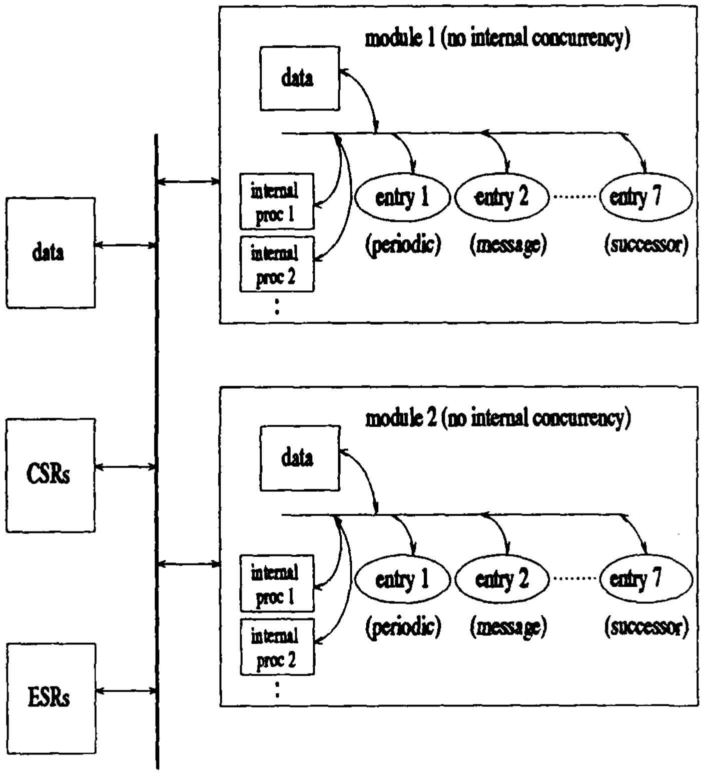


Figure 2: A typical model of previous generation software.

In addition to functionality, modules exhibit other characteristics. *Concurrency* and timing properties are stated by defining periodic module entries. Each of these executes once per period and may have a deadline by which any particular execution of the entry must complete. At most one entry may be active within a module at any time (i.e., modules are monitors). A complex system is composed of many independent activities (or threads of control), which are implemented via calls to module entries, ESRs and CSRs, and which may directly access global tables. Due to the lack of layering, all modules, tables, CSRs and ESRs are visible to each activity. Complex systems must adapt to times of overload by shedding less critical tasks in favor of more critical ones. This shedding is termed throttling, and is specified by associating a criticality with each module entry. *Throttling* allows some degree of adaptability, but the system must also be able to function correctly in the presence of hardware faults. This is normally stated implicitly by designing redundancy into the hardware.

In addition to concurrency, it is also necessary to obey various security levels. A module entry or a piece of data may have access restrictions, only permitting users having the appropriate security level, password, or key to use them. Security requirements cannot be stated in the programming paradigm, but are implicitly coded into systems.

Fault tolerance is provided in the previous generation of complex systems by replicating CPUs, memories, and interprocessor communication links. A typical hardware system employs fewer than 10 nodes, each consisting of a few CPUs with private memories, and a shared node memory (see Figure 3). The execution paradigm is usually MIMD. An approach frequently used to tolerate faults is to maintain dual memories, one containing the current values of all data and the other keeping current values of critical data. In the event of a failure of the first memory, the system automatically switches to the second memory. Both memories also contain complete copies of the code assigned to the node. Normally, both CPUs execute the instructions. Each instruction is statically tagged with the CPU which is to execute it, and the instruction fetching mechanism insures that each instruction is executed by the correct CPU. To perform a "hot restart" when a node becomes over-

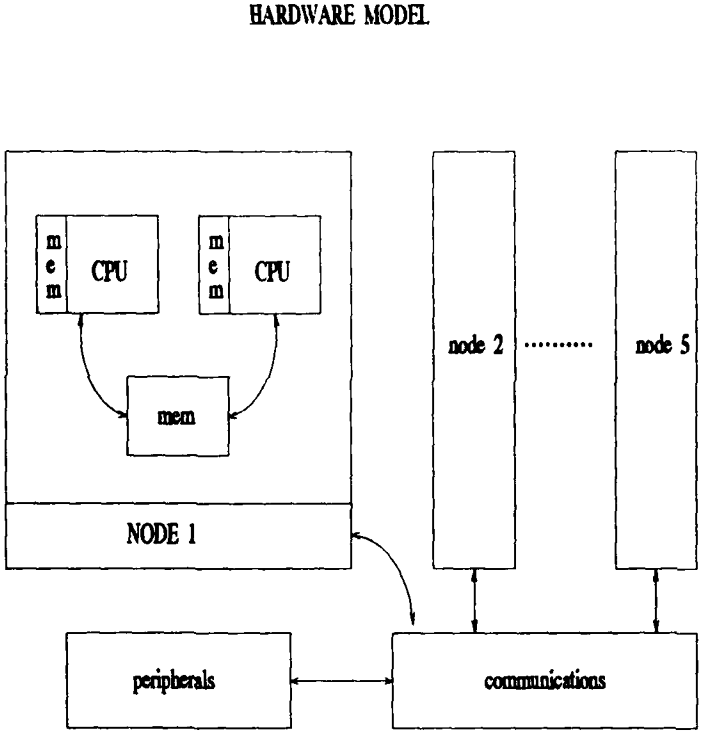


Figure 3: A typical model of previous generation hardware.

loaded, execution switches to the copy of code in the backup memory. In that way, all noncritical data are reset to their initial values. Additionally, a backup node is installed. If one node crashes, its software processes are restarted on the backup node. For security, there are hardware mechanisms (such as keys) that must be switched on in order to execute secure functions.

The execution model used in the previous generation of systems typically relies on an executive that provides features to handle such things as hardware interrupts, memory management, application module scheduling, and fault tolerance. The application programs use services provided by the executive for module (re)initialization, intermodule and intercomputer communication, scheduling of other modules, error processing, input/output channel communications, and periodic scheduling. These systems are almost always non-portable and in most cases are coded in high-level languages and assembly languages available only on a specific class of hardware. The Aegis Tactical Executive System, ATES/43, is such an executive, as described in the following quote from [1]:

> The ATES/43 is an interrupt-driven and table-directed executive system designed to meet a broad range of requirements specified for AEGIS DDG combat system computer programs that run in AN/UYK-43 computers. It responds to all AN/UYK-43 interrupts on a real-time basis, determines the processing to be performed, and performs the processing or passes control of a CPU either to firmware module Fault Tolerant System Reconfiguration Module or a scheduled software module for processing. ATES/43 supports user-controlled rapid recovery and online reconfiguration, and provides flexibility for the design of diverse computers programs.

The executive restricts the execution to one entrance per module at any instant. In other words, only one entrance of a module may be either executing or in the execution queue at any time. A secondary queue is provided to handle this restriction. The user can also specify the CPU on which a module entrance is to execute.

There are several reasons for migrating from the aforementioned paradigm. As the functionality of complex systems increases, it is desirable to increase the concurrency in order to meet the timing requirements. Thus, systems should be portable to different hardware platforms. The frequent use of assembly language to implement entries significantly decreases portability. Furthermore, the lack of user-defined concurrency in system designs makes it difficult to exploit a large parallel processor, and the use of programming constructs like pointers makes the automatic analysis of parallelism troublesome. Also, the flat module hierarchy structure permits the access of global data structures by every procedure, leading to inefficiencies due to synchronization of accesses to such structures. Another deficiency is the lack of usage of modern software engineering concepts like abstract data types and objects, and generics (as can be implemented by Ada packages and C++ classes). The use of such constructs increases layering, improves reusability, and simplifies development, reengineering, timing analysis, and parallelism extraction [6, 15, 8]. The hardware model makes it impossible to achieve the large scale parallelism necessitated by the massive capabilities of modern software systems. Additionally, the ATES-like executive systems are restrictive in terms of portability and also in terms of services provided to the application program. Additionally, there is no clear distinction between services available for interfacing with the hardware and those provided for the application program. In other words, one major portability issue is that the operating system functions are not disjoint from the application run-time type functions. Another issue is the limitation of the memory management functions—the executives often do not provide virtual memory. There is also room for improvement in real-time scheduling techniques. Additionally, the degree of parallelism managed by the systems is very low, and modern parallelism paradigms such as asynchronous remote procedure call [15] are not supported. Another void in the systems is in the run-time support for modern software engineering constructs such as abstract data types and abstract data objects.

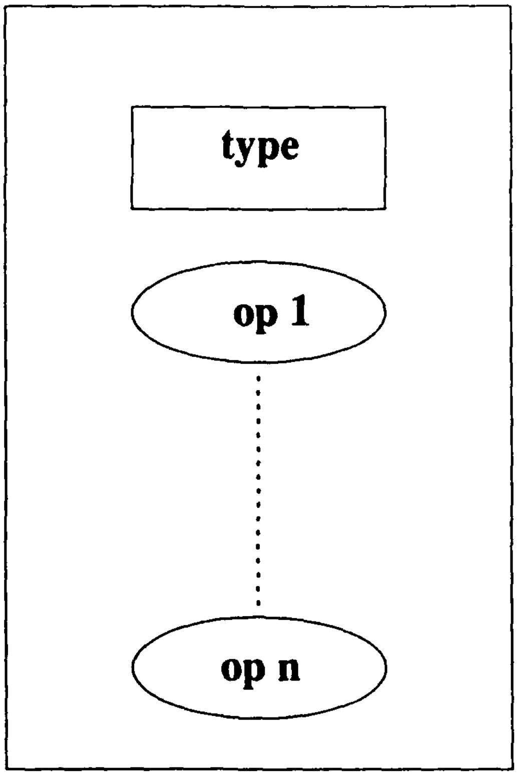


Figure 4: An abstract data type (ADT) module.

3 Desired Characteristics of Reengineered Systems

Many of the shortcomings of "yesterday's" system developmen' paradigms have been overcome by modern paradigms. Layering, loose coupling, reuse [6, 7], high cohesion, layering, encapsulation, and information hiding are facilitated by the proper use of programming constructs such as abstract data types and abstract data objects. The ability to define generic ADTs and ADOs enables the development of parameterized abstractions, resulting in increased reuse and in a high payoff when such a component is produced by reengineering (since the cost of reengineering a component can be amortized over multiple uses of the component). The specification of concurrency can be performed in modern languages such as Ada by defining tasks within ADT or ADO modules. The ability to lexically nest modules reduces the visibility of data structures to only those needing to access them, thus lowering module coupling, simplifying analysis of parallelism, and leading to systems with fewer bugs. The amount of direct (assembly) code is minimized in modern systems, leading to increased portability.

Many modern software engineering paradigms (as supported in Ada, Clu, Modula-2 and RESOLVE) provide techniques supporting implementation of templates. Normally, each template encapsulates an abstract data type (ADT). A typical ADT (as illus-

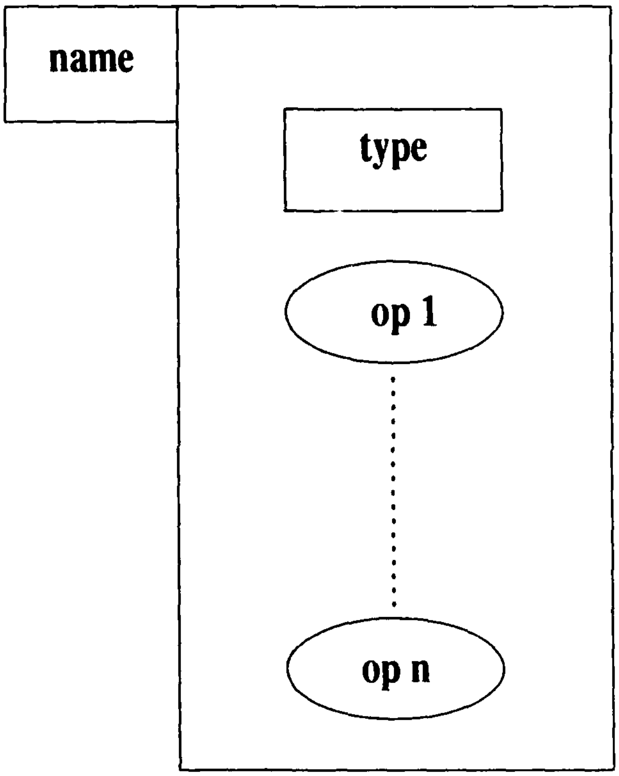


Figure 5: A facility—an abstract data type (ADT) module instance.

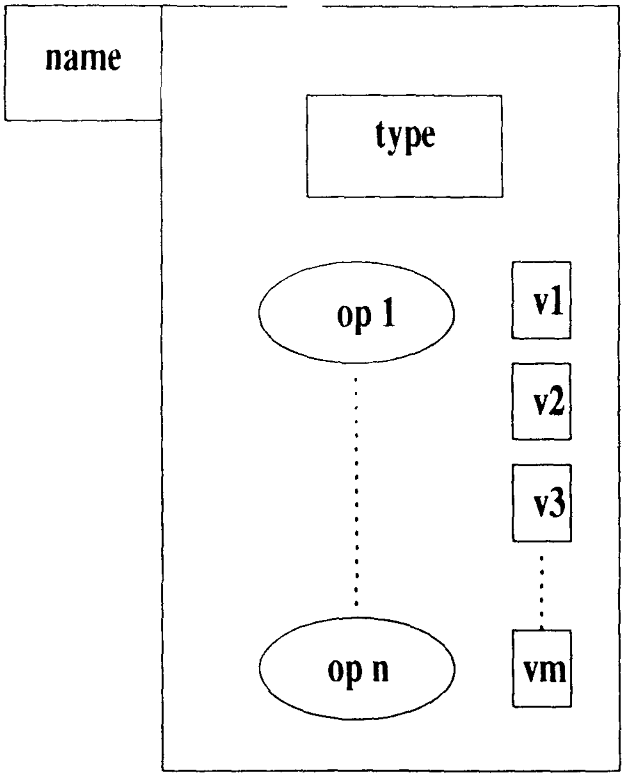


Figure 6: Variables managed by a facility.

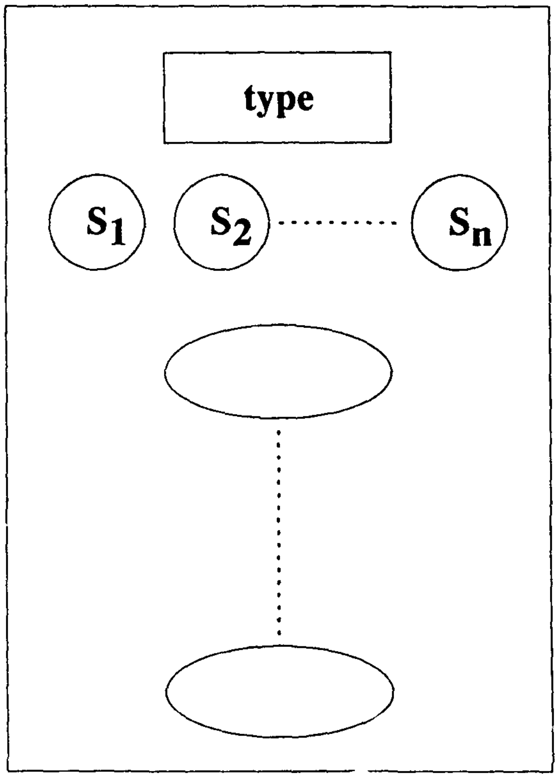


Figure 7: A facility with state.

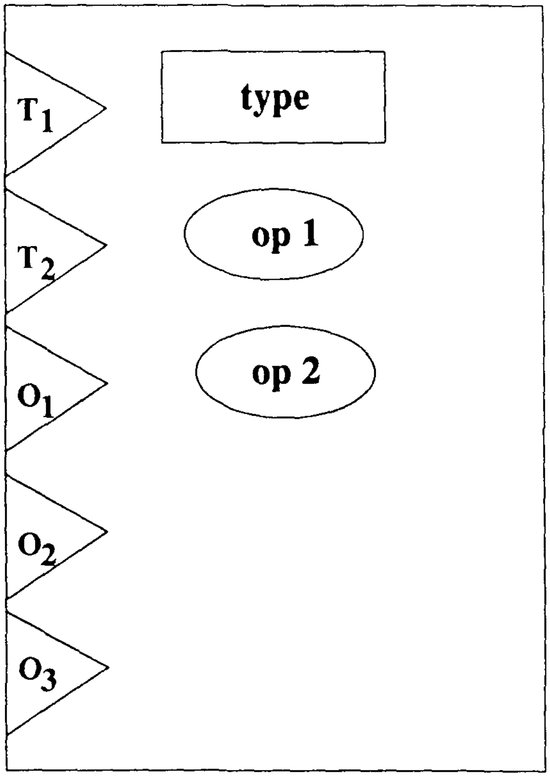


Figure 8: A generic abstract data type (ADT) module.

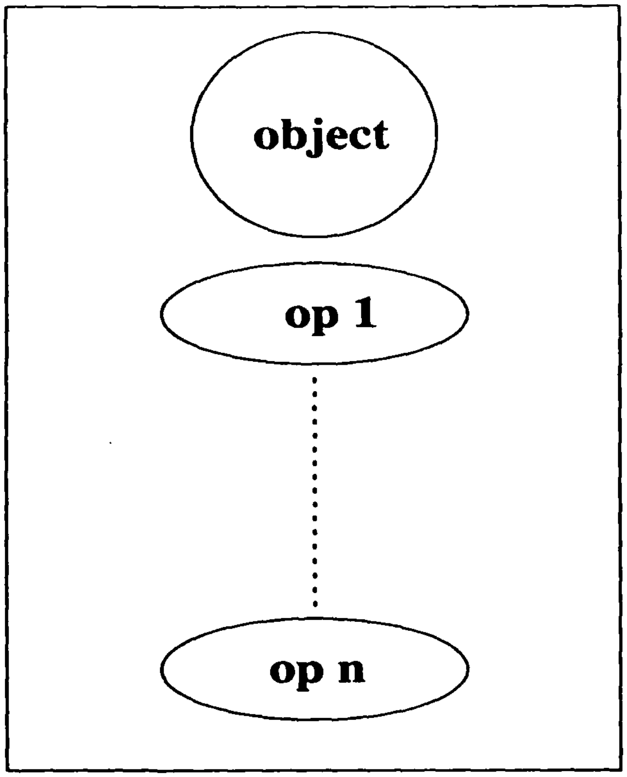


Figure 9: An abstract data object (ADO) module.

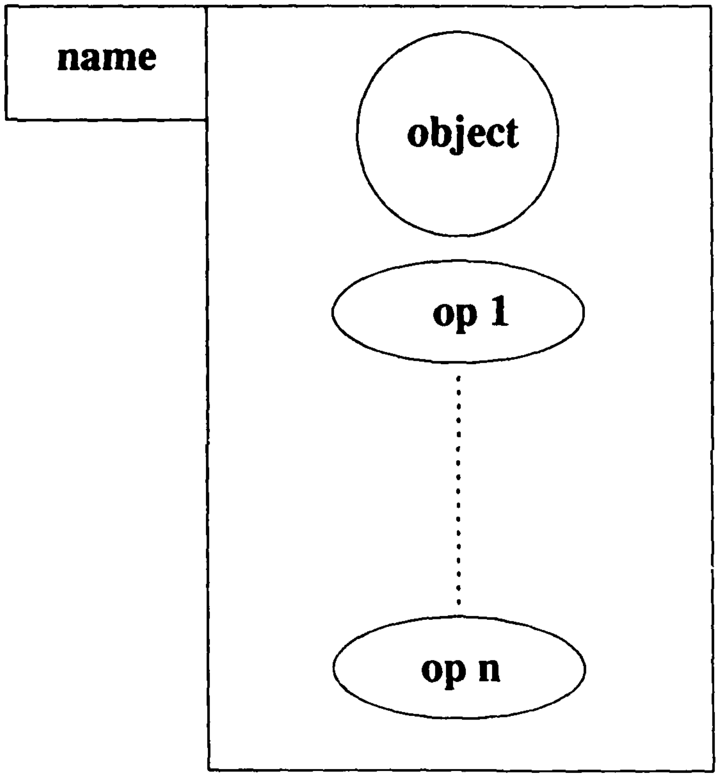


Figure 10: A facility—an abstract data object (ADO) module instance.

trated in Figure 4) exports (1) a type that can be used to declare variables and (2) operations to manipulate variables of the provided type. To use an ADT module, it is instantiated by giving it a name (see Figure 5). In Ada, for example, instantiation is performed with the *use* and *with* clauses. Instantiation creates a module instance, or a facility. With ADTs, instantiation does not create a data value. Instead, a variable must be explicitly declared to be of a type exported by a facility, and may be accessed by calling operations exported by the module. A set of variables managed by a module instance is shown in Figure 6. ADT instances may encapsulate state that is accessible only by the operations of the module, as shown in Figure 7. Tailorable templates can be developed in languages permitting development of generic modules (modules parameterized by types, by operations, or by other modules). A generic module is represented graphically in Figure 8 A generic module is instantiated by fixing its parameters.

A template may also encapsulate an abstract data object (ADO), the unit of reuse in languages such as C++, DEAL, Eiffel, and Smalltalk. An ADO is a special case of an ADT—one that exports no type and that encapsulates state (the value of an object), as illustrated in Figu"es 9 and 10. Thus, ADOs can also be developed in ADT-based languages such as Ada, Clu, Modula-2 and RESOLVE. An ADO is defined using a template module (called a class in most object-oriented languages) which exports a set of op-

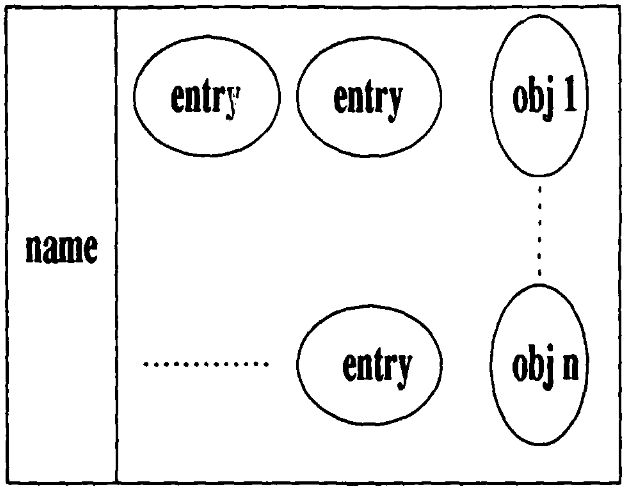


Figure 11: A task type.

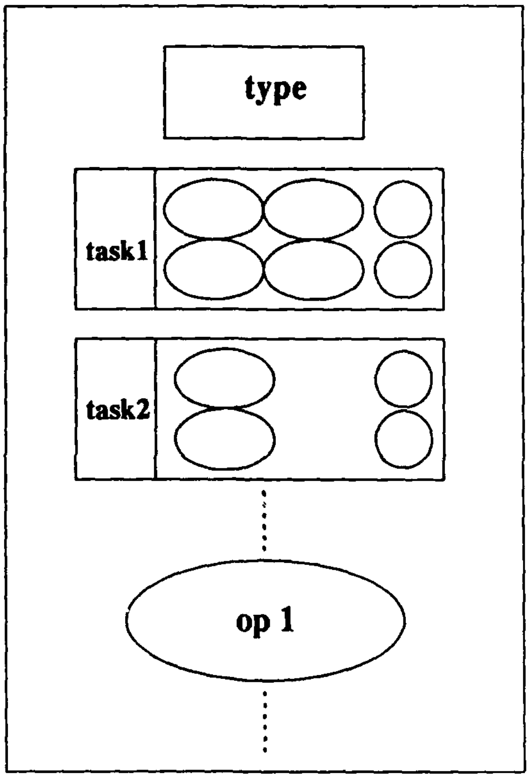


Figure 12: An ADT that exports task types in addition to a data type.

erations, and defines the structure of objects created from the class.

In the Ada programming paradigm, a template may encapsulate tasks, in addition to either exporting a type or encapsulating state. (See Figures 11 and 12. Each task represents a unit of parallelism. When encapsulated in a template, a task becomes active upon the creation of an instance of the template. Tasks proceed independently and may execute concurrently, except when synchronizing (rendezvousing) with other tasks at points called entries. Entries allow tasks to exchange data synchronously, and to provide mutually exclusive access to shared state information, allowing clients to cause synchronous execution of a specific portion of a task body. Exchanging and sharing of information between two tasks can either be done conditionally or unconditionally. Only one client may be active within a task at any instant. Furthermore, a template may export one or more task types, allowing many copies of a task to be obtained from a single task object. To obtain a task from such a template, one declares a variable to be of the appropriate task type and then invokes a task initiation operation on the variable. When such an approach is used, multiple tasks may be active concurrently within an instance. Tasks templates can be used to create *families* of tasks by specifying a discrete range to indicate the indices of the tasks in the family.

In addition to the explicit parallelism available in a task-based system, an abundance of parallelism is available when the asynchronous remote procedure call model of parallel execution is applied to programs constructed from ADT and ADO modules. Facilities (consisting of code and possibly state) are statically assigned to PEs (multiple facilities may reside on the same PE). Facilities' operations are invoked by sending call messages between PEs. To hide the latency of a remote call, an operation is permitted to continue execution until it attempts to access a "locked" variable (this model of parallel execution is termed asynchronous remote procedure call, or ARPC [15]). A variable is automatically locked when it is passed as a parameter to a call and is unlocked upon return of the call. An operation attempting to access a locked variable must wait for a remote call to return before retrying the access. ARPC can achieve parallel execution at multiple levels in the abstraction hierarchy. Thus, potential parallelism within a program increases with the number of levels of abstraction, and the model encourages development of highly cohesive, loosely coupled modules.

With the increase in potential concurrency comes the added complexity of exploiting the concurrency. Software components must be partitioned/clustered according to some binding relationships (such as communication, concurrency or shared data access), and the clusters assigned to processors in a way that causes efficient utilization of hardware resources and simultaneously obeys system constraints [11, 12, 13, 9].

The explicit concurrency available in task-based systems, and the implicit concurrency available via ARPC can be exploited on modern hardware platforms, which are characterized by a large number of interconnected processing elements (PEs). For example, the Intel Paragon computer contains thousands of computing nodes, running according to the MIMD paradigm, and interconnected by a 2-dimensional mesh network. Its computing nodes contain multiple CPUs that share memory. Although there is shared memory within a node, there is no globally shared memory. Additionally, one CPU per node is dedicated to communication processing and the others are general-purpose processors.

The modern execution model provides greater flexibility than the older execution model. Additionally, it provides the same services that the old model provided, such as features to handle hardware interrupts, memory management, application module scheduling, and fault tolerance. Additionally, the system provides real-time capabilities, which provide the application programs with a guarantee of resource availability for critical elements. Furthermore, the execution model is capable of supporting a wide variety of computer architectures, including uniprocessors, loosely and tightly coupled distributed architectures, and a heterogeneous mix of systems. Also, the model aids portability and scalability. Two execution models that partially satisfy this group of requirements are the Mach Operating and the OSF/1 operating system. These operating systems provide the portability and scalability that is desired, as well as the capability to operate on many different architectures. These systems emphasize a layered approach of providing services to the operating system. The operating systems are designed as highly optimized microkernels in which functionality can be added to create a complete operating system.

4 Transformation of Systems

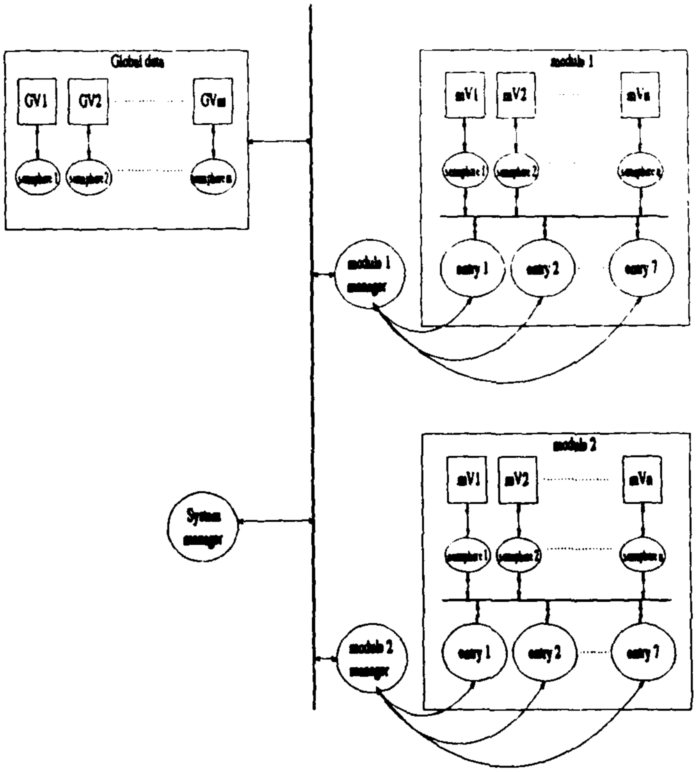


Figure 13: A transformed system.

In our approach to reengineering, we increase concurrency while maintaining consistency and serializability, and preventing deadlock. Figure 13 shows a system following transformation.

The software modules described in Section 2 and depicted in Figure 2 can be described using generic packages encapsulating ADTs, ADOs and task types. There at least two approachs which can be used to model the module structures. In the first approach, a module is represented by a single task whose task entries correspond to the original module entries. With this approach, obeisance is given to the semantic constraint which dictates that a single entry may be active within a module at any instant. However, due to subtleties in the Ada task rendezvous construct, the periodic module entry cannot be accurately modeled. Furthermore, this approach does not take advantage of intramodule concurrency. In the second approach, a module is represented by a packaage containing one task for each module entry. Thus, the amount of concurrency allowed is increased, but consistency of module state is not guaranteed.

To maintain consistent values for module level state variables as well as global variables, semaphores are used to insure single access to each of them. With the addition of the semaphores into the system, the potential for deadlock is introduced. To prevent deadlock, module managers are used to control accesses to module entries. For example, if two entries *a* and *b* of the same module could enter a circular wait state, the module manager may not allow *a* and *b* to execute concurrently. It may not always be necessary to be this strict with intramodule concurrency, and the module manager could make optimizations where appropriate to increase concurrency. For example, segments of entry *a* could perhaps be executed concurrently with segments of entry *b*. To prevent deadlock when accessing global variables, a system manager is used to control concurrency among modules, ESRs and CSRs to avoid circular waiting conditions.

It is the task of reengineering to:

- insert calls to semaphore operations before and after shared variable access;
- partition global and module variables into segments that can be accessed concurrently;
- generate module managers that maximize *intramodule* concurrency, subject to constraints such as liveness and serializability;
- generate system managers that maximize *intermodule* concurrency, subject to constraints such as liveness and serializability;
- partition module entries into segments that can execute concurrently; and
- replicate modules and procedures to maximize concurrency.

Obviously, software reengineering of the nature just described requires more than a translation of source code written in one language into source code written in another language. Instead, reengineering is a transformation of a software and hardware system from one paradigm into another. To perform such a transformation, it is not enough to have a syntactic knowledge of the system being reengineered; a semantic knowledge must also be obtained. Transformation of a system implementation from one paradigm to another involves the initial task of reverse engineering, that is, capturing the intermediate representation, and collecting the metrics necessary for assessment and optimization. Our language-independent representation allows the redesign of a system with the goals of employing modern software engineering principles, exploiting parallelism safely, and conforming to constraints related to timing, dependability and security. In this Section we discuss the metrics that must be collected for concurrency analysis and show how the metrics are used to exploit parallelism.

4.1 Characterizing Complex Systems

When reasoning about parallelism, it is necessary first to analyze parallelism at a low level of granularity, i.e., at the statement or the instruction level. To obtain parallelism information units of larger granularity, one successively synthesizes parallelism information from lower levels. Our representation allows parallelism to be exploited at the following levels:

1. among statements within an operation;
2. among operations within an instance;
3. among operations of different instances;
4. within a task entry;
5. among entries within a task;
6. among tasks and operations within a package instance;
7. among package instances;
8. among clusters of package instances; and
9. among systems.

We have defined language-independent intermediate forms (or metrics) for capturing the features of complex systems that are essential for reasoning about parallelism, encapsulation, information hiding, reuse, real-time, fault tolerance, and security. The intermediate representations presented in this Section can be derived by a combination of compile-time analysis tools and run-time monitoring tools.

Dependence graphs represent program statements as nodes and use directed edges to denote statement ordering implied by the dependences in a source program. Different kinds of ordering requirements are represented in different dependence graphs. The classical data dependence graph (DDG) and control dependence graph (CDG) are used, in addition to the facility dependence graph (FDG) which was invented for the purpose of clone analysis [14].

A directed acyclic graph (a DAG) is used to show the call relationships among the facilities of a program and to represent inter-module parallelism. A program is modeled by a DAG, G = (V, E), where:

1. a vertex v in V denotes the operations of a facility, f(v);
2. an edge (x, y) in E indicates that the code of facility f(x) calls some operation(s) provided by facility f(y).

When assessing the timing properties of distributed/parallel, periodic, time-constrained processes, it is important to capture several metrics [9]. Obviously, absolute timing constraints such as periods and deadlines must be determined. Additionally, relative timing constraints can be identified. For scheduling and assignment/allocation optimization, it is useful to represent the system as a set of (autonomous) activities, and their constituents, which we call beads.

The module call DAG is used to represent the fault tolerance and security properties of systems. The degree of redundancy is given for nodes and edges of the graph, indicating the number of redundant software modules and intermodule access paths, respectively. Additionally, the reliability attributes of modules and interconnections are specified as real-numbers between zero and one. Security is represented as classification levels, which may be associated with DAG edges and nodes. When associated with a node, a security level indicates the classification level required to execute the code represented by the node. Classification levels on edges indicate the minimum degree of security that must be associated with any transmission of the call parameters represented by the edge.

4.2 Concurrency Analysis

Given the statement dependence graphs and the module call DAG, concurrency information is obtained first at the statement level. That information is then aggregated to the procedure level, then to the module level, and so on, until concurrency information is obtained for the desired level of granularity.

Extraction of ARPC parallelism is achieved by first augmenting the call DAG to indicate two kinds of parallelism. An edge drawn using parallel lines indicates parallelism between a client and an exporter. The DAG is also used to indicate which co-exporters of a client can execute in parallel with each other; this type of parallelism is denoted by labeling edges with sets of facilities. To enable one to determine whether two arbitrary facilities can execute in parallel, we have defined theorems that state how the parallelism information contained in the DAG can be propagated among nodes of the graph (for details see [11, 13]).

Frequently, an ADT module is needed simultaneously by multiple clients, thus causing contention. To decrease queueing delays to execute module operations, cloning of module instances (facilities) is used in conjunction with the ARPC paradigm [14]. The detection of statements that contend for a facility is accomplished by considering the DDG, CDG and FDG in conjunction. An edge in an FDG shows where it is beneficial to clone a facility, assuming that data and control dependences do not prohibit parallelism between the statements involved in the facility dependence.

To detect contention for a facility, the statements of an operation are partitioned into *units*. A unit is a sequence of statements that must execute in order, due to the data dependences among them. (Hence, the statements of a unit cannot contend for a facility, but different units may contend for the same facility.) For any statement S_i in a unit, except the last statement of the unit, S_i must complete execution before S_{i+1} can begin execution. Thus, each unit can utilize only one clone of each of the facilities that it uses. However, if two units can execute in parallel, they may contend for clones. Two units can execute in parallel with each other as long as neither is an ancestor of the other (i.e., there is not a directed path from one to the other). This notion can be used to construct a matrix, P, showing which units can run in parallel. The information in the matrix is used to group units with others, such that each member of a group can run in parallel with all others. Given the groups, the parallelism matrix is used to determine the maximum number of clones that can be used simultaneously among all groups.

After the software components of a complex system have been reengineered, it is necessary to assign them to the nodes of a parallel and/or distributed computer [11, 13, 9]. Due to the large number of software and hardware components in complex systems, the cost of obtaining an optimal assignment of software components to processors is quite high. Thus, heuristics are used for assignment. Furthermore, it is useful to cluster components before assignment in order to reduce the problem size.

We have developed a technique to distribute the facilities of a program over the processing elements (PEs) of distributed memory parallel computers. The technique uses a random neural network (RNN) to assign facilities to PEs with the objectives of enabling maximum parallelism among facilities, and achieving this level of parallelism with the minimum number of PEs possible. We also incorporate minimization of communication costs into the objective function by using a prepass to the neural network. The prepass forms clusters of heavily-communicating software units. Following the formation of clusters, the random neural network assigns clusters to PEs using the objectives of maximum parallelism and PE conservation.

To provide the required dependability and security properties, one must insure that the specification is heeded. To achieve redundancy, redundant copies of a component are assigned to different physical processors. Similarly, redundancy in software component connections is achieved by a careful assignment of components to processors. Reliability is achieved by selecting, for each software component, a hardware component with a sufficiently high degree of reliability. Note that the communication links must also be considered for reliability of messages between software components. Security of message flows can be achieved with either encryption or with dedicated networks for each security class. For code modules, security is attained by either run-time system mechanisms or by dedicated code processors for each security level.

5 Conclusions

We have described a framework for automated reengineering of complex systems. Our intermediate representation for complex systems is not linked with any specific combination of programming, hardware or execution paradigms. Furthermore, it allows the capture of essential system metrics relating to parallelism, timing, fault tolerance, and security. The representation enables the incorporation of object-based software engineering techniques during reengineering. Additionally, we describe how the representation is used to enhance parallelism during the reengineering process, to assess timing properties, to achieve dependability and security, and to optimize software-to-hardware assignments.

We continue to evolve our reengineerin? analysis tools, which drive the evolution of the metrics. Thus, additional metrics for parallelism analysis continue to be identified. We are also beginning to develop reverse engineering tools to capture the metrics for languages such as assembler, CMS-2, and Ada. Furthermore, we plan to develop reengineering tools that transform previous generation complex systems into layered, highly concurrent designs and to feed these designs to backend code generators for languages such as Ada, C++, Smalltalk and Eiffel. Finally, we plan to use parallel processors such as the Intel Paragon to implement the synchronization and deadlock avoidance techniques incorporated in the module and system managers.

References

[1] *AEGIS TACTICAL EXECUTIVE SYSTEM (ATES/43) USER'S MANUAL*, Volume 1—System Design Guide, July 1991, p. 3-1.

[2] T. J. Marlowe, A. D. Stoyenko, S. P. Masticola, and L. R. Welch, "Schedulability-Analyzable Exception Handling for Responsive Languages," *Journal of Real-Time Systems*, to appear.

[3] W. B. Scott, "Navy May Accelerate Missile Defense," *Aviation Week and Space Technology*, pp. 283-284, May 30, 1983.

[4] K. J. Stein, "Aegis Fleet Defense Nearing Sea Test," *Aviation Week and Space Technology*, pp. 32-35, August 13, 1973.

[5] K. J. Stein, "Aegis System Tested Successfully," *Aviation Week and Space Technology*, pp. 36-40, April 7, 1975.

[6] M. Sitaraman, L. R. Welch and D. E. Harms, "On Specification of Reusable Software Components," *The International Journal of Software Engineering and Knowledge Engineering*, volume 3, number 2, 1993.

[7] , R. A. Steigerwald and L. R. Welch, "Reusable Component Retrieval for Real-Time Applications," *Proceedings of the First IEEE Workshop on Real-Time Applications*, May 1993.

[8] A. D. Stoyenko, L. R. Welch, and B. C. Cheng, "Response Time Prediction in Object-Based, Parallel Embedded Systems," to appear in *Euromicro Journal*, 1994, Special Issue on Parallel Processing in Embedded Real-Time Systems.

[9] J. P. C. Verhoosel, L. R. Welch, D. K. Hammer, and A. D. Stoyenko, "Assignment and Pre-Runtime Scheduling of Object-Oriented, Hard Real-Time Parallel Processes Using Bead Partitioning," New Jersey Institute of Technology Technical Report CIS-93-16, December, 1993.

[10] J. P. C. Verhoosel, L. R. Welch, D. K. Hammer, and A. D. Stoyenko, "A Model for Scheduling of Object-Based, Hard Real-Time Parallel Processes," *The Journal of Real-Time Systems*, (submitted).

[11] L. R. Welch, "Assignment of ADT Modules to Processors," *Proceedings of the International Parallel Processing Symposium*, March, 1992.

[12] L. R. Welch, A. D. Stoyenko, T. J. Marlowe, "Modeling Resource Contention among Distributed Periodic Processes," *Fourth IEEE Symposium on Parallel and Distributed Computing* (December 1992).

[13] L. R. Welch, A. D. Stoyenko and S. Chen, "Assignment of ADT Modules with Random Neural Networks," *The Hawaii International Conference on System Sciences*, IEEE, Jan. 1993.

[14] L. R. Welch, "Cloning ADT Modules to Increase Parallelism: Rationale and Techniques," *Fifth IEEE Symposium on Parallel and Distributed Computing*, December 1993.

[15] L. R. Welch, "A Parallel Virtual Machine for Programs Composed of Abstract Data Types", *IEEE Transactions on Computers*, accepted for publication—to appear.

Dynamic (Re)Generation of Software Documentation

W. Lewis Johnson
USC / Information Sciences Institute
4676 Admiralty Way
Marina del Rey, CA 90292-6695
johnson@isi.edu

Abstract

We are developing an authoring tool called I-Doc that will automate the process of generating documentation and user help for software systems. The focus of the tool is on capture of the requirements and design decisions that form the content of software documentation. This information can then be used to generate summaries and explanations of the software on demand. The objective of this research is to provide on-line assistance for software maintainers and other software professionals that can take the place of conventional bulk documents. I-Doc is designed to support the reengineering of software systems, since some of the necessary design information will have to be captured by annotating existing code. Reengineering technology, specifically transformation technology, is employed during the generation process to simplify and reorganize design information when describing software.

1 Introduction

Conventional documentation for software systems has surprisingly little value, given the amount of time and effort spent to create it. This is particularly true for large, mature systems. Such systems typically have voluminous design documents, in which it is difficult to find information relevant to any specific maintenance task. If the documentation is not maintained in lock step with the code, it quickly becomes inaccurate, so maintainers cannot rely upon it. There is an increasing need for alternative technologies that can provide maintainers and users of software systems with the information that they need to operate and maintain those systems.

We are developing a documentation authoring tool that will automate the process of generating documentation and user help for software systems. This tool will result in dramatic improvements in the way documentation is developed, maintained, and used. If the design, requirements, and assumptions underlying code are made explicit, generation of documentation can be substantially automated. This underlying knowledge can be acquired and formalized in a natural, incremental fashion that does not overly burden developers.

Documentation will be generated dynamically, in response to specific requests for user information. When the user requests information, the documentation system determines what information content should be presented. It composes a response by combining textual descriptions previously entered in a database, and automatically generating natural language output to fill in the rest. The content of the generated output depends upon the level of expertise of the user, and the history of previous user documentation requests. This fundamentally changes the nature and role of documentation. There will be less need for users to search through bulk documents in order to obtain answers to specific questions. Instead, the documentation system will search its own knowledge base for the information that the user requires, and compose explanations meeting the user's needs.

Other CASE (computer-aided software engineering) tools have been developed to support the authoring of documentation. These tools differ in that they tend to be oriented toward the generation of specific reports, such as those mandated by Department of Defense procurement standards. They make it easier to produce such reports, but that does not make the reports themselves significantly more useful. The I-Doc approach is designed to make such reports unnecessary for most purposes, although it will still be possible to generate bulk reports from I-Doc's design repository.

The approach employ^- reengineering technology, and is designed to be compatible with reengineering efforts. Information required to support documentation is entered in a reengineering knowledge base, in the form of annotations on source code parse trees. Transformations are employed to generate simplified and reorganized sections of code that highlight the aspects of the system being documented. Design recovery activities have the effect of bringing documentation up to date, facilitating subsequent software maintenance.

2 The State of Current Documentation

Let us examine the problems associated with conventional software documentation, to see how new techniques can alleviate those problems.

The primary emphasis of conventional system documentation is on amassing information. Documentation standards, especially government standards such as MIL-STD-2167A or SDD, require developers systematically to describe all details of a design, such as the inputs and outputs of each function. The structure of such documents is fixed and standardized.

The first problem with such system documentation is that is not sufficiently *activity-oriented*, i.e., it is not designed to support the activities of the intended readership. User manuals are activity-oriented in this sense: such manuals are designed to help people whose activity is to use the software system. System documentation is not, or if it is the set of activities being supported is incomplete. System documentation can potentialy support a number of activities, including the following:

- review by the customer to check that all stated requirements are met in the design,
- design reviews in which the quality and validity of the design is evaluated, and
- maintenance activities, in which maintainers seek to obtain information about the design so that modifications and enhancements can be performed correctly.

These activities are very different, yet documents often must support more than one of them. The activity that is least supported, of course, is maintenance. Maintenance manuals are common for physical devices, but are rare for software systems. Of course it is harder to write maintenance manuals for software than it is for devices, because maintenance tasks change as the software evolves. Nevertheless, maintenance activities in general are vastly different from specification and design reviews. Maintainers rarely perform methodical reviews of entire systems; rather, they inspect specific modules in detail in order to determine how they can be modified. Interrelationships between modules can be extremely important. Documents such as design documents, which describe all components in a uniform way, are more suited to design activities than maintenance activities.

In addition to being activity-oriented, good documentation is *task-oriented*, i.e., designed to help readers perform specific tasks. Tutorial user manuals are frequently written in a task-oriented fashion. For example, a word processor manual might have the user work through sample tasks such as composing a business letter or printing mailing labels.

A task-oriented approach centered on hypothetical tasks is not necessarily the best way to design documentation in general. It requires the reader to take the time to work through exercises, whereas document users typically are impatient and skip through the documentation trying to find out what they need so they can get on with their actual job. This is the motivation for the new "minimalist" approach to documentation, which uses overviews, structured exercises, and any information that the user cannot discover through experimentation with the system [3]. However, the minimal approach is not a rejection of task orientation per se, just of manuals that are oriented around lengthy hypothetical exercises and that contain information one can figure out on one's own.

It is certainly true that system documentation can be improved simply by learning lessons from other types of documentation such as user documentation. However, even well-written paper documents suffer from basic limitations. A person writing a document can only make rough guesses about what tasks the reader might be performing, what information he or she might want to know, and the level of expertise of the reader. Detailed exercises can help eliminate the guesswork—if the reader works through an exercise, the writer can try to anticipate what kinds of questions the reader might want to ask at each point in the exercise. This does not work if readers lack the patience to work through the exercises, as the minimalists argue.

The key to a substantial improvement in documentation is an on-line system that can construct presentations dynamically, reducing the reliance on guesswork. Interaction with the system should be in the form of a question-answer dialog; that way, the reader indicates to the system what he or she wants to know. The system can present information in the context of the reader's activities, simply by asking the reader questions about those activities. If the reader does not understand the descriptions generated by the system, the reader should be able to request a clarification, and have the system adjust its estimate of the reader's level of expertise. The process of supporting documentation then becomes less an activity of writing text and more an activity of providing the system with the information that it needs to produce a range of descriptions of the system.

Such a capability constitutes a clear advance of the state of the art in documentation support. However, the technologies needed to realize such a capability, such as design repositories, hypertext, program analysis and transformation, and natural language generation, are well developed and in a state where they can be brought to bear effectively on the documentation problem.

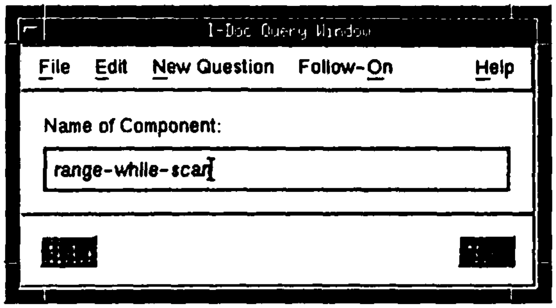
3 An Example from the Reader's Standpoint

The following example illustrates how I-Doc is intended to function, from the viewpoint of the reader, i.e., the person asking questions about the system.

The system in question is real-time embedded control software of a fighter aircraft radar system. This example was studied by Hughes in a research effort sponsored by Wright Patterson Air Force Base [4]. Hughes built a demonstration hypertext documentation system to support a hypothetical maintenance task on this system. We have been using the same example as an initial test case, to design I-Doc so that it can generate descriptions automatically that aie similar to what the Hughes group constructed manually in their demonstration.

One of the functions of the radar system software is a Range While Search function, which electronically controls how the aircraft's radar scans the airspace. Normal Range While Search scans a volume of air space that is wider in azimuth than it is in elevation, say 60 degrees to the left and right of the aircraft, and 10 degrees above and below the horizon. The volume is scanned by sweeping back and forth horizontally, top to bottom. The hypothetical maintenance task is to change the code so that it can scan volumes that are wider in elevation than in azimuth, by scanning vertically rather than horizontally.

Figure 1 shows the window that the user interacts with in order to initiate the query. The user types in the name of the component that he or she is interested in. Mechanisms for selecting components from a menu of alternatives will also be provided.



Name of Component:

range-while-scan

Figure 1: I-Doc Query Window

Before I-Doc can accept a query, however, it first requests information about the user and the task being performed. The user parameters are input via a menu such as that shown in Figure 2. The user is requested to indicate what role the user plays on the project, and indicates Maintainer. I-Doc will therefore include in the system descriptions that it generates information relevant to maintenance, e.g., inputs, outputs, and functional decomposition of each module. If the user had chosen a different selection, such as User, descriptions would be more functional in nature, and limited to those aspects of functionality that would be visible to the user (in this case the pilot or radar intercept officer responsible for controlling and monitoring the radar).

Additionally I-Doc requires an initial estimate of the user's degree of familiarity with the system. In this case the user selects Low, which causes I-Doc to limit the extent to which it refers to implementation details such as data representations.

Next, I-Doc requests a characterization of the task the user is performing. Four types of activities are known relating to system maintenance: adding functionality, fixing bugs, optimizing, and validating documentation. Add Functionality is the choice in this case. In this context, it causes I-Doc to generate high-level overviews of the functionality in question. If Fix Bug or Optimize were chosen, the description would focus more narrowly on those system components involved in generating the behavior that must be optimized. Validate Documentation is chosen when the user (typically a developer) wishes to see a variety of descriptions generated by I-Doc, to verify that the system can generate valid documentation in each case.

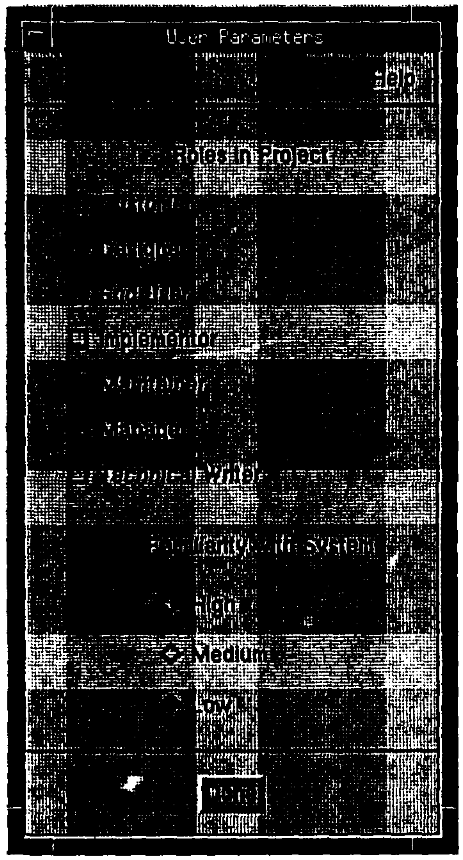
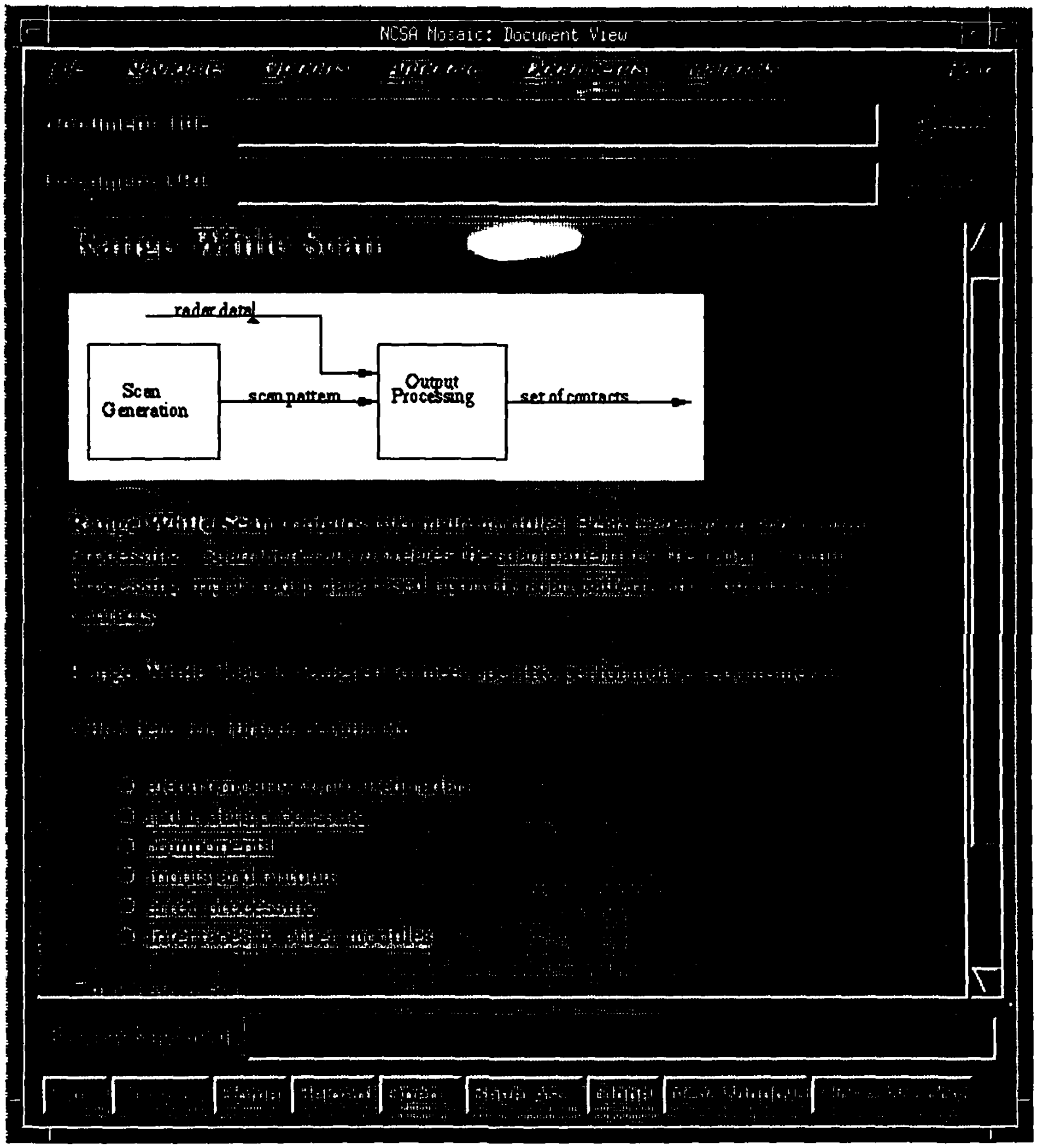


Figure 2: User Parameter Window

Figure 3 shows a sample output, based upon the parameters selected above. I-Doc cannot yet generate this output, as the project is just getting started; this is merely an illustration of the type of output that will be generated. The figure is a display generated by the Mosaic hypertext system [2], which is the hypertext system used as an output interface by I-Doc. The figure contains a simplified decomposition diagram showing the major components of Range-While-Scan: Scan Generation and Output Processing. It summarizes the function of each component, and the main inputs and outputs of each.

Several points are illustrated by this output example. First, the output is selective both in terms of what components of Range-While-Scan are described, and what properties of those components are mentioned. In this overview the major components of Range-While-Scan are shown, but not those components responsible for checking and reporting errors. (For example, if Output Processing detects erroneous radar input, an error is signaled.) It characterizes the function of the components (e.g, Scan-Generation creates a scan pattern), and the inputs and outputs of each component. If the task or user parameters were different, the summary would have changed accordingly, perhaps including more detailed information about the system.

Because the presentation medium is hypertext, it is not necessary to enumerate all relevant properties of Range-While-Scan. It is sufficient to provide hypertext links which, if selected, will permit the reader to obtain further information. Some of these buttonable items are interspersed through the text, and appear underlined in the figure. Other items appear at the bottom. Because Range-While-Scan's performance requirements are particularly important for anyone attempting to add functionality to it, a special hypertext link is included to access this information. Other relevant topics are included at the bottom. The links named "electronically controlled radars" and "radar data processing" provide background about the application domain that might be useful to a maintainer who is unfamiliar with the application. Below are listed links for obtaining more information about Range-While-Scan's components. Further down, below the bottom of the scrolling window in this example, are pointers that allow the reader to see the source code from which this description is derived, either the full text or a simplified version corresponding to what appears in this hypertext description.



Range While Scan

[illegible]

Figure 3: Hypertext Description of Range-While-Scan

4 System Architecture

I-Doc contains the following major functions.

- An acquisition interface is used to input the annotations necessary to generate system descriptions.
- This information is stored in a repository, and in annotations embedded within the source code itself.
- A query interface, as shown in Figure 1, is used to input queries from the user.
- The source code and repository are processed to extract the information to be presented.
- A presentation layout for the information is constructed.
- The presentation layout is displayed as hypertext. Requests to traverse hypertext links are intercepted and passed back to the extraction and layout subsystems to generate new presentations.

These components will be described in further detail below, but first the information content that these components operate on will be discussed.

5 Underlying Knowledge

In order to generate appropriate software descriptions, I-Doc requires a variety of information about the software and its design. Some of this information can be extracted directly from the code and from CASE repositories. Other information must be added to the design in the form of annotations.

First, a hierarchical decomposition of the design into functional components is required, ꞏogether with the types of the components' inputs and outputs. The pattern of data flow among components is necessary as well.

In order to present the information flow between modules in natural language, some additional characterizations of the data and the operations on the data are required. First, it is useful to classify the type of operation being performed by the component. Classifications that have been identified so far as useful include create, destroy, filter, insert, remove, retrieve, process, and validate. For example, the module Scan-Generation is classified as creating scan patterns. Modules whose function is to validate data were omitted from the summary in Figure 3, under the assumption that initial descriptions should assume that all data is valid, and methods for handling exceptional data will be described later. The content and use of data structures is characterized as well. Data structures are categorized as to whether they represent objects, aggregates of objects (sets, sequences, etc.), or names of objects. This enables I-Doc to refer to the output of Output-Processing as a set of contacts, regardless of the actual data representation (e.g., an array of pointers to contact objects).

Function categorizations can be applied to segments of components or groups of components, as well as to individual components. For example, a set of routines may be employed to process radar data, or a set of statements may be employed to validate the data.

Another type of information that plays a prominent role in I-Doc descriptions is information about requirements, particularly nonfunctional requirements. A set of attributes such as speed requirements or accuracy requirements may be associated with functional components and data.

In order to determine how to render data dictionary elements most effectively in natural language, I-Doc uses grammatical annotations. The annotations used in I-Doc are based on those used in the ARIES requirements acquisition system for annotating specifications [7]. Data expressing relationships between objects are categorized as to whether they are attributes, actions, circumstances, classes, or relations. Attributes describe properties of the object, actions describe actions that involve the object, circumstances describe states of the object, classes identify categories to which the object belongs, and relations are default data relationships. Objects participating in such relationships can assume one of several grammatical categories, e.g., actor, goal, location, or beneficiary. These categories are drawn from case grammars for natural language [5].

Finally, descriptions of classes of behavior, called *scenarios*, are useful in the description process. Scenarios are useful for defining system requirements, as a way of describing types of behavior that a system should or should not exhibit [1], which can be used to validate system specifications. They are intended to serve two roles within I-Doc. First, scenarios can be used to illustrate system behavior. Second, scenarios provide the context in which to describe systems. If the I-Doc user is trying to fix a bug, for example, then if a scenario illustrating the bug is available it can be used by I-Doc to focus on describing those components of the system that are relevant to the bug.

6 Knowledge Acquisition

The information described above is acquired from a variety of sources. These sources will be described below. It is important to emphasize, though, that I-Doc can still generate comprehensible system descriptions without much of this information. The added design information is used to improve the quality of system descriptions, and can be acquired when and as appropriate.

Some information is available in front-end CASE tools such as Software through Pictures [6]. I-Doc will have the ability to query one or more such CASE repositories in order to extract such information if available.

Another means of acquiring the documentary information is through a special acquisition interface. This method is used especially for inputting grammatical annotations and design component classifications. These annotations need not be selected directly; instead, the person entering the information can request that I-Doc attempt to provide the annotations automatically, and present samples of natural language output based upon those annotations. Although the grammatical annotations are based on linguistic concepts that may be unfamiliar to software engineers, it is easy to see when the generated natural language is awkward or incorrect. Once the user has selected from among alternative descriptions generated by I-Doc, I-Doc saves the annotations used to produce that sample output.

There are three other ways in which information for constructing system descriptions is obtained. One means is through the use of object hierarchies. If an object class has a particular set of attributes, its specializations are likely to have similar attributes. A second approach is to annotate the models used in automated program synthesis systems. The knowledge bases of specialized knowledge-based synthesis systems, such as user interface development systems, can be augmented to support the generation of documentation and help as well [12]. Integration of I-Doc with one or more such systems is an option being considered for future development.

The third source of design information for documentation is code analysis. Analysis routines can detect components that appear to be creating objects, inserting into or removing from data aggregates, validating data, etc. Such analysis is further facilitated when some design components are already annotated; e.g., when data is designated as an aggregate, it makes sense to look for routines that add and remove elements from that aggregate. The analysis capabilities of Reasoning Systems' reengineering technology, in particular Refine/Ada, are being used as the basis for such analysis capability.

Use of the above capabilities for extracting relevant design annotations is one way in which reengineering technology plays a prominent role in the construction of documentation in I-Doc. In fact, in order for automated generation of documentation to succeed it must be viewed at least partly as a reverse engineering enterprise. Preparing a design for document generation involves adding design information that is not present in the original design. Front-end CASE tools, executable specification languages, and other advanced forward engineering technologies can reduce the amount of information that must be recaptured, but even then some information must be made explicit that is implicit in the design. Thus a combination of interactive design capture and design recovery techniques appear to be essential.

7 Repository Storage and Maintenance

As design information is acquired it must be associated with the code and maintained. The basic mechanism being used to achieve this is to add attributes to the Refine representation of program parse trees. In order to permanently associate these attributes with the code, the following techniques are planned. First, Ada pragmas will be used to insert the information directly into the code. Pragmas were originally intended to record information to guide compilers in generating object code. In an analogous fashion they can be used to guide the generation of system descriptions.

In the longer term, it may be appropriate to extend the data model of a CASE tool such as Software through Pictures in order to record the design information. However, that should not be a substitute for inserting design information directly into the source code. In order for the captured design information to be useful, it must continue to be associated with the code as it is maintained over time. At the present time the best way of ensuring this is to integrate the design information into the source code, so that maintenance using a front-end CASE tool is not required. For languages that do not have constructs similar to pragmas, the approach will be to add grammatical extensions to the source language to provide such constructs, which can then be removed from the source code via transformations, using a tool such as Refine.

8 Inputting User Queries

If the design knowledge associated with a design is sufficiently rich, it can be used as the basis for answering a variety of queries. Following the approach taken in Lehnert's original work on question answering [8] and further developed by research in expert system explanation, such as the work of Moore and Swartout [10], common questions about software have been categorized into types. In the initial version of I-Doc, these question types will be available to the user as explicit choices from which the user can choose. Examples of question types include Describe Function, Describe Design, Describe Interface, and Describe Use. In subsequent versions these choices will be automated to a greater extent, so that the user can simply request "Describe" and the system will construct a combined description appropriate to the context.

The main difference between question input in I-Doc and question input in expert system explanation is the use of hypertext as a medium for posing questions. Question-answering systems such as Moore and Swartout's facility in the Explainable Expert System (EES) system allow the user to point to an element of a description and ask one of several follow-up questions about it. In hypertext, the interaction is more limited—the user clicks on a section of text where a link begins, causing the system to jump to the other end of link. The user does not have the option of selecting one of several operations to perform on the link. In order to overcome this difficulty, we are experimenting with making choices explicit as lists of selectable items at the bottom of the hypertext display, as shown in Figure 3. It remains to be seen how effective this technique is in comparison to more ordinary menu-based approaches.

9 Extracting Relevant Information

Once the type and the object of the query have been chosen, the information suitable for inclusion in the presentation must be retrieved and presented. Retrieval of relevant information can be easily accomplished using the retrieval and query mechanisms available in Refine, Software through Pictures, and other tools. Difficulties arise, though, when the amount of retrieved information is too much to present in such a way that the reader can assimilate it. In such cases filtering techniques may be employed to focus on the information of greatest potential interest to the reader.

The filtering procedures in I-Doc proceed by marking code as either central, interesting, or ignored. Central components are the primary focus of the description, and consist of all code satisfying a particular criterion, such as code matching steps a scenario. Components are designated interesting because of their interactions with the central components. Ignored components are removed because they can be explicitly filtered out, or because they are not found to be central or interesting. The following are some criteria that have been found in pencil-and-paper studies to be useful in determining code to be central or ignored; others are expected to be identified.

- Exceptional case handling. In some descriptions, such as the one shown in Figure 3, code for handling exceptional cases is unimportant; in other cases (such as when fixing bugs) such code may be central. Either way, such can be detected in a significant number of cases. Ada has explicit constructs for raising exceptions. Additionally, if a variable is being used as an error flag, tests of the variable can be detected automatically, and branches of the code marked accordingly.

- Code that sets and/or reads a variable or attribute.

- Code that assigns an attribute or variable to a specific value, or checks that the attribute or variable has a specific value.

Once components are found be central, surrounding code is often marked as interesting and is therefore included. The motivation for this is to provide some context so that the focussed operations are more easily understood. For example, if a routine sets an interesting attribute of an object, assignments to other attributes of the same object in the same procedure may be included as well.

The transformation facilities in Refine are to be employed to implement this filtering process. The transformation pattern language can be used in some cases to detect the code that is of interest. In other cases, transformations can be used to perform simple expression simplifications in order to facilitate pattern matching. A tool that could recognize all instances of potentially interesting code would require a more powerful simplifier than is envisioned for I-Doc. Instead, I-Doc will either inform the reader when the view being presented is possibly incomplete, or simply provide a view that is somewhat broader than is strictly necessary.

10 Presentation Layout

Once relevant information has been identified, I-Doc must determine what media to use to present the information, and how to present the information using those media. In the long term, we hope to be able to employ a variety of presentation media, including mixtures cf text and diagrams. At first, :hough, the focus will be on natural language generation. Such generated text may be supplemented with diagrams if such diagrams have already been constructed and are available, as is currently the case with CASE-generated documents.

The overall structure of each description that is generated will be determined by a presentation template, a library of which will be included in I-Doc's knowledge base. These templates provide standard ways of presenting different aspects of a system. Each template will have a set of selection criteria, based upon the amount of various types of information that is to be presented, and the assumed level of expertise of the user. Slots within the template are then filled out using a natural language generator.

The natural language generator consists of two components. The basic generation work is performed by a Functional Unification Generator, which can construct arbitrary sentences from attribute-value descriptions of the content to be expressed. A second phrase selection component constructs attribute-value patterns in a form suitable for input to the Functional Unification Generator, according to directives contained within the system description templates. This architecture was used successfully in ARIES and the KBSA Concept Demonstration [11] to generate text descriptions rapidly. Functional Unification Grammar has been evaluated against competing methods for natural language generation, and has been found to be both flexible and efficient [9].

11 Presentation Delivery

As indicated in Section 3, Mosaic is being employed as the hypertext delivery mechanism for I-Doc. Although there are various commercial products available that provide hypertext capability, Mosaic has a set of features that make it particularly suitable to dynamic documentation generation.

- Mosaic provides interfaces to external programs, so that the user buttoning on a hypertext link can cause a program to be invoked.

- It interprets hypertext formatting commands dynamically, as needed; this makes it possible to construct a hypertext document dynamically and display it.

- It runs on a variety of platforms, including X Windows, Microsoft Windows, and Macintosh.

- It is public domain, and the source code is readily available, making it possible to customize the system for use within I-Doc.

12 Further Plans and Prospectus

The I-Doc project has just begun; it is expected to continue for another three years. The project plans to make available intermediate versions of the system on a periodic basis. In the near term, the capability will be demonstrated on a military application selected in collaboration with the US Air Force's Wright Laboratory.

If resources are available, I-Doc will be extended so that it can describe systems to other classes of users besides maintainers, such as clients and end-users. This will further enhance the value of documentation generation technology tc ..ftware development projects.

In the long run, dynamic generation of documentation will prove successful if the perceived benefits outweigh the costs of additional design capture. The benefits will be particularly apparent in projects that undergo periodic design reviews, since automated documentation should prove to be a great benefit to the design review process. Reengineering and program synthesis technology will gradually reduce the amount of interaction between developers and I-Doc necessary for design capture. In the near term, making enhanced hypertext capabilities available to developers is likely to bring benefits in itself. Hypertext is gradually being adopted in software development practice, and the activities of the I-Doc project will aim to help accelerate this trend.

13 Acknowledgements

The author wishes to thank Bill Swartout and Richard Angros for their contributions to this effort, and John Salasin and Marc Pitarys for their support. Sheila Coyazo assisted with preparation of the article. This work is sponsored by the Advanced Research

Projects Agency and administered by Wright Laboratory, Air Force Materiel Command, under Contract No. F33615-94-1-1402. Views and conclusions contained in this paper are the author's and should not be interpreted as representing the official opinion or policy of the U.S. Government or any agency thereof.

References

[1] K. Benner, M.S. Feather, W.L. Johnson, and L. Zorman. The role of scenarios in the software development process. In *Proceedings of the IFIP W8.1 Working Conference on Information System Development Process*, 1993. To appear.

[2] E. Bina and M. Andreessen. NCSA mosaic home page. Available from World Wide Web server www.ncsa.uiuc.edu.

[3] J.M. Carroll. The minimal manual. *Human-Computer Interaction*, 3(3):123–153, 1988.

[4] R.A. Falcioni and R.L. Buvel. Modular embedded computer software (MECS): Interim report. Technical Report WL-TR-92-1113, Wright Laboratory, Wright Patterson AFB, OH, 1990.

[5] C.J. Fillmore. The case for case. In *Universals in Linguistic Theory*, pages 1–88. Holt, Reinhart and Winston, New York, NY, 1968.

[6] Interactive Development Environments. *Software through Pictures: Fundamentals of StP*, 1993.

[7] W.L. Johnson, M.S. Feather, and D.R. Harris. Representation and presentation of requirements knowledge. *IEEE Trans. on Software Engineering*, 18(10):853–869, October 1992.

[8] W.G. Lehnert. *The Process of Question Answering*. Lawrence Erlbaum Associates, Hillsdale, New Jersey, 1978.

[9] K.R. McKeown and M. Elhadad. *A Contrastive Evaluation of Functional Unification Grammar for Surface Language Generation: A Case Study in the Choice of Connectives*, pages 351–392. Kluwer Academic Publishers, Norwell, MA, 1991.

[10] J.D. Moore and W.R. Swartout. *A Reactive Approach to Explanation: Taking the User's Feedback into Account*, pages 3–44. Kluwer Academic Publishers, Norwell, MA, 1991.

[11] J.J. Myers and G. Williams. Exploiting metamodel correspondences to provide paraphrasing capabilities for the concept demonstration. In *Proceedings of the 5th KBSA Conference*, pages 331–345, Syracuse, NY, September 1990. Defense Technical Information Center.

[12] P. Szekely, P. Luo, and R. Neches. Facilitating the exploration of interface design alternatives: The HUMANOID model of interface design. In *Proceedings of CHI'92, The National Conference on Computer-Human Interaction*, pages 507–515, May 1992.

A Case Study of Software Reuse in Vertical Domain

Václav Rajlich and João Silva

Department of Computer Science
Wayne State University
Detroit, MI 48202
vtr@cs.wayne.edu

Abstract

This paper presents a case study of domain specific software reuse, also called vertical reuse, where both the architecture and individual classes are reused. The applications domain we deal with is the domain of visual interactive software tools. The paper describes the architecture itself, the reverse engineering process by which it was obtained, and the forward engineering process by which it was reused. The architecture is called orthogonal architecture, and it consists of classes organized into layers and threads.

Key Words

Software reuse, vertical domain, object-oriented programming, visual graphical tools, layers, threads, program families, process of reuse, reengineering.

1. Introduction

The reuse in vertical domain is characterized by a reuse of the whole software architecture, which is being adjusted to satisfy a new set of requirements. In this respect, reuse in vertical domain, software evolution, and perfective maintenance overlap to a large extent. In our case study, we studied a process of reuse in vertical domain of visual interactive software tools. We developed an orthogonal architecture for that domain, which is particularly suitable for vertical reuse. Evolutionary domain life cycle was the process used in the case st.dy. The architecture was obtained through reengineering of an earlier tool.

The major difference between the evolutionary domain life cycle and the waterfall model are the activities that span the whole application domain. These are domain analysis [21, 22, 23, 24], and domain design [8]. Domain analysis examines the requirements of a family of systems. The end result of the domain analysis phase is a document called the domain specifications. Domain design includes definition of the data structures, file formats, and important algorithm descriptions for a specific domain of applications. The final product of the domain analysis phase is a domain architecture that describes the architecture of a family of systems. This domain architecture reflects the design of all software systems that constitute the domain.

We studied the domain of visual interactive tools. Our experience in building these systems dates to 1986 when we developed a prototype of VIC: Visual Interactive C [25, 26]. Other visual interactive editors developed in our group were VIFOR [26], and EDG (described here -- see Appendix A). We defined a set of classes -- some of them general, others specialized -- and all interactions among them. In this model classes may evolve, but the overall architecture remains unchanged.

2. Previous Work

The concept of "program families" originates with Parnas [16, 17, 18, 19, 20]. Program families are defined as sets of programs whose common properties are so extensive that it becomes advantageous to study the common properties of these programs before analyzing individual differences. Since then, other researchers such as Neighbors [13, 14, 15], Barstow [6], Lubars [10, 11, 12], Bailin [4, 5], Kant [9], Arango [1, 2, 3], and Prieto-Diaz [21] have associated the concept of a "family of programs" with the idea of application domain.

Barstow [6] investigated the issue of domain specific automatic programming in the context of the two application domains, both related to oil well logging. Similar studies have been concluded by other researchers for other domains of knowledge. Examples of such studies includes those of Kant et al. [9], Bailin [4, 5], and Dunn [7]. Our work, is in a sense, related to these works because we also performed domain analysis and domain design studies within a specific domain.

Prieto-Diaz [21] proposed a methodology for domain analysis. He described three distinct steps: pre-domain analysis, domain analysis, and pos-domain analysis. Pre-domain analysis comprised the definition and scope of the domain, the identification of sources of knowledge, and information about the domain.

Barstow and Kant [6, 9], among others, performed domain analysis and domain design studies that led to code generation within well defined domains of applications. These domains, quite different in nature, possessed a high degree of cohesiveness, a well understood variability, and a low complexity level. Code generation is advocated for such cases. For more complex systems with a higher degree of variability and a lower degree of cohesiveness, code generation is not recommended. Instead, a knowledge-base that captures important aspects that characterizes the family of systems can facilitate its development. The knowledge-base consists of : (1) well defined code components, (2) a generalized systems' architecture for the family of systems that constitute the domain, and (3) a set of domain rules (i.e., domain knowledge). The domain knowledge includes relationships among different components that make up the system.

3. Orthogonal architecture for interactive software tools.

We performed the domain analysis and design study for an experimental and reusable software system known as "The Environment for Decomposition and Generalization (EDG -- pronounced "edge"). EDG supports the OODG methodology of design [22]. The EDG system has an architecture, which can be easily adapted for other systems within the domain of visual interactive editors. Our goal was not to show that the architecture of EDG is the best possible, but we were satisfied with an architecture that possessed the following attributes:

(1) reusable: The architecture can be transformed to perform the functions expected of other visual interactive tools.

(2) Understandable: The architecture is simple. Domain users who have a minimum knowledge of visual interactive tools are capable of understanding what each class does and what the roles are within the domain, by reading the classes specifications.

(3) adaptable: The classes are capable of evolving. This capability involves the mechanics of changing specific members within a class while it still preserves the structural relationships between classes (i.e., maintaining the overall generic architecture for the whole system).

In order to accomplish these objectives, EDG system is partitioned both horizontally and vertically. Horizontal partitioning represents layers or levels of abstraction. On layer 1, function "main" controls the application. Layer 2 implements the menu interface, and triggers callback functions chosen from the menu. Layer 3 encapsulates the interfaces for all operations supported by EDG. At this layer, we defined a standard dialogue for each operation. For example, if the operation is "Save As," we know that there is a standard dialogue between the user and the system. First, after this operation is selected, a panel is presented and the user is asked to enter a file name. Then if the file name is valid and the user hits the return key or presses the "ok" button, the file is saved. In any other sequence of events, this "Save As" operation is aborted.

The actual functionality of the operations was defined at layer 4. Some of the classes which were defined at this layer include: a class to draw graphs on canvas, a class with algorithms for representing graphs, a class with functions to scan C++ source code and to extract relationships between objects, etc.

Finally, most operations in EDG need services of a database. The fifth and sixth layers perform these services. Layer 5 encapsulates the specific data model and database interface, while layer 6 is the database itself. They are both custom made specifically for EDG. Database is resident in the main memory, and is stored on the disk in a flat file. The efficiency was not of a concern, and hence relatively simple data representations and search algorithms were employed.

Vertical partitioning of the EDG system involves the division of the system into threads that are orthogonal to layers. Threads are sets of classes related to each other by relationship of "use". Threads in our architecture are largely independent of each other, with very few classes from one thread using services of the classes in other thread. Only layers 1 (highest) and 6 (lowest) are shared among the threads. The threads of EDG are:

(1) Project, supporting commands that operate on entire projects and its files.

(2) Graph, which supports graph display and editing.

(3) Views, which selects information to be presented in a window.

(4) Browser, which supports navigation through the database.

(5) Analyzer, which extracts architectural information from programs.

(6) Run, which interfaces EDG with other tools (compilers, debuggers, etc.).

These six threads are truly universal within the domain of visual interactive software environments. Examples of software development environments that use one or more of these threads include: Software through Pictures, Powertools from Iconix, and Teamwork from Cadre.

Taken together, these two orthogonal system partitionings provide an interesting map of the EDG system. For each specific layer and thread, there is at least one class implementing the required functionality. This orthogonality substantially improves the understanding of software, and therefore facilitates its reuse.

4. Reverse engineering: Creating EDG.

EDG was created from earlier projects, which had the same or overlapping functionalities but did not possess the orthogonal architecture described above. The reverse engineering was applied to a code written partially in C and partially in C++. The effort involved domain analysis, which had to be done for the whole domain of visual interactive tools. That was followed by a domain design, where the threads, layers, and individual classes of EDG were defined. Finally the existing code was analyzed and reengineered into the new code, fitting the new architecture. The classes of the new code fall into the following categories: Classes transfered from the previous projects with modifications, and classes written from scratch. There were no classes which could be reused from the previous project without any change.

Of the total 11,558 lines of the code of EDG, total of 4,360 belongs to the classes written from the scratch. These classes belong mostly to top and bottom layers, where the impact of the new architecture was most felt. The rest belonged to the classes that were modified to a larger or lesser degree. As far as the effort is concerned, the effort to reengineer old code into EDG was approximately 40% of the estimated effort it would take to implement EDG from scratch. For more detailed numbers, see [29].

5. Forward engineering: From EDG to EDFD.

The reusability of orthogonal architecture of EDG was tested in a case study where it was reused for a new visual interactive tool: Environment for Data Flow Diagrams (EDFD) which partially supports the methodology described in [27]. A more detailed description of both EDG and EDFD is included in Appendices A and B. As the first step, we identified the set of operations required by the new system, and organized them into a top-menu.

Via the second step, "assign operations," we mapped the set of operations identified in the previous step on the pre-existing set of threads. For example, the operation "Hide Object" is mapped to thread "graph." After this phase, when all operations for a new system are assigned to a thread, all unnecessary threads are removed from the system. In our case EDFD required three threads: the Data-Flow Editor, the Data Dictionary Editor, and the Defining Functions Editor. The first thread was mapped on Graph Editor thread of EDG, while both remaining threads were mapped on Project thread of EDG. Hence the project thread is modified in two different ways, once to serve as Data Dictionary thread and a second time as Defining Functions thread. The rest of the threads of EDG were no longer needed, and therefore were discarded.

The next step consists of modification of the classes in the threads. When modifying a class, we mapped the new specifications of the class on the old one, and compared the new requirements with the existing code. For "Add an Object," we identified the set of functions in "graph" associated with that operation. First we searched the interface of class "graph" and checked whether or not that function exists. If so, we considered that function and all functions called by it for possible modifications.

Function modifications were effected "top-down" until all of the selected functions were completely defined. Three different scenarios have occurred:

(1) The function can be reused "as is." No changes were required.

(2) The function needs to be "adjusted" to conform to a new set of requirements. Here, we would isolate the portion of the function which can be reused without other changes, and then we would add code to perform the new required functionality.

(3) The function does not exist. Here, we implement the new function to accommodate a new set of requirements.

The whole architecture was scanned through this process, and adapted for the new set of requirements.

The following statistics on the extent of modifications were gathered:

Layer 1 Reused without adaptations.

Layer 2 Very easy adaptation
74.2 % of code was used without adaptations
25.8 % of code was used with some modification.
Unnecessary to introduce new functions.

Layer 3 Very easy adaptation.
94.6 % of code was used without adaptations.
4.5 % of code was used with adaptations.
0.9 % of code was implemented from "scratch."
Unnecessary to introduce new functions.

Layer 4 Refers to "graph_editor" (our worst case scenario).
Very easy adaptation for already existing functions.
73.5 % of code was used without adaptations.
2.2 % of code was used with adaptations.
24.3 % of code was implemented from "scratch."
Necessary to introduce new functions.

Layer 5 Very easy to modify.
88.3 % of code was used without adaptations.
3.8 % of code was used with adaptations.
7.9 % of code was implemented from "scratch."
Unnecessary to introduce new functions.

Layer 6 Very easy to modify.
78.4 % of code was used without adaptations
2.3 % of code was used with modifications
19.3 % of code was implemented from "scratch."
Unnecessary to introduce new functions

Analyzing this data, the classes requiring least amount of work were either high in the class hierarchy (layer one) or low (layers five and six). Classes in the middle of the hierarchy required more work.

The total size of EDFD is 4,606 lines. The total effort of the reuse represents 37% of the estimated effort to build the system from scratch. For more detailed numbers, see [29].

6. Conclusions.

We found that building the "perfect architecture" is a step by step process, very similar to exploratory programming. As we coped with the new requirements of EDFD, we were able to perfect the existing architecture of EDG and improve its classes, thus making the architecture more universal and adaptable. We believe that the architecture of EDG developed in the case study is suitable for most visual interactive tools, and future reuse and retroactive improvement will make it even more reusable.

We developed a process of three steps to adapt a generic orthogonal architecture to perform a new set of requirements. Please note that the process is domain independent. Hence we are expecting to be able to use this process in other domains as well.

We conjecture that the orthogonal architecture can be developed for domains other than visual interactive tools. We did some preliminary studies of nucleus of operating system, and found the same methodology and architecture ideas applicable there.

In our case study, we found encouraging productivity figures. To reengineer an unconstrained architecture into an orthogonal one, we spent approximately 40% of the time compared to implementation from the scratch. Adapting this architecture to a new set of requirements, we spent approximately 37% of the time compared to building the system from the scratch. Hence to reengineer a system into orthogonal architecture, and then forward engineer it to a new set of requirements, was cost effective already on the first system, where it cost 77% of the estimated original cost. We conjecture that each subsequent system within that domain should cost again approximately 40% of the original cost to build. We find these preliminary figures to be very encouraging, and hope to verify them by studies in other domains.

Bibliography

[1] Guilhermo F. Arango, "Evaluation of a Reuse-Based Software Construction Technology," Proc. of the Second IEE/BCS Conference: Soft. Engineering 88, pp. 85-92. IEE, London, UK, July, 1988.

[2] Guilhermo F. Arango, "Domain Analysis-From Art to Engineering Discipline." IEEE Computer Society Press, pp. 81-88. 1991.

[3] Guillermo Arango , Josiah Hoskins, and Eric Schoen, "Product Modelling for Software Re-engineering." Proceedings of the 13th Int. Conference on Software Engineering, pp. 14-17, May 13-17, 1991 Austin, Texas, USA.

[4] Sidney C. Bailin, "Generic POCC Architectures," Report prepared for NASA Goddard Space Flight Center." Associates, Laurel, MD, April, 1989.

[5] Sidney C. Bailin, "The KAPTUR Environment: An Operations Concept." Report prepared for NASA Goddard Space Flight Center, Associates, Laurel, MD, June, 1989.

[6] David R. Barstow, "Domain-Specific Automatic Programming." IEEE Transactions on Software Engineering, vol. SE-11, no.11, pp. 1321-1336, November, 1985.

[7] Michael F. Dunn and John C. Knight, "Software Reuse in an Industrial Setting: A Case Study." IEEE Computer Society Press, CH2982-7/91, pp. 329-337, July 1991.

[8] Hassan Gomaa and Larry Kerschberg, "An Evolutionary Domain Life Cycle for Domain Modeling and Target System Generation." Proceedings of the 13th Int. Conference on Software Engineering, pp. 65-71, May 13-17, 1991-Austin, Texas, USA.

[9] Elaine Kant and Ira Baxter, "Domain Modeling in SINAPSE for Synthesizing Mathematical Modeling Programs." Proceedings of the 13th Int. Conference on Software Engineering, pp. 23-25, May 13-17, 1991-Austin, Texas, USA.

[10] Mitchell D. Lubars, "A Knowledge-based Design Aid for the Construction of Software Systems.

PhD. Thesis, University of Illinois, Urbana-Champaign, 1987.

[11] Mitchell D. Lubars, "A Domain Modeling Representation." Technical report STP-366-88, Microelectronics and Computer Technology Corporation, Austin, TX, November, 1988.

[12] Mitchell D. Lubars, "Domain Analysis and Domain Engineering in IDeA." Technical report STP-295-88, Microelectronics and Computer Technology Corporation, Austin, TX, September, 1988.

[13] James M. Neighbors, "Software Construction Using Components." PhD. Thesis, University of California at Irvine, 1980.

[14] James M. Neighbors, "The Draco Approach to Constructing Software from Reusable Components." IEEE Transactions on Software Engineering, Software Engineering, vol. 10, no. 5, pp.564-74, September 1984.

[15] James M. Neighbors, "Report on the Domain Analysis Working Group Session." Proceedings of the Workshop on Software Reuse, Rocky Mountains Institute of Software Engineering, Boulder, CO, October 1987.

[16] David L. Parnas, "On the Design and Development of Program Families." IEEE Transactions on Software Engineering, vol. 2, no. 1, pp.1-9, March 1976.

[17] David L. Parnas, "The Influence of Software Structure on Reliability." Current Trents in Programming Methodology: Software Specification and Design vol. 1. ed. R. Yeh, Englewood Cliffs, NJ: Prentice-Hall.

[18] David L. Parnas, "Designing Software for Ease of Extension and Contraction." IEEE Transactions on Software Engineering, vol. 5, no. 2, pp.128-138, March 1979.

[19] David L. Parnas, Paul C. Clemens and David M Weiss, "Enhancing Reusability with Information Hiding." Proceedings of the Workshop on Reusability in Programming, Stratford , CT: ITT Programing.

[20] David L. Parnas, Paul C. Clemens, and David M Weiss, "The Modular structure of Complex Systems." IEEE Transactions on Software Engineering, vol. 11, no. 3, pp.259-266, March 1985.

[21] Ruben Prieto-Diaz and Peter Freeman, "Domain Analysis for Reusability." Proceedings of COMPSAC 87: The Eleventh Annual International Computer Software & Applications Conference, pp. 23-29. IEEE Computer Society, Washington, DC, October 1987.

[22] Ruben Prieto-Diaz, "A Domain Analysis Methodology." Proceedings of the 13th Int. Conference on Software Engineering, pp. 138-140, May 13-17, 1991-Austin, Texas, USA.

[23] Ruben Prieto-Diaz, "A Domain Analysis: An introduction." ACM Software Engineering Notes vol. 15, no. 2, pp. 47-54.

[24] Ruben Prieto-Diaz, "A Domain Analysis Methodology." Proceedings of the 13th Int. Conference on Software Engineering, pp. 138-140, May 13-17, 1991-Austin, Texas, USA.

[25] Vaclav Rajlich, Nicholas Damaskinos, Wafa Korshid, Panagiotis Linos, and João Silva, "Visual Support for Programming in the Large." IEEE Conference on Software Maintenance, 1988.

[26] Vaclav Rajlich, Nicholas Damaskinos, Wafa Korshid, Panagiotis Linos, and João Silva, "An Environment for Maintaining C Programs." CASE'88 Second International Workshop on Computer-Aided Software Engineering July 12-15, 1988, Cambridge, Massachusetts, USA.

[27] Vaclav Rajlich, "Decomposition/Generalization Methodology for Object-Oriented Programming." To be published in the Journal of Systems and Software.

[28] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, William Lorensen, "Object-Oriented Modeling and Design." Prentice hall, Englewod Cliffs, New Jersey 07632, 1991.

[29] Joao Silva, "Vertical Reuse in Software Tools: A Case Study, PhD Dissrtation, Department of Computer Science, Wayne State University, 1993.

Appendix A

EDG Requirements

EDG is an experimental software tool used in the development of C++ programs; it enforces the use of the Object-Oriented Decomposition and Generalization methodology (27). With EDG, C++ programs can be displayed and edited in two forms: class diagrams and code. A more detailed description of the EDG follows.

Six major threads constitute EDG:

- the project manager thread
- the graph editor thread
- the view manager thread
- the code analyzer thread
- the browser thread
- the code generator thread

The Project Manager

The project manager subsystem implements the commands which operate on entire projects (project descriptions and files). A brief description of each commands follows.

Open: Opens an existing project or starts a new project.

Close: Resumes the currently loaded project.

Add File: Adds a new file to the project description.

Remove File: Removes a file from the project description.

Save: Updates the current project description in a file.

Save As: Updates the current project description in a new file -- possibly the same.

The Graph Editor

Graph operations are used to build and manipulate the architecture of a program. By using these operations, graphs may be created, deleted, and modified. Specially designed canvas windows are used to display these graphs. In association with the graph editor, we implemented Sugiyama's algorithm to display hierarchical graphs.

New: Starts a new graph editor window and initializes the graph database.

Add: Adds objects to the graph database. In EDG, objects can be of two kinds: classes and relationships between classes - use or inheritance relationships.

Delete: Removes objects form the database.

Rename: Changes the name of an object.

Select: Changes the active object.

Color: Changes colors of objects.

Hide: Removes an object and associated relationships from a graph. These objects are not displayed, but are kept in the database.

Move: Changes object positions in a displayed graph.

Load Graph: Reads and displays the information from a file which contains the description of a graph.

Save Graph: Writes the graph description in a named file for future retrieval.

The View Manager

"View" operations are used to control and display the information in text and canvas windows. However, "view" operations do not affect data itself. The view manager implements commands which control the presentation of the information within a canvas window and/or a text window.

New: Creates a new instance for the view manager.

Show File: Displays a named file in a text window. This file cannot be modified using the view manager.

Show Graph: Displays a named graph on a canvas. This graph cannot be modified using the view manager.

Text Editor: Starts a text window that can be used for documentation.

Print File: Sends a text file to the printer.

Snapshot: Retrieves information of some portion of a displayed graph, saves it in a named file, and displays that portion enlarged.

The Code Analyzer

This thread contains commands used in the extraction of architectural and procedural information of C++ programs. The analyzer uses a parser to scan C++ code files and identify C++ code components. In addition, the analyzer searches for relationships between C++ code components. Examples of these code components include classes and functions. While examples of relationships between code components include: "use" and "inheritance."

The menu interface to this thread includes the following operations:

New: Create a new instance of the analyzer. Initialize the analyzer database.

Analyze: Start the interface template for the user to type the filename(s), activate the parser, and populate the analyzer database.

Display: Invoke the Sugiyama's algorithm to compute object positions and display the graph.

The Browser

The browser, helps programmers to understand object oriented software systems written in C++. To achieve this goal, this thread provides information about the set of classes and files comprising the system and relationships among them. With the set of operations described below, users of this thread may "browse" through the system based on relationships among classes, files, and identifiers. Useful cross-reference information about class methods and variables may also be found.

New: Create a new instance of the browser and initialize the browser database.

Show Class: Reads all classes from the database and shows all classes in a panel. When any one of the project classes is selected, its declaration file is opened and displayed on a text-window. The selected class becomes the "active class."

Select Class: Changes the current "active class." The selected class text is validated against class names stored in the database. If the class name is valid, then the active-class is changed. Otherwise, an error message box is shown to the user.

Class Info: Returns information about the "active class." This information is shown on the information panel and includes:

- -- class name
- -- "uses" relationship
- -- "used-by" relationship
- -- "is_a_subclass" relationship
- -- is_a_superclass" relationship
- -- inheritance path
- -- header file
- -- number of "use" relationships
- -- number of "used-by" relationships

Declarations: A text window is shown, which opens the declaration file for the active class. To select the active class one may use the mouse and highlight a class identifier or may highlight a function identifier. Either way, the declaration file for that class or function is shown in a text window.

Definitions: Similar to "declarations." Instead of displaying the header file, it displays the definition file for the selected identifier.

Show Info: This operation is used to give a summary of information about an identifier. Identifiers may be classes or functions. The following information related to identifiers is preserved in the database and displayed when this operation is selected:

- identifier_name
- is_overloaded
- variable_or_function
- type
- declaration_file
- definition_file
- parameter_types
- declaration_file_index
- definition_file_index
- comments
- call_relationships
- called_from_relationships
- total_number_of_calls
- number_of_calls_from

The Run Project

The run project contains the set of commands used to interface EDG with other tools (eg: compilers, debuggers, and code generators).

Generate: Generates C++ code skeletons for classes.
Make : Generates a Makefile for an entire project based on project descriptions.
Compile: Executes the make file.
Execute: Executes the object code for the project.
Debug: Invokes a symbolic debugger -- For example dbxtool.

Appendix B

EDFD Requirements

EDFD supports:

1. Identifying input and output values.
2. Using data Flow Diagrams as needed to show functional dependencies.
3. Describing what each function does.
4. Identifying constraints.
5. Specifying optimization criteria.

To provide support for 1 and 2, we provide a data flow diagram editor. To provide support to 1 and 4, we provide a data dictionary editor. To support 3 and 5 we provide a describing functions editor. We continue this discussion with a description of the operations required to support each one of the editors.

The Data Flow Diagram Editor

Data elements flow from one process node to another process node where they are processed. Data flow diagrams consist of four graphical components: Processes, data flows, data stores, and actors. Processes denote functions of the system. Data Flows represent data elements flowing between process nodes. Data stores represent places where data elements are stored. And, actors are independent objects that produce and consume values -- source or sink of data.

Data flow diagrams depict a system from the data's point of view. Using data flow diagrams, the analyst is able to show how data flows in a system, how data is transformed by the system, and where to store data in the system. To help us with this data flow description, we provide the following data flow diagram editor.

New: Starts a new graph editor window and initializes the data flow diagram database.
Add: Adds objects to the graph database. In EDFD, objects can be of four kinds: processes, data stores or file objects, and actor objects. In addition, there are association objects: data flow between processes, data flow that results in a data store, and control flow. We should also provide the possibility of having composition of data values, decomposition of data values, duplication of data values, and access and update of data values.
Delete: Removes objects from the database.
Rename: Changes the name of an object.
Select: Changes the active object.

Color: Changes colors of objects in a specific data flow diagram.

Hide: Removes an object and associated associations from a data flow diagram. These objects are not displayed, but are kept in the database.

Move: Changes object positions in a displayed graph.

Load Graph: Reads the information from a file which contains the description of a data flow diagram, and displays it.

Save Graph: Writes the data flow diagram description in a named file for future retrieval.

The Data Dictionary Editor

Data flow diagrams are documented by a data dictionary. The data dictionary defines the meaning of each data flow, and data elements. To support this activity we provide the following data dictionary editor:

New: Starts a new data dictionary.

Add Entry: Inserts a new object in the data dictionary database.

Delete Entry: Removes an existing object from the data dictionary database.

Show DD: Displays the data dictionary on a text window.

Load DD: Copies a specified data dictionary into main storage.

Save DD: Stores an existing data dictionary in a file for future retrieval.

Print DD: Generates a file with a copy of an existing data dictionary and sends that file to the printer.

The Describing Functions Editor

Ultimately, leaf processes in the data flow diagrams must be specified directly as operations. The use of structured English, pseudocode, and/or any other form of textual documentation is recommended. To support this activity, we provide the following describing functions editor:

New: Starts a new describing functions instance.

Add Function: Inserts a new function description in the database.

Delete Function: Removes an existing function from the database.

Show Functions: Displays a specific function in read-only text window.

Load Functions: Copies a specified set of function descriptions into main storage.

Save Function: Stores an existing set of function descriptions in a file(s) for future retrieval.

Print Function: Generate a file with a copy of an existing function description and sends that file to the printer.

Print All: Generate a file with a copy of all function descriptions existing in the database, sorts them, and sends that file to the default system printer.

Reengineering to Increase Maintainability and Enable Reuse

Grady H. Campbell, Jr.

Software Productivity Consortium
2214 Rock Hill Road
Herndon, Virginia 22070

As existing systems are changed to keep up with changing needs, their structure becomes less coherent and cohesive making it difficult and increasingly expensive to make further changes. In addition, as the legacy of complex automated systems grows while the available resources for upgrading or replacing them shrink, there is increasing concern for finding ways to leverage these systems as a base for new or improved existing systems. Reengineering is the concept of creating an improved system by judiciously reorganizing, revising, and extending an existing system. Reuse is a related concept in which a set of existing similar systems provide the basis for a product line of new systems. The Consortium's Synthesis methodology integrates reengineering within the framework of a systematic reuse-driven process that promises more cost-effective development and maintenance of software and systems in the future.

Motivations for reengineering

Reengineering is commonly viewed as a variant of system maintenance. Maintenance differs from other phases of system engineering in that its focus is an operational system that requires modifications to correct errors, to support customer needs more effectively, or to satisfy changed needs. Over its useful lifetime, a system must be repeatedly modified to stay responsive to the needs of the customers it is intended to serve. However, modifying a system in response to changing needs inevitably undermines the conceptual and structural integrity and subsequent modifiability of the system as needs continue to change. Reengineering is distinguished from other forms of maintenance because it presumes the need for a redesign of the existing system to make current and future changes more cost-effective and less error-prone. It is a type of maintenance because the system is not rebuilt from scratch but is derived in large part from the artifacts of the existing system.

Reengineering of a system involves first the analysis of the existing system, referred to as reverse engineering, and then reformulation, restructuring, and modification of the system so that required changes are easier to make reliably. Reengineering may be needed for several reasons:

- Documentation of the system's requirements, design, and implementation has either been lost or become unreliable because of subsequent changes to the implementation. Original needs may not be well understood. This makes it difficult and risky to change the system because of uncertainties in how parts of the system interact or why certain functions behave as they do.
- The needs served by the system have changed sufficiently that the original design is no longer a good solution. A redesign is required for the system to continue to be acceptable to its users.
- The technology upon which the system is based has become obsolete. To accommodate improved technology and better serve user needs, the system must be substantially redesigned.

In the worst case, a system may have all of these problems. Although reengineering can accomplish its intended purpose of creating a better structured, more maintainable system in the short run, it may do nothing to avoid recurrence of the problems that led to the need for reengineering in the first place. If recurrence of these problems is not somehow prevented by the reengineering or avoided in subsequent maintenance of the system, then after some time reengineering will again be necessary. Taking a different view of reengineering can reduce recurrence of these problems.

A framework for reengineering

The driving concern for reengineering is the ability to create a system that can be easily changed as customer needs change. The driving concern for reuse is the need to field multiple systems or system versions that satisfy similar yet differing needs, of one or several customers, without having to repeatedly develop similar software from scratch. In reality, these two concerns are the same:

This material is based in part upon work funded by the Virginia Center of Excellence for Software Reuse and Technology Transfer, sponsored by the Advanced Research Projects Agency under Grant # MDA972-92-J-1018. The content does not necessarily reflect the position or the policy of the U.S. Government, and no official endorsement should be inferred

the ability to produce similar systems, whether serially or concurrently, to satisfy similar needs.

Looking more closely at the possible motivations for reengineering a system reveals several alternative objectives:

- To make changes in the functioning or operational properties (e.g., reliability, performance) of an existing system
- To make it easier or safer to make current and future changes in an existing system
- To use existing systems as a foundation for similar future systems

When reengineering is motivated only by the first of these objectives, the situation is not particularly different from that of conventional development and maintenance. In this case, the objective is most likely addressed adequately by traditional approaches to maintenance in which the architecture of a system is upgraded or particular data structures or algorithms are replaced by improved alternatives. The other two objectives warrant a different approach based, the Consortium believes, on the concept of program families [1, 2].

When the objective of reengineering is either to make a system easier to change or to use legacy systems as a foundation for future systems, Dijkstra's and Parnas' concept of orienting development to a family of systems provides significant opportunities for leverage in comparison to a traditional, single-system orientation. Even a single system inevitably evolves through multiple versions because of poorly understood requirements or to accommodate changing requirements or technology. The Consortium's approach to reengineering is based on families of systems, within the framework of the Synthesis methodology for reuse-driven software processes [3]. Reengineering within a Synthesis process comprises conventional reverse engineering capabilities for the analysis of existing artifacts combined with an innovative reuse-driven approach to creating and using families of systems as a basis for both the development and maintenance of systems.

A reuse-driven software process

An organization's primary motivation for instituting a Synthesis process is that the organization perceives itself as having expertise in and serving the market for a cohesive business area. The market has the need for either a single evolving system or several similar systems, which in either case offers a basis for conceiving a family of systems.

The essence of our approach is that development should result in a family of similar systems from which it is possible to mechanically derive alternative members of the family for rapid delivery to customers. A family is not just an abstract conception but a concrete formulation. It is designed and constructed as the means for systematic production and modification of systems to satisfy diverse or changing needs.

A Synthesis process, as depicted in Figure 1, consists of two major activities: domain engineering and application engineering. These activities, described briefly here, are defined fully and in detail in [4], along with extensive practical guidance.

Application engineering is concerned entirely with the needs of a particular customer and with producing a system that effectively addresses those needs. Application engineering prototypically consists of four subactivities:

- Project management. Planning, monitoring, and controlling an application engineering project to respond to customer needs.
- Application modeling. Formalizing customer needs and analyzing alternative solutions in terms of a set of decisions that are sufficient to distinguish a particular instance of a supported family of systems.
- Application production. Producing a system by means of a prescribed mechanical selection, adaptation, and composition of reusable components, directed by the decisions made in application modeling.
- Delivery and operation support. Installing a system in its operational environment, training users, assisting them in effective system operation, and identifying changes that will make the system a better fit to customer needs.

Domain engineering focuses on how to make application engineering most effective in meeting both the objectives of the business and the needs of the targeted market. To achieve this, domain engineering formalizes a family of systems as a domain by identifying the common and varying features of the type of systems that the market requires. Typically, domain engineering supports multiple application engineering projects. Domain engineering consists of five subactivities:

- Domain management. The planning, monitoring, and control of the domain engineering effort. This encompasses coordination with application engineering project management and concern for all facets of process management including configuration management and quality assurance disciplines.

- Domain definition. Establishing the scope and extent of the domain and formalizing the variabilities that differentiate instances of the targeted family of systems.
- Product family engineering. Formalizing standardized (adaptable) requirements, design, and implementation for the family of systems and all associated deliverable and supporting work products.
- Process engineering. Formalizing a definition of a standardized application engineering process and creating automated support for its performance. The prototypical description of application engineering described above is tailored to suit the specific needs of the domain and associated projects.
- Project support. Assisting application engineering projects to make effective use of the domain. This includes validating whether the domain is responsive to project needs and identifying needed improvements and changes.

A domain is a formalization of a family of systems and an associated process for producing members of the family. A system is represented by a set of artifacts (i.e., work products). A system is not just a collection of implemented (i.e., code) components but includes associated requirements/design/user documentation, test materials, management plans, and any other artifacts that result from development or support the use or maintenance of the system. Synthesis is concerned with producing all of these when a system is needed. Creating a family of systems means creating a representation of each type of relevant artifact as a family in its own right. Furthermore, artifacts may be made up of components which are in turn instances of component families. An essential objective of a Synthesis process is to create a material representation of all necessary system, artifact, and component families.

The essence of a family in this sense is that it represent a set of 'similar' individuals, by which we mean individual things that are identical relative to a specified set of traits. A family is formulated as an abstraction that denotes a set of similar individuals and identifies the particular traits that determine membership in the family. Materially representing a family requires expressing not only the substance of similarity but, equally important, the details of variation (out of which come distinct individuals). Variations are additional traits that together make each of the individual members of a family unique and correspond to decisions that are necessarily deferred until a particular family member (i.e., individual system, artifact, or component) is needed. Production of an individual then reduces to resolving these deferred decisions as needed to designate and mechanically derive the corresponding member of the family.

Methods for creating and using component families are referred to as metaprogramming [5]. A metaprogramming technique specifies how to create a component family and subsequently transform it into concrete

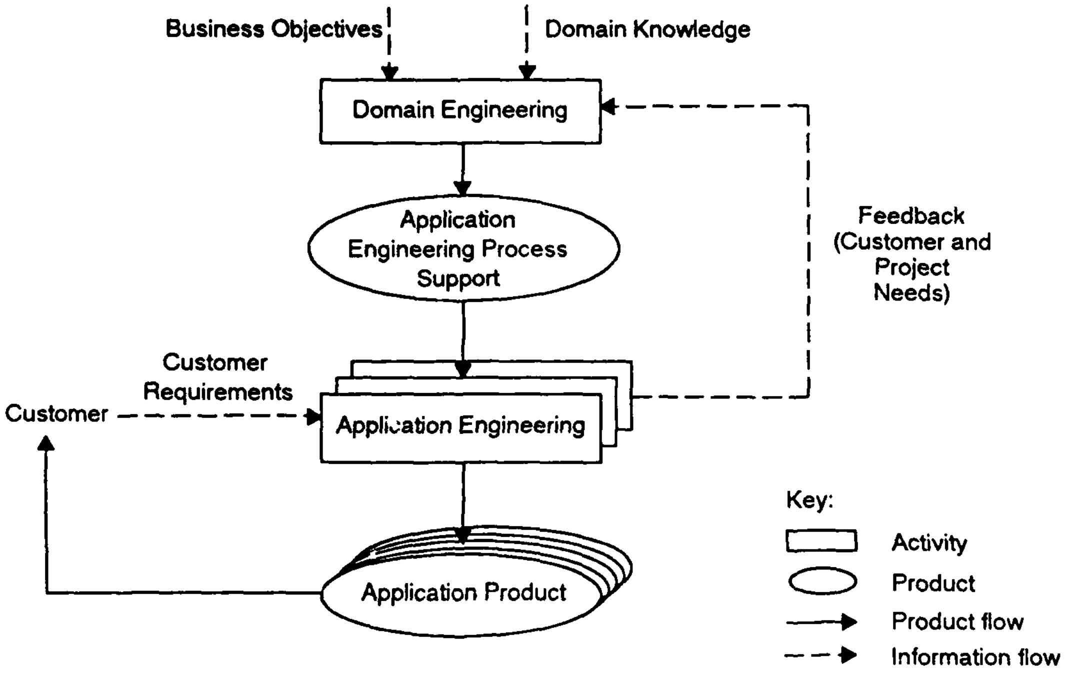


Figure 1. A Synthesis Process

instances. Mechanisms such as C preprocessor constructs, Ada generics, and form-letter capabilities of word processing software have proven sufficient for a viable Synthesis process. Other, special-purpose mechanisms, which are more complete but experimental, may provide additional leverage.

Experience with Synthesis

The Synthesis approach, until now emphasizing reuse with only limited concern for reengineering, has been used extensively by several industrial organizations. Two organizations, in particular, have contributed significantly to understanding Synthesis and how to achieve effective reuse:

- Rockwell Command and Control Systems Division. Rockwell began using Synthesis experimentally in 1990. They have now progressed to the point that they are evaluating its use in support of a substantial business area. Their experience is described in [6].
- Boeing/NAVAIR STARS* demonstration project. Boeing evaluated and selected the Synthesis methodology as the basis for its demonstration of megaprogramming and reuse [7]. This experience is now being transferred into trial use of Synthesis by the Naval Training Systems Command of NAVAIR.

In addition to these two examples, Synthesis is in experimental use on projects in Martin-Marietta, Lockheed, and other organizations. Based on this experience, Synthesis is proving to be a viable and sound approach for systematic reuse-driven software engineering. The experience so far in all of these is that a Synthesis process provides an effective capability for rapidly building multiple systems or system versions and subsequently modifying them as customer needs change. We believe that Synthesis also provides an effective framework for systematic reengineering as an aid to leveraging existing systems in producing new or improved software and systems.

Reengineering within a Synthesis process

An organization institutes a Synthesis process because it has expertise in a targeted business area and intends to serve the associated market. As a rule, requisite evidence of sufficient expertise to justify such a business commitment is that the organization has produced systems for this market in the past. Such legacy systems are a good initial source from which to create a domain as the formalization of a family. For effective use of legacy systems, reengineering is an integral element of the Synthesis process.

Just as the emphasis in Synthesis is on creating a family of similar systems, the result of reengineering within Synthesis should be not just an 'improved' variant of the legacy system(s) but a family of similar systems from which alternative instances of the family can be derived. Derivable instances include alternative systems that are equivalent to an initial legacy system but improved in some way as well as systems that are useful hybrids or modifications of initial legacy systems.

Within Synthesis, reengineering is not viewed as a separate activity. Since a Synthesis process is meant to be a comprehensive engineering process, many of the necessary aspects of reengineering are already a part of the process. Currently, whenever a Synthesis activity involves the creation of a work product, it accommodates the analysis of existing systems as one source of the information in that work product. For example, requirements specifications of legacy systems can be a source for determining how best to express the requirements for the family as a whole. Similarly, test cases used in regression testing of those systems can be a source for creating test cases to be used in testing future systems. Only the use of reverse engineering capabilities need further elaboration as integral elements of Synthesis activities. In large part, this means the enhancement of the product family engineering activity of domain engineering to describe and explain the use of such capabilities.

A family of systems can be derived initially from either a single or several similar legacy systems. Reengineering may be concerned with any and all of the work products associated with a system. When a system is modified, changes are rarely limited to code components; reengineering should facilitate coordinated change across the entire set of artifacts associated with a system, including requirements, design, code, documentation, and test support.

One aspect of a Synthesis process is the design and implementation of component families. This takes the form of reengineering when components are available from legacy systems. Reengineering of legacy components to create a family can start with a bottom-up analysis of similarities among existing components. However, in a Synthesis process, analysis is guided by a top-down specification of component families based on an organization's business objectives. The challenges in creating a viable family by reengineering are to identify components that fit sufficiently within the scope of the envisioned family and to distinguish essential variations among identified components (i.e., driven by sound customer requirements or engineering concerns) from incidental (and therefore unneeded) variations. The leverage from this approach to reengineering comes from

* STARS is the Software Technology for Adaptable Reliable Systems program of the Advanced Research Projects Agency.

recognizing that distinguishable instances can be derived from the unified abstraction of a family.

System reengineering as a generalization of software reengineering

System engineering is concerned with hardware, software, and manual procedures and the interactions among them. Much of the interest in reengineering has focused on software because of the increasing cost of maintenance associated with software changes. However, as defense spending shrinks, the useful life of individual systems grows longer. To respond to new and changing needs, there is a corresponding need and benefit in reengineering complete systems comprising hardware, software, and manual procedures. Reengineering a system can involve coordinated changes to any of:

- The system architecture, including physical and informational connections and the number, identity, and capabilities of the system's hardware and software components
- The design and implementations of individual hardware and software components
- The business/organizational and user processes within which the hardware/software system operates

Another dimension of reengineering at the system level, in contrast with the software level, is that the tradeoff between hardware, software, and manual procedures can be reconsidered. As technology advances, it becomes easier to move software functions into custom hardware. Alternatively, moving a hardware function into software can increase flexibility for modifying it in response to changing needs. Similarly, as system usage matures and manual procedures become more standardized, it becomes feasible to implement more of them in software.

As with software-oriented reengineering, the goal of system reengineering should not be narrowly to reconstruct a system to meet current needs but also to facilitate and reduce the costs of future changes as well. From this perspective, significant leverage arises from considering overall system concerns, as well as those related to each hardware, software, and procedural component of a system, within a reengineering approach. Furthermore, the similarity of motivations for reuse and reengineering justifies a unified approach for systems as well as for software.

Issues in reengineering

Most of the same issues that make system engineering a complex task, such as:

- Understanding the real requirements for a system so that effort is not wasted solving the wrong problem
- Evaluating alternative solutions and making engineering tradeoffs to attain a proper balance among system properties such as performance, reliability, development costs, and operating costs
- Verifying process performance and intermediate work products and validating the final product to ensure that the problem has been solved properly and correctly

are similarly a concern for reengineering. In addition to these common concerns, reengineering raises additional issues, specifically an extended verification problem and a deoptimization problem in reverse engineering. These problems are inherent to reengineering, regardless of approach.

Whenever a system is constructed, it must be validated to determine whether it satisfies the customer's actual needs. Because in the context of reengineering an operational system already exists, it is reasonable to expect that validation reduces to a problem of verifying the replacement system as the equivalent of the existing system. When the replacement system is supposed to have identical functionality to the current system, this equates to a total regression test of the replacement system. However, creating a replacement system with identical functionality is seldom feasible or necessary; creating only near-identical functionality is usually sufficient and less costly. Unfortunately, a divergence from identical functionality makes regression testing much more difficult. When, as is often the case because of changed needs, the replacement system also must differ in certain functionality from that of the current system, the problem takes on the characteristics to a greater or lesser degree of a complete revalidation. For reengineering to be practical, the effort of not only development but of verification and validation as well must be significantly reduced as compared to that of completing the system from scratch.

Reengineering generally requires the reverse engineering of a system for recovery of missing or obsolete design information or to establish precise, as-implemented requirements. Unfortunately, particularly in the case of software, the as-implemented structure is often an optimized equivalent of the intended design. For example, real-time embedded software usually entails responding to asynchronous events in the environment; logically, this corresponds to an architecture consisting of multiple concurrent processes. However, to satisfy stringent performance constraints, such an architecture has traditionally been implemented in a cyclic executive in which the logic of the processes is interleaved in a nonobvious fashion. Trivial reverse engineering would not reveal the true logical structure of the software but instead would describe a much more complex linear structure. Reverse engineering techniques must be developed that help discover the original requirements and design while recognizing that the implementation is actually an optimization.

Conclusions

The Synthesis methodology for domain-specific software development offers a comprehensive framework for a reuse- and reengineering-based approach to revitalizing existing operational systems and producing new, more maintainable systems. Issues remain in the specific methods and technologies of reengineering, reuse-driven product lines, and process automation. However, based on extensive trial use by industry and government, the essential process framework is sound. Further work will demonstrate the benefits of taking such a product line perspective whether the motivation is to reduce the costs of new development or the costs of maintenance and whether the focus is on software or on systems.

Acknowledgments

Rich McCabe, Steve Wartik, and Roger Williams each provided helpful comments that greatly improved this paper.

References

[1] E.W. Dijkstra. "Notes on Structured Programming." *Structured Programming*, O.J. Dahl, E.W. Dijkstra, and C.A.R. Hoare, Eds. Academic Press, London, 1972, pp. 1-82.

[2] David L. Parnas. "On the Design and Development of Program Families." *IEEE Trans. Software Eng.*, SE-2, 1976, pp. 1-9.

[3] Grady Campbell, Stuart Faulk, and David Weiss. *Introduction to Synthesis,* INTRO_SYNTHESIS_PROCESS-90019-N. Software Productivity Consortium, Herndon, Va., 1990.

[4] Software Productivity Consortium. *Reuse-Driven Software Processes Guidebook*, SPC-92019-CMC. Software Productivity Consortium, Herndon, Va., 1993.

[5] Grady Campbell. "Abstraction-Based Reuse Repositories." *AIAA Computers in Aerospace VII Conference*, Monterey, Ca., 1989, pp. 368-373.

[6] James O'Connor, Catharine Mansour, Jerry Turner-Harris, and Grady Campbell. *Exploring Systematic Reuse for Command and Control Systems*, SPC-92020-CMC. Software Productivity Consortium, Herndon, Va., 1993.

[7] B. Freemon. *STARS PSA S01 Experience Report*, D495-20154-1. The Boeing Company. 1993.

A Reuse Approach To Computer-Assisted Software Reengineering*

Daniel E. Wilkening
Joseph P. Loyall

TASC
Reading, Massachusetts 01867

Marc J. Pitarys
Kenneth Littlejohn

USAF Wright Laboratory
Wright Patterson AFB, Ohio 45433

Abstract

The United States Air Force's Wright Laboratory and TASC are developing an environment for the reengineering of software from one language to another. Our approach engineers a program in the new language by reusing portions of the original implementation and design. We use reverse engineering to facilitate understanding, design recovery, viewing, and navigating of the subject system. We use computer-assisted restructuring to aid the engineer in developing a program using design and implementation information recovered from the subject system. We use automatic translation of low-level program statements to free the engineer from the tedium associated with syntactic differences between languages. This paper describes our reengineering process model, the design of our reengineering environment, and the current state of the implementation.

1 Introduction

The reengineering of software from one language to another is becoming a necessity as Department of Defense organizations strive to modernize and improve the maintainability of their systems while avoiding the excessive costs of new development. Systems that have been in use for years often incur large maintenance costs [6] for a number of reasons, including:

- Continual maintenance has made the current implementation and original design inconsistent, made the code harder to understand and error-prone, and made the documentation out-of-date.
- They are written in languages that, while once popular, have fallen out of favor. The limited selection of support tools for these languages, the corresponding expense of these tools, and the shrinking pool of qualified programmers to maintain the software adds to the expense of maintenance.
- They were developed without modern software engineering practices, resulting in code that, by today's standards, lacks structure and is difficult to understand.
- Employee turnover has reduced the amount of understanding and "intimate" knowledge of the system.

TASC, under the auspices of Wright Laboratory, is currently developing an environment for reengineering software from one language to another as part of the Avionics Software Reengineering Technology (ASRET) project. We are initially concentrating on the reengineering of avionics simulation software written in FORTRAN to Ada, but the environment is designed so that additional languages can be supported in the future.

Under the ASRET project, we have performed an extensive investigation of existing reengineering ınd reverse engineering processes, techniques, and tools [23]. Based upon this study, we have developed a process model that defines reengineering in terms of (non-destructively) engineering a new program by reusing the design and implementation of the original program. The process model is consistent with well-accepted reengineering models [4, 5], and improves on them by dividing automated restructuring from restructuring that requires human insight and by defining restructuring tasks in terms of modern software engineering practices.

We have designed and are currently implementing a reengineering tool (RET) that automates portions of the process model an⁻ⁱ incorporates selected techniques from the study. With our system, the engineer non-destructively *develops* a new Ada program by *reusing* parts of the original FORTRAN design and implementation, as opposed to *changing* the original FORTRAN into Ada. For example, an engineer can run an automatic packaging routine that extracts FORTRAN subprograms, translates their declarations into Ada, and arranges them into packages based upon their data usage. The engineer can then rearrange the resulting Ada subprograms interactively. When satisfied with the package structure, the engineer can direct the system to automatically translate statements in the bodies of the subprograms. We are currently implementing the RET.

Most existing reengineering tools fall into one of two categories:

*This work is sponsored by the Avionics Directorate of Wright Laboratory under Contract #: F33615-92-D-1052.

- Reverse engineering and redocumentation tools [8, 23, 20] that present different views of the structure of a program, such as control flow and data flow graphs, to aid in program understanding and manual reengineering.

- Other tools [18, 24] support automatic translation from one language to another or forms of automated restructuring, such as the removal of GOTOs. These tools require little human interaction but, because of this, provide little support for design recovery or improvement.

Our approach can be described as computer-assisted reengineering. It provides automated reverse engineering, redocumentation, and translation of low-level program entities, but also provides a combination of user interaction and automated analysis to reorganize program statements and data into new modules. The RET will relieve the tedium associated with syntactic minutia, i.e., differences between the source and target programming language syntax, and allow the engineer to concentrate on the more important design and implementation decisions that will make the reengineered system more maintainable.

The rest of this paper is structured as follows: Section 2 introduces our reengineering process model and compares it with existing reengineering process models. Section 3 describes the RET design. Section 4 describes the current state of the RET implementation. Section 5 presents some concluding remarks.

2 The Reuse-Based Reengineering Process

Our reengineering process model as applied to the RET domain is illustrated in Figure 1. Steps in the process label the boxes in the figure and inputs and outputs for each step label the icons between boxes. The process model specifies a set of tasks (the steps of the process) that should be performed and the sequence in which they should be performed to reengineer a program in one language, e.g., FORTRAN, to another language, e.g., Ada. The source and target languages can be the same if the goal of reengineering is simply to improve the structure of the program without moving to a different language. The process model also specifies the information necessary and desirable to support these tasks. The process model does not specify how the tasks are to be performed, i.e., they might be automated (as many are in the RET) or they might be performed manually.

The first step in the process model is to perform some ***preliminary restructuring*** of the source code of the original implementation. Preliminary restructuring improves the layout of the source code by removing unstructured program constructs, such as GOTO statements, dead code, and implicit types. Preliminary restructuring is separated from the later restructuring step because it can be completely automated by commercial tools, and placed first in the process model because the structured version of the source program is usually easier to analyze, understand, and restructure.

After preliminary restructuring is complete, the improved source code is ***analyzed*** and representations of the program are constructed. Some of the representations, e.g., abstract syntax graphs (ASGs) and symbol tables, are machine-readable representations used only by automated restructuring and redesign tasks. Others, e.g., flow graphs and structure charts, aid in program understanding, redocumentation, and manual restructuring. For manual restructuring, the set of representations will certainly contain a source code listing.

The *restructuring*, *redesign*, and *redocumentation* steps of the process model are performed multiple times, each time building upon the results of the previous pass. A multi-pass approach is necessary because, in programs of reasonable size, it is easier and less error-prone to reengineer a program in stages, verifying the program after each pass. ***Restructuring***, i.e., changing the structure of the program without changing its functionality, should be performed first, possibly in several passes. These passes should perform the following steps:

- *Macro control restructuring* - Grouping statements and control structures of the program into modules, such as procedures, functions, and packages. This includes recovering modules of the original program, generating new modules, and specifying a declaration nesting structure for modules.

- *Macro data restructuring* - Grouping of data items, such as types, variables, and constants, and associating them with modules created during macro control restructuring. This includes recovering data groupings of the original program, creating new groupings, and creating abstract data types and records.

- *Micro control restructuring* - Manipulation of individual control structures. This includes the translation of individual statements and functionality-maintaining alterations, such as code lifting [1].

- *Micro data restructuring* - Manipulation of individual data items. This includes actions such as translation, changing names, changing types, creating symbolic constants, and changing the scope of variables.

Macro control and data restructuring should be performed first to develop a modular structure for the target system, followed by micro control and data restructuring to restructure individual components of the program.

After restructuring is complete, code in the target language should be generated and the program should be tested to ensure that the restructuring did not introduce any errors or undesired functional changes. The test data of the original program can be used and the results compared with the results of testing the original program. In many cases, the test data will need to be reengineered to work with the reengineered program. Any differences in the results of testing will

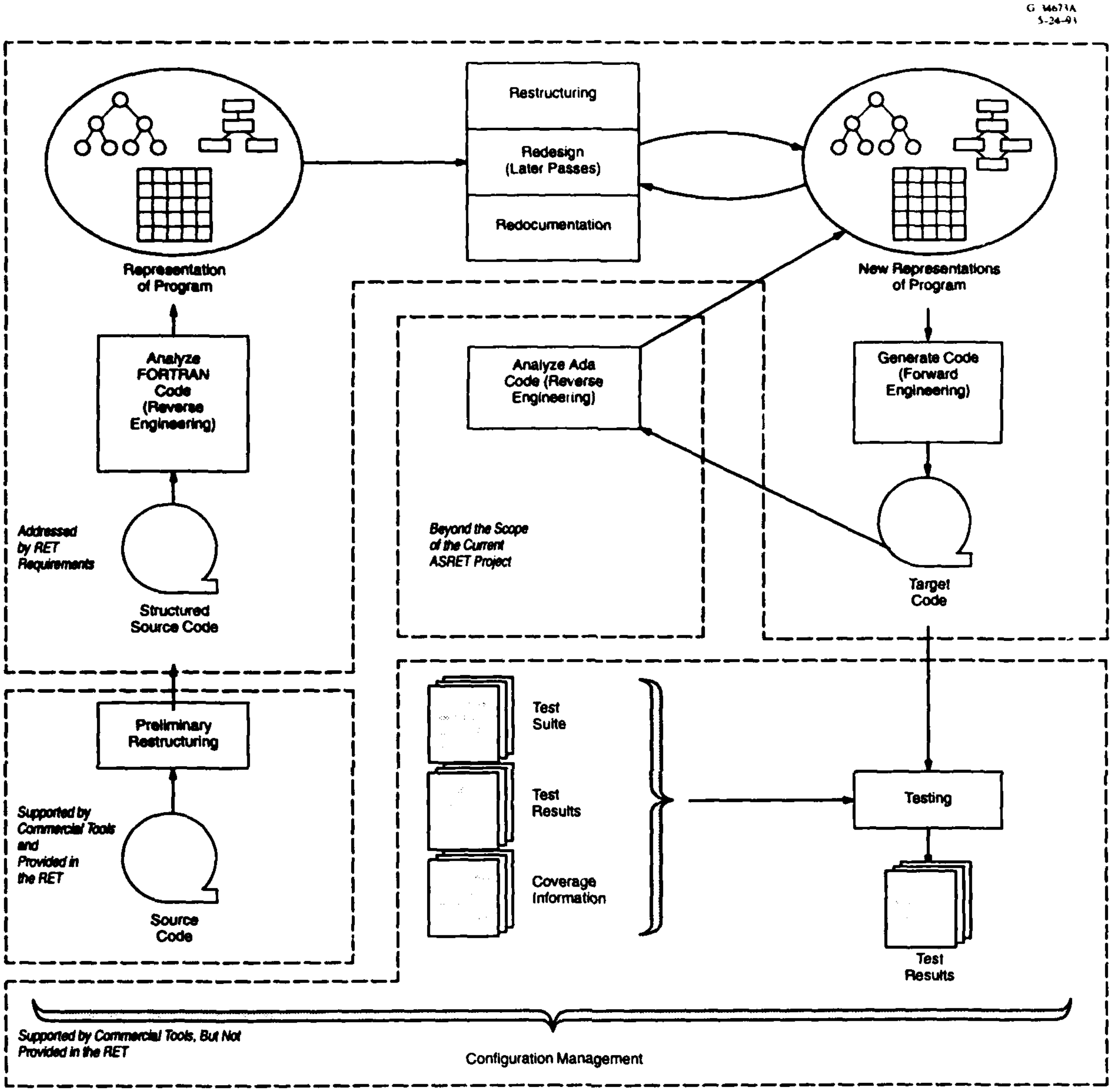

G M673A
5-24-91
Restructuring
Redesign (Later Passes)
Redocumentation
Representation of Program
New Representations of Program
Analyze FORTRAN Code (Reverse Engineering)
Analyze Ada Code (Reverse Engineering)
Generate Code (Forward Engineering)
Addressed by RET Requirements
Beyond the Scope of the Current ASRET Project
Structured Source Code
Target Code
Preliminary Restructuring
Supported by Commercial Tools and Provided in the RET
Source Code
Test Suite
Test Results
Coverage Information
Testing
Test Results
Supported by Commercial Tools, But Not Provided in the RET
Configuration Management

Figure 1: **The reengineering process model.**

indicate the introduction of an unexpected functional change during restructuring. Coverage analysis should be performed during the testing of the target code because restructuring could have introduced or altered control and data characteristics of the program. When an error in the target program is indicated, the program can be corrected by amending the target code directly or by restructuring the representations and regenerating the target code.

Once the program has been restructured and a functionally equivalent program in the target language has been created, the engineer can perform additional restructuring and redesign actions on the program. These steps use the same set of actions, i.e., macro control, macro data, micro control, and micro data, but have different goals. Further restructuring is performed to further improve the structure of the program. Redesign has the goal of changing the functionality of the program, e.g., to correct design flaws or improve the design. If the target program code was edited to correct errors indicated during testing, the code is analyzed to generate representations before subsequent restructuring and redesign is performed.

Redocumentation is performed simultaneously with the restructuring and redesign steps, e.g.,

- The generated representations can be saved and serve as documentation of the program structure and design.

- The engineer can add comments and annotations during restructuring and redesign as he gains insights about the code or design.

The RET reengineering process model includes modern software development processes, such as continuous testing, iterative restructuring and redesign, and configuration management. The process model is a specialization of the Chikofsky-Cross process model [5, 8]. The entire Chikofsky-Cross model is represented, although there are the following changes:

- Inclusion of program management extensions to the process model [4], such as configuration management and testing.
- Separation of easily automated steps, such as preliminary restructuring, so they can be addressed by commercial tools.
- Decomposition of Chikofsky-Cross steps, such as restructuring being decomposed into macro control, macro data, micro control, and micro data restructuring.
- Explicit introduction of the iteration steps that are implicit in the Chikofsky-Cross process.

3 RET Design Overview

The Reengineering Tool (RET) assists the software engineer in understanding and improving the structure of an avionics software support system while translating its source code from the source programming language (FORTRAN) to the target programming language (Ada).

The RET comprises two distinct logical parts called the Left-Hand Side (LHS) and the Right-Hand Side (RHS). The LHS provides views of the original FORTRAN program, or *subject system*. The RHS provides views of the Ada program being developed, i.e., the *target system*. The LHS allows the engineer to navigate and view aspects of the subject system, but does not support changing the subject system. The RHS supports constructing, refining, viewing, and navigating the target system. The engineer *constructs* a basic structure for the RHS (macro restructuring) using information extracted from the LHS. Once the basic structure of the RHS is established, the engineer *refines* the target system (micro restructuring) on the RHS.

Semi-automated RET components support construction activities; they suggest large-scale reorganizations of the subject system and populate the RHS with the basic structure of the target system. The components that support refinement allow the engineer to apply knowledge, which is beyond the RET, and human insight, which is lacking in the semi-automated support provided by the RET, to modify and improve the RHS representations.

3.1 Representations

The RET provides two primary internal representations (PIRs): the Abstract Syntax Graph (ASG) and the Symbol Table (ST). Secondary internal representations (SIRs) are derived from the PIRs. The SIRs are the underlying data structures for the views presented to the engineer. The RET provides six views for each side:

- *Source Code Listing (SCL)* - displays the FORTRAN or Ada source code
- *Declaration Diagram (DED)* - displays the program declaration nesting structure
- *Call Diagram (CD)* - displays the subprogram calling structure
- *Data Flow Diagram (DFD)* - displays how data flows through the program
- *Hypertext Annotations (HA)* - displays trees and networks of textual commentary provided by the engineer
- *Data Dictionary (DD)* - displays various cross references

3.2 The RET Architecture

Figure 2 shows the RET architecture. It depicts the organization of major RET components, and indicates the data flow relationships among them. The Preliminary Restructurer (PR) performs control flow restructuring. The Source Code Processor (SCP) generates the LHS PIRs.

The engineer constructs the RHS PIRs using the Packager (PACK) and Transformer (TRAN). The Representation Generator (RG) creates the SIRs on both sides from the PIRs. The User Interface and Display (UID) creates the corresponding views, and provides the means by which the engineer interacts with the views on both sides and alters the views on the RHS. The Transformer implements the changes by transforming the RHS PIRs, and the Representation Generator propagates the changes to the RHS SIRs. The User Interface and Display refreshes the RHS views in response to the changes.

The File System Interface (FSI) manages external persistent data and the Object Base (OB) manages internal data.

The views, PIRs, and SIRs are thus interdependent, but the engineer need not be aware that the PIRs and SIRs exist. Any changes that the engineer makes to the target system through the views provided by the User Interface and Display may appear to affect the views exclusively. The components are described in more detail below.

Preliminary Restructurer. The Preliminary Restructurer (PR) restructures the control flow of the original FORTRAN source code by eliminating branches into or out of loops and decisions. It eliminates all GOTO statements, leaving only the pure structured programming constructs: sequence, selection, and iteration. We refer to this specialized form

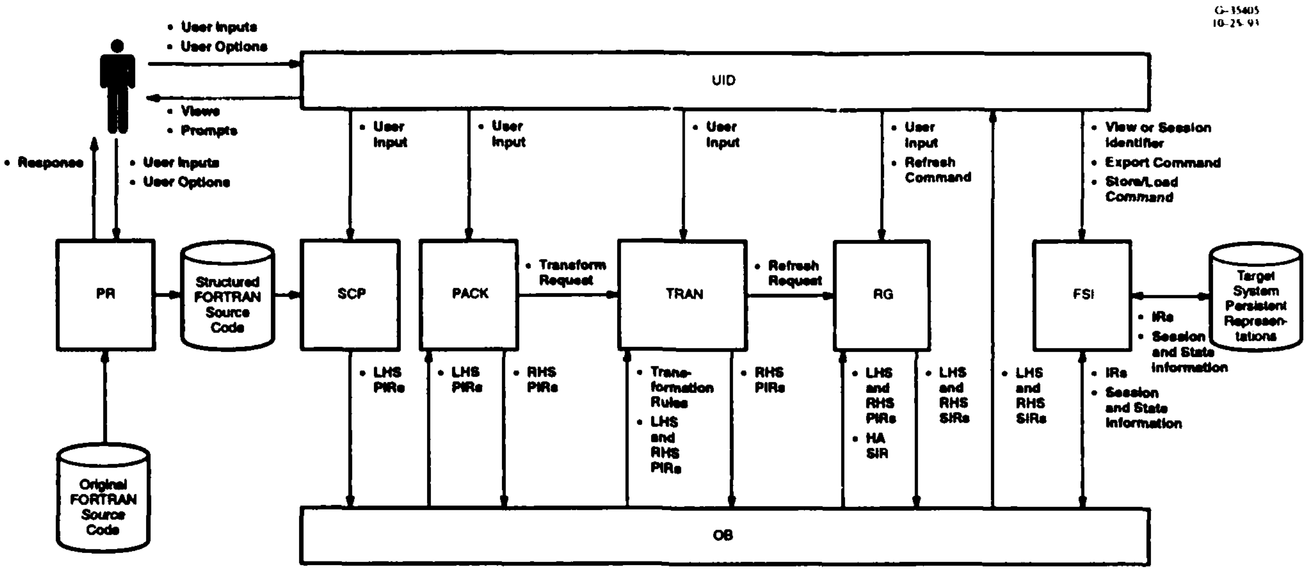


Figure 2: **The RET architecture.**

of restructuring as control flow restructuring to distinguish it from the more general concept of restructuring described in Section 2.

The Preliminary Restructurer is applied as a preprocessing step and the RET assumes, but does not require, that the FORTRAN source code has already undergone control flow restructuring. There are two reasons for this design. The first is that the subject system is not always poorly structured with respect to control flow, so the step should be optional. The second reason is that the design allows any control flow restructuring product to be used without integrating it into the RET.

Source Code Processor. The Source Code Processor (SCP) reads the FORTRAN source code, performs analyses, and generates the PIRs. The PIRs represent the structure of the program, semantic information about program components, and data flow information.

The Source Code Processor will also detect apparent undefined-reference (U-R) data flow anomalies (DFA) on the LHS by static analysis of the LHS. A U-R/DFA is an occurrence of a variable binding which is referenced before it is assigned a value. An instance of this involving some local VAX/FORTRAN variable in a subprogram is called an apparent U-R/DFA because it is not necessarily a data flow anomaly, even though it may be detected as such by the static analysis. The (VAX/FORTRAN) variable enjoys static extent [21], and may have been assigned a value on a prior call of the subprogram. Ada variables have automatic extent [21], so it is important for the RET to identify these situations and promote the variables to an appropriate enclosing scope.

Representation Generator. The Representation Generator (RG) creates the LHS and RHS SIRs. The Representation Generator generates each SIR when its corresponding view is requested by the engineer. Once the RHS SIRs are created, they may become inconsistent with the PIRs as the latter are restructured. The engineer can request a refresh, which regenerates the SIRs from the current PIRs or an environment variable can be set so that SIRs are refreshed periodically.

The Representation Generator creates SIRs for the DFD, CD, and DED views. The Representation Generator produces the DFD according to a method for creating Hierarchical Data Flow Diagrams given in [2]. It generates the CD in a straightforward manner from information in the ST. The DED is a canonical organization of information in the ST.

The Representation Generator does not create SIRs for the SRC or HA views. The SRC view is produced by the DIALECT [12] printer. Hypertext annotations are provided by the engineer during use of the RET. The RET allows the engineer to enter textual comments, i.e., *annotations*, describing insights, recovered design information, or any other information. The engineer can associate each annotation with any part of the views, PIRs, or SIRs in a hypertext network. The HA is maintained by the Representation Generator and can be incorporated as in-line comments or notes during code generation or redocumentation by the File System Interface.

Restructurer. The Restructurer (RES) component comprises the Packager (PACK) and the Transformer (TRAN). The Restructurer helps the engineer develop an Ada program on the RHS by transforming and reusing components of the original FORTRAN program from the LHS. The Packager assists the engineer with macro restructuring. The Transformer supports both macro and micro restructuring.

All restructuring activities performed by the engineer using the Restructurer manipulate the PIRs exclusively, by initially creating RHS symbol tables and abstract syntax graphs and then by populating and transforming them. The information is entered by the engineer through the views and, once the PIRs have been transformed, the SIRs are regenerated and the views are refreshed. Thus, the underlying PIRs and SIRs are hidden from the engineer and it appears as if the views are being transformed directly.

Packager. The Packager constructs or initializes the RHS ASG. It recognizes subprogram, object, and type entities from the LHS PIR and requests the Transformer to transform them from the LHS domain model to the RHS domain model, and to insert them into the RHS ASG. (A domain model [3] is a kind of object-based database schema.) The Packager builds an Ada ASG on the RHS and calls upon REFINE/Ada [15, 16] for semantic analysis.

The Packager groups the entities into Ada packages on the RHS by applying an interactive clustering technique based upon [7, 9, 10, 19]. The clustering technique provides only a first approximation to a reasonable Ada package structure. We expect that the engineer will need to interactively refine the generated grouping.

Transformer. The Transformer assists the Packager with both macro and micro restructuring. For macro restructuring, the Transformer automatically transforms low-level entities from the FORTRAN domain model to the Ada domain model by applying rules that insert subgraphs and other information into the Ada PIRs corresponding to subgraphs and information in the FORTRAN PIRs. For micro restructuring, the Transformer implements changes to low-level Ada entities by allowing the engineer to select a portion of the Ada program under development and change it by applying a rule or by editing, deleting, or inserting text.

The engineer may indirectly provide input to the Transformer by selecting portions of a view and then interacting with the view to change its SIR. The Transformer implements the change by transforming corresponding portions of the PIRs while the Representation Generator propagates the results of the transformation to the other SIRs. The User Interface and Display updates the views so that it may appear to the engineer that the change was made directly to the selected view.

User Interface and Display, File System Interface, Object Base. The RET provides two external interfaces. The engineer communicates with the RET through the User Interface and Display (UID). The UID shows the views, prompts the engineer for input, receives commands and selections from the engineer, and delivers them to the other components.

The File System Interface (FSI) is responsible for the storage and retrieval of persistent data. The FSI inputs the (FORTRAN) source code of the subject system, reads and writes intermediate data, and outputs the (Ada) source code for the target system and other subject and target system views. The intermediate data is stored in the Object Base (OB).

4 Implementation

4.1 RET Development Environment

We are developing the RET on a SPARCstation 10/40 under Sun OS 4.1.3. The RET utilizes several commercial tools, including the following tools by Reasoning Systems:

- The REFINE language [14] - a high-level language that includes object-oriented, rule-based, and iterative characteristics. It includes an object database and an environment that facilitates interactive development and testing.
- DIALECT [12] - A tool for building parsers and code generators.
- INTERVISTA [13] - A tool for building user interfaces, including mouse-sensitive windows and menus.
- REFINE/FORTRAN [17] - A tool that parses and analyzes FORTRAN code. It includes a printer for FORTRAN that generates source code from a symbol table and abstract syntax graph.
- REFINE/Ada [16] - A tool that parses and analyzes Ada code. It includes a printer for Ada that generates source code from a symbol table and abstract syntax graph.
- A number of unsupported systems that can be used in the RET development, such as hypertext and fast dump/load facilities.

4.2 Implementation of RET Components

We are implementing the components of the RET as follows:

- **Preliminary Restructurer** - The SPAG component of the plusFORT product [11] will provide the entire Preliminary Restructurer.
- **Source Code Processor** - REFINE/FORTRAN provides most of the functionality for the Source Code Processor component of the RET. We have made slight extensions to REFINE/FORTRAN to gather and summarize information needed by other RET components. REFINE/FORTRAN also provides most of the LHS PIRs.
- **Representation Generator** - We are implementing the RG component in the REFINE language, using the REFINE Object Base for the SIR structures. REFINE/FORTRAN and REFINE/Ada provide the Source Code Listing SIRs for the LHS and RHS, respectively.

- **Restructurer** - We are developing the RES component using the REFINE language. The Restructurer will extract information from the LHS PIRs either interactively or automatically, and apply REFINE transformations to generate the RHS PIRs.

- **User Interface and Display** - We are developing the UID using INTERVISTA. INTERVISTA provides support for developing mouse-sensitive windows, menus, and diagrams. It is based upon Allegro Common Windows.

- **File System Interface** - REFINE provides facilities for saving the state of a session and for storing some of the representations to disk. We are developing additional code to save the SIRs to disk.

- **Object Base** - The object base is provided by the REFINE Object Base.

4.3 Status of the RET Implementation

We have developed and tested most of the Source Code Processor, Representation Generator, and File System Interface components. We are developing the associated User Interface and Display facilities along with each component. We have just begun to implement the Restructurer component. We expect to complete the first prototype/demonstration system by mid-Spring 1994, after which development will continue.

5 Conclusions

We discovered during a literature and tool survey that most existing reengineering tools are limited to reverse engineering and redocumentation. Language translation tools have been around for some time, but they are not adequate for reengineering [22]. Few tools provide automated help with restructuring and forward engineering, although techniques for these exist.

We have developed a reengineering process model that:

- Is consistent with previous reengineering process models

- Considers software life cycle issues such as configuration management and testing

- Separates completely automated restructuring (i.e., control flow restructuring) that can be performed as preprocessing from restructuring and redesign that requires human intervention and input

- Promotes structured programming techniques, i.e., macro restructuring preceding micro restructuring

- Promotes development and testing of a functionally equivalent program before undertaking design changes that might introduce errors, i.e., restructuring preceding redesign.

We are developing a reengineering tool (RET) that implements parts of the process model. The RET will assist in reengineering avionics simulation support software written in FORTRAN to Ada. The RET emphasizes increasing the maintainability of the software, preserving its functionality, and improving the structure of the code. The RET does not concentrate on configuration management, testing, or preliminary restructuring because there are many commercial tools that can be used in conjunction with the RET to provide those capabilities.

The RET will relieve the engineer from syntactical minutia, i.e., differences between the source and target programming language syntax, that divert attention from the more important design and implementation decisions requiring human judgement. We believe that by concentrating on tasks that are well-suited to automated support, the RET will reduce the resources needed to reengineer avionics support software and will help the human engineer produce a more maintainable system.

References

[1] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers - Principles, Techniques, and Tools.* Addison-Wesley, Reading, MA, 1988.

[2] P. Benedusi, A. Cimitile, and U. De Carlini. A reverse engineering methodology to reconstruct hierarchical data flow diagrams for software maintenance. In *Proceedings of the Conference on Software Maintenance*, pages 180–189, Miami, Florida, October 1989.

[3] S. Burson, G.B. Kotik, and L.Z. Markosian. A program transformation approach to automating software re-engineering. In *Proceedings of the IEEE Computer Society's International Software and Applications Conference*, pages 314–322, 1990.

[4] E.J. Byrne and D.A. Gustafson. A formal process model for software re-engineering: The analysis phase. Technical Report TR-CS-91-12, Kansas State University, November 12 1991.

[5] E.J. Chikofsky and J.H. Cross II. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, pages 13–17, January 1990.

[6] T.A. Corbi. Program understanding: Challenge for the 1990s. *IBM Systems Journal*, 28(2):294–306, 1989.

[7] D. Hutchens and V.R. Basili. System structure analysis: Clustering with data bindings. *IEEE Transactions on Software Engineering*, SE-11(8):749–757, August 1985.

[8] J.H. Cross II, E.J. Chikofsky, and C.H. May Jr. Reverse engineering. *Advances in Computers*, 35:199–254, 1992.

[9] H.A. Muller, M.A. Orgun, S.R. Tilley, and J.S. Uhl. A reverse engineering approach to subsystem structure identification. Submitted for publication.

[10] H.A. Muller and J.S. Uhl. Composing subsystem structures using (K,2)-partite graphs. In *Proceedings of the Conference on Software Maintenance*, San Diego, California, November 26-29 1990.

[11] Polyhedron Software, Oxfordshire, UK. *plus-FORT Reference Manual, Revision B.* U.S. Distributor: OTG Systems, Inc., Clifford, PA.

[12] Reasoning Systems, Inc., Palo Alto, CA. *DIALECT User's Guide.*

[13] Reasoning Systems, Inc., Palo Alto, CA. *INTERVISTA User's Guide.*

[14] Reasoning Systems, Inc., Palo Alto, CA. *REFINE User's Guide.*

[15] Reasoning Systems, Inc., Palo Alto, CA. *REFINE/Ada Programmer's Guide.*

[16] Reasoning Systems, Inc., Palo Alto, CA. *REFINE/Ada User's Guide.*

[17] Reasoning Systems, Inc., Palo Alto, CA. *REFINE/FORTRAN User's Guide.*

[18] J.M. Scandura. Cognitive approach to systems engineering and re-engineering: Integrating new designs with old systems. *Software Maintenance: Research and Practice*, 2:145–156, 1990.

[19] R.W. Schwanke. An intelligent tool for re-engineering software modularity. In *Proceedings of the 13th International Conference on Software Engineering*, pages 83–92, Austin, Texas, May 13-17 1991.

[20] C. Sittenauer, M. Olsem, and D. Murdock. Re-engineering tools report. Technical report, Software Technology Support Center (STSC), Hill Air Force Base, Utah, July 15 1992.

[21] William M. Waite and Gerhard Goos. *Compiler Construction.* Springer-Verlag, New York, 1984.

[22] Richard C. Waters. Program translation via abstraction and reimplementation. *IEEE Transactions on Software Reengineering*, SE-14(8):1207–1228, August 1988.

[23] D.E. Wilkening, R.J. Kreutzfeld, and J.P. Loyall. Avionics software re-engineering technology (ASRET) software re-engineering study report. Technical Report TR-6661-1, TASC, Reading, Massachusetts, February 17 1993.

[24] Xinotech Research, Inc., Minneapolis, Minnesota. *The Design of the Xinotech Language Translator - Jovial to Ada*, second revision edition. Xinotech Technical report XRI 8911_04.

Formal Specification and Software Reuse in Reengineering Embedded Real-Time Systems

Farnam Jahanian
Department of EECS
University of Michigan
Ann Arbor, MI 48109-2122
e-mail: farnam@eecs.umich.edu

Abstract

With the increasing reliance on computer control of embedded real-time systems in diverse civilian and military applications such as avionics, air-traffic control, patient monitoring, and automated manufacturing, the problem of re-engineering an *aging* software base will confront a growing number of organizations in this decade. Facing this challenge is particularly important because of the economic and security ramifications of maintaining complex embedded systems that operate under increasingly strict dependability and timing requirements. After discussing some of the trends in the field of computing that have directly influenced the changing requirements on the existing systems, this paper presents a re-engineering approach based on the formal specification of a system through decomposition into a collection of services with well-defined interfaces. An approach based on formal specification nicely complements the development of a toolkit of common services that can be reused in re-engineering multiple embedded systems. Precise specification of re-engineered components also supports automated tools for testing, fault-injection, verification and run-time monitoring.

1 Introduction

Complex embedded real-time systems are being used in diverse applications such as avionics, air-traffic control, automated manufacturing, and patient monitoring. With the increasing reliance on digital computers in monitoring and controlling embedded real-time systems, both industry and government organizations are faced with the problem of extending and modifying an "aging software" base. Many software systems that are pivotal to the operations of commercial and defense systems are becoming more and more costly to operate [7]. Each change to the software results in one or more patches that contribute to a system that is often less efficient, less reliable, and more difficult to maintain. Consequently, *re-engineering* existing software is a significant growing challenge that confronts us in the 1990s and beyond.

Re-engineering, also called renovation or reclamation in the literature [1], refers to the recovery of design information and its uses to alter or reconstitute the existing system to improve its quality or to meet new requirements. The increasing importance of re-engineering has several economic, security and technological ramifications:

- The economic competitive in a global market is directly linked to the thousands of aging information system and engineering applications that are being used across large and small companies.
- Embedded real-time systems often have strict dependability and timing requirements. They often control and monitor defense systems as well as commercial safety-critical operations. Interruptions in a timely delivery of services may have significant national security or economic implications.
- The existing software base is the infrastructure on which future technology is built. Achieving a technological edge in the next decade is dependent on the strength of this infrastructure.

This position paper argues that the decomposition of a system into a collection of services (or building blocks) with precise specifications and well-defined interfaces may be the key to re-engineering complex embedded systems. [1] The paper further advocates a toolkit approach to re-engineering of these systems: By identifying a collection of distributed services common to a large class of embedded real-time systems, these building blocks can be reused to reduce the complexity and the cost of modifying and extending large systems. We illustrate one such toolkit that was developed at IBM Research to support distributed fault-tolerant systems. Finally, the paper argues that an approach based on formal specification and precise interface definitions also facilitates the development of automated tools that aid in demonstrating that a re-engineered system meets the new requirements. A brief discussion of a suite of tools, currently under development, for testing,fault-injections, verification and run-time monitoring of embedded real-time systems is also presented.

2 Distinguishing Characteristics of Embedded Real-Time Systems

Embedded real-time systems often interact with the external environment and operate under strict timing and dependability requirements. As shown in Figure 1, an embedded real-time system can be decomposed into three components: the *controlled object*, the *computer system*, and the *operator*. The controlled object and the operator are the *environment* of the system. The interface between the real-time computer system and the controlled object is called the instrumentation interface, consisting of sensors and actuators. The interface between the computer system and the operator is called the man-machine (or the operator) interface. The operator monitors and controls the object via this interface to the computer system. Embedded real-time systems are in essence *responsive*: they often interact with the environment by "reacting to stimuli of external events and producing results, within specified timing constraints" [3]. To guarantee this responsiveness, the system must be able to tolerate failures. Hence, a fundamental requirement of fault-tolerant real-time systems is that they provide the expected service in a timely manner even in the presence of faults.

The above description depicts a real-time computer system as a single entity. With the introduction of inexpensive microprocessors and dense memories in recent years, a rapid shift from large centralized computer systems to clusters of closely-coupled nodes has taken place. In fact, most real-time computer control systems are now distributed (Figure 2) in the sense that they consist of a set of nodes interconnected by a real-time communication subsystem. Conceptually, a real-time computer system is providing a set of well-defined services to the environment. These services must be made fault-tolerant to meet the availability and reliability requirements on the entire system.

In summary, re-engineering embedded software is complicated by three characteristics of real-time systems:

1. *Timing Constraints:* The system must provide timely service even in the presence of faults. The correctness of a computation is dependent not only on the correctness of its results, but also on meeting stringent timing requirements.

2. *Dependability Requirements:* Embedded real-time systems often have strict availability and reliability requirements. Failure of a system may result in catastrophic loss of life or property.

3. *Interaction with Environment:* A real-time computer system reacts to stimuli from the external environment. The interaction with the external world makes the goal of meeting strict timing and dependability requirements more difficult.

3 Changing Requirements and the Search for a Silver Bullet

The ever-changing requirements on the existing systems is often mentioned as a source of continuing modifications and extensions to a software system. These changes are sometimes needed because the original design was not robust enough or because new software bugs (or features) are discovered. However, as suggested in the previous paragraph, several technological trends in the computing field have contributed to the changing requirements on our existing software-based applications. These trends include:

- Shift from centralized systems to distributed computing,

- Shift from monolithic operating systems to microkernels,

[1] A more detailed version of this paper can be found in [4].

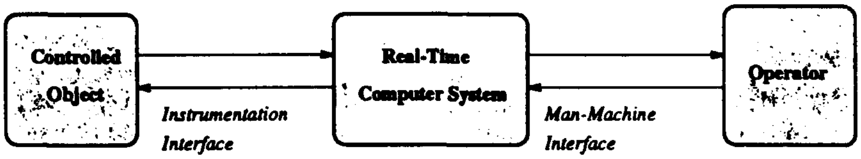


Figure 1: An Embedded Real-Time System.

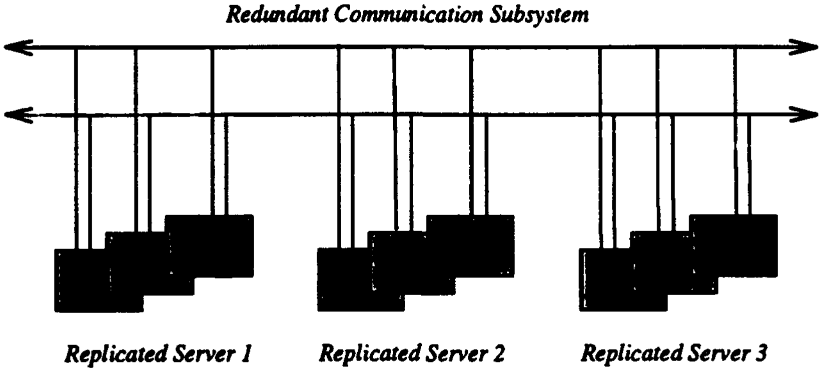


Figure 2: A Distributed Real-Time Computer System.

- Shift from proprietary hardware/software components to inter-operability of systems.
- Significant improvement in cost/performance ratio for processors, memory, and storage devices, and
- Shift from human-control to computer-control of embedded systems.

These dramatic trends in the computing filed have contributed significantly to the increasing changes of the requirements on the existing embedded systems. Hence, we are faced with an aging software base that must be modified, extended, or in some cases completely redesigned and re-implemented. However, there is no silver bullet. The cost of re-engineering is often very high and the existing tools are in most cases very primitive: tools for extracting syntactic or structural information are available where as tools for capturing semantic information are rare.

Re-engineering a complex embedded system becomes manageable only by a careful decomposition of the system into a collection of services with precise interfaces that are preserved from one version to another. Decomposition of a system into a collection of formally specified services also lends itself to the reuse of software in different systems: by building toolkits of services that capture common functions in a range of embedded real-time systems, one can reduce the cost of re-engineering through the reuse of software. We elaborate on this point in the following sections.

4 Precise Specification and Well-Defined Interfaces

A key to re-engineering complex embedded software systems is the capturing of important attributes of a system at an appropriate level of abstraction. This includes capturing both the requirements specification and the key design attributes of the system. At one extreme, an existing implementation itself can be viewed as a concrete specification of the system. However, this information is far too detailed and it exposes much of the implementation that may not carry over to the re-engineered system. At the other extreme, an informal (English) requirements specification of the system can be viewed as a high-level description. The problem here is the informality of the specification and the lack of precise semantics. Hence, specification tools are needed that aid the software engineer in capturing system requirements and key features of an existing design.

An issue related to the specification of requirements and design attributes is the precise definition of internal and external interfaces for a system that is being re-engineered. A complete or a partial re-engineering of a system is feasible if we view an embedded system as a collection of building blocks (or services) with well-defined external interfaces. Each service, in turn, may be implemented using other services. This model naturally supports composition of more primitive building blocks so that more complex services are constructed. This is the same philosophy that guided the initial design of the new generation of the air-traffic control system [2]. Localizing the modifications or extensions becomes manageable if there is a precise functional specification of each building block with well-defined interfaces. As discussed in the section 6, a precise and formal specification is also helpful in supporting automated tools for testing, verification and run-time monitoring of a re-engineered system.

5 Reuse of Building Blocks in Re-Engineering Embedded Systems

The reuse of software can provide additional leverage in re-engineering embedded systems. Since we will be faced with an increasing number of complex systems that will require partial or complete redesign and re-implemented, it makes sense to develop a collection of building blocks that are reused in different systems. By capturing some of the changing requirements that are common to a large collection embedded real-time systems, one can develop a set of services with well-defined interfaces that are used in re-engineering large systems. *One can view this approach as a toolkit approach to re-engineering.*

As mentioned earlier, the shift toward distributed computing and the emphasis on open systems are among the most important factors contributing to the changing requirements on the existing software base. Hence, a collection of distributed fault-tolerant services that can be used as building blocks in re-engineering real-time systems may prove to be a logical first step. In the following subsection, we introduce a testbed that was built in IBM Research for developing fault-tolerant servers[6]. A new version of this architecture specifically designed for real-time systems is currently under development at the University of Michigan.

5.1 Toolkit of Services: An Example

The testbed consists of a collection of protocols for managing replicated and distributed resources in a system. It consists of six software layers, each exporting a well-defined interface to the other layers or to applications that are built on top of the testbed. Figure 3 illustrates the software layers in the testbed. Each software layer, referred to as a service, supports one or more protocols. A brief description of each service layer follows:

- *multicast communication service:* provides a reliable datagram communication service for sending a message to a collection of destinations. This service allows the exploitation of available communication protocols (e.g., Netbios vs. UDP) and possible hardware support (e.g., hardware broadcast facility) on a given system without exposing the implementation to the higher layer services.

- *processor membership and monitoring service:* provides a consistent view of the operational status of a group of processors in the presence of processor/process failures/joins and communication failures. Three membership protocols with varying degrees of consistency in the views of the members are supported.

- *clock synchronization service:* provides a bound on the deviation between logical clocks on processors in the presence of hardware clock drifts and failures.

- *reliable naming service:* provides a reliable service mapping the name of an object to a list of processors in the system. This layer supports multiple namespaces.

- *distributed cache service:* provides shared/exclusive access to remote objects with local caching. This layer supports multiple coherency protocols including cache invalidation and write-through policies.

- *replication service:* provides a mechanism for maintaining multiple copies of objects in a cluster. This layer supports several replication protocols with different consistency semantics for updating replicas.

- *distributed synchronization service:* provides fault-tolerant and scalable synchronization protocols for serializing access to shared/exclusive resources. The distributed synchronization service can recover from the failure of a lock holder/coordinator and communication failures.
- *service failure detectoi:* provides notification/query service for monitoring status changes of a collection of subsystems grouped together as a server. A status change to a group can occur because of a subsystem failure or an update to an application-defined status field.

The above software services are the building blocks from which much larger systems can be developed. These building blocks can be used to replace partially or completely a system component that is being re-engineered. The objective is to develop sufficiently general building blocks once and to reuse them in extending and modifying existing software systems.

6 Testing, Verification, and Monitoring Tools

One can view the re-engineering of existing software as a *magic trick*. The existing system and the new requirements are fed into this complicated process, and the output is a new system that meets these requirements. However, ensuring that the new system meets the imposed requirements is an important problem that must be addressed. Embedded real-time systems have strict timing and dependability requirements. Rigorous methods must be applied to demonstrate that the system meets these constraints.

Decomposition of a system into a set of services with precise specifications and clean interfaces provides an advantage in achieving this. We advocate a three-way approach in demonstrating that a re-engineered system meets its specification:

1. *Testing and fault-injection methods*
2. *Formal verification methods*
3. *Run-time monitoring techniques*

Testing and fault-injection techniques exercise numerous execution paths in the new system to detect design and implementation faults. Fault-injection techniques are used to test the fault-tolerance capabilities of the new protocols and the robustness of the system with respect to failures. Formal verification techniques are effective in proving the correctness of an implementation with respect to a specification. These techniques will be effective in ensuring the correctness of certain critical tasks in a system. Formal verification complements testing and fault-injection by demonstrating that critical operations, for example, meet certain safety assertions. Testing and formal verification techniques may not guarantee against violation of design assumptions and unpredictable behavior of the external environment. Hence, monitoring of a system is necessary to detect violation of safety properties and design assumptions at run-time[5, 8]. It should be noted that a formal specification of a system and the interfaces of its building blocks are important for developing automated tools for testing, verification, and monitoring. We are currently investigating a collection of automated tools for testing, fault-injection, verification and monitoring that aid a software engineer in modifying and extending existing software.

7 Conclusion

Re-engineering embedded real-time systems is an important challenge that confronts us in the coming decade. With the increasing reliance on digital computers for monitoring and controlling embedded systems, the problem of modifying and extending an existing software base that meets new requirements is a key issue. The changing requirements on the existing systems have been influenced by several trends in the computing field including the shift to distributed computing, the need for inter-operability of hardware/software components, and the exploitation microkernel based O.S. Strict timing and dependability requirements introduce additional complexity in the re-engineering of embedded real-time systems.

This paper advocated that the decomposition of a system into a collection of services with precise specification and well-defined interfaces is crucial in reducing the complexity of re-engineering large systems. Furthermore, by developing a collection of building blocks (or services) that can be reused in different systems, the cost of extending and modifying existing systems may become more acceptable. However,

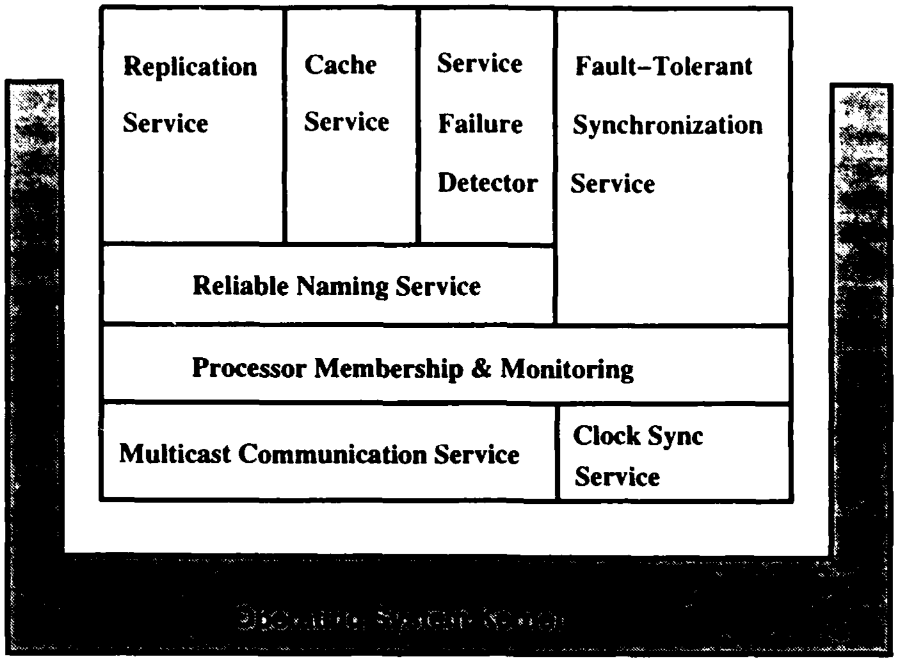

Replication Service
Cache Service
Service Failure Detector
Fault-Tolerant Synchronization Service
Reliable Naming Service
Processor Membership & Monitoring
Multicast Communication Service
Clock Sync Service

Figure 3: The Software Layers of the Testbed.

these building blocks must contain a common collection of services that may be needed when addressing the changing requirements of a large subset of embedded real-time systems. Finally, this approach is amenable to the development of a collection of tools for ensuring that the new requirements are satisfied by a re-engineered system.

References

[1] E. Chikofsky and J. Cross III. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, pages 13–17, January 1990.

[2] F. Cristian, R. Dancey, and J. Dehn. Fault-tolerance in the advanced automation system. In *Proceedings of the 20th Annual Sumposium on Fault-Tolerant Computing*, pages 1-12, 1990.

[3] H. Hopetz and P. Verissimo. Real time and dependability concepts. In *Distributed Systems, 2nd Edition, S. Mullender(editor)*, pages 411–46 (Chapter 16). Addison-Wesley, 1993.

[4] F. Jahanian. Formal specification and software reuse in reengineering embedded real-time systems. Technical Report Technical Report, Department of EECS, University of Michigan, 1994.

[5] F. Jahanian and A. Goyal. A formalism for monitoring real-time constraints at run-time. In *Proc. of Fault-Tolerant Computing Symposium (FTCS-20)*, June 1990.

[6] F. Jahanian, R. Rajkumar, and J. Turek. A testbed for prototyping distributed and fault-tolerant protocols. In *Proc. of Complex Systems Engineering and Synthesis Workshop*, Silver Spring, MD, July 1993. Naval Surface Warfare Center.

[7] R. Pressman. *Software Eng'neering A practitioner's Approach.* McGraw Hill, New York, 3rd edition edition, 1992.

[8] S. Raju, R. Rajkumar, and F. Jahanian. Monitoring timing constraints in distributed real-time systems. In *Proc. of Real-Time Systems Symposium*, December 1992.

MK 86/UYK-7 Enhanced Memory Unit Project
Jacking the Computer Up and Putting a Powerful New Engine Under it!

Joe S. Gaines, Richard W. Williams and Jay Roske

Authors

Joe Gaines serves as the Project Manager for the MK 86/UYK-7 Enhanced Memory Unit (EMU) development effort at Naval Surface Warfare Center Crane Division, Crane Indiana. He is an electrical engineer by profession with over 12 years combined experience at both Crane and the Northrop Corporation.

Richard Williams serves as the Project Engineer for the EMU development at Naval Surface Warfare Center Crane Division, Crane Indiana. He is an electrical engineer by profession with over 25 years in systems engineering experience at Crane.

Jay Roske serves as the Technical Direction Agent for the EMU development. He is employed at the Naval Surface Warfare Center Port Hueneme Division, Port Hueneme, California as the MK 86 Software Support Agent. An electrical engineer by profession, he has over 20 years experience with shipboard systems.

Background

The AN/UYK-7 Computer Set is a Navy standard computer originally fielded in the late 1970's and manufactured by Unisys. The computer set is configured with a variety of modules such as the memory module or the Central Processing Unit (CPU) module. AN/UYK-7 computers are used across a wide array of platforms with a population of hundreds of units.

The MK 86 Gun Fire Control System (GFCS) uses the AN/UYK-7(V) Computer Set for a variety of functions. The MK 86/UYK-7 computer as it exists today faces several memory problems. Planned upgrades in targeting radar systems require more memory and faster access to provide a performance increase in computer execution of new application software. The existing memory units available for use in the AN/UYK-7(V) Computer Set are inadequate for these upgrades. Additionally, the existing memory units face severe obsolescence problems. The mated film technology which forms the basis of the Double Density Mated Film Memory Unit (DDMFM) is no longer in production. The day is rapidly approaching when the UYK-7 as it exists will not support MK 86 GFCS requirements. The MK 86 GFCS In Service Engineering Agent (ISEA) at Port Hueneme Division (PHD) of the Naval Surface Warfare Center (NSWC) recognized the approaching problem in 1991 and took steps to counter it. NSWC Crane Division was enlisted to provide product engineering support to develop a technology upgrade for the MK 86 UYK-7 Computer Set. In an age of ever-decreasing defense budgets, there is more and more pressure to keep existing equipments upgraded and fielded. This paper describes the project to develop an Enhanced Memory Unit (EMU) for use in the AN/UYK-7(V) Computer Set and the features of said unit. The EMU contains an embedded state-of-the-art processor which essentially gives the AN/UYK-7 computer access to new found powers. In other words, we've jacked the computer up and put a powerful new engine under it.

Teaming Arrangement

The project is a teaming arrangement between NSWC Port Hueneme Division (PHD), NSWC Crane Division and PMS-412 UYK-7 Program Office. NSWC Crane is developing the MK 86/UYK-7 Enhanced Memory Unit (EMU) and will undertake the MK 86 production run of the EMU. PHD is writing the application software for use with the MK 86 upgrades and will perform the ORDALT to install the EMU into MK 86/UYK-7 computer sets. PHD is also responsible for overall project direction. PMS-412 is providing physical assets for refurbishment and effecting the field change to the UYK-7 computer set.

Goals & Objectives

The original goal of the project was to develop a semiconductor memory replacement for the UYK-7 Double Density Mated Film Memory (DDMFM). The Enhanced Memory Unit was to have 4 times the memory and 30% performance enhancement. A further goal was to provide units to meet the scheduled ship installation of other MK 86 upgrades. During the course of the project, the goals changed to attempt to develop a unit which could provide a solution for MK 86/UYK-7 requirements into the next century.

Accomplishments

All of the original design goals will be met or exceeded by the EMU which has been developed with major improvements in power consumption and MTBF. More significantly, the EMU will provide performance enhancements of approximately 1000% or better through a technology insertion. The technology insertion is the use of an embedded state-of-the-art processor, the M68040. A simple drawer replacement (slide out the old unit and slide in the new EMU) will give the UYK-7 state-of-the-art processing capabilities.

The project is currently on schedule and within budget. The first Engineering Developmental Model (EDM) was delivered in November 1993. According to a study by PHD, it is anticipated that this systems re-engineering technological solution will result in a $60M cost avoidance.

Finally, it is anticipated that the EMU will provide an excellent option for other UYK-7 users faced with the same DDMFM obsolescence, lack of memory and needed performance enhancements which prompted MK 86 to initiate this project. PMS-412 estimates that as many as 800 EMU's could be needed by the UYK-7 community.

Technical Features

The Enhanced Memory Unit (EMU) had many design goals. The first was to remain form, fit, and functionally equivalent to the current Double Density Mated Film Memory (DDMFM) which is a 32K x 32 memory module used in the UYK-7 Computer Set. This would lead to easy installation of the EMU on board ships. The second goal was to increase memory capacity to 128K x 32 in a memory module so that single and dual bay configurations of the UYK-7 could reach full memory capacity without adding more cabinets. The third goal was to increase the performance of the UYK-7 by decreasing the memory access time. Early tests showed that only marginal performance increases could be obtained with decreased access time, so a processor was added to the memory module to offload computational processing tasks from the UYK-7 CPU.

The choice of the processor was quickly narrowed to the Motorola 68000 family since the current compiler for the UYK-7, CMS2K, presently supports the Motorola 68000 series processors. Since most of the computational tasks to be offloaded and future upgrades require floating point calculations, an integral floating point processor was desired. With these considerations, the Motorola 68040 became the choice.

The Enhanced Memory Unit (EMU) is composed of 5 main functional areas: 128K x 32 non-volatile memory array, 8 access channels, control section, power distribution, and EMU Processor unit. The block diagram for the EMU is shown in Figure 1.

The memory array is formed of 8K x 8 non-volatile static RAM microcircuits. These microcircuits consist of a fast static RAM section with a shadow EEPROM section. During normal operation, the memory operates from the static RAM section providing fast access for both read and write operations. The contents of the static RAM are stored in EEPROM when the input power is detected to be below normal operating voltage. The time required to make this transfer is 12 milliseconds. Therefore, the internal voltages generated in the EMU must be maintained during this time. This is accomplished by having a large diode isolated capacitor on the input of the EMU power converter. On power up, the contents of the EEPROM are transferred to the static RAM section. The time required to do this is 25 microseconds which occurs after the EMU power converter reaches normal operating voltage and while the UYK-7 master reset is still active. This gives the memory unit its non-volatile appearance and does not affect any of the normal operating time of the UYK-7. The EEPROM section has the same characteristics of any other EEPROM which has a minimum of 10,000 write cycles. Since writes to the EEPROM only occur during power removal, this will provide several years of failure free non-volatile operation.

Eight access channels are provided to interface the memory unit to the UYK-7 CPU and Input Output Controller (IOC). The CPU requires two interface channels; one to fetch instructions and one to obtain operands. The IOC requires only one interface channel. Any combination of CPUs and IOCs can be interfaced to a memory unit within the limits of 8 channels. The eight access channels share the memory array through a simple channel ranking priority arbitration scheme. This means the highest ranking channel with an active channel request will get the next available memory cycle. The EMU contains 4 times the available memory of the DDMFM which means that the EMU must provide faster access time to provide equivalent performance of the DDMFM in certain configurations. The DDMFM provided a single cycle read access time of 1100 nanoseconds with a memory cycle time of 750 nanoseconds. The EMU was designed with a single read access time of 750 nanoseconds with a memory cycle time of 150 nanoseconds. The 750 nanosecond single cycle access time approaches the limit of the UYK-7 CPU. A single EMU provides better performance than two DDMFMs used in a software interleaved configuration.

The control section manages the data flow through the memory unit. The control unit is divided into three functional blocks. The first is the local controller. The local controller handles all the control signals from the UYK-7 required for the transfer of information over one channel. There is one local controller for each channel. The second functional block is the priority arbiter which determines which of the requesting channels will get the next memory cycle. The memory controller is the third major control function. The memory controller provides the timing and the decoding required to access the memory array.

The power distribution section generates the 5 volt power required by the EMU logic from the -90 volt DC provided from the UYK-7 power supply. Detection logic determines when the incoming power is going out of tolerance and generates the timing control signals required to perform the store from static RAM to EEPROM. A diode isolated capacitor provides the energy required to perform the store after incoming power has been removed. After the store operation is complete, the EMU power converter is shut off to prevent undesired store operation as the capacitor slowly discharges.

The EMU Processor unit was designed to offload processing tasks from the UYK-7 CPU. Independent processing tasks can be downloaded from the UYK-7 CPU to the EMU Processor. The UYK-7 CPU initiates tasks providing input parameters and receives processed results. Communication from the UYK-7 CPU to the EMU processor is accomplished through shared memory which is the EMU Memory Array. The EMU Processor has no access to any external peripherals of the AN/UYK-7.

The EMU processor unit consists of a 68040 that has local memory which is 128K x 32 words of volatile static RAM and 64K x 32 words of EEPROM as shown in Figure 2. The static RAM section is used to store tasks loaded from the UYK-7 CPU and to set up data buffers to be used during the processing of tasks. The EEPROM section contains three program functions; the floating point code provided by Motorola to provide full function floating point processor, built-in-test routine for checking the EMU Processor module, and a monitor routine to allow for initial download and start of tasks. The EMU processor module is connected to the address and data buses of the EMU memory array which gives the 68040 access to the entire EMU memory array as shown in Figure 1.

The upper eight memory locations of the EMU Memory Array are reserved for control of the EMU Processor. These memory locations are called the EMU Processor Control Registers although they are dedicated memory locations with additional decoding circuitry for special functions. These registers provide a Control Register, a Status Register, a Software Reset Register, a UYK-7 Interrupt Register, and a Built-in-Test Register.

The Control Register allows the UYK-7 CPU to send commands to the EMU Processor Monitor Software. When the UYK-7 CPU writes to this register, an interrupt is generated to the 68040. The 68040 can read the register and perform the required function.

The Status Register is used by the EMU Processor to provide status information to the UYK-7 CPU. The information passed in the Status Register can indicate such things as the EMU Processor being initialized, a valid command being received, and the status of a completed task.

The Software Reset Register allows the UYK-7 CPU to reset the EMU Processor. The reset performed is equivalent to a power-up reset.

The UYK-7 Interrupt Register provides a means of signalling the UYK-7 CPU when the EMU Processor completes a task. When the EMU Processor writes to the UYK-7 Interrupt Register, a Class II Interprocessor interrupt is generated on the UYK-7 CPU. This feature requires that a wire be added on the UYK-7 backplane from the EMU module to the UYK-7 CPU module.

The Built-in-Test Register is reserved for use with the EMU Processor built-in-test software. The built-in-test software uses this register to test the interface from the EMU Processor to the EMU Memory Array. The built-in-test software on the EMU Processor tests the 68040 functions, the RAM and EEPROM on the EMU Processor module, and the interface to the EMU Memory Array. The built-in-test does not test any other function of the EMU. This check out is performed with the standard UYK-7 Diagnostics.

To allow the UYK-7 Diagnostics to be executed without change, an enable feature was designed into the EMU Processor. When the EMU is powered up, the EMU Processor is not enabled. The EMU Processor can not access the EMU Memory Array and will not respond to commands from the EMU Processor Control Registers. Therefore the EMU Processor Control Registers respond the same as normal memory locations. This allows the UYK-7 Diagnostics to run unchanged. The EMU Processor is enabled by writing a coded sequence to the Status Register memory location. This enables all the functions of the EMU Processor. The EMU Processor is disabled by any reset including the software reset.

As a result of the redesign effort, two other improvements were obtained for the UYK-7 Computer Set: decreased power consumption and an increased reliability over the DDMFM. The present UYK-7 system has marginal cooling capacity. Some long term failure problems have been attributed to inadequate cooling. The present DDMFM dissipates 250 Watts. In a single bay configuration, there can be up to three DDMFMs. One EMU has the memory capacity of four DDMFMs and has a measured power dissipation of 80 Watts. If no memory capacity increase is desired, this offers up to a 670 Watt power dissipation saving in this one cabinet and provides additional cooling capacity for the remaining modules in the cabinet.

The DDMFM has one of the highest failure rates of the modules in the UYK-7 Computer Set with the primary failures being the mated film stack and the associated drivers. The EMU is designed with all solid state devices which have inherently lower failure rates. The calculated MTBF per MIL-STD-217E is 8600 hours which is approximately 4 times better than the actual failure rate of the DDMFM. With the fact that fewer modules are required for the same memory capacity, this offers a considerable reliability improvement for the UYK-7 Computer Set.

Software Re-engineering

In the late 1960's, the original MK 86 operational program was written in assembly language and hosted in a MK 152 (UNIVAC 1219B) computer with 32k of 18 bit memory. In the early 1970's, as MK 86 outgrew the MK 152 computer, efforts began to translate the program into UYK-7 code. The product that resulted was very much like the MK 152 program with changes made to utilize the UYK-7 features and new system hardware. The significant computer improvements of that era included the faster CPU, memory, and an independent Input/Output Controller. In accordance with U.S. Navy policy to use standardized "high order languages," MK 86 began coding system improvements in CMS-2. Developed in the late 1960's, CMS-2Y was a "Compiler Monitor System" for the 32 bit instruction set architecture machines. Compilations were performed on the UYK-7 itself.

As a language, CMS-2 has continued to be updated, and is now supported on platforms other than the UYK-7 system. MK 86 now uses the Navy's standard set of support software tools called "Machine Transferrable And Support Software (MTASS)" hosted on a VAX to perform life cycle software support activities.

In order to discuss the re-engineering of our product, the structure of the current MK 86 operational program is depicted in Figure 3. There is a basic one second cycle of events that must be completed to effect system operation. The one second window is broken down into 16 segments, each having specific functions to complete which support other "sixteenths" tasks. When four identical tasks are performed during each sixteenth, the task is said to be a "64 hz" task. 64 Hz processing is initiated by the IOC Monitor Clock. While the major Executive cycle executes at a 16 Hz rate, there are also a number of tasks which execute at rates of 8 Hz, 4Hz, 2 Hz, and 1 Hz. These are referred to as "Variable Sequences" and consist of a number of MK 86 functions which must complete during an allocated amount of time, but need not be executed at a 16 Hz rate.

Looking forward to operational software integration with the EMU, the basic program cycle will not change. The 64 hz, 16 hz, etc. processing will continue. Our challenge is to off-load tasks that the UYK-7 struggles with onto the EMU Processor. The EMU drawer will function as a two port UYK-7 memory with the M68040 processor providing high speed execution and "mathpac" capabilities. The M68040 integer and floating point capabilities are extensive. In anticipation, we have begun recoding mathematically intense operations such as coordinate conversion routines, various predictors, and stabilization subprograms to utilize the EMU processor's power. It will be very interesting to compare M68040 performance with UYK-7 and MK 152 data where trigonometric functions were computed utilizing look-up tables. For example, early MK 86 project notes indicate that a MK 152-hosted sine or cosine function, with "14 1/2 bit accuracy" could be accomplished in as little as 400 microseconds! Recent investigations measure the same basic MK 86 algorithm, executing in the UYK-7, at 87 microseconds. Given the EMU processor's integer and floating point computational power, we fully expect to reduce our math execution times by an order of magnitude.

A depiction of the new structure is shown in Figure 4. All standard processes will be shared with the processor to the greatest extent possible. This will allow Variable Sequence processes to be performed earlier (as required). The UYK-7 will have more time available to perform additional background processes. This will allow for improvements in on-line data reduction and analysis, for example. Certain, rigid processes will remain unchanged.

With regard to language, a major factor in selecting the EMU processor was the ability to compile existing CMS-2 code with a new target in mind. The MTASS CMS-2K compiler supports the M68030 target processor as utilized in the AN/UYK-43's "Time Critical Subfunction (TCS)". Therefore, we felt secure in selecting the M68040 which fully supports the repertoires of the M68030 and M68882 coprocessor - only faster and more efficiently. In the development of a computer program which could be executed by the EMU processor, we discovered a problem. We could not, using our MTASS tools, build a bootable object tape containing

M68040 code that could be loaded into the UYK-7 and executed in the processor's upper 128k of memory. That upper 128k space is where the EMU processor's high speed RAM is located. The reason for this problem is mainly due to the fact that the UYK-7 and the EMU processor have two separate memory maps for the same physical memory space. In these maps, the EMU processor has read/write access to the UYK-7 memory, but the UYK-7 cannot access the "non-shared" processor memory. This is depicted in Figure 5. The UYK-7's lower 128k memory space is represented by locations 0 to 377,777 (octal), while the same space represents locations 2,000,000 to 4,000,000 (octal) to the EMU processor (its second 128k space).

Due to the difference in memory mapping, and the fact that the UYK-7 does not have access to the EMU processor's upper memory, no programs can be loaded directly into the EMU processor's upper memory via the UYK-7's NDRO bootstrap. This problem can be overcome by loading the EMU processor code into UYK-7 memory via the bootstrap and "downloading" the code from UYK-7 memory to EMU processor memory via the EMU's Download command.

The download method would seem to be the answer to the loading problem, however, this method unveils another problem. In order to execute a program somewhere in memory, the program must be linked to that particular memory area so that instruction operands can evaluate correct absolute addresses for data movement and branches. On the other hand, when building a bootable tape using MTASS, the Tape Builder program (TBL) uses the absolute start address of the program calculated during link time as the address at which to load that program. In other words, a program which is compiled and linked to execute in EMU processor memory, cannot be loaded into UYK-7 memory.

Until a more elegant solution is found, a temporary "loader" program is being created for the processor program which ignores the starting address associated with the NTDS boot record and, instead, loads the EMU Processor program into a predetermined location in UYK-7 memory. This allows the EMU processor program to be linked relative to an area of EMU processor memory, and loaded into UYK-7 memory for downloading.

While the overall implementation of changes in the MK 86 project will be incremental, the processes governing each upgrade remain the same over the next 3 years: identify candidate tasks to be off loaded to the EMU processor, recode whatever assembly language exists, recompile under CMS-2K, and recertify performance. This method supports our existing maintenance philosophy, requires no additional personnel, and is supported by our existing standard test procedures.

Lessons Learned & Conclusions

Perhaps the most difficult part of a design is accurately defining the interface specifications from the unit to be designed and the rest of the system. Most of the requirements of the design were derived from the technical manuals and specifications in the fabrication drawing package. When trying to duplicate the exact functions of a device, measured data defines the interface requirements accurately. But when increased performance is desired, minimum and maximum timing requirements and relationships are required. In over three cases during the design, timing relationships defined or implied in the available data sources were found to be incorrect which resulted in design changes during testing.

Another problem was noted when attempting to change device types. The UYK-7 channel busses were implemented with open collected devices. An attempt was made to use tri-state drivers to perform the same function. Early testing showed that every thing would work properly with tri-state drivers, but when the full system was tested, intermittent problems occurred due to decreased noise margins introduced by the tri-state drivers. This caused a design change to open collector drivers which are not as readily available in certain functional type devices.

These examples emphasize the advantage of performing early confidence testing to help define design requirements and parameters.

Finally, this project demonstrates the advantage of providing technology insertions for existing systems. Benefits are derived in system capabilities and performance as well as substantial cost advantages.

References

AN/UYK-7 Technical Manual

SE610-AW-MMA-010 Maintenance Manual

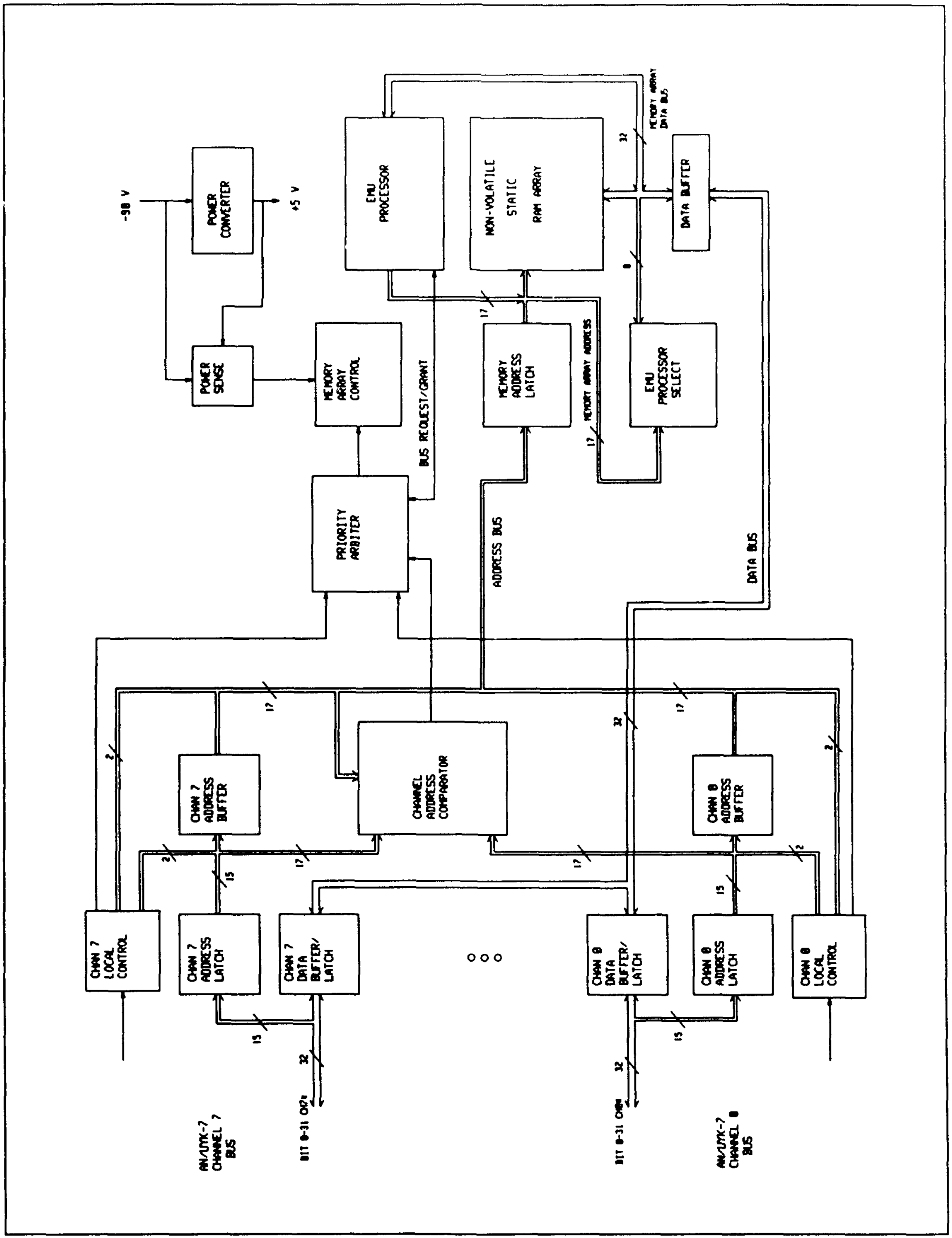
Enhanced Memory Unit Critical Item Development Specification, 53711-6891400, Rev B.

Product Fabrication Specification, Memory Module, Double Density Mated Film, SB-12857

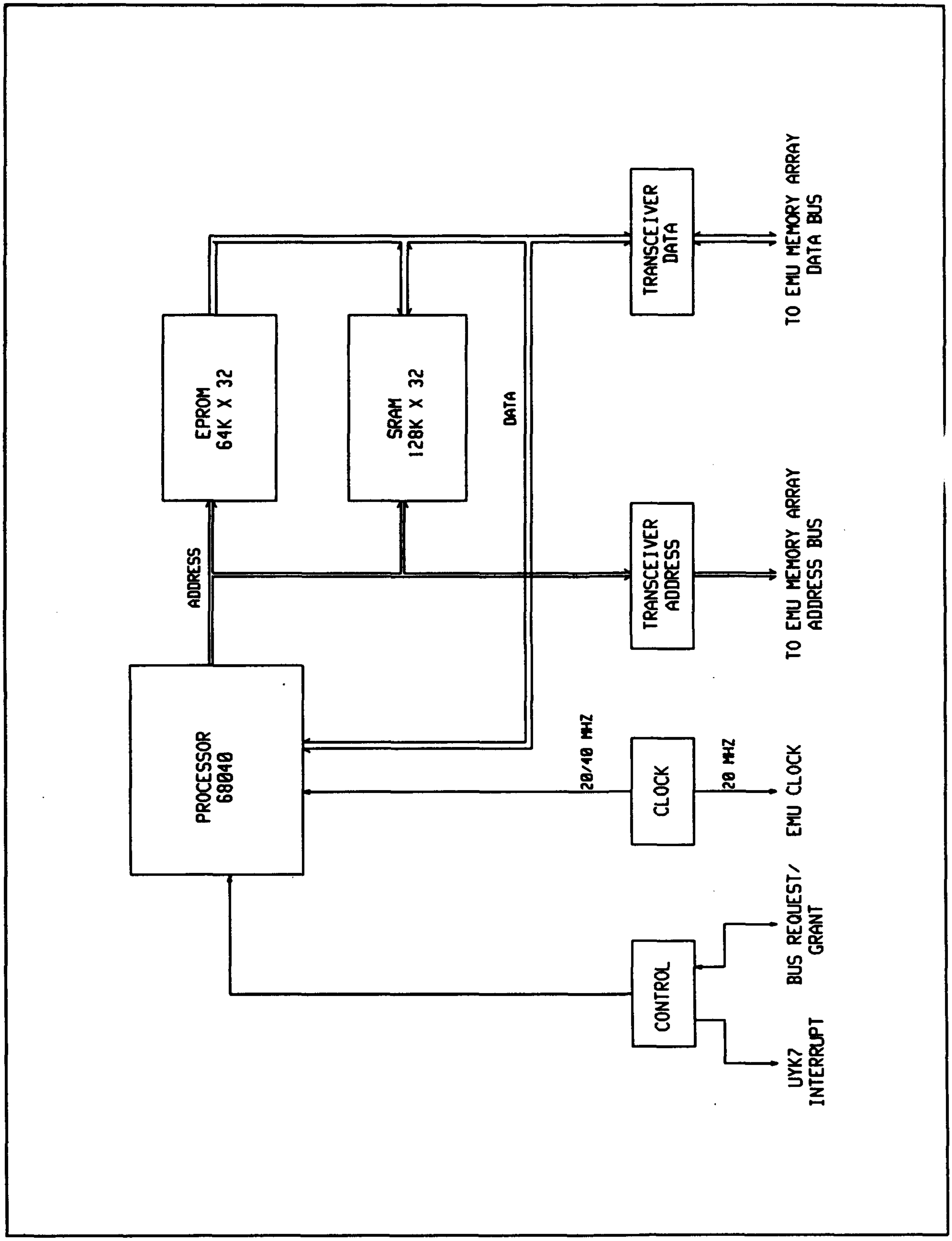
CMS-2 Compiler, Users Handbook, 0967-LP-598-8020

Linkage Editor, Users Handbook, 0967-LP-598-8060

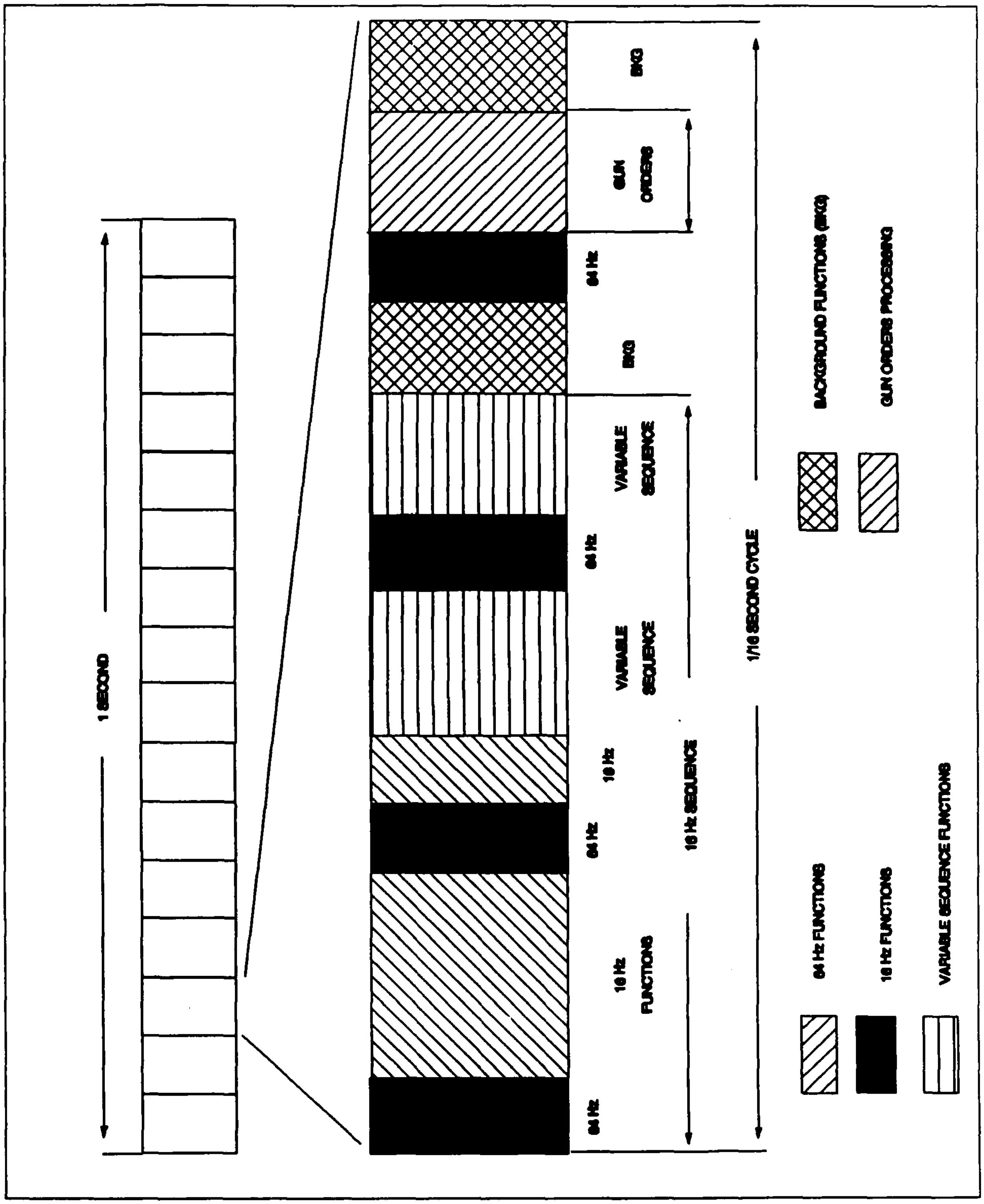
SDEX/7, SDEX/43, & SHARE/43 Tape Builder, Users Handbook, 0967-LP-598-8070



AN/UYK-7 EMU BLOCK DIAGRAM
Figure 1

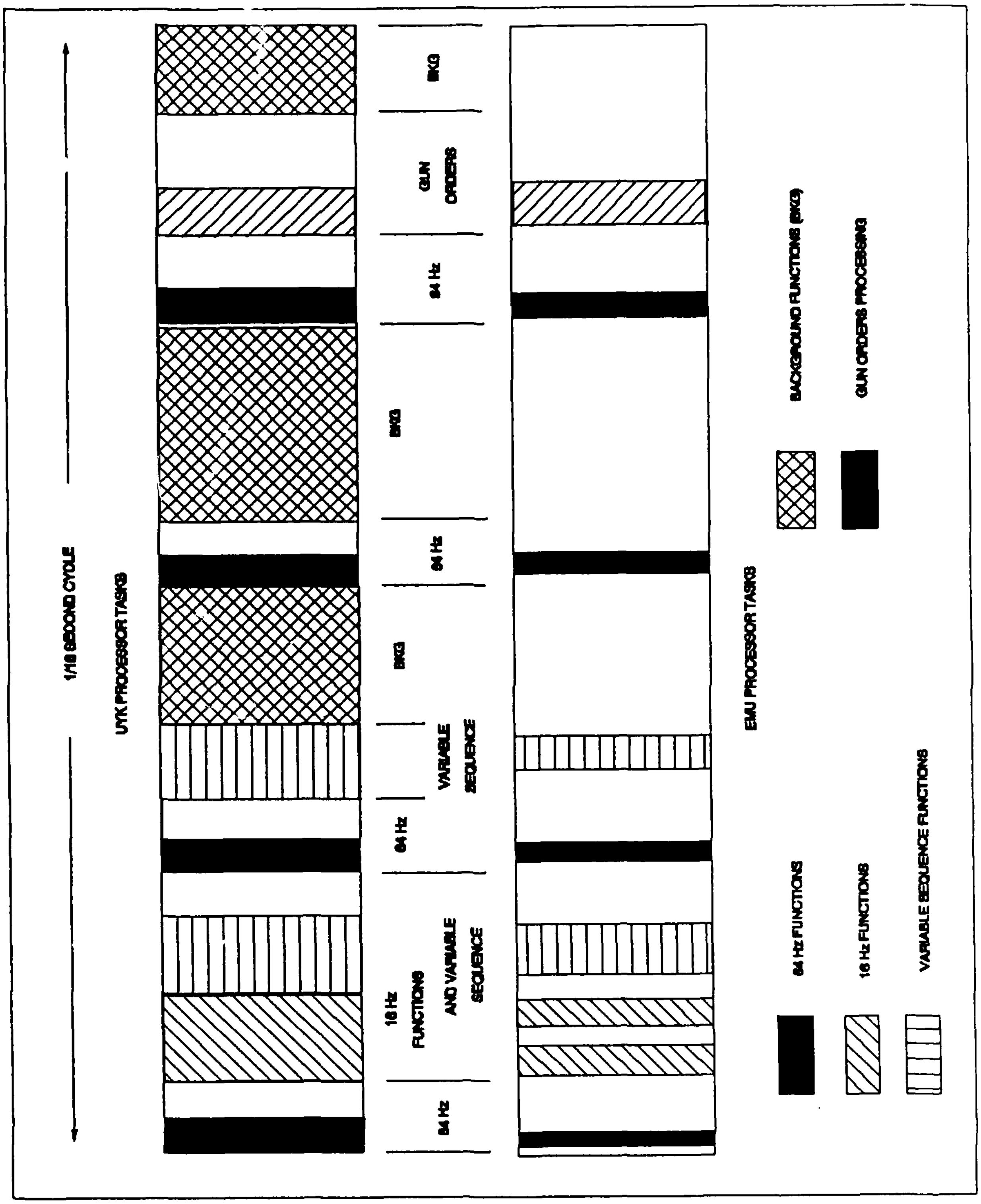


EMU PROCESSOR BLOCK DIAGRAM
Figure 2



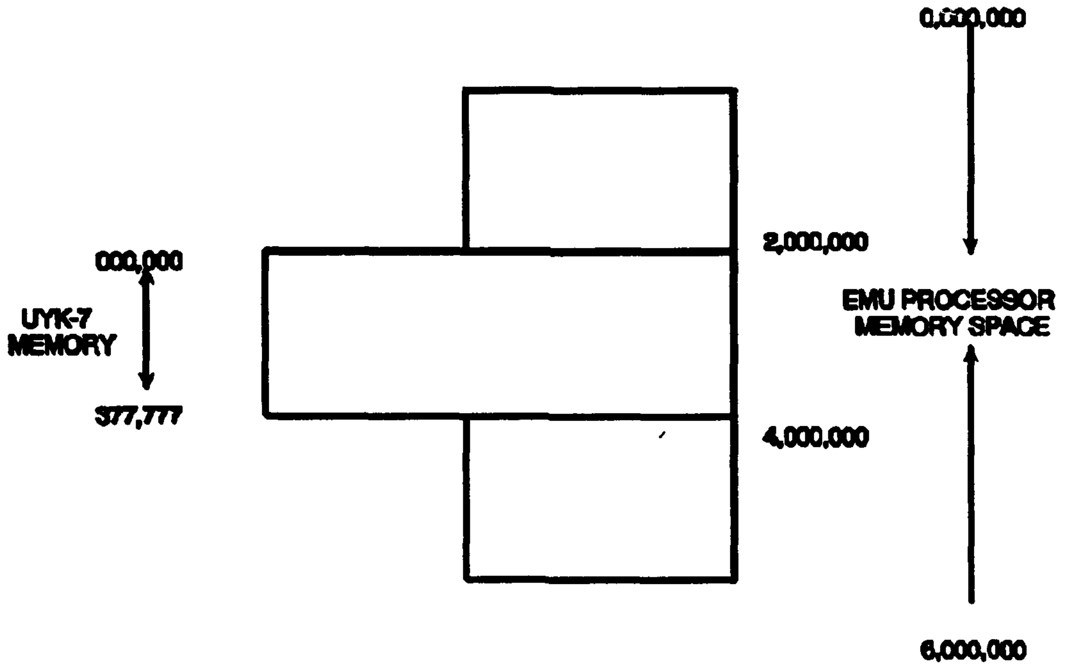
CURRENT MK 86 CONTROL PROGRAM TIMING SEQUENCE

Figure 3



FUTURE MK 86 CONTROL PROGRAM TIMING SEQUENCES (DEPICTION)

Figure 4



ADDRESS SPACES VISIBLE TO UYK-7 AND EMU PROCESSORS

FIGURE 5

Reengineering the LAMPS MK III to Provide a LOS Ship-to-Ship Teleconferencing Mode

James P. Rahilly

Naval Command, Control and Ocean Surveillance Center RDT&E Division
San Diego, CA 92152-5000

ABSTRACT

This paper describes the reengineering of the NAVY's LAMPS MK III to provide a new high data rate (HDR) ship-to-ship LOS communication capability. The application of present LAMPS MK III, with added capability, permits ship-to-ship LOS data communication rates in excess of near three times the T 1 rate of 1.544 Mbps. A cooperative effort is underway to demonstrate this capability with the support and resources of CINCLANTFLT. This demonstration will consist of a HDR LAMPS MK III extension of the existing USS Mount Whitney Satcom Video Teleconferencing (VTC) to an HDR modified LAMPS MK III ship. Since the VTC extension will only operate at a 386 Kbps rate, other Multimedia loading of the link will be used to more fully load the communication link. This VTC demonstration using LAMPS MK III will be discussed in in this paper.

1. INTRODUCTION

The LAMPS MK III reengineering challenge is largely related to the achievement of this new HDR communication capability without making system changes that could impact LAMPS MK III primary mission of ASW and ASST (anti-ship surveillance and targeting). The exploitation of this system's design features has been done in order to satisfy the R&D program's HDR of T-1 or 1.544 Mbps objective. The use of LAMPS MK III is the most viable and cost effective way for the Navy to provide a large number of combatant ships with LOS HDR ship-to-ship communications. Navy R&D program plans for FY 95 call for creating an even higher data rate communication system. This new system only uses the LAMPS MK III waveguide and high gain antenna system. These LAMPS MK III elements are coupled, when required, to a new wideband receive-and-transmit system to achieve a ship-to-ship or ship-to-shore communication data rate of T 3 or 44.7 Mbps. This capability would be particularly useful to satisfy the shore-to-ship theater extension (TENET) requirements of Global Grid. Figure 1.1 provides a photograph of the present shipboard LAMPS MK III system (SRQ-4) and Figure 1.2 shows the LAMPS MK III (SH60R) helicopter, with it's LAMPS MK III system on-board; just after its takeoff from FF 8 of the Fast Frigate Class. ASW and ASST data that the LAMPS MK III helicopter collects while on station, up to 100 Nmi from its mother ship, is transmitted at a HDR back to the ship. The Navy has in excess of 80 ships with the LAMPS MK III. The LAMPS MK III antenna installation is shown high up on the ship's mast of a DD 963 in figure 1.3.

2. DISCUSSION

In the early phases of the R&D investigation, technical issues were examined and conclusions reached that led to the belief that LAMPS MK III was the optimum way to achieve a HDR ship-to-ship communication capability. For example, the scope of the investigation required the consideration of the entire SHF band. However, as a result of a review of the allowable electromagnetic frequency regions in the SHF band, where nonsatellite ship-to-ship communications is permitted by international agreement, it was found that only the spectral region from 4.4 to 5.0 GHz was allocated for this purpose. Therefore, this is the operating frequency band used by LAMPS MK III to establish a communication link between a ship and a helicopter.

Another issue relates to the assessment of the electromagnetic compatibility of any newly created communication system with the existing LAMPS MK III shipboard transmitter and receiver operating environment. For example, any newly created system

Mr. Rahilly's work is sponsored under 6.2 block funding by the Office of Naval Research, contract N00014-94-WX-35038AD

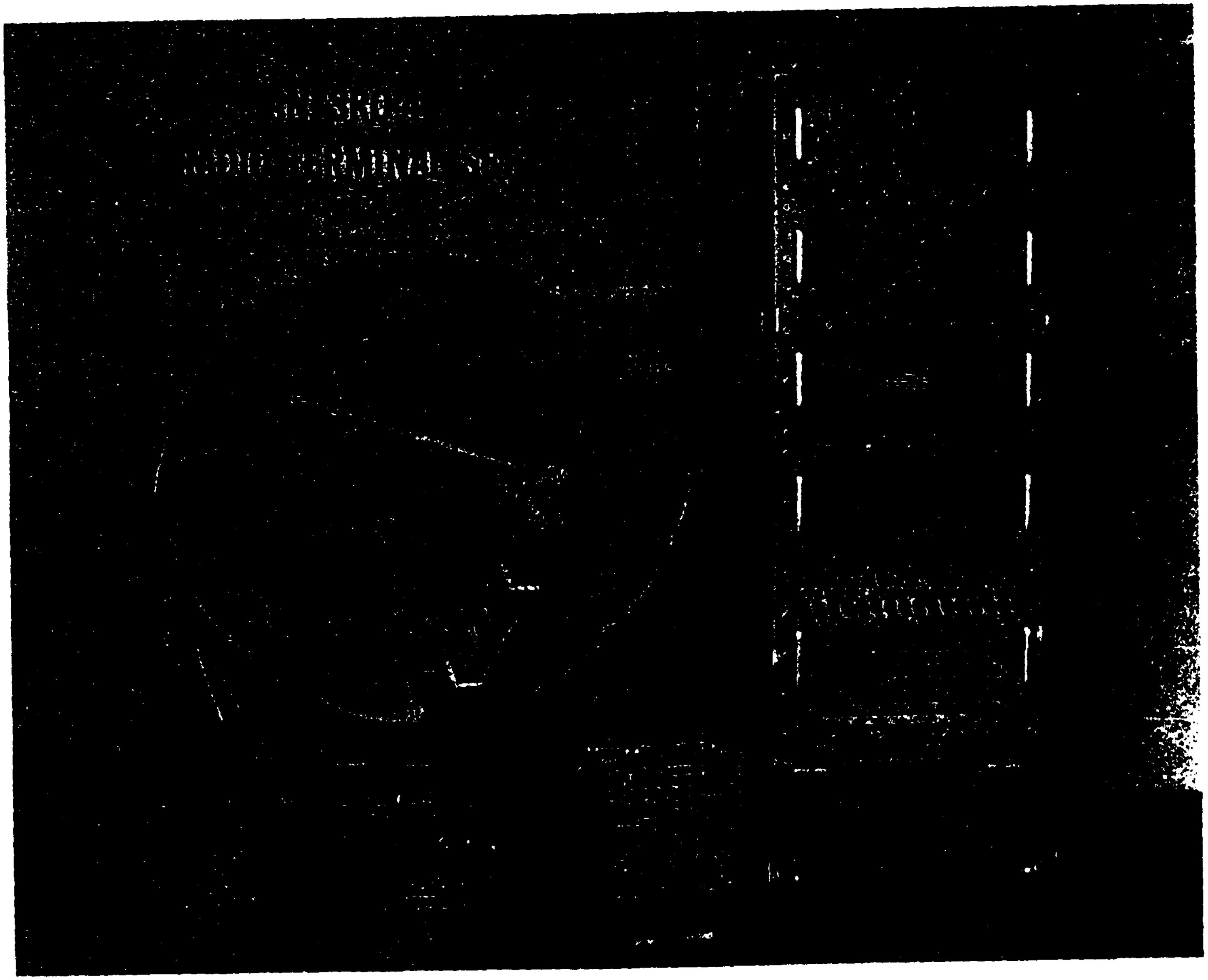


Figure 1.1. AN/SRQ-4 radio terminal set.

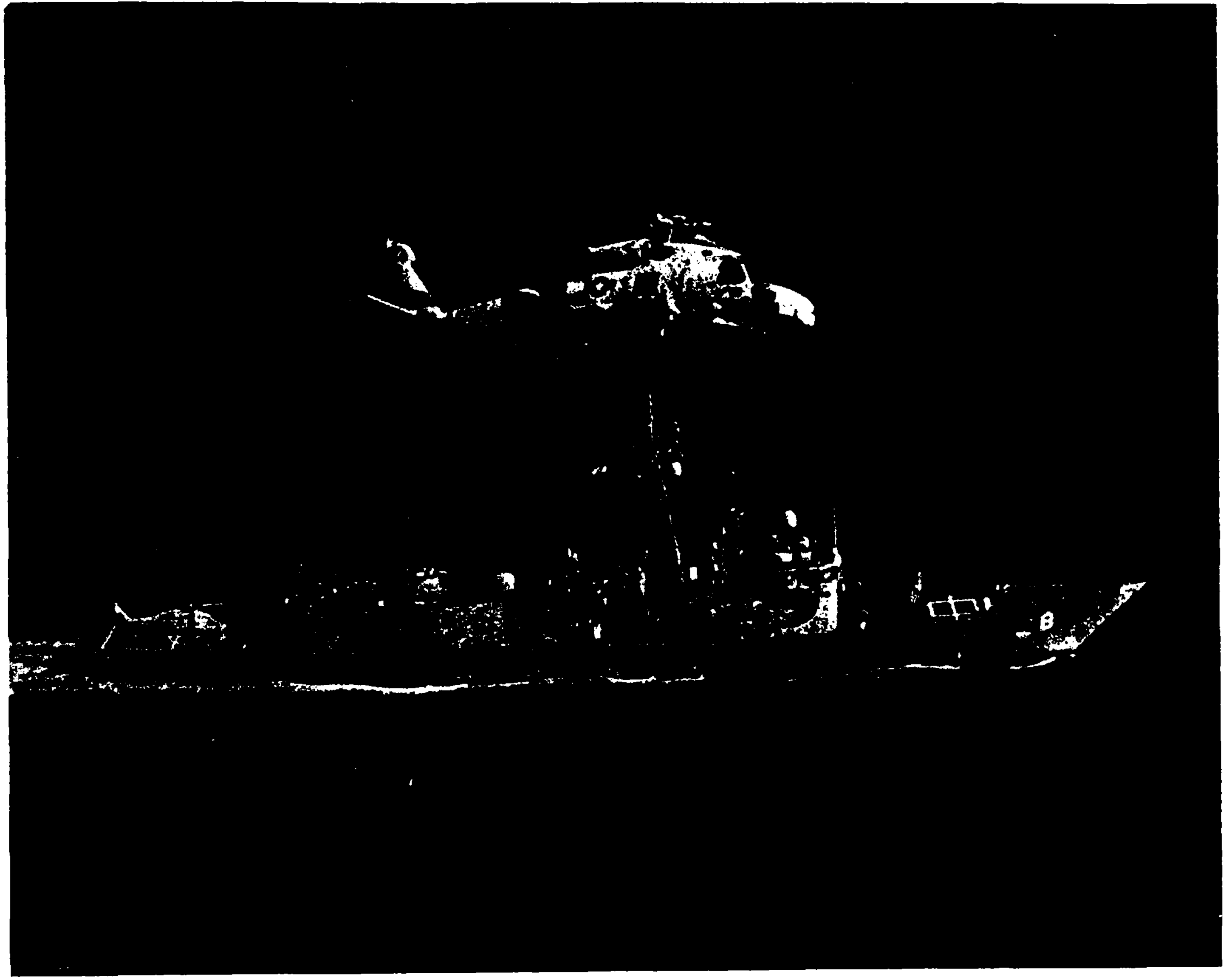


Figure 1.2. LAMPS III SH60R helicopter after takeoff from a LAMP III equipped FF8.

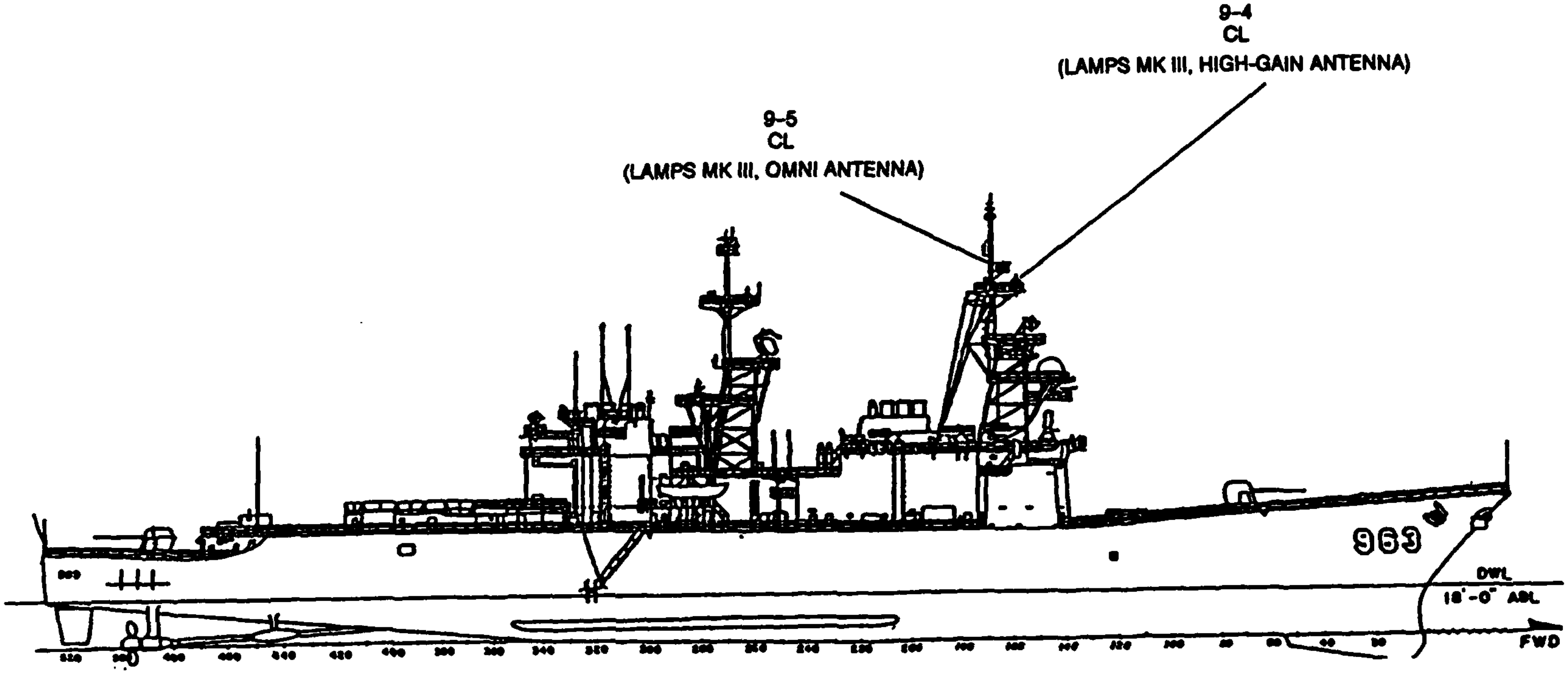

9-4
CL
(LAMPS MK III, HIGH-GAIN ANTENNA)
9-5
CL
(LAMPS MK III, OMNI ANTENNA)
963
DWL
18'-0" ABL
FWD

Figure 1.3. DD963 Starboard profile.

would be operating in the same 4.4 to 5.0 GHz frequency band as the LAMPS MK III system and must not interfere with the high priority communication linkage between the ship and the SH-60B helicopter. The approach taken in this new ship-to-ship communication concept is to not create a new system that might interfere with LAMPS MK III but rather to use the LAMPS MK III system itself to provide the ship-to-ship communications when it is not performing its operational role with the LAMPS MK III helicopter.

Another advantage to using the existing LAMP MK III system to achieve a ship-to-ship HDR capability includes the fact that the LAMPS MK III high gain antenna system, because of its high operational priority, is located very high up on the ship's mast, thus maximizing the LOS range. Real estate is very scarce at these heights and a new system would find it difficult to get a location as high as LAMPS MK III currently has.

Also, by using the LAMPS' antenna system, there are no major developmental or procurement costs to obtain a high gain, stabilized antenna that provides monopulse azimuth tracking, has passed mil-spec shock and vibration testing, and has demonstrated highly successful operational reliability while providing mission support in conjunction with the LAMPS MK III helicopter system.

It is the system goal, in this proposed use of the LAMPS MK III shipboard assets, to not only use the above-deck assets but also to use to a maximum degree all the below-deck subsystems. These subsystems include the frequency synthesizers, modulation, demodulation subsystem transmit and receive subsystems, mux and demultiplex subsystems, and KG-45 crypto systems. However, variations from the present LAMPS MK III must carefully consider the requirement to be able to rapidly reconfigure the system from a posture that supports ship-to-ship communication to one that can immediately support the LAMPS MK III mission. The LAMPS MK III mission must always be the primary operational mode, and any design changes must always be evaluated from this perspective.

3. DESCRIPTION OF LAMPS MK III HDR COMMUNICATION CONCEPTS

In order to most effectively discuss this new concept, it is best to first examine how LAMPS MK III communications presently operates. Figure 3.1 shows the basic facets of LAMPS MK III ship-helicopter communications. The shipboard Lamps transmit-and-receive equipment is below-deck and is connected by waveguide through the waveguide switch shown in figure 3.1 to either the high gain or omni antennas, mounted high on the mast. This figure shows the high gain antenna in use. This occurs when the helicopter is a significant distance from the ship. The uplink requires less than 10% of the bandwidth required by the helicopter-to-ship downlink. The term wideband (WB) is used for the downlink and narrowband (NB) is used for the uplink.

The WB downlink includes the NB response data, since it is multiplexed in with the wideband downlink data stream. Thus, a full duplex communication is achieved relative to the narrowband data communication. The wideband link is simplex in nature, since there has been no need for WB transmission from the LAMPS MK III ship until now.

3.1 LAMPS MK III COMMUNICATION CONCEPT: BASELINE SYSTEM WITH MAXIMUM CAPABILITY.

The baseline LAMPS MK III HDR communication concept consists of using the shipboard LAMPS MK III (SRQ-4) and integrating these assets together with the LAMPS MK III transmit, receive and mux/demux assets that are used on the SH-60B helicopter (ARQ-44). Figure 3.2 illustrates this concept. As may be seen in this sketch, a new dual rotary waveguide switch now replaces the present single action switch. With the waveguide switch position shown, the SRQ-4 receive-and-transmit system is connected to the high gain antenna, while the ARQ-44 is connected to the omni antenna. This is the ship A configuration. Ship B is at the other end of the link and its connectivity is just reversed. Therefore, it can be seen from figure 3.2 that the high gain antennas are transmitting on the narrowband (NB) data links and the zero db gain omni antennas are transmitting the wideband (WB) data links. It should also be noted that by rotating the SMA switches the NB or WB operation can be conducted in either LB (lower band) or in the UB (upper band) This is required in order to achieve sufficient frequency separation to afford the needed isolation between the transmitted energy from a given antenna and the low level signals that are being simultaneously received by the other. This signal isolation is presently obtained using two broadband RF bandpass filters in the lower

LAMPS MK III SHIPBOARD SYSTEM - AN/SRQ-4

SIMULTANEOUS Tx and Rx

RECEIVE - WB (any one of) 16 Channels
TRANSMIT - NB (any one of) 16 Channels

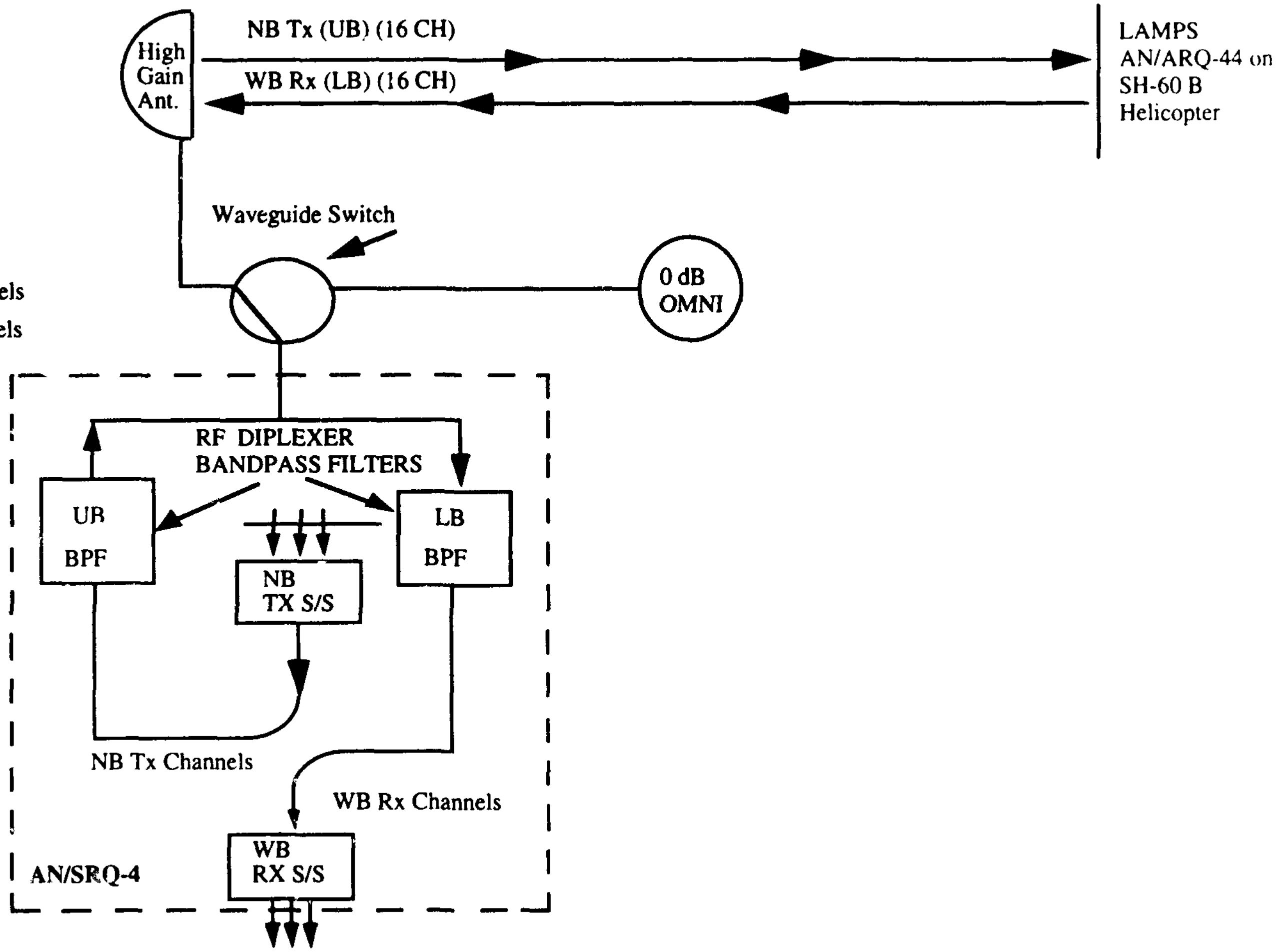
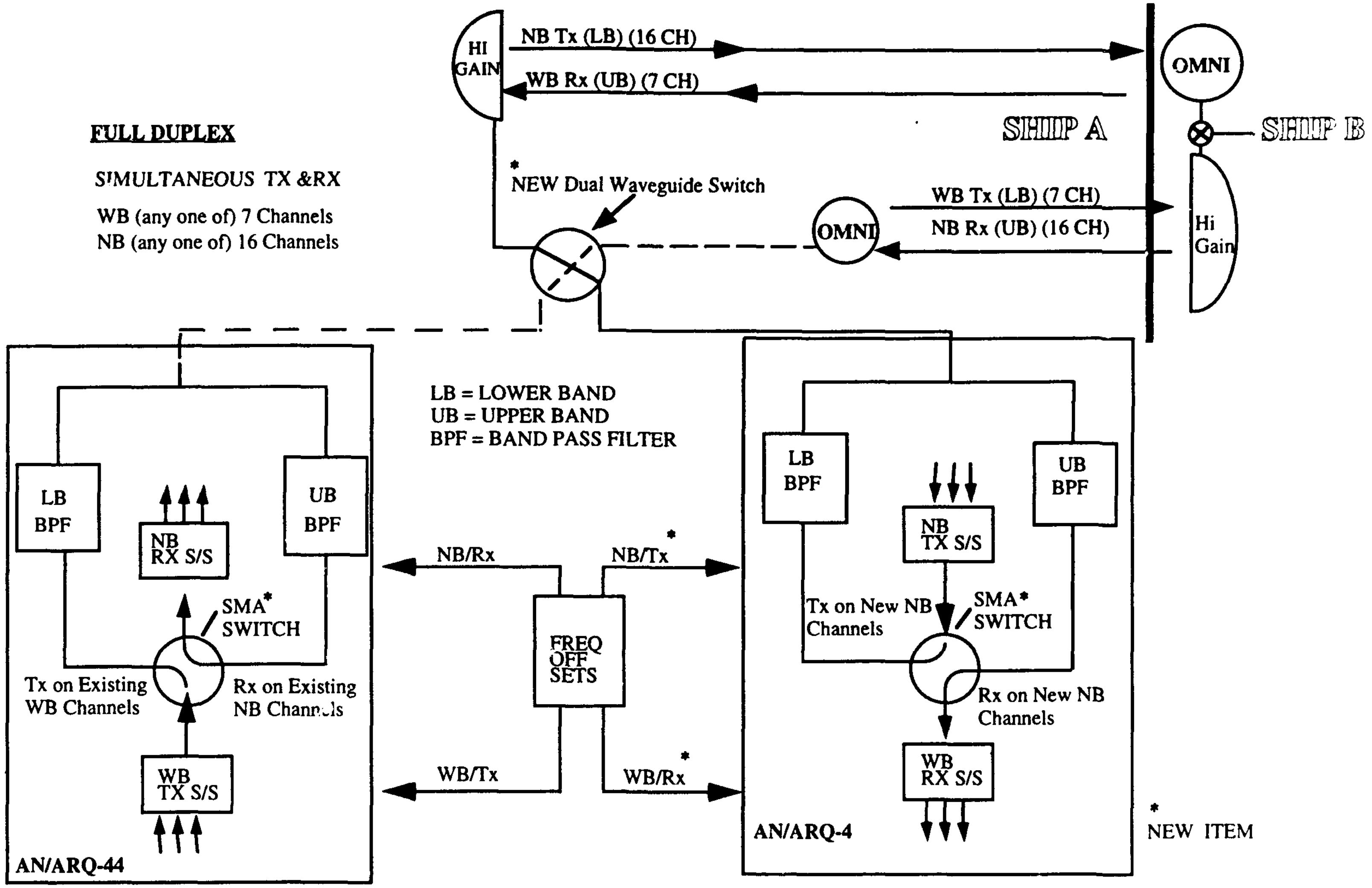


FIGURE 3.1

MAX. CAPABLITY: FULL DUPLEX WB & FULL DUPLEX NB LAMPS MK III COMMUNICATION SYSTEM



FULL DUPLEX

SIMULTANEOUS TX &RX

WB (any one of) 7 Channels
NB (any one of) 16 Channels

LB = LOWER BAND
UB = UPPER BAND
BPF = BAND PASS FILTER

FIGURE 3.2

and upper frequency bands in each system. This arrangement provides a diplexer with as much as 90 dB of isolation between the receive and transmit signals when the ship is using either the high gain antenna or the omnni antenna. From Figure 3.2 we can see that the ARQ-44 is transmitting WB in the LB filter and the SRQ-4 is receiving WB in the UB to achieve the needed transmit to receive isolation. Therefore, on each of the platforms the separation between the diplexer bandpass filters, located in the UB and LB, is the same and is sufficiently great that transmission and reception can occur simultaneously without causing corruption of the quality of the received data.

For this test system the UB bandpass RF filter used for the WB transmit and receive functions and LB RF filter used for NB transmit and receive functions have not been tailored for this test application. It has been found by testing the LAMPS MK III diplexer filters that the bandpass characteristics of the UB filter was about an order of magnitude larger than was expected. Consequently, it will be possible to allow operation of the WB transmit through it in about seven channel locations. For this reason, it is estimated that full duplex operation could occur in seven channels. In the foregoing example we have proceeded, using the present LAMPS MK III communication architecture, to create two simultaneous full duplex loops; one for narrowband and the other for wideband communication. Whether the present 90 db isolation can be achieved or not remains to be determined. If the isolation level is unsatisfactory for simultaneous full duplex WB and NB operation, then nonsimultaneous WB or NB full duplex communication mode could solve the resulting interference problem. To achieve the WB/NB capability shown in figure 3.2, the following would be added to the basic HDR LAMPS MK III capability in order to equip two ships with a maximum capability:

1. (2) dual waveguide switches
2. (2) SMA type switches at input to the R/T subsystems.
3. (2) mods to Voltage Tuneable Signal Sources (VTSS) to permit shifting frequency references in accordance with the new WB UB and NB LB operation while still being able to meet LAMPS transmit and receive bands and channelization in the normal mode (1).

(1) Note: The mods of this new VTSS subsystem have been now built and tested.

4. SHIP TO SHIP HDR LAMPS MK III COMMUNICATION DEMONSTRATION SYSTEM

Operational demonstrations that might be associated with the evolution of this new ship-to-ship high data rate communication capability can take many forms. The first of these demonstrations begins with tests in the laboratory. At the next level we will make a bridge between the laboratory and the world of the Navy combatant ship that already has a LAMPS MK III system on-board. The specifics of the areas of investigation, in each of the above categories, is contained below.

4.1 NAVY LABORATORY TESTING

The work in this area was begun last year because of the support provided by NESEA, St. Inigoes, MD. Although the documentation supporting the LAMPS system is quite extensive, there are still many areas where documentation deficiencies were noted. Where information deficiencies occurred detailed design information and data on subsystem characteristics were derived by NESEA. In these situations NESEA conducted various measurements on the LAMPS equipment to reveal this needed design information. Because of security classification issues, some of these results cannot be presented.

The laboratory testing in preparation for the at-sea demonstration has been conducted at NAVSEA. To date, testing has shown that the new frequency references for upper band WB and lower band NB operation are functioning properly. A considerable effort has been devoted to the data interface for the test system. The LAMPS MK III WB downlink has evolved along telemetry lines, and the transmitted data words are commutated into specific data-framing locations. Therefore, along with the WB data which may be in either the LAMPS Radar or ASW channels, there are also discrete data frame locations that are used concurrently to carry 26-bit blocks of data plus 6 bits of EDAC. These slots are used to carry computer data, voice, and DME data. The computer channel is used for command and control information between the ship and the helicopter. The other two framing slots are available for the data block transmission that results in an independent 56 Kbps digital voice link and a 26-bit data block that is used for distance measuring equipment (DME) between the ship and the helicopter. The HDR data interface challenge develops from the fact that the LAMPS MK III data flow is not a continuous flow of one type of data as occurs in the normal commercial T-1 communications world. Data buffering using a PC-based FIFO (first-in, first-out) buffering system and a special serial I/O card plus strict timing controls have been developed by the Navy to achieve this interface compatibility. This required data interface has been developed jointly by NRaD and NESEA.

The next phase is the laboratory testing of the system which includes all the VTC equipment, Timeplexer multiplexers, data interfaces, HDR SRQ-4 and HDR ARQ-44 terminals, including the Cesium and Rubidium time standards and cryptos that would represent video data to RF interconnection of the two HDR LAMPS equipped ships. This would duplicate full duplex VTC at-sea testing using the HDR LAMPS that is to take place before the end of February 1994.

4.2 AT-SEA VTC DEMONSTRATION TESTING OF THE HDR LAMPS MK III COMMUNICATIONS SYSTEMS

Figure 4.1 presents a pictorial representation of the at-sea VTC testing with the USS Mount Whitney that is planned for the April/May 1994. Prior to this test there will be an at-sea test using only two LAMPS MK III ships operating on a point-to-point basis. In this early at-sea test there will be no requirement to interface with the Satcom VTC. The April/May testing will require the Satcom interface that will permit video teleconferencing between the CINCLANTFLT Headquarters and the HDR LAMPS MK III ship. The USS Mount Whitney does not currently have a LAMPS MK III installation, so it is being equipped with a shelter to house the HDR LAMPS MK III capability and will use an omni antenna that is telescopically mounted to the roof of the shelter.

Ship-to-ship VTC tests will be conducted at sea at various ranges and shipboard EMI (electromagnetic interference) conditions. This demonstration will reveal the level of electromagnetic interference that will be experienced by the HDR LAMPS MK III receive system in the WB mode. Also, the level of RF interference with other ship systems that will be created by LAMPS transmitter emitting a WB signal, using either the omni or the high gain directional antenna at SHF, will be examined.

4.3 MINIMUM COST HDR LAMPS MK III VTC DEMONSTRATION SYSTEM

Figure 4.2 presents the HDR LAMPS MK III minimum-cost system to be used in at-sea VTC tests. This system will allow full duplex WB data communications between two ships to support the VTC. As in the HDR configurations discussed, it is necessary to integrate an ARQ-44 with the shipboard SRQ-4 to achieve this WB capability. Figure 4.2 shows that, since this is a test, simplification of the system is possible. It can be noted that ship A and ship B have particular connections of the bandpass filter to the WB receiving and WB transmitting subsystems. Both of these connections are manually made via SMA lines within the SRQ-4 and the ARQ-44. The restriction to WB operation removes the need for the dual waveguide switches shown in figure 3.2. When both ship A and ship B are LAMPS MK III ships, then either one could be ship A or ship B, since they both have the same antenna systems. In this drawing it can be noted that in the ship B configurations the ARQ-44 would be configured as it normally is in that it transmits WB on the LB/BPF. On the SRQ-4 side, we can see that the shipboard system, which normally receives WB on the LB/BPF, is now to receive it from ship A in the UB/BPF. When the Satcom VTC testing occurs, the shelter that contains the HDR LAMPS MK III equipment will be represented by the configuration shown as ship A since it will use the 0 db omni antenna. The data interface developed for this system is as shown and for a number of reasons will operate only at twice the T-1 rate or about 3 Mbps. Other than the SMA switching shown and the VTSS changes, the rest of the ARQ-44 and SRQ-4 radio terminal equipment will remain the same as it is today.

5.0 PREDICTED COMMUNICATION PERFORMANCE

Using a Navy computer program (PC) called SLAM that allows a parametric examination of the communication performance of any particular system, including the effects of multipath, propagation diffraction, and ducting, the plot shown in Figure 5.1 was developed. The actual characteristics of the present LAMPS system were utilized including the system losses that affected both the transmitted signal as well as the received signal. These losses were provided in the engineering documentation in the Navy files on the LAMPS MK III. The term on the ordinate in Figure 5.1 is called margin and stands for the signal power level,

AT-SEA TEST OF HDR LAMPS MK III EXTENSION OF CINCLANTFLT SATCOM VIDEO TELECONFERENCING SYSTEM

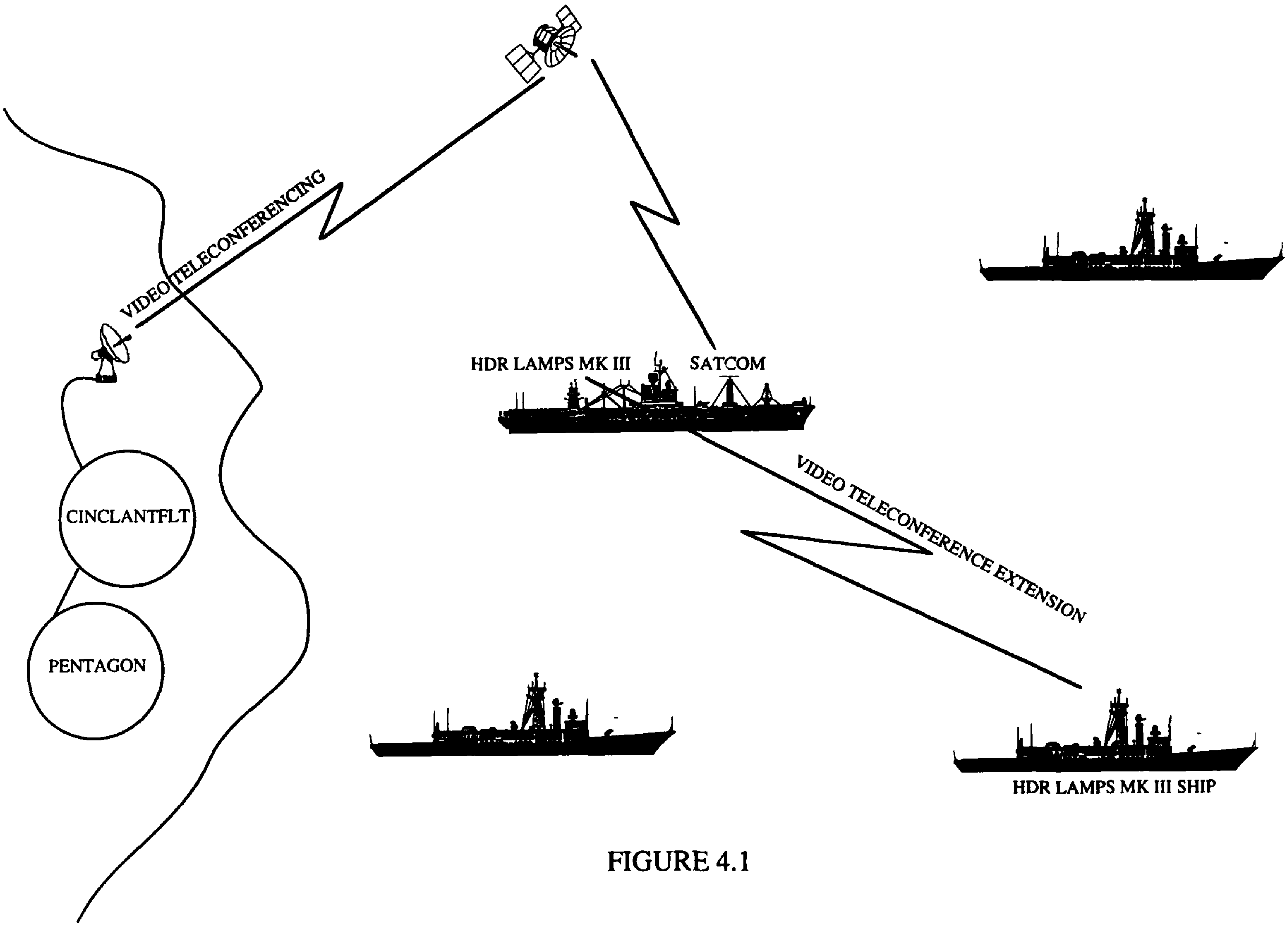
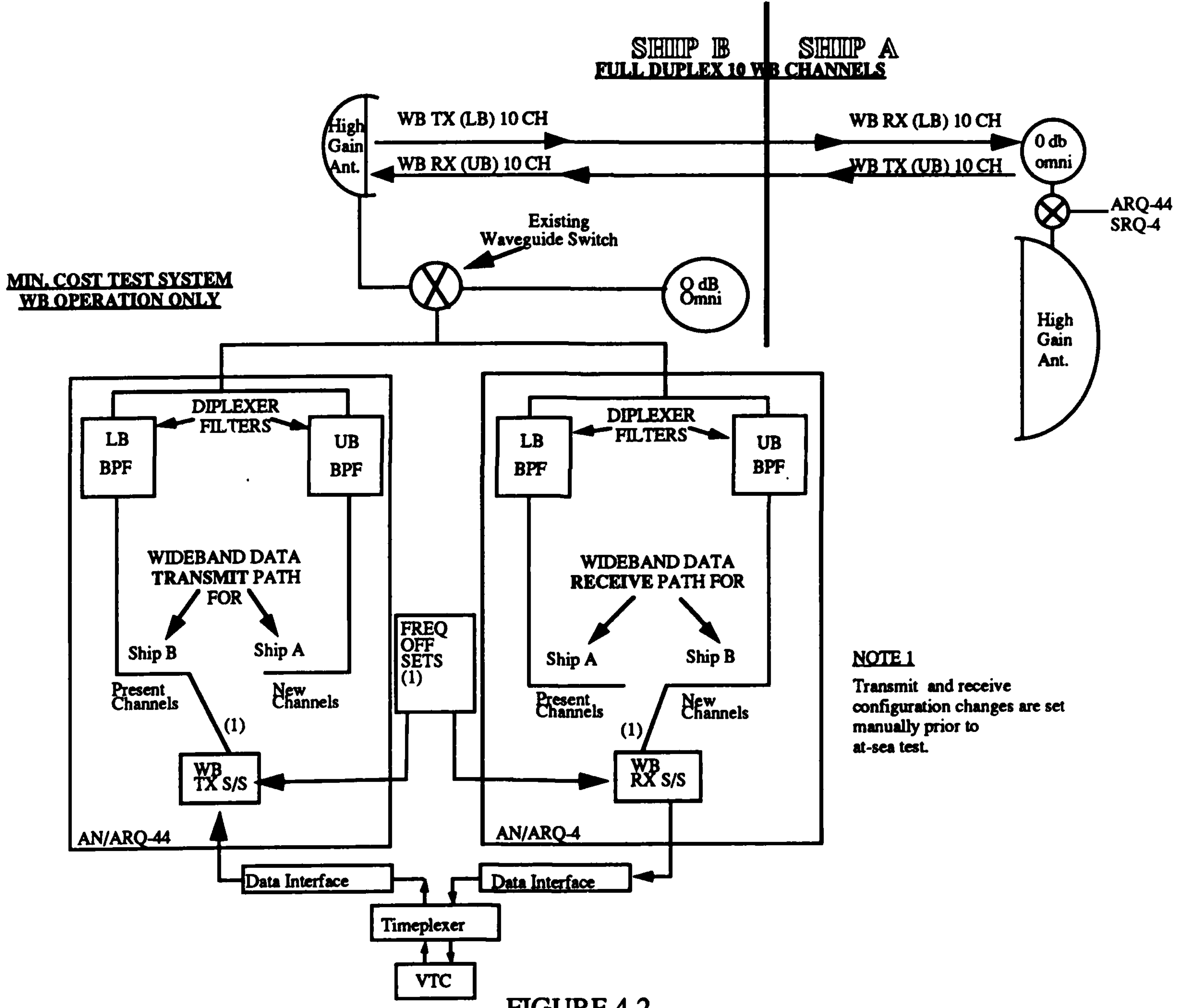


FIGURE 4.1

LAMPS MK III HDR TEST CONFIGURATION FOR AT-SEA VIDEO TELECONFERENCING TEST



NOTE 1

Transmit and receive configuration changes are set manually prior to at-sea test.

FIGURE 4.2

PREDICTED SHIP TO SHIP COMMUNICATIONS LINK
PERFORMANCE USING HDR LAMPS MK III DURING
AT-SEA VIDEO TELE-CONFERENCE TEST

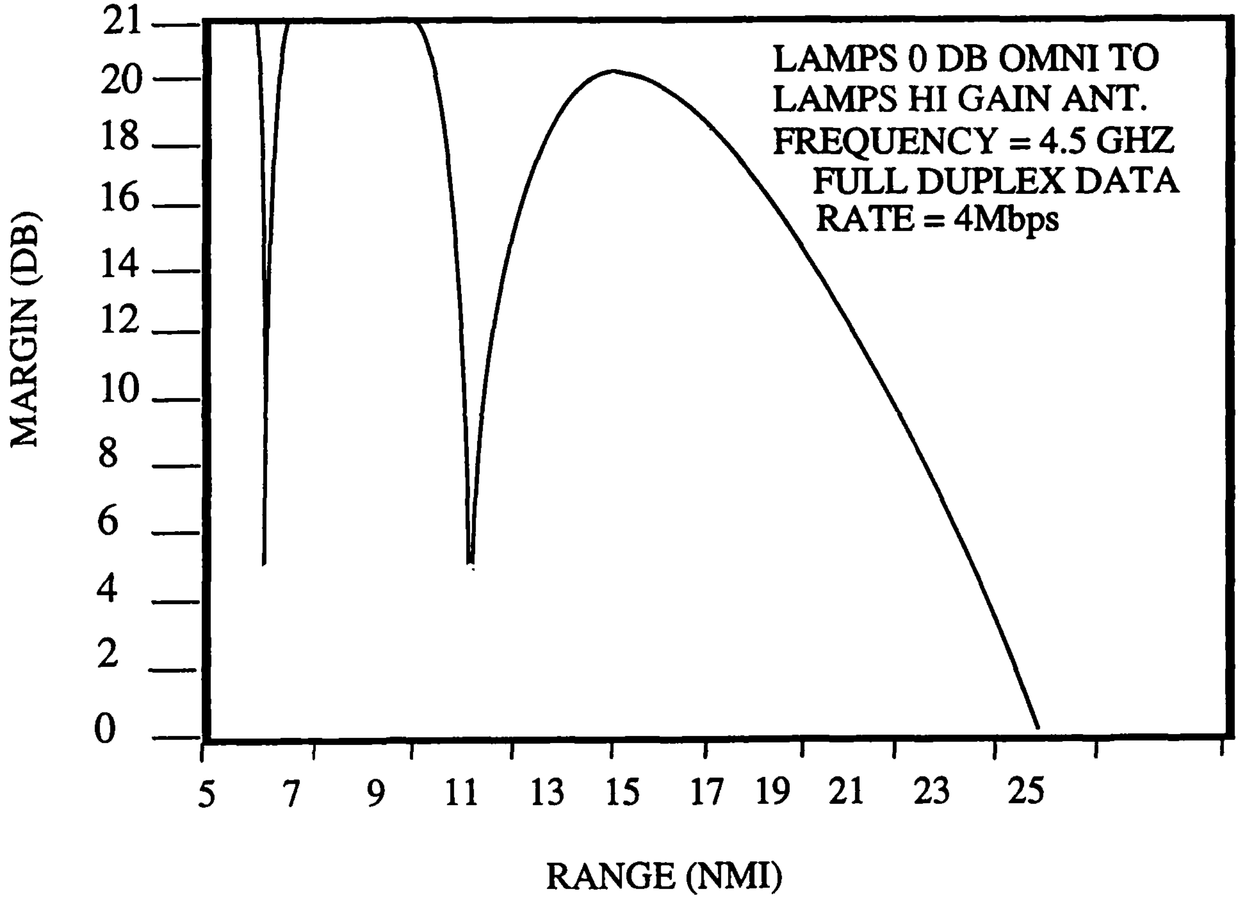


FIGURE 5.1

in db, greater than that level which will just allow achieving the desired data bit error rate (BER).

For the link computations that resulted in Figure 5.1, it was assumed that the antenna on each ship was at 100 ft above sea level, and this results in a maximum line-of-sight range of about 20 nmi. When sufficient excess power is available range values will be in excess of the LOS limit out to a diffraction loss limit. Figure 5.1 shows the expected performance of the LAMPS communication link between the high gain LAMPS antenna and the LAMPS "0" db antenna as a function of range assuming a communication data rate of 4 Mbps with a Eb/No = 12 db. The sharp downward spikes in figure 5.1 are due to multipath signal degradation effects. The worst of these can be seen to be about 5 db above the level which would create signal degradation. This is regarded as the multipath safety margin that exists in this link. The received signal power is sufficiently strong as to allow communication out to ranges of about 25 nmi including the link diffraction losses.

6.0 CONCLUSIONS

The US Navy has a valuable resource in the LAMPS MK III system and this has been demonstrated in its remarkable operational performance. It has been operational in the fleet for at least 10 years. It is a well engineered system that in many ways is still very sophisticated. Its excellent performance results from the soundness of its engineering. It is an example of what is possible in operational fleet electronics. For these reasons it represents a very sound foundation upon which to create another remarkable level of LAMPS MK III usefulness to the Navy. It presently has an operational role which it performs efficiently and effectively. Therefore, variations from its present design have to be very carefully evaluated to be sure that no reduction in present operational capability will occur if any design changes are to be implemented.

This paper has touched on some specific communication and operational scenarios. Others being examined, but not discussed, includes networking of an ethernet type between HDR LAMPS MK III ship's that uses the high gain to omni link between ships to achieve a 4 Mbps network trunk. In another version networking is achieved only between the HDR LAMPS MK III omni antennas at a NB rate. What the HDR mode offers to the Navy is a communication trunk that can serve many forms of data communication simultaneously. It is, in fact, ideally suited to serving the new world of multimedia including large file transfers, digital interactive video, high resolution image transmission and digital voice. There are also a variety of possible operational military configurations, that appear to have very desirable features. At this juncture the technical investigations and operational testing will continue so that greater insight will be developed as to its most desirable and achievable operational capability.

Biography

Mr. Rahilly originated the concepts presented in this paper and is the Program Manager over the HDR LAMPS MK III development. The program is being conducted in the RDT&E Division, Communication Technology and Systems Branch. After graduation from what is now New York Polytechnic University in 1951 he was a systems engineer at Westinghouse Air Arm Division, Raytheon Missile Division, General Electric Technical Military Planning Operation, and Philco Ford Western Development Laboratories. He joined the NAVY laboratory on Pt Loma, San Diego Ca, that has now become the RDT&E Division, in 1968 and has been employed there on a variety of communications programs since that time.

SUCCESSFUL PROCESS IMPROVEMENT EFFORT USING CLEANROOM SOFTWARE ENGINEERING

S. Wayne Sherer, AMCCOM LCSEC
Paul G. Arnold, IBM
Ara Kouchakdjian, SET

Abstract

The results and lessons learned from a STARS (Software Technology for Adaptable Reliable System) sponsored process technology transfer demonstration are presented in this paper. The Armament, Munitions and Chemical Command (AMCCOM) Life Cycle Software Engineering Center (LCSEC) at Picatinny Arsenal was selected to demonstrate that Cleanroom Software Engineering (CSE) and Process-Guided Project Management (PGPM) could be successfully applied in a typical DoD Software Support Activity (SSA). Results indicate that:

- CSE practices and PGPM can be successfully transferred to a typical DoD SSA,
- engineering staff productivity and quality was increased while simultaneously increasing job satisfaction, and a
- return on investment of at least 5.9:1 has been realized on the first project to which CSE and PGPM techniques were applied.

Technology Transfer Goal

The goal for the technology transfer effort for the AMCCOM LCSEC at Picatinny Arsenal was to conduct a demonstration of CSE practices and process-guided project management at a typical DoD SSA.

AMCCOM LCSEC was selected in response to their expressed interest in improving the process by which they maintain software in general and, specifically, in using the CSE technology. Additionally, as a typical DoD SSA, it was deemed important to improve the means by which the government spends their largest portion of software money; i.e., in software maintenance and re-engineering (as opposed to new software development). The demonstration was to be facilitated by IBM and SET (Software Engineering Technology, Inc.).

The LCSEC at Picatinny Arsenal is a representative DoD Software Support Center that wants to apply a more formal approach to software support. The current state of software re-engineering at the AMCCOM LCSEC varies from project to project but the majority have not achieved the desired level of productivity and quality. A major goal of AMCCOM LCSEC is to achieve an Software Engineering Institute Capability Maturity Model (SEI CMM) Level 3 rating by adopting an evolutionary process improvement approach to software re-engineering. Currently AMCCOM LCSEC is receiving support, under STARS Task IA02 from IBM and SET, in applying the Cleanroom approach on the re-engineering of the Mortar Ballistic Computer (MBC). Initial results have been successful in terms of the projects employing the Cleanroom engineering practices and adopting a process driven team organization.

Organization Overview

DoD Software Support Centers (SSCs) provide important opportunities to demonstrate STARS efforts to improve software quality and productivity. SSC activities represent a major portion of the DoD software budget and the proportion is expected to be increased during the next decade. This will occur as the many systems in the DoD development pipeline are turned over to SSCs for support. It is likely that, as fewer new systems come into the inventory, DoD managers will attempt to extend the useful life of old systems through software enhancements and re-engineering..

The AMCCOM LCSEC provides a number of services including: software acquisition support to program managers, computer resource life cycle management plans, pre-planning for software support, manage contracted post deployment software support efforts, software configuration management, and design and *implement software changes.* The types of battlefield automated systems supported include air defense, cannon and tank gun systems, smart mines and munitions, ballistics computers, gunnery simulators, trainers, and nuclear biological chemical detection systems.

The MBC project was a re-engineering of the current system used by the US Army to aim Mortars for combat support. The existing MBC was implemented in DTL (Display Terminal Language) and Z-80 Assembler that is not easily upgraded for new requirements. The re-engineered system was implemented in Ada and as a result can be moved to new updated hardware platforms.

The STARS program is a DoD funded research and development effort funded under ARPA (Advanced Research Projects Agency). The main thrust of this effort is that software development is process-driven, domain-specific, reuse-based, and supported by an integrated software engineering environment. This concept is called Megaprogramming and the STARS program is currently engaged in several demonstration projects of the technologies developed earlier in the program. Picatinny MBC effort is the first demonstration project to use STARS concepts. IBM is one of the prime contractors for this effort and SET is a principal subcontractor for the IBM/STARS effort.

The desire for process-driven technology was the result of a Software Process Assessment (SPA) conducted by a team of representatives from the AMC LCSECs with coaching from the Software Engineering Institute (SEI). AMCCOM LCSEC management has developed a close relationship with the SEI because they desired help with identifying areas to achieve the desired level of productivity and quality. Review of the SPA findings lead AMCCOM LCSEC management to realized that the software engineering process was not under intellectual control. Each new software project, whether performed by contractors or civil servants, was treated largely as new activity that did not necessarily draw on prior experience for process improvement. The only factor that perpetuated experience was people, be it government or contractor, who participated in the same projects time after time. Documentation received by Picatinny, when they were given systems to maintain, was poor or not up to date and no defined process existed for maintaining continual project control. In other words, the state-of-the-practice consisted of traditional software engineering practices that are ad-hoc in nature, as opposed to a disciplined, defined software engineering process. These realizations and the results from the SPA were the basis for their move to enhance their software engineering capabilities.

Typical DoD SSA organizations have immature processes and are subject to morale problems among software engineers due to the combination of an undefined manner of doing work, along with a lack of task-oriented scheduling. The software engineers at the AMCCOM LCSEC did their work well because of individual skills, but often seemed to be stuck in the same "groove," where the same situations, in terms of schedule, would arise year after year. A general lack of enthusiasm pervaded our initial discussions with project teams.

Despite these difficulties, however, the customers (various users within the US Army) indicate that they are basically content with the quality of the products. Not many field reports of failures are submitted by their customers, due to extensive, pre-release user testing. Unfortunately, evidence suggests that this may also result from the absence of formal failure observation and reporting mechanisms, making the field quality of AMCCOM LCSEC developed products difficult to ascertain.

AMCCOM LCSEC management recognized the problems with their state of the practice and took the initiative to recognize Cleanroom Software Engineering (CSE) and Process-Guided Project Management (PGPM) as the mechanisms with which to facilitate the desired cultural, technical and process changes.

Cleanroom Software Engineering (CSE)

CSE was chosen as the process driven technology because it addresses the deficiencies identified during the LCSEC Software Process Assessment (SPA). CSE's management and development team approach was consistent with quality management philosophy, e.g. workforce empowerment, process focus, and quantitative orientation. It provides for the transition of process technology to the project staff and integrates several proven software engineering practices into one methodology. LCSEC management anticipated productivity gains and morale enhancement from the introduction of the technology.

CSE consists of a body of practical and theoretically sound engineering principles applied to the activity of software engineering. Cleanroom consists of a thorough specification phase; resulting in a six part specification, including a precise, **black box** description of the software part of a system. Software development proceeds from the black box specification via a **step-wise refinement** procedure using box-structured design concepts. This process focuses on defect prevention, effectively eliminating costly error removal phases (i.e., debugging) and produces verifiably correct software parts. Development of software proceeds in parallel with a **usage specification** of the software. This usage profile becomes the basis for a **statistical test** of the software, resulting in a scientific certification of the quality of the software part of the system.

A quick high level comparison between the typical development and CSE philosophy of software development is summarized in Table I. The typical development environment can be characterized by craft based techniques which are highly dependent upon the skills of the individuals involved whereas CSE is an engineering discipline with associated rigor and formality.

Table I: Comparison between Typical Development and CSE

Characteristic	Typical Development	CSE
Programs regarded as	Lines of Instructions	Correct rule for a function.
Specification focus	Incomplete description of external behavior and internal design details.	Complete, precise description of external behavior; design details left for development.
Specification to code transformation process	Informal, debugging to verify code.	Stepwise refinement and verification using Box Structures.
Failures are	Expected and accepted.	Unacceptable.
Testing strategy	Futile attempt for coverage and little insight on field reliability.	Random sample based on usage model that predicts field reliability.

Transfer of CSE Technology The transfer of CSE technology was achieved through formal, classroom-style training courses and follow-on coaching of demonstration team members. The courses involved instruction on the underlying specification, development, and certification methods of CSE and included in-class workshops so that students gained experience applying the technology. As often as possible, workshops were supplemented with examples extracted from the MBC project. Training provided the introduction to and initial experience with the tools that would help enhance individual and team performance.

Project support was given to the team members through repeated on-site coaching visits by CSE experts from IBM and SET. This activity helped to solidify the new ideas as team members saw how the techniques were applied to their specific problems.

The major intent of the training and coaching was to establish the human behavioral changes necessary to develop better software. Implementing CSE is an intellectually challenging process that instills specific values into its participants. For example, the focus on product quality, a major Cleanroom theme, instills a "get it right the first time" attitude into the members of CSE teams. As successes were made and milestones conquered, the CSE teams reported significant improvements in job satisfaction, team spirit, and the desire to continue quality improvements. A significant focus of the coaching effort was to positively reinforce each project success in order to create a stronger identity with the project.

Such behavioral changes within a project are improved by active participation from all levels of the organizational hierarchy from contributing technical leads to engineering management. The initial plan was for the project staffs to work closely as teams, rather than as individuals. Additionally, the intention was for the staffs to be motivated and excited about what they were doing; that is, have a strong identity with the process and project. Thus, coaching contained a "cheer leading" aspect, designed to create a healthy Cleanroom environment.

Reinforcement of CSE was provided through the availability of a six volume set of process manuals to the demonstration groups. These process manuals were an integral part of the training program and were discussed in detail, both during the formal training sessions and off-line as a part of the follow-on coaching activities. Their purpose was to augment the training by providing reference information to AMCCOM LCSEC engineers using Cleanroom concepts. They serve as a single reference source for resolving questions about specific issues concerning process adherence. The process manuals are organized as follows:

Volume 1:	Cleanroom Engineering Process Introduction and Overview
Volume 2:	Organization and Project Formation in the Cleanroom Environment
Volume 3:	Project Execution in the Cleanroom Environment
Volume 4:	Specification Team Practices
Volume 5:	Development Team Practices
Volume 6:	Certification Team Practices

The division of the volumes represents a separation of concerns for the various project stakeholders.

Process-Guided Project Management (PGPM)

CSE is a formal process that clearly defines the tasks necessary for the engineering effort to progress, the completion conditions for each task, and the control flow that dictates the order of work on each task. Process-guided project management entails the use of a clearly defined process as the approach to be used to complete the particular project. The intent with process-guided project management is to give engineers a clear and understandable road map which they can follow and by which they may track progress towards project completion.

Transfer of PGPM methods : Awareness of software process is a key issue in successfully transferring technology to an organization and to an organization's long term success with applying CSE. The project staffs at AMCCOM LCSEC received an introduction to process definition and process-guided engineering in the context of CSE. Coaching also reinforced the importance of following the defined process and using the process definition, which defines the possible project alternatives, to support the selection of correct project choices.

In addition to training and coaching, the engineering handbooks provide a key reinforcement of the concepts of process-guided engineering. Each volume defines the tasks and the control flow between the tasks necessary to conduct the specific process which is the focus of the manual. Engineering processes are defined as formal control-flow procedures with specific completion conditions. Collections of engineering processes also have the same level of formalized control flow and completion conditions. Thus, each engineer, manager or other staff member has well defined roles and tasks that exist as a part of a larger software process.

The application of the process is supported by formal enactment of the tasks defined in the handbook. For the MBC team, this enactment was automated in the Cleanroom Engineering Process Assistant (CEPA), an automated process support system which had the following mission:

1. To minimize time lost because supporting activities are not properly coordinated. CEPA was to significantly improve the probability that all of the pre-requisites, tools and data that an engineer needs to do a task are available with no wasted time on his or her part.

2. To enable engineers to follow the Cleanroom process and thereby obtain all of its benefits.
3. To enforce the Cleanroom process in the most unobtrusive way possible by being user-friendly.
4. To enable all levels of management to plan, schedule and control project tasks and to ensure that the required reviews and verifications take place.
5. To facilitate the collection of all required metrics for providing statistical control of the process and for providing better estimates of development time and cost.
6. To update on-line state data, the data needed to develop the product, and make it immediately available to all members of the project team.
7. To improve formal and informal communication between the members of the group.

The engineering handbooks, and the automated enactment gave project staff a way to use a project framework (the process model for the project) that facilitates scheduling, task dispatching and task statusing.

Technology Transfer Description

To conduct the demonstration, both control and demonstration groups were identified. The control group consisted of a sample set of ongoing and completed software projects at the AMCCOM LCSEC. These projects represent the use of "typical" software engineering methods at the AMCCOM LCSEC. Enhancement projects at Picatinny typically include the correction of observed problems, the addition of new capabilities, and in some cases, re-engineering of software. The demonstration project was the Mortar Ballistics Computer (MBC) re-engineering effort. The demonstration aspect of this project was the adoption of the CSE technology and PGPM techniques as provided by the participation of IBM and SET. The hypothesis to be confirmed or rejected in this demonstration was: **The use of CSE practices and process-driven project management improves the effectiveness (quality and productivity) of the AMCCOM LCSEC software support mission.**

The technology transfer package was implemented as follows:

(1) the *transfer of Cleanroom Engineering practices* to give team members the technical tools that provide the human behavioral changes necessary to create high quality software with increased productivity, and

(2) the *transfer of process-guided project management* to orient both individuals and teams to thinking and working within a PGPM environment.

In order to transfer the technology, process and culture for a Cleanroom environment, four different tools were employed:

(1) training, in a formal classroom setting which integrated lecture material and numerous hands-on workshops (tailored for this effort),

(2) coaching, both for project planning and execution as well as a medium to promote ongoing education,

(3) process handbooks (evolved for this effort), which act as a written source of education material and as a reference during project execution, and

(4) an automated process support system (developed for this effort), that helps enforce process adherence and monitors task completion, by automating non creative tasks.

Baseline Metrics for Control Group Projects

The control groups represent the state-of-the-practice at the AMCCOM LCSEC. Baseline metrics were collected in order to gain insight into project practices and to establish a basis of comparison to the demonstration Cleanroom groups. Table II presents the baseline metrics for the control group. These metrics are presented with the caution that some data collection mechanisms are unreliable, resulting in inaccuracies. The numbers in Table II are similar to results reported by Mosemann for other projects within the DoD [Ada and C++: A Business Case Analysis, July 1991].

Table II: Baseline Metrics for Control Group Projects

Project / Measure	Control Group Projects
Number of Projects	5
Range of Effort - Staff Months	21-58
Total Technical Staff Months	192
Total KLOC (*)	23.14
DERIVED METRIC: Productivity - LOC/Staff Month	121

(*) KLOC computed using NASA/Goddard formula of:
(New Lines of Code + 0.2 * Modified Lines of Code) / 1000

Observations

The following observations are a compilation of IBM and SET experiences with the MBC teams. These observations are in the context of IBM's and SET's other experiences with replacing craft-based practices with engineering-based practices, both in the private sector and with government organizations. One must keep in mind these observations are preliminary since the project has not been completed.

1. *The assigned project teams were able to assimilate and even adapt the Cleanroom Software Engineering practices and process-guided project management.*

A common worry among managers when hearing about Cleanroom is that it is too hard or too mathematical for their staff. At Picatinny, engineers were able to apply and adapt the Cleanroom practices to the needs of their project. Engineers learned, used and extended the ideas successfully for their project. The evidence of this observation is the products they have produced.

Disciplined engineering in a team environment requires rigor, cooperation of individuals, and the creativity to apply theory to real world problems. This creates a challenging work environment that tends to bring out the best in both individuals and teams.

A prime example of the accomplishments of the MBC team was the tailoring of the box structures algorithm to meet both their application environment and the target programming language, Ada. MBC team members have made original contributions to the expression of box structure constructs in Ada, which will have applicability across many Cleanroom projects. This has benefited both the project, in terms of constructive methods, and the individual team members, in terms of a sense of accomplishment. The team has enjoyed using the various Cleanroom techniques and have seen many real accomplishments. The specification team is convinced that this is the most complete and precise specification they have ever written. The step-wise refinement and verification, which drives engineers to define one small step to take at a time, take that step, and then confirm its correctness, has also been successful. The development team is convinced that they have a great design and have minimized the amount of code they need to develop.

Furthermore, as the MBC team has almost completed their second increment, it has already shown major gains in productivity. Early estimates show that productivity has tripled despite the learning curve of working with a new methodology. Moreover, this measure includes time spent toward an entire product specification, which will make future increments less time consuming. Thus, team members are optimistic about continued increases in their productivity (although future predictions can only be assertions and remain to be confirmed at project completion).

2. *Staff morale has improved on the project teams.*

Another common fear of managers when hearing about Cleanroom is that their staff members will not like it due to the rigor of the process and the absence of positive feedback through debugging. This has not been our experience at other places where we have introduced Cleanroom. Picatinny is no exception. When an organization replaces craft-based practices with engineering-based practices, morale improves. The reason seems to be that people now know what to do, when to do it, and how it should be done. This eliminates the uncertainty and anxiety for project teams that now have a good knowledge of exactly what has been done and what is remaining to be completed. When engineers learn to use the Cleanroom practices, they know they can do the high quality job they have been striving to achieve. Engineers are convinced that they are producing a better product. As a result, they are excited about it.

At the AMCCOM LCSEC, all the engineers, both in informal contacts and in a questionnaire distributed to the engineering staff, reported morale improvements. The AMCCOM LCSEC management has also confirmed the existence of the improved morale and, of course, is favorably impressed.

3. *Facilitation of work effort is greatly enhanced through process-guided project management.*

Team leaders managed by process definition and task lists which allowed more visibility of project status by management. CEPA was the tool used to clearly define the tasks necessary for the software development process to continue including completion conditions for each task and the control flow that dictated the order of work on each task. The intent with process-guided project management is to give engineers a clear and understandable road map which they can follow and by which they may track progress towards project completion. Awareness of software process is a key issue in successfully transferring technology to an organization and to an organization's long term success with applying a given process-driven approach to project management. PGPM must provide for:

- the reduction of time lost because supporting activities are not properly coordinated,
- enable engineers to follow the defined process and thereby obtain all of its benefits
- enforce the defined process in the most unobtrusive way possible by being user-friendly

- enable all levels of management to plan, schedule and control project tasks
- facilitate the collection of all required metrics for providing statistical control of the process and for providing better estimates of development time and cost.
- update on-line state data, the data needed to develop the product, and make it immediately available to all members of the project team.
- improve formal and informal communication between the members of the group.

4. *The team-oriented approach of CSE saw immediate acceptance and realized both tangible and intangible benefits.*

A key ingredient of Cleanroom is that a team amplifies human performance. People took advantage of the insight of others in order to bring the best possible project result. Good people working together brought in better results. The simple idea that many minds are better than one makes the outlook for quality good. However, some less tangible benefits were realized as well. The fact that the entire team is responsible for quality, in a series of checks and reviews, puts pressure on the team and **not** on individuals. This pressure creates a reliance on team activity over individual performance. Furthermore, as successes are encountered, the entire team takes credit, not a single individual, thus, cementing the teamwork concepts. The bottom line is that teamwork improves individual performance.

Our observation is that the MBC team now works within an effective team-oriented environment. We believe that further use of Cleanroom will establish a strong team mentality that will serve to further improve the initial good results.

5. *Coaching is a key ingredient of technology transfer success.*

Although the training was rigorous with a mixture of theory and workshops, students learn at different rates. Coaching allowed IBM and SET staff to re-educate the slower-to-adopt project staff members and keep the entire team on a common level of knowledge and expertise. IBM and SET technical presence at project inception and during project execution helped solidify the transfer of the technology and ensured that the project got started in the most efficient manner.

Furthermore, there was a gap between the end of training and the start of the project and some of the education was forgotten. Coaching became the mechanism to re-educate and supplement the original training. Further, as good ideas were conceived by some team members, it was possible to see that all members were supplied with the new ideas.

As the project progressed, the CSE ideas needed to be adapted to the *specific Picatinny environment.* Coaches were used to discuss design alternatives and to help in refining the technology to best serve the application.

Perhaps the most unnoticed but effective use of coaching was in the positive reinforcement the CSE trainers were able to give to the team members and the team as a whole. Coaches are recognized as experts. When experts comment positively on original ideas by a team member, the effect can be enormous in terms of self-esteem and sense of accomplishment and contribution. The CSE trainers tried to positively reinforce the behavior of those making such contributions and encourage others to seek answers beyond the limits of current knowledge. The "cheer leading" approach increased project satisfaction, which motivated greater project performance.

The idea of coaching with positive reinforcement was first formally tried out by IBM and SET on the Picatinny project based on the hypothesis that it would be helpful in technology transfer. The realized benefits far exceeded our expectations. Based on this experience, it is now believed that coaching should be a formal part of any technology transfer effort.

6. *Communication among teams (and between team members) is greatly enhanced through process-guided project management.*

An important ingredient of any proccss-guided activity is communication among contributing teams and individuals. One aspect of this was that no team culture existed at the AMCCOM LCSEC; meaning that no real notion existed of how teams are *supposed* to behave during project execution. This problem manifested itself in many different ways. Testing teams often did not receive specification updates (and failed to ask for them). Also, work tended to be duplicated by multiple team members because the division of tasks was unclear and communication among members occurred too seldom.

There were two aspects of solving this problem at Picatinny. The first was to establish effective communication among team members and the second was to establish communication among the different departments involved in the project. The adoption of a well defined process includes a vocabulary that is of great help to the understanding and discussion of the process. This well defined vocabulary makes communication between team members much more effective and productive. Our observation indicates that communication among team members significantly improved via the team approach and strengthened through the use of CEPA. The improved communication also started a shift in the culture of the teams. Team members report that they readily use each other as information sources, quality checks, etc. Team reviews are effective and informative. However, the second aspect, communication between departments, continues to be a problem. The MBC certification team members work for a different department than the specification and development teams. Resulting problems are that the certification team finds themselves working from outdated specifications. Furthermore, the certification team seems to duplicate each other's work. A future goal is to be able to duplicate the success of the specification and development teams in the certification team, primarily by improving communications. A more concerted effort should have been made by the coaches to minimize these communications problems.

7. *Process-guided project management supports engineers in mastering a new technology.*

Process-driven, now referred to as "Process-guided," project management is one of the two basic technologies being advanced by the STARS program. The Picatinny project was the first project on which this key idea has been employed.

The reason process-guided project management seems to support technology transfer can be summarized as follows. When doing something for the first time, one often asks, "What do I do next?" or "When will I be done?" This indicates a lack of understanding the big picture, where engineers can clearly place their efforts in a project context. This is not only an attribute of first time usage of techniques or a process, but also an indication that a clearly defined process does not exist or is not effectively managed.

By placing the Cleanroom techniques within a fully defined process, AMCCOM LCSEC engineers knew precisely what step they were currently on, as well as what had been completed and what remained to be done. Giving each individual the foresight that showed where they were in the context of the entire project strengthened project identity and boosted morale.

The results of this project also indicate that experienced engineers will also gain productivity benefits by employing a process support system but that can only be tested by comparing the performance of experienced teams which was not possible at Picatinny.

8. *CEPA, the Cleanroom Engineering Process Assistant, despite some shortcomings, provided valuable process-guidance support for the project.*

There were a number of known, as well as discovered, shortcomings in developing and using CEPA. It was an enhancement of a prototype system developed during a previous STARS Phase. The enhanced system was to provide support to engineers using a specific Cleanroom process model. This approach was known to be somewhat limiting, but was used in order to determine the level of constraint necessary for engineers to easily adopt process-guided engineering. Although the engineers did report finding the product constraining, CEPA did allow engineers to identify the tasks assigned to them and locate all files necessary to complete the tasks. Team leaders could also focus their management effort based on assigned/outstanding and completed tasks. This status reporting feature allowed team leaders to manage project tasks at a more reasonable level of granularity, which permitted them to maintain the project under greater intellectual control.

CEPA was viewed as being tightly coupled with the process. As a result, formal training in using it was not given, which would have also made its use more effective. The lack of CEPA training was a significant shortcoming that needs to be rectified in future technology transfer efforts. Additionally, formal training in using the underlying tools in CEPA would have been useful. Other problems with the CEPA implementation used at Picatinny included a clumsy user interface and difficulties in using the software on a network.

It was observed that automated process support is quite helpful in supporting technology transfer. This is in spite of some of the shortcomings of the system that the MBC staff was asked to use. The developers of CEPA learned a great deal about how people use such a system; and consequently, requirements for an enhanced process support system were identified and modified. The automated process support system that is to be transferred to Peterson Air Force Base has been improved as a result of this usage.

9. *Specific technological aspects of the Cleanroom Software Engineering practices were easily and successfully used.*

Using specific techniques are means by which engineers change their behavior and improve their performance. Three techniques in specific were discussed by project staff as being major sources of their improved performance. These techniques are team reviews, Cleanroom specifications and box structured design, and are described in greater detail below:

Team reviews, although experiencing a slow, awkward start, were cited by team members as one of the most successful aspects of the new activity. Members report that the team shared responsibility eased misgivings about participating in such a big project. This negated "finger pointing" that existed in previous projects and allowed even difficult personality combinations to work together. The result was that everyone participated and worked as a team toward project success and completion. Morale increased sharply as groups of individuals transformed into an effective software team.

Cleanroom specification, most notably black box documentation, was cited as being responsible for gains in productivity. Many talented engineers existed on the project and their productivity was significantly enhanced when working from a well defined problem statement. The completeness of the specification was the main reason cited for the team's confidence that they were producing a high quality product.

Box structured design is credited with focusing the code generation process and with making team reviews more effective. The team enjoyed the orderly process of developing software. It got them started more quickly on solving a particular problem and they were able to measure the progress of the development activity with more precision than in the past. Since the process relies a great deal on logical thinking as opposed to programming skill, less experienced programmers are able to take a bigger share of the development burden. Therefore software engineers can make the most of their software engineering skills without having to develop in-depth programming language expertise.

Results

The most important result noted by this effort, even in its preliminary form, is that the motivation to continue to use Cleanroom practices and PGPM at Picatinny has been established. This demonstration effort was sponsored by STARS and the continued effort is being sponsored by the AMCCOM LCSEC. This result is an instance of the STARS program fulfilling its mission by being the catalyst for introducing improvements to the software engineering capabilities in the DoD. In one sense, the effort is to be expanded across the entire organization, is the most definitive conclusion of this effort.

In addition to the above mentioned conclusion to this effort, the following conclusions can be drawn based on the current status of the MBC project.

1. *It is possible to transfer CSE and PGPM practices to project teams operating within a typical immature DoD SSA organization.*

This was shown by the fact that the MBC project has progressed to a point where CSE and PGPM are being successfully applied. This result shows that a specific maturity rating is not necessary in order to benefit from Cleanroom Software Engineering or Process-Guided Project Management. The engineering staff also enjoyed using the ideas, and all were interested in using the ideas again. Additionally, nearly all were interested in supporting and participating in the establishment of a "Cleanroom Competency Center" at the Picatinny Arsenal.

2. *Typical immature DoD SSA organizations can realize important benefits, in terms of improved process productivity, product quality, and staff morale, from the application of CSE and PGPM.*

This conclusion is supported by the apparent tripling of productivity of the MBC team. Table III shows the results for productivity for increments 1 and 2. The productivity increases because the training and learning curve are not required for successive increments. Future increments should show additional improvement.

Table III: Productivity Change for MBC Re-engineering

Increment	LOC per Staff Month	Change in Productivity based on Picatinny Baseline Metrics
1	370	3.06:1
1 + 2	428	3.54:1

Two important observations from the MBC project are that (1) PGPM has aided the learning process and helped ease the transfer and application of the CSE technology and (2) following a well defined process significantly improves team productivity and morale.

Early indication are that quality has appeared to have improved over previous product quality according to Picatinny's customers. The quality of the software developed during the first increment is very high when compared to quality for traditional software development. The MBC team is excited about the prospect of the upcoming test of their second increment. Thus, the result achieved will be viewed by the MBC team as the mark to better on the next increment of this project. The incentive and motivation for continual improvement is firmly in place among MBC team members.

Table IV: Quality: Number of First Increment Failures

Failure Type	Number of Failures	Description
Process	19	Approved improvements made to design but not reflected in updates to the specifications.
Spelling	3	Misspellings on displays.
Behavior	2	Coding Errors, did not work as specified.

$$\text{First Increment Failure Rate} = \frac{\text{2 Behavior Failures}}{\text{8500 LOC}} = 0.24\ \text{Failures/KLOC}$$

3. *The return on investment at Picatinny cannot be definitively calculated, but indications are that there is a significant return on investment.*

Since the project is not yet complete, a preliminary estimate of return on investment can only be based on estimates from the information currently available. The resulting return on investment (ROI) calculations appear on the next page in Table V.

Table V: Return on Investment for MBC Re-engineering

Increment	LOC per Staff Month	Staff Months MBC would have taken without CSE	ROI Base	ROI with Coaching	ROI with Training
1	370	8500 LOC/121 LOC/SM = 70.3	5.9:1	8.9:1	14.1:1
1 + 2	428	18200 LOC/121 LOC/SM = 150.4	9.8:1	20.4:1	17.1:1

SM = Staff Months

$$\text{ROI Base} = \frac{(\text{SM MBC would have taken - SM MBC took})}{\text{Staff Months of Coaching Effort + Staff Months spent in Training}}$$

$$\text{ROI with Coaching} = \frac{(\text{SM MBC would have taken - SM MBC took - SM of Coaching Effort})}{\text{Staff Months spent in Training}}$$

$$\text{ROI with Training} = \frac{(\text{SM MBC would have taken - SM MBC took - SM spent in Training})}{\text{Staff Months of Coaching Effort}}$$

If these assertions are correct, one must also realize that productivity will increase with the later increments because specifications are complete for the entire system. Once again, the final calculation of return on investment awaits project completion.

4. *An Automated Process Support System (PSS), that is consistent with the process defined for the project, facilitates technology transfer*

Automating the non-creative tasks of a new technology, such as file access and simple process flow facilitates the adoption of the new technology. This was true even for a system with limitations known and subsequently observed in CEPA. CEPA's successor system (being developed for deployment on the STARS Air Force demonstration project) applies many of the lessons learned from observing CEPA use at Picatinny.

5. *Based on this demonstration we now believe that a technology transfer program to support individual projects at a typical DoD SSA organization must begin with a defined process for the project and should consist of the following five components:*

(1) formal CSE training,

(2) training in PGPM,

(3) the availability of engineering handbooks,

(4) the use of a PSS (e.g., CEPA and its successor), and

(5) the availability of qualified coaching.

The combination of technology transfer components created a series of successes at Picatinny; including productivity gains, expected quality gains, and the increased motivation of the engineering staff.

The MBC project has realized significant gains from the CSE ideas. Once the learning curve had been completed;, initial successes in creating the Black Box specification served to cement commitment to CSE.

The resulting conclusions from the overall evaluation are preliminary because the demonstration project is still in its early stages. However, the original hypothesis that Cleanroom improves the effectiveness of the software re-engineering activities at Picatinny looks very promising. Indeed, management and staff agree that morale and motivation is extremely high, that teamwork is now the normal mode of operation, and that people are excited about the software process being established and are motivated to produce high quality products.

A good technical road map is in place at Picatinny; the technical personnel are developing the skills that appear to show significant gains in productivity. Even more promising is the fact that these gains were made with minimal exposure to CSE. Future gains are likely to be of greater magnitude as projects are carried out by experienced teams of Cleanroom engineers well advanced on the learning curve.

Future Directions

The next steps for LCSEC/STARS cooperation have been defined. The impressive results to date in the areas of productivity, quality, return on investment and moral have convinced AMCCOM LSCEC management to continue the work begun under this demonstration project and to expand it further throughout the organization and this will include the following:

Complete remaining phases of MBC project of which there are three increments in the initial plan. This will result in a completely re-engineered system.

Gain insight from a second experimental project, the Bradley Institutional Conduct of Fire Trainer, a software block update of a FORTRAN system. This project is a maintenance software block update for the aiming system on the Bradley fighting vehicle. CSE and PGPM techniques are being adapted for use in this important area. Much needs to be learned in adopting the successful techniques used on the MBC to the area of software maintenance.

Creating a Cleanroom Process Team to build CSE and process expertise in house. This group is responsible for continuous review and study of the application of Cleanroom and PGPM at Picatinny. This group's charter is to internalize and refine CSE and PGPM to the software practices at Picatinny.

Planning to use CSE for all future development and re-engineering projects internal to the LCSEC. It is cost effective to use these techniques for any maintenance project that requires effort equal to changing one third or more of the original code. Since productivity increases are three times conventional methods, a whole system can be re-engineered for the same cost or less of a maintenance update. The technology transfer package developed by IBM and SET can be used by the trained staff to support these projects. The MBC team is fortunate to have talented team members who can carry this out.

Evolve the Process Support System. The understood and observed shortcomings in CEPA are primarily addressed by the PSS being developed by the IBM STARS team for use on the STARS Air Force Demonstration Project at Peterson Air Force Base. The major source that provided input to the specification process for the PSS was the experience gained by using CEPA at Picatinny. As a result, many of the improvements desired by Picatinny are being developed as a part of the PSS. The PSS needs to be delivered to Picatinny, as well as to Peterson Air Force Base. As a result, Picatinny will receive their desired functionality and will help provide a second test bed for the PSS.

Evolve Cleanroom Software Engineering into a complete life cycle process as far as is feasible using standard Cleanroom techniques. Work is currently underway to perform a mapping of the evolved Cleanroom Software Engineering process, as used at Picatinny, against SEI developed Software Process Frameworks (SPF) for the SEI CMM Level 2 and 3. The results of this mapping will provide Picatinny with a road map of all areas that are either not covered or are poorly covered by the CSE process. These results will be used to identify areas for Cleanroom process definition as far as is practical. Areas that are not practical for Cleanroom extension will be defined with non Cleanroom extensions. The results will be a complete life cycle process definition for Picatinny to support their move to SEI CMM level 3. Additional work is being done in the area of Metrics to determine a baseline of measures to support a level 3 organization.

Design Capture Views Applied to the WAA System

Daniel J. Organ

Naval Undersea Warfare Center
Newport Division
New London, CT 06320

Abstract

This paper illustrates two system definition methodologies developed by the Engineering of Complex Systems (ECS) project. The methodologies are the System Design Views and the System Design Factors. These methodologies are used to provide a detailed characterization of the hardware components of the AN/BQG-5 Wide Aperture Array (WAA) system. The system hardware components are characterized using the Informational, Functional and Implementational System Design Views. Properties including performance, physical attributes and future needs of the system are presented using the hierarchical System Design Factors approach. The benefits and utility of the information in a system reengineering environment are discussed.

1: Introduction

One area of evolving system engineering methodologies for complex systems focuses on providing a comprehensive system definition early in the design cycle. Commercially available system engineering tools such as RDD-100 allow system designers the facility to generate various design capture views. The views provide the mechanism to completely define system requirements, notional architectures and system behavior. The goal is to mitigate risk by validating candidate architectures and identifying potential problems prior to building a prototype of the system.

When beginning to specify the requirements for a complex military system replacement or upgrade, it is important to characterize the existing system. The designer uses the information obtained from the characterization of the baseline system in two valuable ways:

1. To identify areas which are candidates for improvement in the new system design.
2. To provide a basis of comparison between the existing system and the new system.

For example, cost versus performance may be the primary tradeoff in design of the new system. The baseline characterization of the existing system may reveal that the ratio of cost to throughput is high. In this case the designer must ensure that the cost to throughput ratio for the new system does not exceed that of the existing system.

The preceding example illustrates how specific information obtained from a characterization of a baseline system can be used to define and evaluate a new system design. The example focused on a specific aspect (cost vs. performance) of the system definition. It is clear that the consideration of only this aspect provides a narrow and incomplete definition of the system.

As part of the Engineering of Complex Systems (ECS) project, scientists at the Naval Surface Warfare Center (NSWC) are actively pursuing research in the areas of system definition using several design capture views. The following design capture views and objectives of each view have been defined by N. T. Hoang[1]:

1. Informational
   - Characterizes system concept of operations
   - Represents system in abstract terms
2. Functional
   - Defines system functions and decompositions
   - Defines data flow requirements
3. Behavioral
   - Defines system critical paths
   - Specifies system real-time characteristics
4. Implementational
   - Define physical hardware, software, and human resources

- Specifies system physical interconnectivity

5. Environmental
   - Establishes conditions and events constraining system operations
   - Specifies performance MOEs and conditions of measurement

Within these design capture views NSWC researchers have defined a set of System Design Factors[2](SDF). The SDF represent a proposed standard methodology to describe the properties, attributes and characteristics of the system. The SDF can also be used to quantify specific aspects of the design and to perform trade-off analysis.

This report presents a baseline characterization of the AN/BQG-5 Wide Aperture Array (WAA) system hardware. The goal of this effort in an overall reengineering environment is to provide a detailed baseline to be used for comparison and trade off analysis between relevant aspects of the existing system and the proposed system. In the following case study the characterization of an existing system will be represented using the Informational, Functional, and Implementational design capture views. In addition, System Design Factors in the areas of performance, physical attributes, and future needs will be used to describe the baseline system hardware.

2: System Design Views

The following three System Design Views are the Informational, Functional and Implementational views. The views focus on the hardware components of the system. Each view contributes a different aspect of the overall system description. Collectively the design capture views provide a detailed characterization of the AN/BQG-5 WAA system hardware. However, the system hardware definition provides an illustration of only one aspect of the research being conducted by the ECS project.

2.1: Informational

This design view provides a textual description of the overall system functions and simple concept of operations. In addition some hardware specific information is included. The information in this view is intended to familiarize the designer with the functionality and use of the baseline system.

The AN/BQG-5 WAA is a stand alone acoustic sensing and tracking system designed to augment the combat system in 688 class attack submarines. The WAA system provides the capability to acquire, track and rapidly analyze an acoustic contact and conduct Target Motion Analysis (TMA) without maneuvering the ship. The inboard system hardware consists of 3 Common Electronic Equipment Enclosures (CEEE). The units are the WAA Receiver (1306), WAA Beamformer (1307) and WAA Processor (1308). A brief description of each unit and the unit resident firmware follows.

The WAA Receiver cabinet receives multiplexed Manchester coded serial-bit format data transmitted via twisted shielded pairs of wire from the canisters in the WAA Passive Receive Outboard Electronics (OBE). The WAA Receiver receives, demultiplexes, and reformats data signals for output to the WAA Beamformer. Required DC power and control signals for the WAA OBE are carried down twisted shielded pairs of wire from the WAA Receiver.

Firmware resident in the WAA Receiver provides control and performance monitoring of data that is input from the WAA OBE and formats the data for output to the WAA Beamformer. The firmware receives OBE controls and parameters from other system software and formats these for transfer to the OBE. The firmware includes control of a Pseudo Random Noise Generation (PRNG)/Cal generation function which can generate a single bit of Pseudo Random Noise (PRN) or calibration signal to be passed to the OBE for generation of a test signal. The firmware performs a gain calculation to be used for OBE gain control. The firmware also performs Digital ACINT Data Acquisition System (DADAS)/ACINT Calibration injection upon command.

The WAA Beamformer cabinet receives WAA hydrophone data from the WAA Receiver; provides required acquisition and track beamforming; implements a post beamformer correlation function; provides selectable beam data output for recording, audio, and calibration; performs integration and requantization of correlogram for display; provides post beamforming AOBT signal injection, and provides resources for tracking, localization, performance monitoring and unit control.

The resident firmware receives DIMUS hydrophone data and linear hydrophone data and the necessary controls to manage the track and acquisition beamforming and signal processing.

The acquisition processing provides DIMUS beamforming; post beamformer AOBT signal injection; post beamformer filtering and correlation processing; selectable digital beam output for recording; stabilization, integration, normalization, and requantization of correlation data for display; PM/FL and processing control.

	BCM ASICS	68000	56001	TMS320C30	TOTAL
	@25 MIPS	@.75 MIPS	@20 MIPS	@25 MFLOPS	MIPS/MFLOPS
BCM	900				900/0
GCM		9			9/0
CFM			300		300/0
SPM			3120		3120/0
FPM				1400	0/1400
GCA			300		300/0
EFM			100		100/0
TOTAL	900	9	3820	1400	4729/1400

Table I. MIPS/MFLOPS Calculations for Unit 1307

The track processing consists of linear beamforming; post beamformer filtering and correlation processing; DEMON classification processing; selectable digital beam and hydrophone data output for recording and audio; stabilization, integration, normalization, and requantization of correlation and DEMON data for display; noise estimation; PM/FL; and processing control.

The WAA processor supports functional processing for acoustic data, On-Board Training, Input/Output (I/O) processing, Database Management processing, disk storage and Workstation processing. The unit functionality is not provided by firmware but by application software executing in processor nodes connected by a LAN called FLEXNET. FLEXNET supports data transfer and communication between nodes within the unit.

2.2: Functional

The functional design view of the WAA subsystem is represented by characterizing system throughput and resource utilization. Information describing the functional partition is also presented. The information presented in this view provides insight into the computational allocation of the baseline system. When beginning to develop a strategy for allocation of processing resources for a system upgrade it is helpful to first examine the baseline system allocation as a starting point. By characterizing the throughput of existing system the designer gains some insight into the overall processing requirements of the system as well as specific functional partitions.

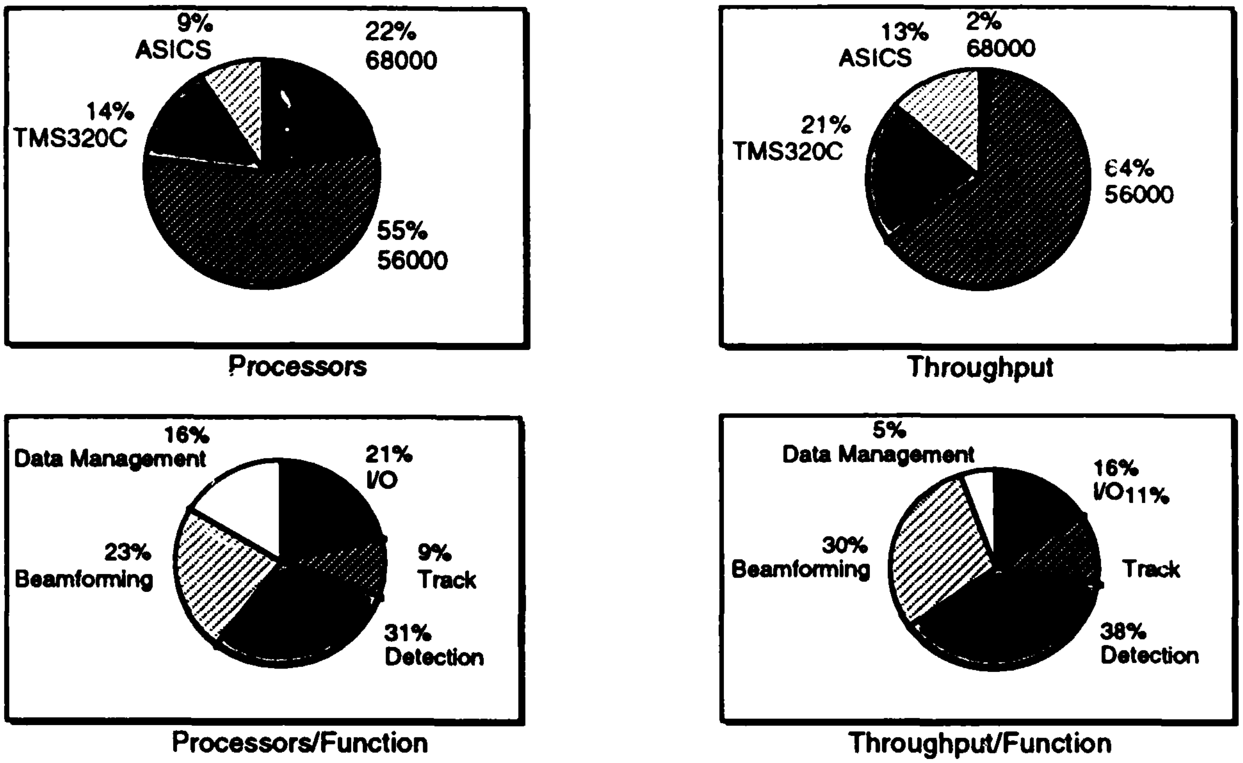
The system throughput is represented in MIPs and MFLOPs. The example unit calculation listed in Table I for unit 1307 are based on analysis of the number and types of processors resident in the unit[3]. The modules which contain processors are listed on the left side of the table. The various processors resident on the modules are listed on the top of the table along with the specific MIP/MFLOP rating for each module type. Modules based on two microprocessors, two Digital Signal Processing (DSP) chips and one Application Specific Integrated Circuit (ASIC) provide the system processing. The DSPs are the Motorola 56001 and Texas Instruments TMS320C30. The microprocessors are the Motorola 68000 and 68030. The ASIC is a special purpose chip called the Beamformer Computational ASIC.

For the throughput characterizations the processor ratings are estimated to be 2 MIPs for the 68030, 0.75 MIP for the 68000, 20 MIPs for the 56001, 25 MFLOPS for the TMS32C30 and 25 MIPs for the BCM. The ratings for the ASIC and DSPs are estimates and are based on published benchmarks, clock speed and have been derated to estimate actual sustained throughput rather than peak throughput. The MIP rating for the 68030 is based on specified CPU module performance.

In unit 1306, a combination of 68000 and 56001 provide a total of 486 MIPs. The 56001 support the demultiplexing of data received from the OBE. The 68000 support unit control functions.

Table I shows that a combination of all processor types contribute to the total 1307 unit throughput capacity of 4729 MIPs and 1400 MFlops. The primary functions performed in this unit are beamforming, detection and tracking. The majority of the processing is performed by the DSPs and the ASIC. Again, the 68000 supports unit control functions and does not contribute significantly to the overall unit computational capacity.

Although a table is not presented for unit 1308, the processing is provided entirely by 68030 processors for a total unit throughput capacity of 95.2 MIPs[3]. The 68030 resident on the CPU module and the NSU module which performs FLEXNET processing have measured throughput of 2 MIPs. Other 68030 are resident on I/O modules (FST, NTDSE, SCSI, SIOC, CSIO) and have



Processors

Throughput

Processors/Function

Throughput/Function

Figure 1. Functional and Processor Partitioning

measured throughput of 1.6 MIPs.

The pie charts in Figure 1 provide an overall system summary which illustrates how the throughput is partitioned between system functions and processors. The figure shows that the DSPs form the majority both in number of processors and in throughput. In particular the 56001 provides 55% of the number of processors in the system and 64% of the total system throughput. Functionally, the majority of processors and throughput are allocated to Beamforming and Detection. As seen in Figure 1, 23% of the total number of processors and 30% of the system throughput are allocated to Beamforming. For Detection 31% of the processors and 38% of the throughput are allocated to this function.

While the throughput data presented above illustrates the available computational capacity of the system and some indication of the partitioning, it does not show how much of the capacity is being utilized by the application software. Table II contains resource utilization for the A3 drawer in unit 1307[4]. Both processor and memory utilization as well as available memory metrics are listed for each individual processor or processor group. Because the available memory and memory utilization in unit 1307 are generally relatively low, the data indicates that the resident functions are more processing intensive than memory intensive. With a few exceptions the majority of the processors are below the system TADSTAND requirement of less than 80% processor and memory utilization.

2.3: Implementational

The implementational design view of the WAA subsystem is represented by describing each hardware unit. The description will include the drawer types, module populations, and finally overall unit metrics for power, weight, volume and available space[3]. The importance of this view is that it provides a detailed characterization of the physical aspects of the baseline hardware.

Three units comprise the WAA subsystem. All the units are nine drawer CEEE populated by SEM-D format modules. The units are water cooled and are powered by 155 VDC. Each unit contains one Power Control drawer which provides EMI filtering, unit and drawer controls and status monitoring and distribution of 155 VDC power. Unit 1306 and 1307 each contain an Interface and Control drawer which acts as a unit controller for synchronization and timing control.

As described earlier, unit 1306 contains nine drawers. In addition to the power drawer(A2) and the interface and control drawer(A5) , Unit 1306 contains six demultiplexor drawers(A1, A3, A4, A6, A7, A9) and one unpopulated drawer(A8). Each drawer, except for A2,

DRAWER	# PROC	MODULE TYPE	PROC UTIL(%)	MEM(RAM) UTIL(%)	MEM(RAM) AVAIL(KB)
A3					
CHANNEL BEAMFORMER	18	BCM	68.0	37.0	32
TIME DELAY CALC.	3	GCA	5.0	13.0	96
CHANNEL CONTROLLER	3	GCA	45.0	21.0	96
DRAWER CONTROLLER	1	GCM	45.0	20.0	128
FILTERING	6	SPM	79.0	7.0	192
SPECIAL OUTPUT COLLECT	3	SPM	65.0	10.0	192
SPECIAL OUTPUT	1	EFM	89.0	10.0	192
FULLBEAM CORRELATOR	6	SPM	71.0	78.0	192
HALFBEAM CORRELATOR	6	SPM	78.0	78.0	192
DEMON	[illegible]	SPM	37.0	94.0	192
ACOUS. FORMAT/OUTPUT	3	CFM	15.0	10.0	384

Table II. Unit 1307 Resource Utilization

contains 66 module slots. The module populations of the demultiplexor drawers are identical. GCM modules which contain 68000 processors, perform control functions. The SPM modules which contain 2 56001 DSP, perform the majority of the demultiplexing. Unit 1306 contains 192 WPR modules which provide DC power to the OBE.

	A1	A2	A3	A4	A5	A6	A7	A8	A9	TOTAL
BCM			18			18				36
GCM	1		1	1	7	1	1			12
CFM	3		3	3		3	3			15
CTM	1		1	1	3	1	1			8
DRM	3		6	3		6	3			21
DDM					3					3
SPM	18		12	18		12	18			78
PSM	1		1	1	2	1	1			7
CDM	3		3	3	7	3	3			22
FPM					28					28
ROM					6					6
WFM	3			3			3			9
WCM	9			9			9			27
GCA	3		3	3		3	3			15
DRD	1		1	1		1	1			5
CIM					5					5
EFM	1		1	1		1	1			5
DAC			2			2				4
KLM			1			1				2
TOTAL	47	0	53	47	61	53	47	0	0	308

Table III. Unit 1307 Module Population

UNIT	POWER (KW)	VOLUME (FT^3)	PROC. THROUGHPUT (MIPS/MFLOPS)	SPARE SLOTS (%)	PROC. DENSITY (MIPS/FT^3)
1306	5.9	24	486/0	38	20
1307	3.6	24	4729/1400	41	197
1308	2	24	95/0	77	4

Table IV. Summary of Unit Metrics

The module population of unit 1307 is listed in Table III[3]. In addition to the power (A2) and interface and control (A5) drawers, this unit contains three identical Acquisition/Beamformer drawers (A1, A4, A7) and two identical Track/Beamformer drawers (A3, A6). Drawers A8 and A9 are empty. The primary computational modules contained in the unit are the BCM (ASIC), GCM (68000), CFM (56001), SPM (2/56001), FPM (2/TMS320C30), GCA (68000/56001) and EFM (56001). The WCM module performs 1 bit DIMUS processing.

Unit 1308 contains two Data Management drawers (A3, A4), a disk drawer (A6) and an EMSP I/O drawer (A1) as well as the power control drawer (A2). Drawers A5, A7, A8, A9 are empty. The modules in the functional drawers (A1, A3, A4) are configured in processing nodes to perform specific functions. Each node contains 1 CPU, 1 or 2 GM16, 1 NSU and various interface cards (FST, NTDSE, SCSI, CSIO).

Table IV summarizes several unit metrics for the WAA subsystem. The percentages of spare slots for each unit was computed by dividing the number of occupied module slots by the number of available module slots. All the units contain available space for additional modules. The processing density for each unit was computed by dividing the unit MIPS by the volume and indicates that the majority of the system processing is performed in unit 1307.

3: System Design Factors

One of the primary objectives of System Design Factors is " to provide a mechanism to quantify and identify a large, complex real-time system's strengths and weaknesses so that effective comparison of different systems is achievable"[2]. By defining a set of SDF, specific system characteristics are more easily compared. The information is presented as a hierarchy in which each

PROPERTIES	ATTRIBUTES	METRICS	QUANTITATIVE VALUES		
			1306	1307	1308
I. Performance	Throughput	MFLOPS	0	1400	0
		MIPS	486	4729	95
	Utilization	percent (%)	43	50	34
II. Physical	Power	kilowatts (kW)	5.9	3.6	2
	Weight	pounds (lbs)	1120	1240	1060
	Volume	cubic feet (ft^3)	24	24	24
III. Future Needs	Expandability	spare slots (%)	38	41	71

Table V. System Design Factors

level provides increasingly specific descriptions of the properties listed in the left hand column of the table. Table V contains a list of SDF defined for the WAA subsystem. The properties listed in Table 9 are only a small subset of the SDF identified by Nguyen and Howell[2].

4: Summary

This paper provides a characterization of the WAA subsystem. The information is presented using two system definition approaches developed by researchers at NSWC. The approaches are the System Design Views and the System Design Factors methodologies.

The System Design Views approach uses 5 unique design views to describe the system. They are the behavioral, environmental, functional, informational and implementational view. Each view provides specific information which can be used separately to define specific system characteristics. When taken together, the set of views provide a comprehensive description of the system.

The System Design Factors approach involves creating a hierarchy of system properties. Each successive layer of the hierarchy provides additional detail to define the property. By presenting the information in a hierarchical format, it allows the designer to compare and contrast the relevant properties of various systems.

System definition methodologies such as System Design Views and System Design Factors are being developed to improve complex system engineering. The goal of these methodologies is to improve the system definition early in the design cycle. The hope is that these methodologies will aid the system designer in the conceptualization and development of complex systems.

When considering system upgrades or improvements it is important to evaluate characteristics of the new system against the existing system. This report has attempted to characterize the WAA subsystem by applying currently evolving system definition methodologies to present the pertinent information. The information presented will be used to perform trade off analysis and evaluate relevant aspects of the proposed system upgrade.

References

1. Hoang, N. , "The Essential Views of the System", 1991 System Design Synthesis Technology Workshop, Silver Spring, MD, September 1991.

2. Nguyen, C.M., Howell S. L., "System Design Factors", 1992 Complex Systems Engineering Synthesis and Assessment Technology Workshop, Silver Spring, MD, July 1992.

3. General Electric Company, "AN/BQG-5 Hardware Notebooks", 9 March 1992.

4. General Electric Company, "AN/BQG-5 System Metric Reports", March 1993.

Translating CMS-2 to Ada

Charles H. Sampson
Computer Sciences Corporation

Abstract

The goal of this paper is a description of TRADA, a CMS-2 to Ada translator that is being developed by Computer Sciences Corporation under the auspices of the Naval Command, Control and Ocean Surveillance Center (NCCOSC) Research, Development, Test and Evaluation Division (NRaD) in San Diego.

This paper begins with a discussion of the translation philosophy behind TRADA and an overview of the translator. Then follow three sections that sketch CMS-2. Those familiar with CMS-2 might be tempted to skip these sections, but they do contain some subtle points that a user might be unaware of. A discussion of CMS-2 features that cause translation problems and several of the translation strategies used in TRADA are next. The paper concludes with a brief discussion of the one known case of inadequate translation and a summary.

1. Philosophy of Translation

What should be the goal of a CMS-2 to Ada translator? There is no single answer to that question. From the user's viewpoint a translator (from any language to any other) should ideally translate all code with 100% accuracy; if the ideal cannot be achieved then at least it should translate the hard parts that the user doesn't want to do himself.

In practice, it is often impossible to achieve either of these goals, particularly when the language being translated is as undisciplined as CMS-2. It is often precisely the parts that the user finds hard that are very difficult or impossible for a translator. Nonetheless, a translator can have a significant impact in transitioning a project from CMS-2 to Ada. It can take care of many of the mind-numbing details, leaving the humans involved free to devote their energies to important parts of the task.

The design and implementation of TRADA is driven by three goals: to produce translated code that is transportable and at least as easy to maintain as the original CMS-2; to translate with 100% accuracy when translation is possible; and to clearly indicate the problem when translation is not possible. Even these goals are not always attainable, but the cases when they are not attained are expected to be very rare or, in the case of inaccurate translation, detectable by some other means.

The desire for transportable, maintainable, Ada code comes from the belief that there is no particular virtue in exercising an Ada compiler. That is, the primary, perhaps unstated, purpose of the translation is to reap the benefits of the Ada language and two of those benefits are transportability and maintainability. While there might be some argument on the issue of transportability, particularly for a specific system targeted to a particular Navy computer, it is hard to justify ever producing code that is harder to maintain than the CMS-2 original, as has been seen from time to time.

Fully accurate translation is required because of the carefully tuned algorithms in much deployed CMS-2 code, particularly fixed-point calculations. If the translation output inefficiently but accurately captures the algorithm, the maintenance programmer can look at the Ada code and decide how to improve the efficiency. This is a much easier process than attempting to determine why a previously correct algorithm is no longer producing correct results.

Clear indication of translation problems is a counterpart of the desire for fully accurate translation. When an accurate translation is not possible, it is necessary to call out to the maintenance programmer that intervention is required.

2. TRADA Overview

TRADA is viewed as a one-shot tool. That is, any CMS-2 program is expected to be translated only once. After that, any modifications will be made to the output Ada code. Because of this one-shot aspect, the emphasis in TRADA has been on the translation process, with minimal regard for execution speed. If a program is only going to be used once per project and only a few times in a single shop, very long execution times can be accepted.

It has been suggested that some might want to translate using TRADA, investigate the translation output, modify the CMS-2 source, translate again, and repeat this process through several cycles. This approach seems to be based on the belief that a slight tweak of the CMS-2 would result in successfully translating many previously untranslatable statements. While this might be true, in these times of limited resources it was decided to carefully document the kind of changes that should be made to the code before translation and reduce costs by implementing with minimum consideration of throughput.

The implementation of TRADA is in progress as this paper is being written. Rather than freezing the description of TRADA at this time, it is described as though it is complete, with all specified features implemented.

TRADA is being developed on a VAX/VMS, using DECAda. It has been designed to simplify transporting to other hosts. In particular, its design does not take advantage of the virtual memory of the VAX; all major data structures have a spill-to-disk capability. (However, TRADA is not a sequence of independently loaded phases. Therefore a reasonable amount of memory will be required to hold the executing code.)

In the following subsections a number of terms are used that might be unfamiliar to those who do not know CMS-2. They are explained in section 3.

2.1 Inputs

TRADA is an unusual translator in that it only translates entire programs. Certain of its translation decisions require knowledge of all uses of the item being translated. Therefore, translation of less than an entire program would conflict with the goal of fully accurate translation.

As a result, one required input to TRADA is the entire source of the program to be translated. TRADA does support the include capability of the MTASS CMS-2 compilation system, so it is not necessary to gather all of the source into a single file. On the other hand, TRADA does not support compools, so the source of the system data blocks that make up compools must be presented as part of the source to be translated. (Compools cannot be used in TRADA because of the need to see all uses of items before their translation can be determined. Even if this were not an issue, compools are not cost-effective because they would be used only once. Thus the cost of implementing compools in TRADA would not be paid back as it is for a CMS-2 compiler by the reduction in resource usage gained when they are input to multiple compilations.)

The presented source program must also be correct CMS-2; if any syntax or semantic error is found, no translation is produced. The potential for chaos that could result from translating an incorrect CMS-2 program is too great. The requirement of a complete, correct, source program means that the user must fix the problems discussed in sections 5.2 and 5.3. Without these fixes, proper translation is impossible.

TRADA must also be told the name of the main procedure. This information is used solely to generate the Ada main procedure, which does nothing but call the translation of the CMS-2 main procedure.

TRADA optionally accepts the input of a *script file*, which directs certain aspects of the translation. Examples of the kind of thing controllable through the script file are whether the use clause or fully qualified names should be used, whether fixed-point conversions should be by truncation or left to the Ada fixed-point accuracy specifications, and whether between-statement comments are attached to the preceding construct or the following one.

2.2 Outputs

TRADA creates one package corresponding to each CMS-2 module of the program being translated. For system data blocks, the package usually consists of a specification only, although there are cases in which the package will have a body. For system procedure blocks the package specification contains the declarations of the module's entry subprograms and any other exported declarations. Each of these compilation units is created in a separate file.

The output also contains a package (specification only) corresponding to the major header and a "main" procedure that calls the translation of the CMS-2 program's startup procedure.

In addition, a number of packages are created that do not correspond to any single construct of the program being translated. One of these contains the Ada translations of all of the CMS-2 types used, along with a number of functions needed to support those types (mostly conversion functions that truncate). This package is tailored to the specific translation; there are too many valid CMS-2 fixed-point types (32,130) to use a canned package that contains translations of all of them. To minimize the number of explicit conversions needed in translated expressions, this package makes heavy use of subtypes. (As an aside, the ranges and deltas used to define these types are expressed in octal, which corresponds clearly to the way the types are expressed in CMS-2.)

The specification of another generated package contains the translations of everything whose definition was incomplete. If any of those items are subprograms, the body of this package is generated to contain their stubbed definitions. The existence of this package is a warning that the requirement of a complete program has not been met. It does, however, permit the translated output to be compiled and linked. The careful user of TRADA might be able to flesh out this package and get working Ada code, rather than fixing the CMS-2 and retranslating.

Other generated packages, generated as needed, contain Ada versions of subprograms predefined in CMS-2 and code to simulate the hardware switches of the Navy computers. The latter will ultimately be replaced, of course, but it is useful in the initial post-translation stages when the translated code is being verified, augmented, and cleaned up.

2.3 Assumptions

TRADA assumes very little about the Ada compiler to be used on the translated code. It assumes that the compiler supports a 32-bit integer, as suggested by the Uniformity Rapporteur Group, but does not assume that it is Standard.Integer. It implicitly assumes that the compiler has enough capacity to compile the translated code. It assumes that binary fixed-point deltas are supported for some reasonable set of values. (I.e., it will properly compile statements of the form

```
    FOR Xyz'Small USE 2.0 ** N;
```

for a reasonable range of values of N.)

3. Description of CMS-2

CMS-2 is a closely related family of languages. By standards of the 1990's, these languages lie at the low end of the spectrum of high order languages. Indeed, when I first became familiar with CMS-2 almost 20 years ago, I characterized them as "disguised assembly languages". I have had no reason to change that characterization since. Of the well-known languages, the CMS-2 dialects are probably closest to C in strengths and weaknesses, although certainly not in syntax.

In this paper I use the term CMS-2 as though it were a single language rather than a family. When I write about a particular dialect, I make that clear.

CMS-2 uses a bit of non-standard terminology, both in its keywords and in the description of features. To minimize the jolt to the sensibilities of the uninitiated, I will describe CMS-2 using Ada terminology as much as possible. For the benefit of those who know CMS-2, and who might be confused by what they consider non-standard terminology, I will put the corresponding CMS-2 term in parentheses.

One Ada term that will be used in the following has not yet received wide acceptance. It is *regional data*, meaning data declared in a package body but outside any subunit of the package.

3.1 Gross Program Structure

There are two structures for modularization in CMS-2: the system data block and the system procedure block, commonly called SYS-DD and SYS-PROC respectively, after their defining keywords. (CMS-2 keywords often contain embedded hyphens. Unlike COBOL, when the hyphen is used as a minus sign it is not necessary to set it off from identifiers in any fashion. This is only one of the unusual problems imposed on compiler writers by the language.) The system data block is a module that contains declarations of global data. A system procedure block contains executing code, along with data whose scope is global, local to that module, or local to a subprogram within that module. Each of these modules can be preceded by a *minor header*, which essentially parameterizes the module. The purpose of a system procedure block appears to have been to afford a mechanism fo encapsulating a subprogram of the system along with any auxiliary subprograms needed to assist it in carrying out its function. In practice, it is often used as a simple collection of more or less related subprograms.

A CMS-2 program consists of a number of these modules. There is no CMS-2 construct for indicating the "main program". The means for transferring control from the operating system to the program depends on that operating system, which is often special-purpose.

A compilation (compile-time system) consists of the source code for one or more modules, preceded by a *major header*, which conceptually is used to parameterize the entire program. A compilation can also make use of *compools*, which are previously compiled system data blocks. As far as the semantics of a compilation are concerned, the use of a compool has the same effect as including the source of those system data blocks. What is gained by using compools is the cost of repeatedly parsing and semantic checking the data blocks. CMS-2 compilers also typically support some form of source file inclusion.

Two distinct forms for source code documentation are afforded. A *comment* is free-form non-executable text that can only appear between statements and declarations. A *note* can appear between any two tokens of the program.

3.2 Declarations

The basic data types of CMS-2 comprise integer, fixed-point, floating-point, boolean, fixed length string (character or Hollerith type), and enumeration (status types). The integer and fixed-point types are specified in terms of the number of bits required to hold their values (size); for fixed-point the fractional part (scaling) is also specified in number of bits. The floating-point types are specified in terms of the floating-point formats available on the target machine. Enumeration literals are delimited by apostrophes. The actual enumeration literal values are thus not restricted to be identifiers; any character from the CMS-2 character set can be used.

CMS-2 contains constructs for declaring typeless numeric constants (ntags) and typed variables, of either the basic data types (simple variables) or composite types—records (item-areas or structured variables) and arrays (tables). However, the components (fields) of a record type are restricted to be of the basic types. It is possible to specify an initial value for most objects (variables and tables). The declaration of an array object specifies how it is to be laid out in memory (similar to row major vs. column major). This is an execution efficiency issue and does not affect the program semantics, although use of one of the forms does impose some restrictions.

Only very late in the evolution of CMS-2 did it obtain an explicit type declaration. A number of unusual syntaxes and concepts were used to compensate for this lack. Specifically, the declaration of an array uses a block syntax. Within that block, the structure of the array's components (items) is declared. Within that block it is also possible to declare other arrays whose components have the same structure (like-tables), variables having the same structure (item-areas), and sub-arrays (subtables), whose components are a contiguous subset of components of the array being declared. Even with these mechanisms, restrictions in the language made it necessary to replicate component declarations in different array declarations. This occurs quite often in old CMS-2 code, in spite of the obvious maintenance problems it engenders.

A *switch* is a CMS-2 construct not found in many other languages. A label switch specifies a number of labels; control is transferred to one of those labels when a particular form of goto statement is executed. A procedure switch specifies a number of procedures; one of those procedures is executed when the switch is invoked, with a syntax very similar to that of a procedure invocation. In both cases the choice depends either on the value of an integer expression (indexed switch) or the value of a specified variable (item switch) at the moment of execution/invocation. Thus, executing a goto using an indexed label switch has an effect quite similar to FORTRAN's computed goto.

It is possible to elevate the scope of many identifiers. Identifiers declared in a system procedure block whose scope would ordinarily be local to that block can often be given global scope.

CMS-2 does not have a project library similar to that of Ada. In order to support building large systems out of a number of small compilations, most of the declarations can be modified (EXTREFed) to indicate that the datum is not to be allocated as a result of the declaration—only its attributes are being specified in order to give the compiler the information it needs to generate the proper code sequences. The code is tied to the proper memory location—allocated as a consequence of an unmodified declaration elsewhere—when the system is linked.

3.3 Subprograms

Subprograms consist of procedures and functions. Procedures can optionally have input and output parameters. Functions can have only input parameters and can return values of only the basic types. It is not possible to nest subprograms. Subprograms are not recursive, but they are compiled with code that permits reentrancy, if the requisite executive support is available.

In some of the dialects of CMS-2, procedures can also have exit parameters, similar to the abnormal exit of FORTRAN. At the point of invocation, the exit parameters are matched with labels. If the invocation is terminated by executing an "abnormal" return, control is transferred to the corresponding label. In this case, the assignment of values to actual output parameters is not performed. (See section 4.3.)

3.4 Expressions

The operands of numeric expressions can be of any numeric type. The rules of the language specify the implicit type conversions that will be performed in order to evaluate a numeric expression. In particular, an expression involving fixed-point values is evaluated according to an elaborate set of *fixed-point scaling rules*, which are highly machine dependent. Generally, these rules prevent loss of significant data, specifying when values are to be shifted right internally to avoid overflow. (There are a few exceptions that are intentionally in the language!) When type conversion results in the loss of information, it is by truncation rather than rounding.

The usual numeric operations are available for data up to 32 bits long (counting the sign bit). These operations can be written for quantities in the 33-64 bit range. Addition and subtraction of such quantities are performed efficiently and accurately. When such quantities are multiplied or divided, the operation is performed in floating-point. (The conversion of a 64-bit value to floating-point is inexact in all dialects of CMS-2.)

Boolean expressions are usually evaluated by short-circuit. This is always the case in conditional statements and is now the case for some compilers in boolean assignment statements. This evaluation is an artifact of the compiler implementations; it is not specified as part of the language definition.

3.5 Statements

Although their syntaxes can be quite non-standard, CMS-2 has the expected collection of statements: assignment, if-elsif-else, loops, case, goto, etc. Many of these have unusual features. There are also a few non-standard statements.

Loops (vary statements) have the usual three forms: FOR-loop (controlled by an index), WHILE-loop (controlled by a boolean expression evaluated at the top of the loop), and loop-UNTIL (controlled by a boolean expression evaluated at the bottom of the loop). A loop index can be of either a numeric or an enumeration type.

One unusual feature of loops is that numeric loop indexes are not required to be integer; a numeric loop index is initialized, incremented on each iteration, and tested for equaling or exceeding its terminal value, regardless of its type. Another is that any combination of the loop controls can be used in a single loop, including multiple indexes; the loop terminates as soon as any one of its controls satisfies its termination condition.

Related to loops is the resume statement. When it is executed inside a loop, it terminates execution of the current iteration. (It causes bottom-of-loop processing to occur.) It can also be executed outside of the loop it specifies. In this case, the loop resumes execution at its bottom-of-loop processing.

A special form of loop (find statement) is supplied for performing a linear search of any array. The search terminates when a component satisfying specified criteria is found. The search can be continued by the resume statement.

The selector of a case statement can be of any basic type. The semantics specify that a search is made for a matching case value, clearly dangerous if the selector's type is an approximate type, such as fixed-point or floating-point.

There is no restriction on the relative locations of a goto statement and its label. Free transfer of control into and out of arbitrary statement blocks is permitted. One dialect even permits transfer of control across subprogram boundaries.

As mentioned in 3.2, there are forms of the goto statement for executing a multi-way branch through a label switch and there are procedure-call-like statements for invoking procedure switches. These statements also allow specification of a label to be transferred to if the value on which the switching decision is to be made is invalid (invalid specification).

4. Unusual and Problematic Features

The above description of CMS-2 concentrated on concepts that it shares, more or less, with other high-level languages. It has however a number of features that are rare, if not unique. In preparing this paper I easily developed a list of 48 of them. In the following, I discuss only those that give particular problems in translating, omitting those that might cause headaches for compiler and translator writers but whose actual translation is simple.

4.1 Static Aliasing

CMS-2 affords an astonishing variety of ways to statically alias objects. (By this I mean causing two objects to share memory. I am not discussing dynamic aliasing, such as the aliasing that occurs in many languages when a global object is used as an actual parameter of a subprogram invocation, for example.)

First among these is the overlay statement, which specifies that the bit string allocated to an object is to also be used for the bit strings allocated to some other objects. Notice that this is much finer than simply specifying that two objects are to use the same memory location. As an example, for an 18-bit object it is possible to specify that bits 1-6 are to be used for another object and bits 9-17 are to be used for a third. This ovtrlaying can be specified for directly allocatable objects (those that hold the values of variables) or for the components of a record type.

This effect can be obtained a second way for the components of a record type, because CMS-2 allows *user-packing*, similar to Ada's record representation clause. Unlike the Ada construct, user-packing permits overlapping of components.

Finally, there is a means for specifying the relative allocations of variable objects. This relative allocation is expressed in words of the target machine; the programmer can specify that XYZ is to be allocated 23 words beyond the place where ABC is allocated. Notice that if ABC is an array, then XYZ might be allocated in the middle of one of ABC's components.

4.2 Bit-Twiddling

By *bit-twiddling* I mean working with bit substrings of values. The problem with bit-twiddling is that the language-defined semantics of the substrings are incomplete at best. While the language specifies which bits of the value are to be extracted or replaced, the meaning of those bits and the relation of them to the larger value is known only to the programmer. It is the ease with which one can bit-twiddle in CMS-2 that caused me to characterize it as a disguised assembly language.

Overlaying, either through the overlay statement or implicitly through user-packing, is a heavily used means for bit-twiddling. In addition, CMS-2 supplies a parameterized operator or pseudo-function (the BIT operator or function) for accessing an anonymous bit substring. There is another parameterized operator or pseudo-function (the CHAR operator or function) for accessing character substrings within a value; the value is not required to be of character type.

It is also possible to access individual target-computer words of an object (word reference). The particular word accessed might be only one of several making up the object or it might contain several components of the object. Again, the exact meaning of such an access is known only by the programmer.

Bit strings can be manipulated by the conventional set of bit operations: AND, OR, NOT, and XOR. Typed operands of these operators become temporarily typeless. The result regains a type dependent on the context of the operation.

4.3 Subprogram Parameter Passage

Subprogram formal parameters are very unusual in that rather than serving as surrogates for the actual parameters during execution of an invocation, they are variables external to the subprogram. At the point of invocation, the values of the actual input parameters are copied to the corresponding formal input parameter prior to transfer of control. Following the invocation, the values of any formal output parameters are copied to the corresponding actual output parameters.

The semantics of parameter assignment, coupled with the parameters being variables external to the subprogram, can give rise to *cross-parameter interference*: assigning a value to one parameter in the list can affect the value assigned to another. An example is a formal parameter that is an integer variable, with that variable being used as a subscript of a later actual parameter. Different dialects of CMS-2 assign the formal parameters in different orders.

There is no restriction on the use of formal parameters; they can be used at any time like ordinary variables. Furthermore, the same variable can be used as a formal parameter of more than one subprogram.

4.4 Indirect Arrays

There is no heap/pointer mechanism in CMS-2, but an undisciplined approximation is available through the indirect array (indirect table). When an array is declared to be indirect it functions as a surrogate for other, "real", arrays during execution. Thus it is very much like a pointer.

Specifying a designated object is accomplished by using the CORAD pseudo-function. When a value is assigned to this pseudo-function its argument can only be the name of an indirect array. That value then becomes the memory address of the designated object for subsequent references to the indirect array.

When the CORAD pseudo-function is used in other contexts, its argument must be a name that has an assigned memory address; it generates that address as its value. Thus the statement

```
SET CORAD(XYZ) TO CORAD(ABC)  $
```

means that for subsequent references, XYZ is to be used as a surrogate for ABC.

There is no requirement that the structure of the designated object be at all related to the structure specified in the indirect array's declaration. It is not even necessary that the designated object be an array. It could be a simple variable, the "tail" of an array (by specifying the address of one of that array's components), a switch, or even code! In all these cases of mismatch the indirect array's structure rules; the designated object is accessed as though its structure were that of the indirect array.

One use of indirect arrays is to effect call-by-reference semantics in subprogram invocation. Because the CMS-2 parameter passage mechanism is call-by-value (section 4.3), if a formal parameter is an array then the entire array must be copied as part of the invocation. It is possible to specify a formal parameter as the address of an indirect array (using the CORAD pseudo-function). By doing this, only the address of the actual parameter is copied.

Because this pointer mechanism is specified as implemented using addresses, an executive could implement a heap. Calls to its allocator would simply return the address of the newly allocated memory area.

4.5 Array Initial Value

In specifying an initial value of an array, it is not necessary that the array be fully initialized. The array's components must be of a record structure and the initial value is specified in terms of values to be assigned to the record's components in successive array components. If fewer values are specified than array components, only the "first" array components are initialized; the values assigned to the remainder are not defined by the language. (However, see section 5.1.) This initialization can be ragged; different numbers of values can be specified for different record components.

4.6 Multi-Receptacle Assignment

An assignment statement can contain multiple *receptacles*, the objects that receive the value. If the receptacles are of different numeric types, implicit conversion of the value being assigned is required. The semantics of a multi-receptacle assignment specify that the value is assigned to the first receptacle, converted if necessary. That possibly converted value is then assigned to the second receptacle, where another conversion might be needed. This value is then used for the third receptacle, and so on. While these semantics could produce some surprising results for the unwary, in practice the overwhelming use of this feature is to assign zero.

Different dialects of CMS-2 order the receptacle list differently, some from left to right and others from right to left.

4.7 Array Assignment

Array assignment is accomplished by transferring target machine words, without regard to the structures of the two arrays. If the receptacle array is shorter (in words) it is filled with the corresponding words from the beginning of the source array. If the receptacle array is longer only the words at its beginning that correspond to the words of source array receive those new values. Thus, if the arrays have identical structures the right thing happens, and if their components have identical structures but their sizes are different something understandable happens. However, when their components have different structures, an insidious form of bit-twiddling occurs.

4.8 String Manipulation

In general, string assignment and string comparisons are accomplished through blank-padding and truncation. String literals are handled specially. If the source of a string assignment is a literal, it is not padded when too short (the value is sliced into the beginning of the receptacle) and it is not permitted to be too long.

4.9 Typed Records

A record type in CMS-2 can be declared to have an associated basic data type. When a variable of such a type is referenced as a whole, it has the semantics of a variable of the associated type. As a benign example, a record type could be declared to have components corresponding to the characteristic and mantissa of a floating-point value, positioned appropriately, and the floating-point type as its associated type. For a variable of this type, the components could be used to create a floating-point value, which could thereafter be used by referencing the variable itself.

Ada has no corresponding capability.

4.10 Boolean Constants

The boolean constants *true* and *false* are not defined in CMS-2; their functions are served by the numeric literals 1 and 0, respectively. In practice, named number declarations are often used to define the identifiers True and False, as well as other convenient identifiers with boolean overtones.

Because of the dual use of 0 and 1, it is possible to have both boolean and numeric receptacles in a single multi-receptacle assignment statement.

4.11 Enumeration Assignment and Comparison

All enumeration types are considered compatible. Assignment and comparison of enumeration values is based solely on the compiler-generated encoding of the symbolic values (0 for the first value, 1 for the second, etc.)

4.12 Data Local to Subprograms

CMS-2 is not a stack-oriented language; data local to subprograms are statically allocated. Thus the values of these data at the end of execution of one invocation are present at the beginning of execution of the next invocation. Any initial values given to such data are only the values at the beginning of execution of the program.

4.13 Conditional Compilation

Conditional compilation is achieved through a blocking construct, where the block's header specifies a flag (CSWITCH) whose value controls the compilation of the code in the block. The blocks can appear among both statements and declarations.

If the flag setting is such that a block is not to be compiled, the code in that block is ignored, other than some minimal checking to detect the end of the block. *Minimal checking* has a number of implications, principal among them being that the code being checked does not have to be syntactically or semantically correct. It also means that an identifier can be declared in any number of blocks, provided that only one of those blocks is compiled.

4.14 Direct Code

Assembly language statements can be included in a CMS-2 program by enclosing them in a special block (direct code block). The programmer is given almost the full freedom (license) available to an assembly language programmer: executing instructions can be placed among high-level data declarations, assembly data declarations can be placed among high-level statements, and the instructions and declarations can be intermixed in any order.

5. Usage of CMS-2

In addition to the official semantics of CMS-2, sketched above, there are facts about the language that might be called "informal semantics". These facts are known and relied on by many users of the language, even though they have no official sanction.

5.1 Initialization to Zero

All of the CMS-2 linkers set all data values to zero unless they are explicitly initialized to some other value. Some CMS-2 programs make use of this to avoid expending memory, which is often limited, on code that zeros some uninitialized data.

5.2 Linking by Name

CMS-2 linkers link by name; no semantic information is passed from the compiler to the linker. Occasionally this is "exploited" in creating a program from independent compilations, when a name is given different attributes in two or more of those compilations.

5.3 Reuse of Named Numbers

The names of named numbers are not passed to the linkers by the compilers. Therefore it is possible—and it does occasionally occur—for "the same named number" to have different values in two independent compilations of a program.

6. Intractable Translation Problems

Certain of the features of CMS-2 give rise to translation problems that cannot be solved by any translator, at least not when the goals of the translation include maintainability and transportability, as they do for TRADA.

6.1 Bit-twiddling

Any attempt to mechanically translate the various bit-twiddling features of CMS-2 is doomed to fail for one of several reasons. The common thread to these reasons is that the exact meaning of the bit-twiddling cannot be discerned from the CMS-2 source, as noted previously. As an example, suppose that a bit string is being extracted from some datum (typically a machine word) to be used in a numeric context and that the numeric value being extracted might possibly be negative. If the extraction of the bit string is "correctly" translated, but the negative representation of the target computer changes (the AN/UYK-7 family of Navy computers uses 1's-complement, while 2's-complement is currently the most popular representation), the meaning of the program has been changed by the translation. As a second example, suppose that bit-twiddling is being used in a utility routine to create a floating-point value by building up its component parts. Even if the translated code could mimic these manipulations, the result is meaningless if the target machine for the translated code does not support the same floating point format.

"Doomed to fail" is too harsh a statement, of course. It might be possible to specify some relatively well-behaved bit-twiddles that could be translated. In that case the issue of cost-effectiveness arises: is it worthwhile to design an algorithm to detect and translate these cases, which are expected to be few, or would it be better to allocate resources to other, more promising areas, and leave all bit-twiddling for human intervention?

6.2 Overlays

Perhaps the greatest single benefit that could be obtained from a CMS-2 to Ada translator would be the analysis of the rat's nest of overlays that are so typical of CMS-2 code and translation into clean Ada. Unfortunately, that benefit is not to be had.

To begin with, overlays are potentially bit-twiddles, as has been noted, and suffer from all of the problems of bit-twiddling. For example, the creation of a floating-point value by constructing its components, mentioned above, can be accomplished using overlays.

There are other uses of overlays that are not, strictly speaking, bit-twiddling. One such is to simulate a multilevel record structure—records within records within records—which is not directly achievable in CMS-2. An algorithm to detect such a use of overlays is possible, but then a second translation problem arises. If overlays are used in CMS-2 to simulate a record within a record, the larger record (the *overlay parent*) has a simple CMS-2 type, usually numeric. Since there is nothing corresponding to this in Ada (a record type does not have an associated scalar type), it would be necessary to check all uses of the larger record to see if it is truly being used as a value of that type or if the type is simply a hook to hang the simulation on. Translation would be possible in the later case, but again the cost-effectiveness question must be asked.

Finally, overlays can be used to create structures somewhat like an Ada record with variant part. The key characteristic of this kind of structure is that no component is ever read without first being explicitly written. While this is perhaps the most benign use of overlays, detecting it is very difficult. At the least an algorithm built on top of a global data flow analysis is required. Even if such an algorithm is implemented, there is no guarantee that a particular structure of this kind can be translated into an Ada record with variant parts. Consider the following "variant record": it has a discriminant and three ordinary components; the discriminant can take on values 1, 2, and 3; component 1 is valid when the discriminant has value 1 or 2; component 2 is valid when the discriminant has value 1 or 3; and component 3 is valid when the discriminant has value 2 or 3. This structure cannot be translated into an Ada record with variant part. The symbol table of the CMS-2Y compiler is this kind of structure; it has been carefully constructed to allow common components to be shared between almost random values of the discriminant.

6.3 Direct Code

There seems to be a hope among those interested in using a translator to assist them in transitioning from CMS-2 to Ada that it can figure out what the direct code is doing and translate it into Ada. This simply cannot be done. One example should suffice: how can a translator determine the meaning of an instruction that loads a register using indirection, where the indirect word used is constructed during execution?

The only approach that has any hope of success is to simulate the direct code as part of execution of the translated Ada. Basically, this means outputting—as part of the translation—a simulator for the original target machine. Even this is not foolproof, because much direct code depends on the way the program's data are allocated by the CMS-2 compiler. A more fundamental challenge to this approach is: what is the point? Execution of an ISA simulator, albeit written in Ada, is not the purpose of translating CMS-2 to Ada.

7. TRADA Translation Strategies

Most of the translations used by TRADA are straightforward: procedures are translated into procedures, functions into functions, assignment statements into assignment statements (usually), etc. There are also a number of special translations used to overcome dissimilarities of the two languages.

Even some of the problematic areas have relatively straightforward translations. String manipulations (section 4.8) are handled through slicing and concatenating with blanks. A multi-receptacle assignment (section 4.6) is translated into a sequence of assignments with the appropriate conversions, the only complication—and a very minor one at that—occurring when the receptacles are a mixture of numeric and boolean.

Among the straightforward translations are an exact duplication of the CMS-2 fixed-point scaling rules. Conversion between fixed-point types is by truncation by default, which means through the use of a TRADA-generated, inefficient, truncation function. The user can specify that a simple Ada type conversion is to be employed instead. It is hoped that this option will be used only if the user verifies that the algorithms being translated are not so finely tuned that a one-bit discrepancy would cause problems or if it is known that the Ada compiler being used generates code that does (always) convert by truncation.

7.1 Untranslatable Constructs

As is common with translators, when TRADA encounters a construct it cannot translate, it outputs the construct as a comment, along with a special comment indicating that user intervention is required. The format of the comment makes it easy to locate using an editor.

To follow this approach slavishly would reduce the effectiveness of a translation of CMS-2 because of the untranslatability of bit-twiddles. Many expressions, and thus the statements in which they appear, would be marked as untranslatable.

When TRADA encounters a bit-twiddle while translating an expression, it is translated into a generated object with the appropriate type. For example, a word reference is translated into an object named Word_reference of a type that is appropriate for the original target computer. Similarly, a reference to the BIT pseudo-function is translated as Bit_reference_*len*, where *len* is the length of the specified bit string. Through this technique, the other semantics of the expression, such as any implicit type conversions, and the statement in which the expression appears, are translated. The comment that user intervention is required is also output, of course.

The declarations of these generated objects are gathered into a single package.

7.2 Array Assignments

The translation of a CMS-2 array is, in general, fairly straightforward. A type is generated to translate the array's component structure. A second type is generated to declare the array type itself. (This type is unconstrained.) This second type is then used to declare the translated array object.

Following this simple approach would result in translating many otherwise acceptable CMS-2 array assignments into invalid Ada, because the corresponding Ada array types would have different names. In order to translate array assignments whose CMS-2 semantics match those of Ada, TRADA analyzes the structure of all array components and record type specifications in the program being translated to determine common structures. Using the information from this analysis, arrays that have common component structures are translated using a common array type and array assignments involving arrays whose components' structures are different are flagged as being untranslatable.

This analysis also uncovers assignments of arrays whose components' structures are not identical. These assignments are flagged as untranslatable.

The "smallest controls" semantics of CMS-2 array assignment (section 4.7) requires some array assignments to be translated using slicing or loops.

7.3 Indirect Arrays

Because the indirect array of CMS-2 has so many of the properties of a pointer in other languages, it is translated as an object of an access type whose designated object is the corresponding array type. The only translation difficulty arises when an indirect array is set to point to a "real" array, one that is not indirect.

TRADA's solution to this problem is to note all "real" arrays that are pointed to at any point in the program being translated. They are then also translated into the designated objects of an appropriate constant access object. The object declaration includes the allocator to create the designated object. If the real array includes initial values for some components, this initial value is supplied either as a qualified expression in the creating allocator or by code in the initialization block of the appropriate package body. (The choice is user selectable.)

This translation strategy illustrates why it is necessary to translate an entire program rather than a program fragment. A program fragment might contain the declaration of a "real" array but no instances of its being pointed to. It would therefore be translated into a static object and this translation could not be used with other parts of the program where its access value is needed.

(When translating into Ada 9X, the same analyses must be done but the actual translation is somewhat simpler because of the *aliased* attribute.)

7.4 Common Enumeration Types

Because of the very loose semantics of the CMS-2 enumeration types, TRADA must detect enumeration type specifications that specify the same set of values in order to effectively translate assignment and comparison of enumeration values. (This is quite similar to, but much easier than, the problem of detecting common array component structures.) Common enumeration type declarations are then used in the translated code.

If assignments or comparisons involve CMS-2 enumeration types that do not specify the same set of values, TRADA translates by forcing a conversion of the Ada encoding, using code of the form

```
To_type'Val(From_type'Pos(Value)).
```

If the source type has more values than the destination type, execution of this expression could raise a Constraint_error exception. This is an example of a translation that is possibly inaccurate, but the inaccurate translation is detectable by other means.

7.5 Named Numbers

CMS-2 named numbers are translated to Ada named numbers, in general. For the most part, expressions using named numbers are left as expressions; they are not folded by TRADA into static values. This allows the named number to continue being used in Ada as a source code parameter.

The primary exception to this strategy arises when a named number is used to specify an attribute of a numeric type, such as the number of fractional bits of a fixed-point type. To leave the type parameterizable in the Ada code would require an elaborate translation. One aspect of this translation would be an execution-time interpretation of the CMS-2 scaling rules. For this reason, the attributes of numeric types are fixed at translation time, even if they are parameterized in the CMS-2 code. (Often this parameterizing of numeric types comes from an ill-considered adherence to the rule that no hard-coded constants should appear in a program.)

7.6 Boolean 0 and 1

All uses of CMS-2 named numbers are analyzed to see if they are ever used in a boolean context. If so, they are translated as boolean constants, rather than named numbers. The cases

```
TRUE  EQUALS 1  $
FALSE EQUALS 0  $
```

are handled specially. They are not translated at all and references are translated into True and False from package Standard.

In the unlikely but not impossible case that such a named number is also used as an integer, the form

```
Boolean'Val(Value)
```

is employed. Other, more elegant, translations are possible, but the circumstance is so unlikely that is seems inappropriate to expend effort on it. (Code that uses the same identifier in both numeric and boolean contexts should probably be rethought.)

7.7 Subprogram Parameter Passing

Initially TRADA intended to mimic the CMS-2 parameter passage faithfully: each CMS-2 subprogram would be translated into a parameterless Ada subprogram, each invocation of a subprogram would be preceded by assignment statements mimicking the assignment of the input parameters and followed by assignment statement mimicking the assignment of the output parameters. This approach would solve the problem of cross-parameter interference. However, it was recognized early on that it would clutter the Ada code quite a bit. This was particularly true in evaluating an expression that contained more than one function reference, when it would be necessary to evaluate the function references in the proper order into temporary locations (which would need largely meaningless TRADA-generated names), each evaluation being preceded by the parameter assignment statements. This clutter would conflict with the goal of producing code that is at least as maintainable as the original CMS-2.

The strategy settled on is to have formal parameters of mode IN corresponding to the CMS-2 input parameters and formal parameters of mode OUT corresponding to the CMS-2 output parameters. Subprogram invocations are then translated in the straightforward manner. Execution of the body of the subprogram begins with assigning the formal input parameters to the external objects that are the translations of the CMS-2 formal input parameters. Thereafter, those external objects are referenced rather than the Ada formal parameters. For procedures, the values of the external objects that are the translations of the CMS-2 formal output parameters are assigned to the corresponding Ada formal output parameters immediately before returning.

This strategy translates the semantics of the CMS-2 subprogram invocation except in the case of cross-parameter interference. TRADA analyzes all invocations for such interference and outputs a warning message when it is detected, calling for post-translation analysis and modification. Cross-parameter interference occurs rarely and is a poor programming practice when it is used. It is better to handle it this way than to faithfully simulate it with elaborate code sequences that are difficult to maintain.

7.8 Procedure Abnormal Exits

Procedures with exit parameters are translated into procedures with an "extra" parameter of mode OUT. This parameter is of an enumeration type which has one value corresponding to each exit parameter and one value corresponding to a "normal" return. In the body of the translated procedure this parameter is set appropriately depending on the return statement executed. The parameter's value is then tested following each invocation to determine the next statement to be executed.

In order to avoid changing the values of the actual output parameters when an "abnormal" exit is taken, the mode of the formal output parameters is made IN OUT and the assignment of the external objects to the corresponding formal output parameters, described above, is not done when an abnormal exit is translated. The mode ensures that the values are not changed for parameters that use the copy in/copy back mechanism. (If the mode were left as OUT, the uninitialized bit pattern that fills the stack location allocated to a formal output parameter would be copied to its corresponding actual parameter at the conclusion of execution. When the mode is IN OUT, that stack location is filled with the value of the actual parameter at the beginning of execution.)

7.8 Procedure Switches

A procedure switch is translated into a procedure, whose name is the switch name. The formal parameters of this procedure correspond to the switch's formal parameters, according to the same scheme as for translating procedure formal parameters. The body of the procedure contains a case statement or an if-elsif-else sequence to accomplish the switch. The form of the body is determined by such considerations as switch value density, but in general indexed switches give rise to case statements and item switches give rise to if-elsif-else sequences.

For an indexed procedure switch, the corresponding procedure has an "extra" input parameter, which is the value to be used in making the switch.

If the switch is ever invoked using an invalid clause, the corresponding procedure has an "extra" boolean output parameter, which is set according to the validity of the switch value.

7.10 Loops

Because of the free use that can be made of a loop index in CMS-2, a simple indexing loop cannot be easily translated into a FOR-loop in Ada, where the use of the index is very restricted. Before such a translation could be made, a heavy analysis of all uses of the index would have to be done, checking for such things as updating the index, accessing the index from within a subprogram that is invoked (directly or indirectly) in the loop, use of the loop value after loop termination, etc. Rather than invest in this analysis, TRADA translates such loops into "infinite" loops, with explicit incrementation and termination testing at the bottom.

A simple WHILE-loop in CMS-2 is translated into a WHILE-loop in the obvious fashion. However, a loop-UNTIL must be translated into an "infinite" loop with an explicit test at the bottom. The "infinite" loop with explicit termination tests must also be used when the CMS-2 loop has a combination of termination conditions, such as multiple indexes, an index and an UNTIL-condition, etc.

Even the "infinite" loop is too restrictive when the CMS-2 loop is resumed from outside. In this case the looping structure is controlled by goto statements, to allow the translation of the resume statement to branch into the loop.

7.11 Labels, Goto Statements, and Label Switches

Labels and goto statements are translated directly. Because of the free transfer of control allowed in CMS-2, this can result in illegal Ada goto statements. The error messages produced by the Ada compiler will be an adequate signal that human intervention is required.

In general, a label switch is translated as a case statement or if-elsif-else statement with a goto statement in each alternative. Whenever TRADA can verify that a code sequence can only be reached by a transfer through a label switch, that sequence is moved into the appropriate alternative of the case statement.

7.12 Data Local to Subprograms

In general, data local to subprograms are translated into regional data in the subprogram's package, in order to preserve their staticness. Such a datum's name is modified by appending the subprogram's name in order to avoid name clashes, if necessary.

To avoid this cluttering of names and to make effective use of the Ada stack, TRADA identifies local data that are always assigned before they are referenced. In the translation, these data remain local to their subprogram.

7.13 Initial Values

Data that are given initial values in CMS-2 are given those values in the TRADA output. For a variable of the basic types, the initial value is specified as part of the corresponding object declaration. This technique can also be used for an array, where the object declaration includes an aggregate specifying the initial value. For a large array, the aggregate might well exceed the capacity of the Ada compiler being used. To avoid this problem, a TRADA option allows the user to specify that arrays should be given their initial values through the execution of assignment statements. For a global or regional array, these executable statements are placed in the initialization block of the array's package and are thus executed only at program startup.

The user can specify whether data that are not given initial values should be left indeterminate or "zeroed", to mimic the de facto zeroing of the CMS-2 linkers. If "zeroing" is chosen, numeric data are given the value zero, character data are given ASCII.NUL, and enumeration data are given their type's first value.

When aggregates are used to initialize array components, the above "zeroing" technique is used for components that have no specified initial value.

7.14 Typed Records

A record type that has an associated basic type is translated into an Ada record type. When translating an object of this type, a "shadow" object of the translated basic type is also created, and a comment calling for user intervention is also output. References to the whole object are then translated as references to the "shadow" object.

7.15 Conditional Compilation

Two features of the CMS-2 conditional compilation feature make translation into "parameterized" Ada impossible: the fact that the code in a block being ignored might not be correct and the possibility of declaring an identifier multiple times in distinct blocks. For this reason, TRADA translates only one configuration of the program, the one corresponding to the settings of the compilation flags at the time of translation.

8. A TRADA Shortcoming

The unusual nature of subprogram formal parameters gives rise to one tractable problem that TRADA does not attempt to handle. If an expression contains two or more function references, then the execution of one of those references could affect the parameter passage of a subsequent one. This can happen two ways: during the parameter passage stage of the earlier function reference or during execution of its body. At either of these times, a value could be changed that plays a role in the actual parameters of the later reference, either by changing an actual parameter or by changing a value that is used to select an actual parameter, such as a value that occurs in a subscript expression.

Although it is not possible to detect this inter-function parameter interference with 100% certainty, the problem could be solved by breaking the expression apart to make certain that the translated function references occur in the same order as in the CMS-2 code. As was discussed in section 7.7, this can result in obscure code that is difficult to maintain. Since using this "feature" is an unconscionable programming practice and its uses are extremely rare, it was decided to aim for the readability/maintainability goal rather than bulletproof translation in this case, even though it is not possible to give the usual warning that intervention is required.

This is the only known case in which TRADA will generate possibly incorrect Ada without either a warning message, an Ada compiler error, or an exception during execution of the translated code.

9. Summary

TRADA is a translator driven by the goals of conservative translation (either faithfully reproducing the semantics of a construct or marking it as untranslatable, requiring user intervention), producing maintainable code, and producing transportable code. Of the three goals, conservative translation is the most important. With it, the user's part of the translation effort can be attacked without worrying about the part done by TRADA.

TRADA achieves its goals to a high degree. It often uses full knowledge of the entire program being translated in order to chose its translation strategy, where relying on only local knowledge might lead to an incorrect translation. Choices between alternative translation strategies have been made on the basis of maintainability and knowledge of the de facto uses of the construct being translated. Lastly, it refuses to translate into non-portable Ada or to translate CMS-2 constructs whose semantics are not fully known.

Bibliography

The following documents define the various CMS-2 dialects and describe their compilers.

CM2Y-MAN-PGR-M5049-R04C0, CMS-2Y Programmer's Reference Manual for the AN/UYK-7 and AN/UYK-43 Computers, FCDSSA, San Diego, 1 October 1986.

CM2Y-MAN-PGR-M5045-R05C0, CMS-2Y Programmer's Reference Manual for the AN/UYK-20 and AN/AYK-14 Computers, FCDSSA, San Diego, 1 December 1986.

CM2Y-MAN-PGR-M5047-R01C0, CMS-2Y Programmer's Reference Manual for the CP642 Computer, FCDSSA, San Diego, 1 October 1986.

CM2Y-MAN-PGR-M5044-R01C0, CMS-2Y Programmer's Reference Manual for the Transferrable Subset, FCDSSA, San Diego, 1 October 1986.

NAVSEA 0965-LP-598-8020, User Handbook for CMS-2 Compiler, Revision 4, Change 1, Department of Navy, Washington, D. C., 30 November 1993.

The San Diego FCDSSA (Fleet Combat Direction Systems Support Activity) was the U. S. Navy organization responsible for the first four of the above documents when they were created and is cited on their covers. The duties of FCDSSA are now part of those of NRaD.

Ada is defined in

ANSI/MIL-STD-1815A-1983, Ada Programming Language, Department of Defense, Washington, D. C., January 1983.

Reengineering Concurrent Software Into Ada[†]

Noah Prywes*, G. Ingargiola, I. Lee*, and M. Lee***
Computer Command and Control Company
2300 Chestnut St.
Philadelphia, PA 19103

Abstract

The paper describes a methodology for translating concurrent software into Ada, where the concurrency is expressed in Operating Systems calls embedded in the source software. There are two important advantages to such a translation. First, the understanding and analysis of concurrency expressed through Operating System calls is very complex and difficult. Translating these calls (as well as the sequential portions) into Ada greatly simplifies the analysis, and understanding of the software by providing the operational semantics of the calls. Second, the translation to Ada eliminates dependence on Operating Systems.

The focus of the paper is on translation of concurrency related software portions of the source software into Ada. It is part of a larger system for translating both sequential and concurrent aspects of real–time applications software. Translating the sequential portions of the source code into Ada has been documented separately in previous reports [4, 13]. These reports provide background to the presently reported work.

The immediate motivation for the work described here is due to the need to automatically translate Navy software in CMS–2 into Ada, where the concurrency is expressed by calls to the SDEX–20 or ATES Operating Systems. Much of the Navy mission–critical systems use these languages. The translation to Ada is needed to modernize these systems.

The methodology we propose for replacing Operating System calls by equivalent Ada code uses a standard message–based kernel oriented architecture. This architecture is specialized for each Operating System and modularly extended to represent the individual features (concurrency control, input/output, etc.) supported by that Operating System. Within this architecture the determination of a generic method for translating into Ada a new Operating System call, though non–elementary, is not overly time consuming (we estimate less than one week of programming for each additional Operating System call). The use in the translation of a standard message–based architecture has a cost in lost performance with respect to the performance attainable in an optimal ad hoc translation using a shared memory model. Upper bounds on this performance cost can be determined, and can be used to establish the characteristics of the hardware required to achieve specific real–time deadlines.

T[illegible]nslation is based on the notion of *functional equivalence* of the source and target software. It is attained by adhering closely in the target Ada software to the data, data layouts, software units and data transformations in the source software. In some cases, additional information, typically found in a software specification, is needed to achieve programs that are free of hardware and scheduling dependencies.

The translation uses an intermediate entity–relation–attribute graphic model of the target Ada software which is created and progressively enriched. It serves as a repository for all the information extracted from the source software and the software specification. The graphic software model is then used as a basis for generating the target Ada software, for analysis and abstracting of the software and for generating an up–to–date software specification.

The concurrency translation to Ada is described in this paper in sufficient detail for executing it manually, or implementing it automatically. The methodology utilizes templates for data structures, tasks and procedures for translating into Ada of multiple Operating Systems. Additionally, for each Operating System call, it is necessary to compose templates of procedures that execute the respective protocols. The templates are placed in the Ada library, provided in [6]. It includes the generic templates and, as an example, templates for seven Unix calls. These templates provide a general framework for the translation of concurrency related Operating System calls to Ada.

*Also affiliated with the University of Pennsylvania
**Also affiliated with Temple University
[†] Prepared under Contract N60921–92–C–0196 Naval Surface Warfare Center

A concurrent software example in C using Unix Operating System calls is provided in the Appendix. It illustrates synthesis of the Ada templates and generation of a complete Ada program. The choice of C and Unix is due to the familiarity of readers with these languages.

Needed future work is also described. It consists of: (i) Extending the present approach to distributed processing, using Ada–9X. (ii) Extending the target software to assure mutual exclusion, progress, and limited postponement, independently of the hardware speeds and scheduling that are used. (iii) Selecting new target hardware, to ensure that timing requirements are met.

1. Introduction

This paper describes the translation of concurrency–related Operating System calls to Ada. Concurrency–related Operating Systems calls generally have been informally documented. This has made understanding and analysis of such concurrent programs extremely difficult. The translation to Ada greatly simplifies complete presentation and understanding of the role of concurrency. Further, the software becomes independent of Operating Systems used with the hardware.

The immediate motivation for the work described in this paper has been due to the need for automatic translation of concurrent real–time Navy software in CMS–2 into Ada. Embedded in the CMS–2 code are concurrency–related calls to ATES or SDEX–20 Operating Systems. Such programs are widely used in Navy mission–critical applications. The translation is needed for modernization of these systems.

Still another motivation has been due to the relatively new field of Software Reengineering. Its objective is to process existing software automatically, or semi–manually, in order to obtain modern software for the same application or new applications for execution on a high speed distributed network.

The overall reengineering process involves code translation and creation of a software model that is used to provide a number of capabilities. These include software analysis, facilitation of software understanding, documentation of the software, reorganization and restructuring of the software and interfacing it with other software.

The translation of concurrency–related Operating System call is an integral part of providing these capabilities.The translation of concurrency–related Operating System calls to Ada is a step in a larger process. The software translation process consists of several progressive steps, illustrated in Figure 1 for the translation of CMS–2 to Ada. They are as follows [14]:

(i) translation of sequential portions of the software into Ada [4, 13];

(ii) translation of the concurrency related statements of the software, expressed in Operating System calls, into Ada (this is the topic of this paper);

(iii) storage of a graphic model of the Ada target software in a database and its display and modification [7,13];

(iv) generation of a software specification [5];

(v) enhancing concurrent software so that its logical correctness is independent of processor speed (requires further work; see Section 6);

(vi) simulating scheduling of concurrent software to determine the new hardware resources needed to meet timing requirements [10] . The hardware resources may be in a distributed processors and communications network (requires further work, see Section 6).

The paper focus is only on item (ii)—the translation of concurrency expressed in Operating Systems calls into Ada. The paper describes this step in sufficient detail to have it performed manually, or to program it. Steps (i), (iii), and (iv) have been implemented previously and are reported in respective references. Step (v) and (vi) require further research.

This paper contains the following sections.

Section 2 is a technical problem statement. It establishes the requirements for the translation by defining the functional equivalence of the source and target software.

Section 3 describes the graphic model of the software. It is created as the output of translating the source software, adhering closely to its data, data layouts, software units (tasks, procedure, functions), data transformations, and their precedences and interactions. The use of a graphic software model is important here because it is needed for the progressive translation steps and because it is used later for analysis, understanding and creating an up–to–date specification of the software.

Section 4 presents the translation process. It consists of generating Ada tasks and procedures to implement concurrency–oriented Operating Systems calls. The translation replaces concurrency–related Operating System calls with respective Ada tasks, procedures and messages. It creates tasks for the source program processes and for synchronizing these processes.

Section 5 describes the translation process in greater detail. It uses an Ada template library for synthesizing the concurrent aspects of the code. The Ada library contains the code needed generally for translating concurrency–related Operating System calls and also example procedures for translating several Unix Operating System calls. The Ada library is given in [6]. The methodology is applicable to other source software languages and Operating Systems (e.g. CMS–2 with ATES and SDEX–20).

The translation of Operating System calls is only partly feasible in some instances. To illustrate this point, the paper includes in the Appendix an example of the translation of the Unix *fork* call into Ada. The computational model of Unix differs greatly from the computational model of Ada. For example, Unix tasks are much "larger" granular objects than Ada tasks: a Unix task has its own address space and maintains by default information about open files, related tasks, and signals; no such information is available by default with Ada tasks and in most implementations Ada tasks share the same address space. More fundamentally, an Operating System controls tasks switching and can make decisions when dispatching a task from the Ready state to the Running state (examples of such decisions are to suspend or to terminate a task). No such fine grained control is possible in Ada since no facilities are provided by the language to control task switching (Ada 9X will correct to an extent this problem); even the drastic "Abort" statement does not take effect until the aborted task, on its own, reaches a synchronization point. One objective of the work reported here is to clarify and bridge over such differences.

The appendix refers to templates for Unix concurrency–related calls. Seven of these calls are shown as having been defined by respective procedures. The implementation of procedures for all of the Unix calls and their placement in the Ada library, is required for attaining the full translation of Unix into Ada. The appendix shows the use of the Ada templates in the library for translating an example source software in C with Unix calls into Ada.

Section 6 concludes the paper with a description of future needed research and development as follows. The objective of the translation is to be able to execute the produced Ada software on new distributed hardware. The currently presented translation is restricted to use of a single processor and Ada. It needs to be extended to use of distributed processing and Ada–9x. It is also necessary to modify the target software to be logically independent of timing dependencies due to the hardware. Finally, it is necessary to determine the processing and communication capacities needed to satisfy the timing requirements.

2. Technical Problem Statement

The basic translation requirement is to produce Ada target software which is functionally equivalent to the source software. *Functional equivalence* is defined as follows:

A *Software specification* defines legal input sequences and respective outputs of the software, as well as additional requirements. Functional equivalence of the source and target software of the translation means that they both conform to the software specification. They both consume legal input sequences and produce respective legal outputs. A software specification may not be reliable or even available. Instead, it is preferable to use the source software whenever possible. The source software that is used as input to the translation is assumed to be a well–tested, extensively used and highly reliable representation of the software specification, though it may be incomplete in some of the cases described below. If the source software is modified prior to the translation, then it is required that it be tested or shown not to affect the respective computations.

(i) ***Sequential Software***: In this case the source software is a complete representation of the specification (assuming no timing requirements). The source software defines all the precedences and operations needed to process legal input sequences and produce respective legal outputs. The target Ada software is then functionally equivalent to the source software as it adheres to the latter's precedences and operations.

(ii) ***Concurrent Software***:

(a) **Source Software where the logical correctness is independent of the hardware speed and Operating System**: In this case the source software also defines all the concurrent execution threads and the synchronizations needed for accepting legal inputs and producing legal outputs, as defined in the software specification . For given inputs execution of the target Ada software may produce different outputs from those produced by processing the source software, but still the target software is functionally equivalent to the source software as the outputs still comply with the software specification. For example, the source and target software may execute a same sequence of inputs, the former may produce sequences of different outputs than the latter, but they both conform with the software specification.

(b) **Source Software where the logical correctness is dependent on the source hardware speed and Operating System:** The source software programmer may have relied on delays, due to relative speeds of the hardware in executing portions of the software, and has omitted entering explicitly the respective synchronizations in the code. In this case the source software is lacking some synchronization statements to retain the logical correctness independent of the hardware and Operating System used. These synchronization statements must be added to either the source or the target Ada software in order to comply with the software specification. (This is further discussed in Section 6.2, as future research.)

(iii) **Timing:** The timing requirements are documented in the software specification, but typically not in the source software. The needed capacity of the network's processors and communications, must be determined for executing the Ada target software while guaranteeing the timing requirements (This is further discussed in Section 6.3, as future research).

Note that satisfying item (i) above is performed by translating the sequential software in the top box in Figure 1. It has been reported previously [4,13]. This is adequate for purely sequential source software. However, by itself it provides an incoherent concept of the software if the software contains concurrencies. Satisfying items (ii)(a) above is performed in the next box in Figure 1. This is the main topic of this paper. Satisfying items (ii)(b) and (iii) above is performed in the third and last boxes in Figure 1. They depend on the existence of information not directly available by inspecting the source code. In particular (ii) (b) requires the availability of specifications that clarify what are the concurrency constraints so that we can determine which are enforced by the source code, and which are voided by assumptions made about hard ware speed and scheduling discipline. These items are discussed in Section 6 as needing future research.

To achieve func ional equivalence, the translation is based on very close adherence to the memory and transformations of the sot rce software. Instead of relying on the software specificatic n, we rely in items (i) and (ii)(a) above on the software code which has been assumed to be a tested, tried–out, reliable, and machine–readable representation of the software specification. The information from the source software is stored progressively in an Ada–oriented graphic software model.

Close adherence to the source software means that all the entities, operations, and precedences of the source software are represented in the graphic software model using Ada semantics. They are as follows.

(i) Declarations of same named variables as declared in source software, using the same memory layout, and same scope (same shared memory).

(ii) An Ada task and communications to perform the functions of the Operating System calls used in source software.

(iii) Ada tasks to represent each of the concurrent processes in the source software.

(iv) Ada procedures and functions to represent respective procedures and functions in the source software.

(v) Ada I/O to represent I/O devices in the source software.

(vi) Ada transformation statements (executable statements) for each data or control transformation statement in the source software. They perform the same operations in Ada in the same sequential order.

The Ada–oriented graphic model also contains directed edges between the above nodes/entities to indicate precedences in execution of statements as well as other relations enumerated below.

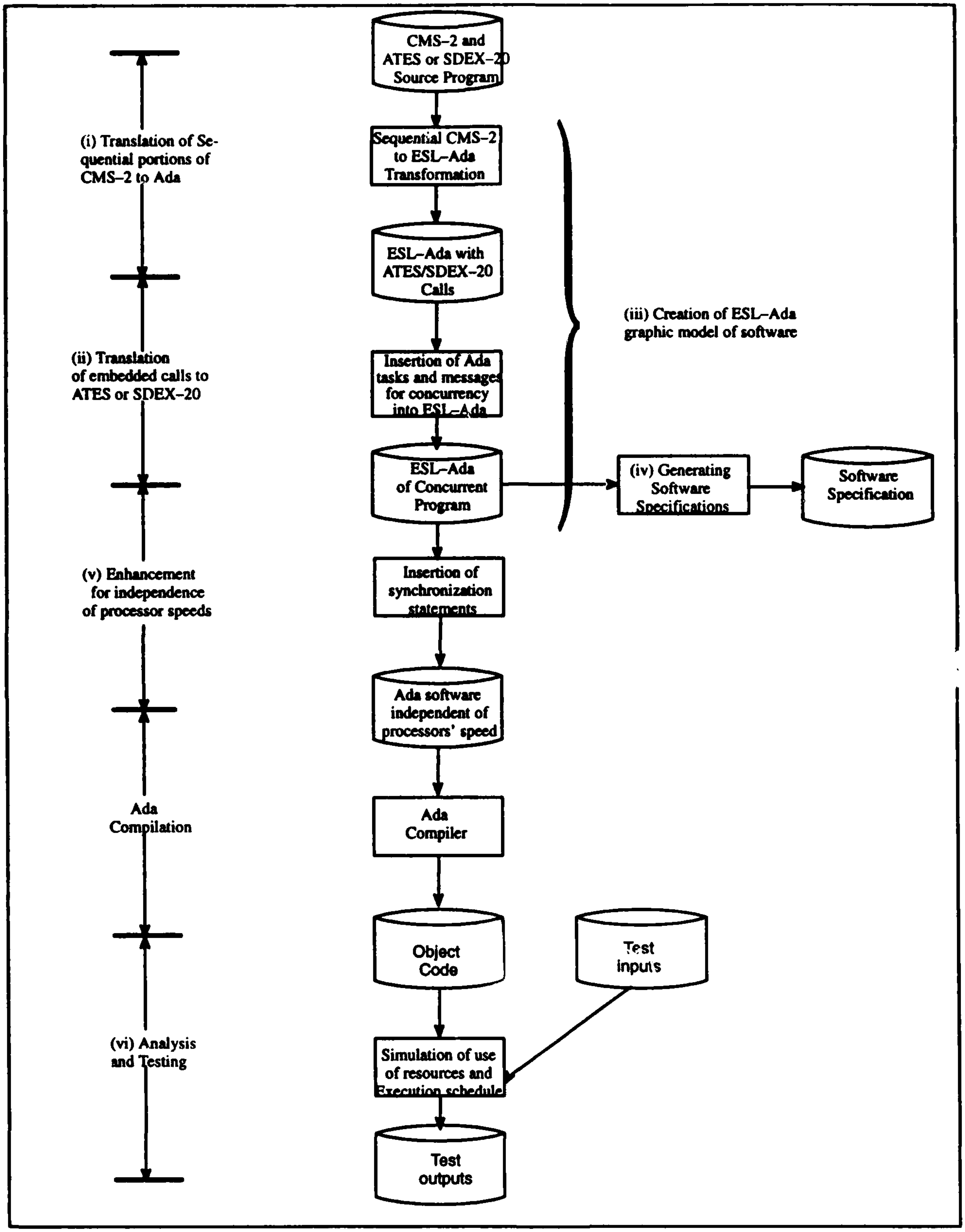

CMS-2 and
ATES or SDEX-20
Source Program
(i) Translation of Sequential portions of CMS-2 to Ada
Sequential CMS-2 to ESL-Ada Transformation
ESL-Ada with ATES/SDEX-20 Calls
(iii) Creation of ESL-Ada graphic model of software
(ii) Translation of embedded calls to ATES or SDEX-20
Insertion of Ada tasks and messages for concurrency into ESL-Ada
ESL-Ada of Concurrent Program
(iv) Generating Software Specifications
Software Specification
(v) Enhancement for independence of processor speeds
Insertion of synchronization statements
Ada software independent of processors' speed
Ada Compilation
Ada Compiler
Object Code
Test inputs
(vi) Analysis and Testing
Simulation of use of resources and Execution schedule
Test outputs

Figure 1: The Six Steps in Reengineering CMS-2 Software with Calls to ATES or SDEX-20 into Ada

3. A Graphic Model of the Target Ada Software

The main translation activity consists of progressively creating from the source software a graphic software model which employs Ada semantics. This model contains first the translation into Ada of the sequential portions of the source software. Then it is augmented by adding the translation of concurrency related Operating System calls. Later the graph is further modified and extended progressively, without affecting the declarations or order of the operations of the source software. For example, changes are made to partition overall large-scale software into hierarchical software units. The graphic model is also reorganized and restructured to conform to the object-oriented Ada programming paradigm. The graphic software model includes tracings to the source statements whose translation caused the respective target Ada statements. Although the target software may be reordered, the respective source statements can be located and the translation remains explainable and justifiable.

A summary of the contents of the graphic software model is given in the following:

The graphic model representation is called an *Elementary Statement Language* for Ada (ESL–Ada). ESL–Ada is a graphic language to describe Ada code [4]. Every Ada statement forms a node in the graph. Thus, there are nodes for declarations of data, software unit types (tasks and procedures), generics and their instantiations. There are also nodes for execution statements (e.g. assignments, conditions, loops, etc.). A node includes attributes of the respective statement such as the parsed statement, a trace to a source statement, and an icon for the node's display. Thus, all the statements in the source software are represented progressively in the graphic software model by nodes.

An ESL–Ada graph contains edges which connect the nodes to form a tree. A tree branch from a node to another node at the same software hierarchical level means that the statement of the first node immediately precedes the statement of the latter node. A branch from a higher software hierarchical level block statement node to a chain of the next lower level nodes represents lexical containment. This type of edge is called a *scope tuple.* These edges specify all the precedences derivable from the source program.

Additional edges are then created between pairs of nodes to represent binary relations between nodes. These edges are also called *tuples*. There are six additional types of tuples indicating respective relationships. These tuples are used in reordering and reorganizing code and for creating software abstractions for documenting the software. They will also be used for analysis of mutual exclusion and simulation of the timing (see Section 6). They are as follows:

(i) a relation between a statement that references or updates a variable and a statement that declares the variable (*memory* tuple);

(ii) a relation between a procedure call statement and the procedure's declaration (*call* tuple);

(iii) a relation between a message call statement and the called entry point declaration (*message* tuple); this includes synchronization calls;

(iv) relation between an i/o call and the respective device declaration; (*i/o* tuple);

(v) relation between a generic or type declaration and respective instantiation statement (*type* tuple);

(vi) relation between a specification and its respective body declaration, or between with/use statement and the respective package declaration (*context* tuple).

Each tuple/edge has attributes, such as its type, icon, etc.

A specification of the sequential translation process, for a specific source language, requires that the syntax and semantics of each construct in the source language are given a syntactic and semantic representation in Ada. This has been done so far for CMS-2 [4].

The translation of Operating System calls consists of their representation in Ada in the graphic software model. As will be discussed, the translation requires using Ada tasks and procedures for Operating System calls.

The graph is stored as an Entity–Relation–Attribute database [2], where the nodes are the entities and the relations between them are the edges. The graphic software model is stored in a graphic database. Retrieval queries and display allow selective viewing of graphic views of the software from different perspectives or of different granularity of the code. These displays are critical to understanding of the software. The retrieved views can be modified graphically by adding or deleting of nodes or edges.

As noted, the ESL–Ada graph is the basis for software abstractions and documentation. The overall graph is partitioned into subgraphs of *hierarchical software units.* These subgraphs progressively portray the partitioned units of the software architecture. Separate graphs are produced for each hierarchical software unit. Each graph for a hierarchical software unit contains edges/tuples to nodes in other hierarchical software units with which the unit interacts. Diagrams and text are then generated as listed below. They provide the abstractions of the software that constitute the software's specifications in accordance with DOD–STD 2167A [8].

Hierarchical Decomposition Diagrams—showing decomposition of the overall software into hierarchical units. (based on the scope tuples)

Flow Diagrams—showing flow of data and control within and between hierarchical units. (based on the memory, call, message and i/o tuples)

Interface Tables—showing the structure of inputs and outputs of each hierarchical unit. (based on the scope, call, message and i/o tuples)

Object/Use Diagrams—showing for hierarchical units where types or generics are defined and where they are used. (based on the type tuples)

Context Diagrams—showing the library units and where they are used. (based on the context tuple)

Comments Text—showing the comments in each hierarchical unit. They are assumed to contain information on the hierarchical unit's capabilities.

These diagrams can be retrieved and displayed (with appropriate layout) based on queries that cite the desired nodes and tuple types as parameters.

4. Strategy for Translating Concurrency–Related Operating System Calls

This section defines the Ada entities that are synthesized to accomplish the protocols of Operating System calls. The translation replaces processes in the source software with Ada tasks called functional tasks. Additional tasks are created for a *controller* to execute Operating System calls and for buffering of communications between other tasks. Procedures are added for each Operating System call. They are illustrated in Figure 2 and discussed below. Large rectangles in Figure 2 denote Ada tasks; bubbles denote task entries; communication paths denote flow of messages (showing direction of the call and direction of the message); columns of small rectangles (inside tasks) denote procedures for executing Operating System calls.

The interprocess communication of the source software Operating System is performed by an Ada task called *controller.* It is shown at the top of Figure 2. It only performs the concurrency related Operating System calls that are actually used in the source software. (Operating Systems error processing and I/O are not considered in this paper. The controller plays the role traditionally played by the kernel in Operating systems. That is, it provides the basic mechanisms for supporting task creation, interaction, and termination, and for communicating with the program's environment.

The source Operating System also performs dispatching of processes and uses a variety of underlying systems. However, these services are provided by the Ada compiler when generating object code by inserting into the code Operating System calls tailored for each vendor's Operating System hardware and communications. These capabilities therefore are are not included in the translation. The scheduling by the source Operating System may differ from that of the target Ada Program but should not affect the correctness of the software, although it may affect the timing (further discussed in Section 6.3). If the source Operating System supports priority assignments for processes, then corresponding priorities are also generated for respective Ada tasks. If the source language or Operating System supports priorities for messages, then they are included in the operations of the mailbox tasks, as described below.

The messages exchanged between tasks are of two kinds:

(i) *control messages* for interpreting Operating System calls; these messages are exchanged by the controller task and the functional tasks.

(ii) *data messages* for communicating variables among processes as specified in the source program; these messages are sent or received by tasks that represent source software processes (the functional tasks).

The Controller task contains in its body a main loop that:

(i) receives a control message from other tasks (via the controller's mailbox task) to perform the equivalent of an Operating System call

(ii) calls a procedure that executes the protocol of the Operating System call. The protocol may involve sending control messages to tasks and receiving acknowledgements of protocol steps.

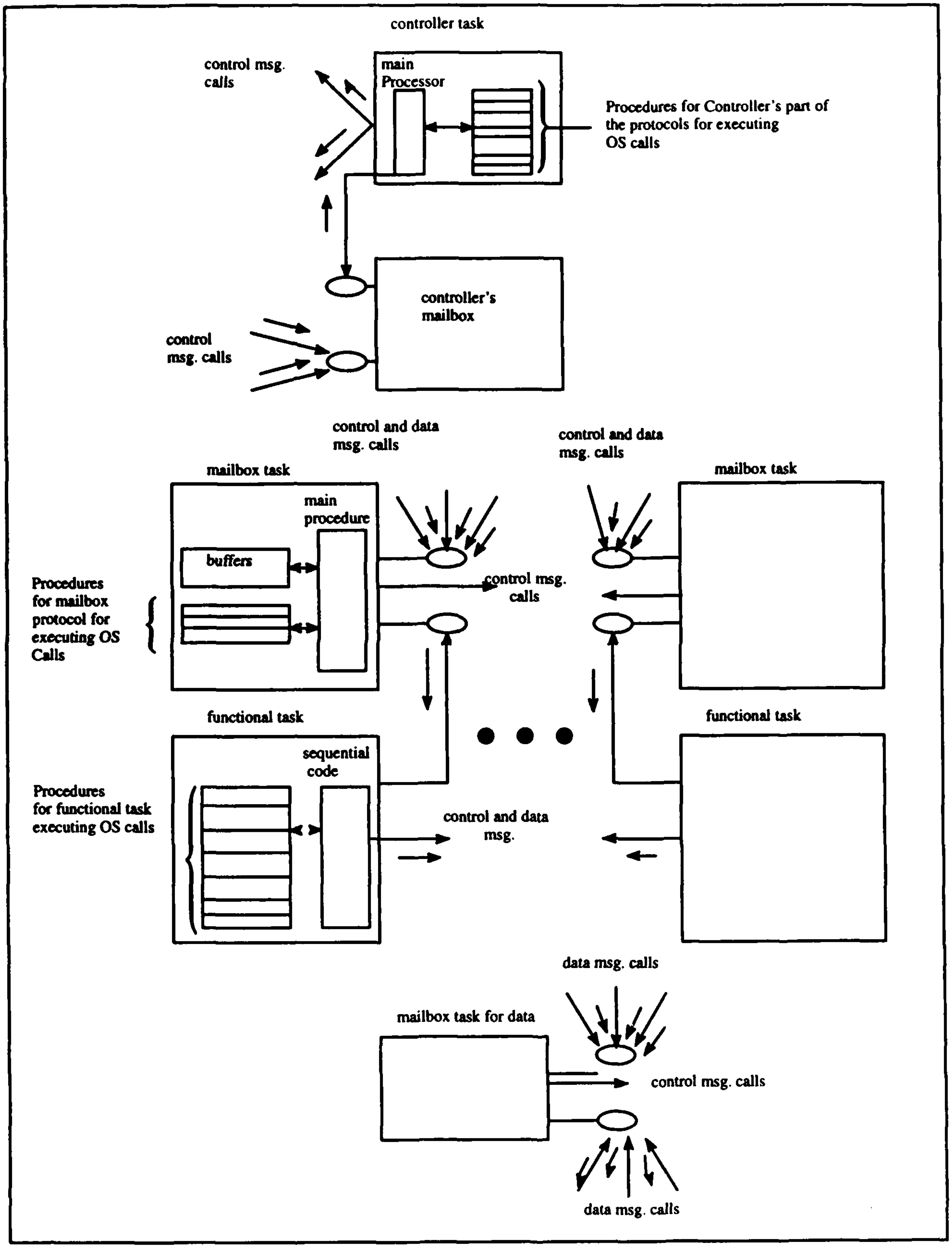


Figure 2: Ada's tasks, procedures, and messages to implement OS Calls

Thus, the Controller task is implemented as a monolithic kernel in the sense that it can execute the protocol of a single Operating System call at a time. This should not be a problem since most of the work of the call is done outside the Controller which only routes messages and updates task control information. If it will prove necessary, we will reconsider the design of the Controller to become an interacting family of tasks, each with a different priority, corresponding to the priority of the callers and to specific segments in the execution of the call. The declaration of a controller task is inserted at the beginning of the target Ada concurrent software. The controller is dynamically provided with data on each task being created. This data is similar to a Process Control Block of an Operating System. It is called *Task Control Block* (TCB).

Processes created by Operating System calls in the source program are translated into declarations of respective Ada tasks, the *functional tasks*. They are illustrated in the middle of Figure 2. The declarations of functional tasks are inserted in the places where there are calls to the Operating System in the source software to create the respective processes. Each functional task contains a main procedure that corresponds to the sequential execution code of the respective process in the source program. In Ada, the sequential execution code is contained in the body of the functional task. The functional task body may contain calls to procedures that send control messages to the controller task, thus causing execution of the protocol of the respective Operating System call. A functional task may be assigned a priority, as indicated, by a respective source software Operating System call. The inserted declaration of functional tasks create the tasks dynamically and their creation is reported to the controller task.

An originating task may call the controller task to execute an Operating System call on a destination task. The controller task then sends the command to the destination task. Each functional task must check periodically if it has a waiting command from the controller. If one exists, then the functional task must execute the command. An example of such a command is a call in one functional task for suspending or terminating another functional task. When the destination functional task receives the command, it suspends or terminates itself normally. The checks for existence of a waiting message add overhead to the execution of these commands. The required response time to these commands determines the frequency of checking for such messages. It must be compensated for by using much faster hardware with the target Ada software than the hardware used with the source hardware.

For the reasons below, it was determined necessary to have all communications between functional tasks or between the controller and a functional task to go via *a mailbox task*. The mailbox task contains intelligence to handle the following:

(i) recognizing and buffering variable length data messages

(ii) recognizing and buffering control messages and giving top priority in delivery of control messages

(iii) delivering data messages in a priority order, in accordance with the message priority requirements of the source software Operating System. Same priority data messages are delivered in first–in first–out order.

(iv) acknowledging receipt or delivery of messages corresponding to requirements of the source software Operating System calls. Note that this can support both guaranteed delivery of messages as well as servicing blocked or unblocked communication commands.

(v) receiving, interpreting, and acknowledging control messages directed to the mailbox task itself. This is necessary to suspend, continue or terminate a mailbox.

Mailbox tasks are created dynamically. They are reported to the controller for each control and functional task. A mailbox task is created at the initialization of the respective controller or functional task. Thus, there is one mailbox task for the controller task and one for the functional task. An additional mailbox task may be created to establish a sending and receiving communication path between multiple processes in the source software. This task is also created dynamically in the corresponding place where the source software communication path is declared in the source software. The mailbox tasks are also shown in Figure 2. Each mailbox task has an entry point for calls of incoming messages and an entry point for delivery of messages.

The use of mailbox tasks may at times add an overhead of as much as double the communication time between tasks. This overhead must also be compensated for by use of hardware that is considerably faster than the hardware used by the source software.

The selection of Ada primitives and their method of synthesis takes into account minimizing the overhead in execution, and maintaining, or even improving the understandability of the graphic model of the software. We distinguish between two types of primitives: those that are generic to many Operating Systems, and those that are specific to a selected Operating System. There are some structures and operations that are common to a number of Operating Systems. The translation of each Operating System call requires composing its own procedures that implement the respective protocol of the call. Both the generic code and the Operating System call procedures are placed in the Ada library as discussed in Section 5. The procedures that execute Operating System calls are shown in Figure 2, inside the respective task boxes.

An Operating system call may not be fully translatable to Ada, or translatable only in a restricted way. For example, the Appendix refers to the Unix Fork call which involves creating a new process. It consists of copying an executable image, creating an appropriate task control structure, and rescheduling. This cannot be done in general directly in Ada. But it can be done in Ada if the code executed in the child thread consists of a call to a predefined pure procedure, i.e. all data used in the procedure is either local or an explicit parameter of the procedure (Out, and In Out parameters of the procedure must have been copied before the procedure is called).

5. Implementing Translation Of Concurrency–related Operating System Calls

5.1 A Two Pass Process

The translation of Operating System calls is shown in Figure 1 as a box titled "Insertion of Ada tasks and messages for concurrency into ESL–Ada." The input of this process is the ESL–Ada model of the software obtained from translating the source sequential code into Ada. It includes source Operating System calls embedded in the sequential Ada code in the graphic software model. This concurrency translation process replaces the source Operating System calls in the ESL–Ada with Ada code.

The translator process is illustrated in further detail in Figure 3. As shown, the ESL–Ada software model is updated in two passes.

The first pass consists of:

(i) scanning the ESL–Ada graphic model of the software to find: (a) the concurrency–related Operating System calls that are used in the source software. (b) which calls create processes and therefore must be replaced by functional task declarations, and (c) the beginning and end of the sequential thread of execution code performed in each of the source processes. This information is tabulated for use in Pass 2.

(ii) generating the controller task which contains calls to the procedures that interpret all the Operating System calls that are used in the source software. As noted above, the code in the library is for a single processor. Only one controller task exists for the entire software. It is generated in place and not included in the Ada library. The controller task declaration is generated in Pass 1. Its specification is inserted at the beginning of the ESL–Ada model of the software. It contains procedures for only the Operating system calls in source software.

(iii) creating a mailbox task for the controller task.

The second pass consists of:

(iv) inserting instantiations of respective functional tasks and their mailbox tasks in place of each Operating System call that creates a process. The inserted code sends a control message to the controller task, reporting the created task control block.

(v) inserting an instantiation of a mailbox task for each Operating System call that establishes an intertask data messages channel.

(vi) Inserting in–place calls to procedures that execute every other type of Operating System call.

5.2 Use of Ada Library

As shown in Figure 3, the creation of Ada tasks and procedures is simplified by instantiation of pre–defined objects in the Ada library. The Ada library is given in [6].

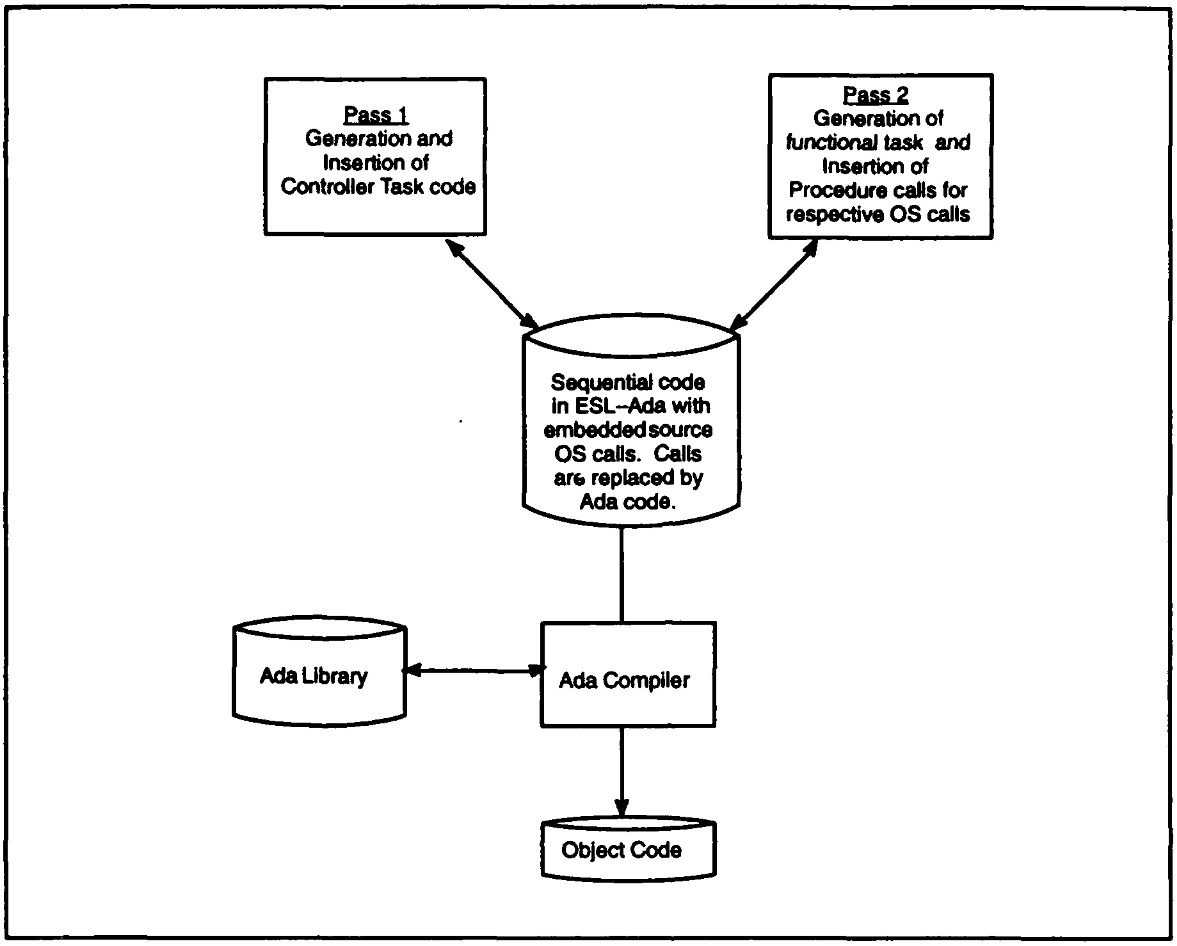

Pass 1
Generation and Insertion of Controller Task code
Pass 2
Generation of functional task and Insertion of Procedure calls for respective OS calls
Sequential code in ESL–Ada with embedded source OS calls. Calls are replaced by Ada code.
Ada Library
Ada Compiler
Object Code

Figure 3: Conversion of Operating System Calls to Ada

I. Data declarations for use in:

mailbox tasks, for receiving, sending and buffering data
and control messages

generic package of a functional tasks.

controller task

II. Types of procedures for:

interpreting the protocols of the source Operating System calls in the:

controller task

functional tasks

mailbox tasks

III. Types of tasks and generic package for:

mailbox task

generic package that contains a functional task

Figure 4: Summary of the Ada Library

The contents of the Ada library is outlined in Figure 4. It contains the data declarations used in the tasks shown in Figure 2 (I). Next, the library contains procedures for the Operating System calls (II) . Finally, the Ada library contains a declaration of a generic package for a functional task and a mailbox type (III). Since there may be a large number of functional tasks, the use of the generic package for the functional task facilitates the software understandability. The instantiation of each functional task in Pass 2 requires providing the generic package with parameters (i) a unique name for the task (ii) the name of the procedure that corresponds to the code executed in the respective source process, and (iii) the names of the procedures for interpreting the Operating System calls by the respective functional task. [6] contains the Ada library.

The synthesis of Ada software for Operating System calls is shown in an example in [6]. It illustrates the Pass 1 and Pass 2 processes through application of this methodology for a program in C for a Producers/Consumer application using the Unix Operating System. This C Program appears in Figure 5. Figure 6 has the correspondent Ada Program. The execution of this program is also documented in [6].

A briefer description of the example in C/Unix translated into Ada is given in the Appendix to this paper. It shows both the C/Unix code input and the Ada code output of the translation process.

6. Future Research And Development

The present paper is in a sense a progress report. As shown in Figure 1, while it reports on the translation of concurrent source software into Ada for execution on a single processor, it leaves to future research the problems of checking and modifying the software to attain independence of the logic on the source hardware and distribution of the calculations. These problems are summarized in this section.

6.1 Distributing Concurrent Software of Large Scale Applications

Large–scale concurrent software is envisaged as consisting of large numbers of Ada tasks divided among software units. Further, the translation from large–scale source software is typically performed in pieces, producing one translation unit at a time (e.g. similar to an Ada compilation unit). Frequently communicating pairs of tasks should be executed in co–located processors. Less frequently, communicating tasks may be distributed geographically. It is contemplated to have a controller task for each set of co–located processors. The translation of distributed software is envisaged to produce one controller task for each set of co–located processors. Communication between distinct co–located processes will be mediated by their controllers. Use of Ada–9x will facilitate exchange of messages in a geographically distributed environment.

6.2 Modifying the Ada Software For Independence of the Hardware

In reengineering concurrent software it is necessary to consider the possibility that the programmer of the source software has relied on knowledge of relative computation delays and omitted some synchronization statements based on knowledge of the speed of the processors and the scheduling discipline of the Operating System used at that time. Therefore, some synchronization statements may not have been made explicitly in the software. Such explicit synchronization commands must then be added to the software in order to attain logic–wise independence of the speed of the processors used.

Finding and inserting missing synchronizations requires the availability of specifications and an understanding of the software. We envisage employing an interactive man–machine procedure. The user may gain knowledge of the software requirements by examining the display of the graphic model of the software. In particular the data flow graph may be useful (It was described in Section 3.). It shows the flow of data and control between hierarchical software units, such as tasks, procedures, or functions, as well as the shared global data structures, files, inputs, and outputs. The data flow graph will then show in detail the interactions of tasks with specific global data structures. The graphs will be examined in detail. The user will be able to check the existence, or lack, of the needed synchronizing messages to assure mutual exclusion. Missing synchronization commands can then be inserted in the graphic model of the software.

6.3 Determining Required Processor and Communication Capacity For Satisfying Real–Time Deadlines.

The objective is to determine the required capacities of network computing and communication resources that must be allocated to the concurrent software in order to meet the real–time deadline requirements. The proposed approach consists of a simulation of the schedule based on the model of the software [3,10].

The timing requirements are typically not specified in the source code. They must be obtained by the human user of the software reengineering facility from the software specification. An interactive procedure is envisaged here as well. The user will utilize the display of the data flow diagram, on a selected level of detail. The timing information may be added to the graph by identifying paths from starting to ending nodes and the required maximum delay. The data flow graph, with the timing requirements, is then used as input to a simulation system which computes the delay in a path and compares it with the required maximum/minimum delays. The computation may be only an approximation if it is based on the execution times of Ada instructions. The simulation must be augmented with testing.

The simulation may also be used iteratively in searching systematically for near optimal allocation of computing resources and communications. It may also find possibilities of bottlenecks [3].

7. References

1. Chang, S., "Visual Languages and Visual Programming," Plenum Press, 1990.

2. Chen, P., "The Entity–Relationship Model: Toward A Unified View of Data," ACM Trans. on Database Systems, May 1976.

3. Computer Command and Control Company, "Software Engineering Environment for Executing Parallel/Concurrent Programs on a Computer Network," Contract No. N60921–89–C–0127, Philadelphia, PA., Nov. 1990.

4. Computer Command and Control Company, "Software Intensive Systems Reverse Engineering," Final Report, Contract No. N60921–90–C–0298, April 1992.
   Also project memoranda: Memo 2—Elementary Statement Language Internal Representation, June 29, 1992. Memo 3—CMS–2 to ESL Translation, January 30, 1992.

5. Computer Command and Control Company, "Software Specification Assistant" : Status Manager and Step–by–Step Guide, Document Manager Guide, Evaluation Guide and Installation Guide, Contract N00014–91–C–0160, December 1992.

6. Computer Command and Control Company, "Reengineering Concurrent Software into Ada," Technical Report Contract N60921–92–C–0196, December 1993.

7. Cvetanovic, Z., "The Effects of Problem Partitioning, Allocation and Granularity on the Performance of Multiple–Processor Systems," IEEE Transactions on Computers, Vol. C–36, No. 4, April 1987.

8. Digital Equipment Corporation, "DECdesign: User's Guide," AA–PABRB–TE, Maynard, MA, May 1991.

9. DOD, "Defense System Software Development," DOD–STD–2167A, September 1988.

10. DOD, "Military Standard Software Development and Documentation (Draft)" DOD Harmonization Working Group, December 1992.

11. Lock, E., Prywes, N. and Andrews, S. "Case For Development And Re–engineering Of Real–time Distributed Applications," Fourth International Conference, Software Engineering and Its Applications, Toulouse, France, December 9–13, 1991.

12. Naval Sea Systems Command, PMS 412, User Handbook for CMS–2 Compiler, NAVSEA 0967–LP–598–8020, 30 March 1990.

13. Naval Sea Systems Command, PMS 412, Program Performance Specification for CMS–2 Compiler, NAVSEA 0967–LP–598–9020 30, March 1990.

14. Prywes, N., Liu, W. and Ge, X. "Software Reengineering Environment," 3rd Reverse Engineering Forum, Sept. 1992.

15. Prywes, N., Ingargiola, G. and Ahrens, J. "Automatic Reverse Engineering of Software to Confirm/Update Requirements Specification," Computer Command and Control Company, Contract No. N00014–92–C–0242, Philadelphia, PA, 19103, June 1993.

16. Prywes, N., "Software Restructuring," Computer Command and Control Company, Philadelphia, PA 19103, July 1993.

17. Prywes, N., Lee, I. "Integration of Software Specification, Reuse and Reengineering," Computer Command and Control Company, Contract No. N60921–92–C–0194, Philadelphia, PA, 19103, June 1993.

APPENDIX

PRODUCER-CONSUMER C/UNIX EXAMPLE EXPLANATION

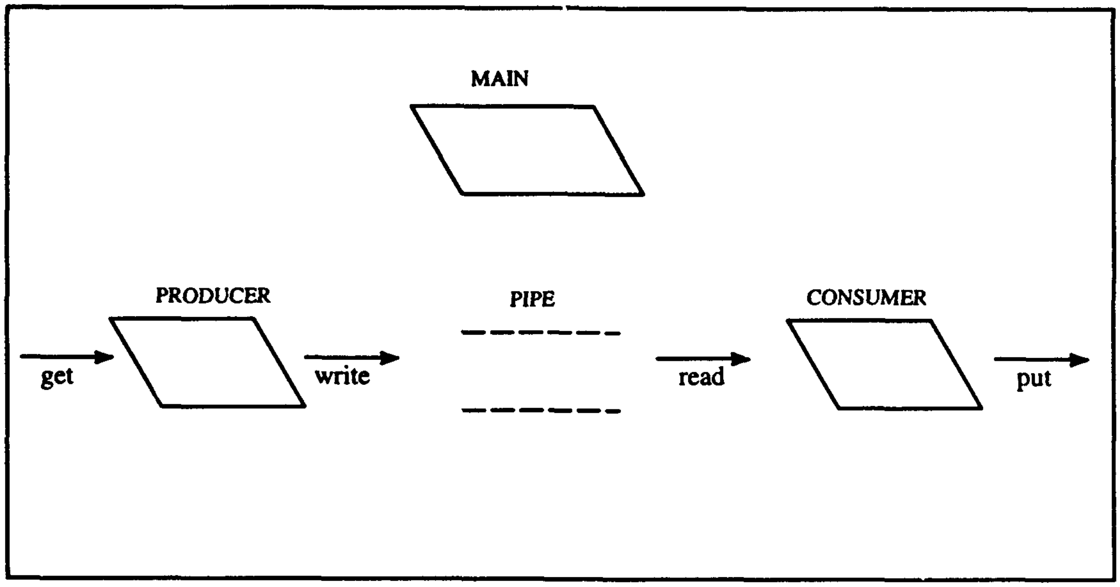
This is a C/Unix program example for the Producer/Consumer problem. This program consists of three entities: main, producer, and consumer. At runtime, there are three processes running concurrently as follows:

18. A process executing the original main program that creates a PIPE, and two child processes.

19. A process executing a copy of the main program in which a PRODUCER function is being called.

20. 3) A process executing a copy of the main program in which a CONSUMER function is being called.

The functions for these processes are called MAIN, PRODUCER, and CONSUMER processes. The runtime structure of these entities and their relationships are are shown as below. It includes PIPE and READ/WRITE operations on it.



In the following, each function and its respective operations are described.

1. MAIN : It creates a PIPE and processes that execute PRODUCER and CONSUMER functions. It passes a WRITE pointer to PIPE to PRODUCER and a READ pointer to CONSUMER for communication of message between PRODUCER and CONSUMER. Once both processes are active, it waits for termination of both processes. After termination, it closes the PIPE and terminates its execution.

2. PRODUCER: It is executed in a copy of the main() function. In execution, a pointer to a PIPE is passed as a parameter to write messages. It writes on the PIPE a message which has been gotten from the

input, in each iteration of a loop. This loop continues until FLAG condition is not met.

3. CONSUMER: It is executed in a copy of main() function. In execution, a pointer to a PIPE is passed as a parameter to read messages. It reads from a PIPE a message which is to be written to the output, in each iteration of a loop. This loop continues until FLAG condition is not met.

4. PIPE : It is an IPC mechanism used in UNIX OS. This is one-way communication mechanism, with two pointers for READ and WRITE. It is created by MAIN, and is passed to PRODUCER and CONSUMER for communication between them.

These entities are described by comments in the following C/Unix

PRODUCER–CONSUMER C/UNIX EXAMPLE CODE

```
void PRODUCER(BUF)                            /* CONSUMER FUNCTION                  */
int BUF;                                      /* WRITE PIPE                         */
{
           char MSG[1];                       /* Message to be written on PIPE      */
int  FLAG = 1;                                /* Flag for wile iteration            */

           while (FLAG)                                   /* Repeat until false     */
{
               gets(MSG);                                 /* Input from Standard    */
               write(BUF,MSG,sizeof(MSG));    /* Write Msg on PIPE                   */
               /* reset FLAG */                           /* Reset flag             */
    }
    exit( );                                  /* Exit from execution                */
}

void CONSUMER(BUF)                            /* PRODUCER FUNCTION                  */
int BUF;                                      /* READ PIPE                          */
{
           char MSG[1];                       /* Message to be read from PIPE       */
           int  FLAG = 1;                     /* Flag for while iteration           */

           while (FLAG)                       /* Repeat until false                 */
{
               read(BUF,MSG,sizeof(MSG));                 /* Read MSG from PIPE     */
               puts(MSG);                                 /* Output to Standard     */
               /* reset FLAG */                           /* Reset flag             */
           }
    exit( );                                  /* Exit from execution                */
}

main( )                                       /* MAIN FUNCTION                      */
{
           int BUF[2];                        /* READ/WRITE file descriptor of PIPE */

           pipe(BUF);                                     /* Open a PIPE            */

           if (!fork( ))                                  /* Create a child process */
           {
               CONSUMER(BUF[0]);                                      /*calling CONSUMER  */
           }
```

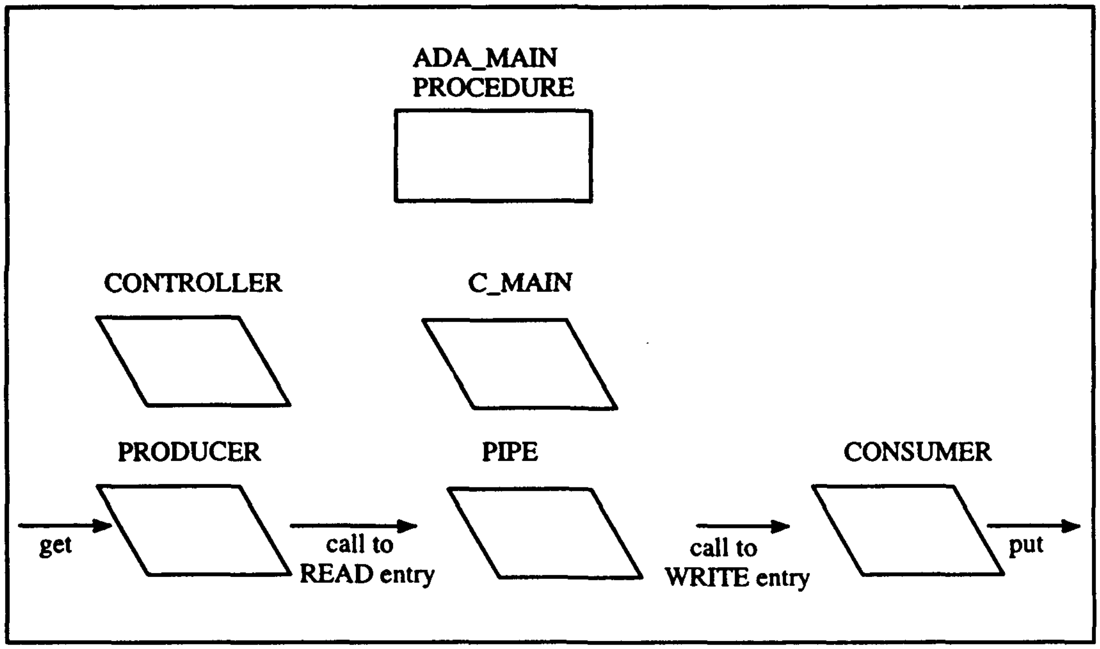
```
        if (!fork( ))                              /* Create a child process */
        {
           PRODUCER(BUF[1]);                                 /* calling PRODUCER */
        }

        wait( );                                   /* Wait for a child terminated  */
        wait( );                                   /* Wait for a child terminated  */
        close(BUF);                                /* Close a PIPE            */
}
```

STRUCTURE OF TRANSLATED ADA PROGRAM

The source software is translated into a procedure. One of the reason for this is to modularize each respective source software. This increases readability and understandability of the generated Ada target code.

The structure of the translated Ada code is shown below Figure. It consists of tasks: a controller tasks with mail box, functional tasks with mail boxes, and a pipe.



Note that the reason for C_MAIN to be instantiated as a task is due to UNIX semantics that MAIN is a process.

TRANSLATED ADA PROGRAM

```
with LIBRARY_PACK; use  LIBRARY_PACK;
with TEXT_IO;      use TEXT_IO;
with Unchecked_Conversion;

procedure ADA_MAIN is

   --<Other Ada Code for Controller and Mail Box Tasks>

 ------------------------------------------------------------------
 --                         PRODUCER
 ------------------------------------------------------------------
```

```
procedure  Producer(CtnlMBoxPtr   : in MAIL_BOX_P;
                      FTMBoxPtr : in MAIL_BOX_P;
                      ArgList   : in MESSAGE_TAIL_T)
is
      -- Local Variables:
      MSG          : CHARACTER;
      FLAG         : INTEGER := 1;

      -- Conversion Dependent Local Variables:
      LocalArgList : PRODUCER_ARG_T:=Byte_Str_T_2_Producer_Arg_T(ArgList);
      ByteStream   : MESSAGE_TAIL_T;
      N            : INTEGER := MSG'SIZE;
      RValue       : INTEGER;
begin

      while (FLAG = 1)
         loop

            -- For demonstration:
            put("Enter a character for PRODUCER(To terminate, enter 'T'):");

            get(MSG);

            -- [C/UNIX] write(BUF,MSG,sizeof(MSG));
            -------------------------------------------------------------
            ByteStream := MSG_2_Byte_Str_T(MSG);
               UNIX_Write(FTMBoxPtr,LocalArgList.BUF,ByteStream,N,RValue);
            -------------------------------------------------------------
               --| The 'write' OS call is translated into a call to a procedure
               --| 'UNIX_Write' defined in the library. This procedure
               --| generates a DATA message to a PIPE mail box by calling an
               --| WRITE entry. In writing, it waits for an acknowledgement from
               --| the PIPE based on blocking information. In return, it receives
               --| a number of bytes written on the PIPE.

               -- /* reset FLAG */
end loop;

-- [C/UNIX] exit( );

  -------------------------------------------------------------------
  --                  UNIX_Exit(CtnlMBoxPtr,FTMBoxPtr);
  -------------------------------------------------------------------

       --| The 'exit' OS call is translated into a call to
       --| 'UNIX_Exit' procedure defined in the library. This
       --| procedure generates a CONTROL message for this OS call and
       --| passes it to Controller. when Controller receives this message,
       --| it calls a procedure 'SYS_EXIT_Routine' procedure defined
       --| in the library. This procedure will perform the protocol of
       --| 'exit' OS call in the source software OS.

end Producer;


  -------------------------------------------------------------------
  --                    CONSUMER
  -------------------------------------------------------------------

    procedure  Consumer(CtnlMBoxPtr : in MAIL_BOX_P;
                       FTMBoxPtr      : in MAIL_BOX_P;
                       ArgList        : in MESSAGE_TAIL_T)
```

```
   is
      -- Local Variables:
      MSG          : CHARACTER;
      FLAG         : INTEGER := 1;

      -- Conversion Dependent Local Variables:
      LocalArgList : CONSUMER_ARG_T:=Byte_Str_T_2_Consumer_Arg_T(ArgList);
      ByteStream   : MESSAGE_TAIL_T;
      N            : INTEGER := MSG'SIZE;
      RValue       : INTEGER;
   begin

      while (FLAG = 1)
         loop

            -- [C/UNIX] read(BUF,MSG,sizeof(MSG));

  ------------------------------------------------------------------------
— UNIX_Read(FTMBoxPtr,LocalArgList.BUF,ByteStream,N,RValue);
—      MSG := Byte_Str_T_2_MSG(ByteStream);
  ------------------------------------------------------------------------

               --| The "read" OS call is translated into a call to a procedure
               --| "UNIX_Read" defined in the library. This procedure
               --| gets a DATA message from a PIPE mail box by calling an READ
               --| entry. In reading, it checks for the right message by the
               --| size and type of message.

               put(MSG);

               -- /* reset FLAG */
      end loop;

     -- [C/UNIX] exit( );

  ------------------------------------------------------------------------
—                     UNIX_Exit(CtnlMBoxPtr,FTMBoxPtr);
  ------------------------------------------------------------------------



         --| The "exit" OS call is translated into a call to
         --| "UNIX_Exit" procedure defined in the library. This
         --| procedure generates a CONTROL message for this OS call and
         --| passes it to Controller. when Controller receives this message,
         --| it calls a procedure "SYS_EXIT_Routine" procedure defined
         --| in the library. This procedure will perform the protocol of
         --| "exit" OS call in the source software OS.

   end Consumer;

  ------------------------------------------------------------------------
 —                                       C_MAIN
  ------------------------------------------------------------------------

   procedure  C_Main(CtnlMBoxPtr : in MAIL_BOX_P;
                     FTMBoxPtr   : in MAIL_BOX_P;
                     ArgList     : in MESSAGE_TAIL_T)
   is
      -- Local Variables:
```

```
      BUF          : MAIL_BOX_P;

      -- Conversion Dependent Local Variables:
      ProducerArg  : PRODUCER_ARG_T;
      ConsumerArg  : CONSUMER_ARG_T;
      ByteStream   : MESSAGE_TAIL_T;
      FTTskPtr     : STANDARD_TASK_P;
      RValue       : INTEGER;
      Name         : NAME_T;

      Consumer_P   : STANDARD_TASK_P;
      Producer_P   : STANDARD_TASK_P;

      -- Instantiation of CONSUMER package:
      package CONSUMER_PACK is new FUNCTIONAL_TASK_PACK
           (Task_Body_Procedure => Consumer);

      -- Instantiation of PRODUCER package:
      package PRODUCER_PACK is new FUNCTIONAL_TASK_PACK
           (Task_Body_Procedure => Producer);
   begin

      -- pipe(BUF);
      -----------------------------------------------------------------
      -- UNIX_Pipe(CtnlMBoxPtr,FTMBoxPtr,BUF,RValue);
      -----------------------------------------------------------------
      --| The "pipe" OS call is translated into an instantiation of
      --| mail box. In instantiation, it makes a call to
      --| "UNIX_Pipe" procedure defined in the library to
      --| inform Controller about this new mail box task named PIPE.
      --| This procedure generates a CONTROL message for new task and
      --| passes it to Controller. when Controller receives this message,
      --| it calls a procedure "SYS_PIPE_Routine" procedure defined
      --| in the library. This procedure creates a new TCB for this
      --| mail box.

            --      [C/UNIX]      if      (!fork())      {      PRODUCER(BUF[1]);      }
-----------------------------------------------------------------


   ProducerArg.BUF := BUF;
   ByteStream      := Producer_Arg_T_2_Byte_Str_T(ProducerArg);
   Name            := Get_Task_Name;

   Producer_P := PRODUCER_PACK.Generate_Functional_Task(Name,FTMBoxPtr,
                 CtnlMBoxPtr,ByteStream);
      -----------------------------------------------------------------
      --| The "fork" OS call is translated into a call to a function in
      --| PRODUCER_PACK package, which is an instantiation of
      --| package called "FUNCTIONAL_TASK_PACK". This package is defined
      --| in the library. This instantiation requires the body of
      --| PRODUCER procedure to be passed as a parameter. A call to a
      --| "Generate_Functional_Task" function in this instantiated
      --| package requires a set of proper parameters to be passed.
      --| These parameters are used to   generate a desired functional task.
      --| These are 1) name of child task being created, 2) a pointer to
      --| this task creating a new child task, 3) a pointer to the CONTROLLER
      --| mail box, and finally 4) a list of parameters for this PRODUCER
      --| procedure in a stream of bytes. In instantiation, this newly
      --| created child task generates its own mail box, informs a controller
      --| about its own task and mail box, and finally calls the body
      --| procedure of its own. When Controller receives messages about these
      --| new tasks, calls "NEW_TCB_Routine" procedure, in library, to create
--| a new TCB for this functional task and mail. This TCB is used
--| to inform parent task about its child task.


-- [C/UNIX] if (!fork( )) {CONSUMER(BUF[0]);}
-------------------------------------------------------------------
Name            := Get_Task_Name;
ConsumerArg.BUF := BUF;
ByteStream := Consumer_Arg_T_2_Byte_Str_T(ConsumerArg);

Consumer_P := CONSUMER_PACK.Generate_Functional_Task(Name,FTMBoxPtr,
              CtnlMBoxPtr,ByteStream);
-------------------------------------------------------------------
--| The 'fork' OS call is translated into a call to a function in
--| CONSUMER_PACK package, which is an instantiation of
--| package called 'FUNCTIONAL_TASK_PACK'. This package is defined
--| in the library. This instantiation requires the body of
--| CONSUMER procedure to be passed as a parameter. A call to a
--| 'Generate_Functional_Task' function in this instantiated
--| package requires a set of propcr parameters to be passed.
--| These parameters are used to   generate a desired functional task.
--| These are 1) name of child task being created, 2) a pointer to
--| this task creating a new child task, 3) a pointer to the CONTROLLER
--| mail box, and finally 4) a list of parameters for this PRODUCER
--| procedure in a stream of bytes. In instantiation, this newly
--| created child task generates its own mail box, informs a controller
--| about its own task and mail box, and finally calls the body
--| procedure of its own. When Controller receives messages about these
--| new tasks, calls 'NEW_TCB_Routine' procedure, in library, to create
--| a new TCB for this functional task and mail. This TCB is used
--| to inform parent task about its child task.

-- [C/UNIX] wait( );
-------------------------------------------------------------------
UNIX_Wait(CtnlMBoxPtr,FTMBoxPtr,NULL,RValue);
-------------------------------------------------------------------
--| The 'wait' OS call is translated into a call to
--| 'UNIX_Wait' procedure defined in the library. This
--| procedure generates a CONTROL message for this OS call and
--| passes it to Controller. when Controller receives this message,
--| it calls a procedure 'SYS_WAIT_Routine' procedure defined
--| in the library. This procedure will perform the protocol of
--| 'wait' OS call in the source software OS.

-- [C/UNIX] wait();
-------------------------------------------------------------------
UNIX_Wait(CtnlMBoxPtr,FTMBoxPtr,NULL,RValue);
-------------------------------------------------------------------
--| The 'wait' OS call is translated into a call to
--| 'UNIX_Wait' procedure defined in the library. This
--| procedure generates a CONTROL message for this OS call and
--| passes it to Controller. when Controller receives this message,
--| it calls a procedure 'SYS_WAIT_Routine' procedure defined
--| in the library. This procedure will perform the protocol of
--| 'wait' OS call in the source software OS.

-- [C/UNIX] close(BUF[1]);
-------------------------------------------------------------------
UNIX_Close(CtnlMBoxPtr,FTMBoxPtr,BUF,WRITE_PIPE,RValue);
-------------------------------------------------------------------
--| The 'close' OS call is translated into a call to 'UNIX_Close'
--| procedure defined in the library. This procedure generates a
--| CONTROL message for this OS call and  passes it to Controller.
```

```
      --| when Controller receives this message, it calls a procedure
      --| 'SYS_CLOSE_Routine' procedure defined in the library. This
      --| procedure will perform the protocol of 'close' OS call in the
      --| source software OS.

      -- [C/UNIX] close(BUF[0]);
      -------------------------------------------------------------------
      UNIX_Close(CtnlMBoxPtr,FTMBoxPtr,BUF,READ_PIPE,RValue);
      -------------------------------------------------------------------
      --| The 'close' OS call is translated into a call to 'UNIX_Close'
      --| procedure defined in the library. This procedure generates a
      --| CONTROL message for this OS call and  passes it to Controller.
      --| when Controller receives this message, it calls a procedure
      --| 'SYS_CLOSE_Routine' procedure defined in the library. This
      --| procedure will perform the protocol of 'close' OS call in the
      --| source software OS.

   end C_Main;

   -- Instantiation of C_MAIN package:
   ----------------------------------------------------------------------
   package C_MAIN_PACK is new FUNCTIONAL_TASK_PACK
                                       (Task_Body_Procedure => C_MAIN);

begin
   -- ADA_MAIN declares C_MAIN as a FUNCTIONAL TASK by calling a
   -- Generate_Functional_Task functional defined in C_MAIN_PACK
   -- package.
   declare
      Name       : NAME_T;
      ByteStream : MESSAGE_TAIL_T;
      MainArg    : MAIN_ARG_T;
      C_Main_P   : STANDARD_TASK_P;
   begin
      Name       := Get_Task_Name;
      ByteStream := Main_Arg_T_2_Byte_Str_T(MainArg);
      C_Main_P   := C_MAIN_PACK.Generate_Functional_Task(Name,
                    ControllerMailBoxPtr,ControllerMailBoxPtr,ByteStream);
   end;
   NULL;
end ADA_MAIN;
```

Software Migration and Reengineering (SMR)
A Pilot Project in Reengineering

Stephen R. Mackey[1] and Lynn M. Meredith[2]

[1]Microelectronics and Computer Technology Corp. (MCC)
Advanced Systems & Networks
3500 West Balcones Center Drive
Austin, TX 78759-5398
mackey@mcc.com

[2]Computing Devices International
U.S. Operations
8800 Queen Avenue South
Bloomington, MN 55431-1996
Lynn.M.Meredith@cdev.com

Abstract

This paper describes the Software Migration and Reengineering (SMR) project being sponsored by the Embedded Computing Institute. The project is nearing the end of the first phase of a four phase effort. The purpose of the initial phase was to define the technical approach, based on the expectations of the customer, investigation of previous approaches, and the availability of existing tools. The paper discusses information uncovered and describes the technical approach that was developed based on this information. The unique contributions of this project are expected to be in the areas of: recovery of design information from CMS-2M and AYK-14 assembler; investigation of representations for the desired state of the system; interactive capture, display, and modification of the recovered information by the user; and integration of the SMR toolset with the F/A-18 Software Engineering Environment via population of a mainstream, commercially available CASE tool.

1. Introduction

The United States military is the world power of today; however, post-Cold War realities have shifted American policy from one of military buildup to one of extreme downsizing. Part of the originally successful buildup included the development and acquisition of highly capable weapon systems. Although the current downsizing effort may save financial resources, it decreases or halts weapon system procurement, thus promoting a possible drop in America's technological competitive edge. The U.S. military seeks to solve this dilemma by increasing current weapon system capabilities with key technological advances.

A specific example of this investment is the Navy's F/A-18 platform. While the operational capability of this particular weapon system was being fine-tuned to meet the expectations of those whose lives depend upon it, other technological advances have occurred. Some of these technological advances (e.g., improved sensors, multisensor fusion) will translate directly into increased operational capability, while others (e.g., advanced software techniques, Ada, 32-bit microprocessors) will translate indirectly into increased operational capability because they contribute to lessening maintenance expenditures. A key issue is how to take advantage of these technological advances, maintain the current operational capability, and prepare to deliver enhanced operational capabilities into the next decade.

Recognizing these advances, the U.S. Navy seeks to upgrade avionics systems from existing hardware platforms to those that better support the Ada language. Because existing software is exhibiting limitations with respect to performance, maintainability, portability, and scalability, there is a need for effective and thorough software migration and re-engineering mechanisms. With the current emphasis on cost and risk management, the solution to this problem must also support an incremental approach.

The Software Migration and Reengineering (SMR) project was established by the Embedded Computing Institute (ECI) located at the Naval Air Weapons Center (NAWC) in China Lake, California. It is a cooperative research and development effort involving Computing Devices International, Microelectronics and Computer Technology Corporation (MCC), and the Embedded Computing Institute.

1.1. Current System State

This subsection discusses the current state of the F/A-18 system. The major points are: system level, including sensors, buses and mission computers; software, including a brief introduction into its structure; documentation, highlighting differences between existing documentation and that required by recent development standards; and maintenance concerns, including the existence of several versions of the F/A-18 operational software.

The entire F/A-18 system is composed of a number of subsystems, including a variety of sensors and two mission computers. These computers are interconnected via several MIL-STD-1553 buses. The SMR project is focusing on the reengineering and migration of the software residing in the mission computers; therefore, there will be no further discussion on the sensors.

The current mission computers are reaching the limits of their capacity, and the need to employ multiple MIL-STD-1553 buses is an indication that they too may reaching their capacity limits. However, it is important to recognize that redundant computers and busses may be present because of survivability requirements (e.g., fault tolerance). Fault tolerance may be utilized to compensate for hardware failures, or battle damage.

The existing software is implemented in AYK-14 assembly language with some CMS-2M. The current operational software possesses existing system properties (as defined in [2]): functionality, performance, and accuracy. We consider these properties as fundamental properties, which must be preserved by any reengineering approach. These properties are a result of:

- Many years of engineering effort.
- Extensive interaction with the end users; that is, pilots of the F/A-18.
- Use of the system in the actual deployed environment; for example, Desert Storm.

The existing "architecture" of the software is dominated by global data (expressed as CMS-2M SYS-DDs), and functionally oriented modules (expressed as CMS-2M subroutines). Although the software is laid out in terms of CMS-2M constructs, almost all of the actual code is implemented in AYK-14 assembler using the direct features of CMS-2M. When an assembler language module wishes to manipulate some global data, the references are hard coded in the assembler modules.

This type of architecture, in addition to being difficult to understand, makes seemingly simple software changes extremely difficult. For example, changes in the representation of a particular data item can require changes in the code of modules throughout the system. Another example is changes in computation may also require modifications to several modules. It is important to recognize that this style of architecture is typical of legacy systems, and was required to achieve the performance that was required by the system.

It is unlikely that all of the kinds of information that accompany a deployed system can be extracted from source code alone. The next logical step is to look for documentation of the existing system. The existing documentation for the F/A-18 is a mixture of MIL-STD-1679 documents, and documents created by the F/A-18 community to meet specific operational needs. The existing documentation is for the most part complete and consistent; however, there are no formal software requirements or software design documents in MIL-STD-1679.

Last it should be noted that the operational software for the F/A-18 has been in existence for a number of years. As is normal for deployed software with a long lifetime, several versions of the operational programs are in existence and must be maintained. Sources of variation include:

- Differences in aircraft configurations.
- Differences in mission requirements.
- Different customers.

As is evident from the preceeding discussion, the "current state" of the existing system is conceptually very large, extremely complicated, more than just source code, and some of the representations are not amenable to automated understanding.

1.2. Desired System State

It is difficult to describe what the desired system state should be with a high level of confidence. Many factors contribute to this difficulty. For example:

- What point in time should the desired state be achieved?
- What constitutes an acceptable desired state?
- Is it standards based?

Instead of a definitive answer regarding the structure of the desired state one is usually presented with "requirements" like the following:

- Retarget the software from 16-bit target hardware to 32- or 64-bit target hardware.
- Move that software to a more modern language (i.e., Ada).
- Enable more effective software maintenance.
- Lower cost.

An important part of any systems reengineering effort is to create a definition of the desired state that meets these kinds of requirements. The concept is illustrated in Figure 1, and the remainder of this section discusses several, but probably not all, completed or ongoing industry efforts that had an influence on the concept. This concept has been extremely useful in guiding the search for widely accepted approaches to more detailed definition of the desired system state.

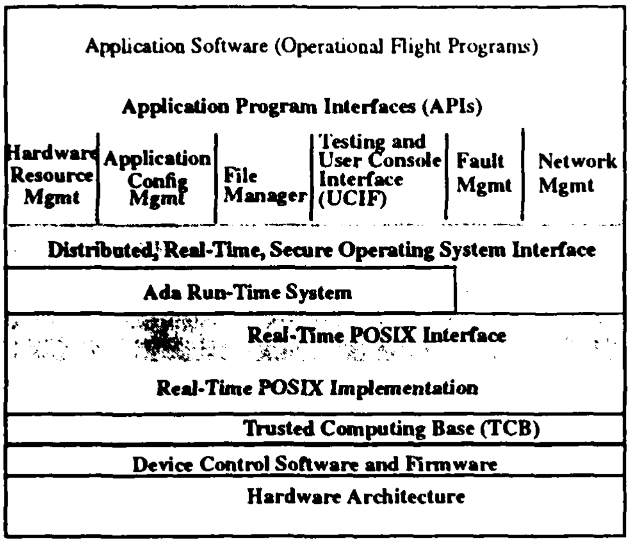

Application Software (Operational Flight Programs)
Application Program Interfaces (APIs)
Hardware Resource Mgmt
Application Config Mgmt
File Manager
Testing and User Console Interface (UCIF)
Fault Mgmt
Network Mgmt
Distributed, Real-Time, Secure Operating System Interface
Ada Run-Time System
Real-Time POSIX Interface
Real-Time POSIX Implementation
Trusted Computing Base (TCB)
Device Control Software and Firmware
Hardware Architecture

Figure 1. A Layered View of the Desired State

The software engineering community at large has identified other desirable properties of a software system. These properties, which are also nicely summarized in [2], reflect software engineering principles, facil'tate continuous change, and attempt to capture existing system know how.

Examples of these continuous change properties are: modularity, standard interfaces, virtual machine abstractions, localization of information about the application domain and implementation technology, parameterization, use of Commercial Off the Shelf (COTS), and reuse.

Examples of software engineering properties are: portability, structure, readability, testability, data independence, documented system understanding, openness, interoperability, and seamless integration.

Examples of existing system know-how properties are: domain models, domain-independent software architecture principles, domain-specific architectures, and adaptable components.

Many of these principles are well understood by some software engineers. Several observations can be made about these properties. First, the properties are not mutually exclusive. For example, modularity, virtual machine abstractions, localization of information, structure, and domain independent software architecture principles all hint at the same idea. Second, some of the properties are not the result of a programming language per se. For example, use of high level languages makes software more portable and more readable than the use of assembly languages. Third, some of these properties may be contradictory to the fundamental properties. For instance, abstractions may degrade performance. Finally, some of these properties may be in conflict with each other. One example of potential conflict is the ability to achieve seamless integration while employing COTS products or reusing existing software. Based on the foregoing discussion, it is clear that these principles need more explicit definition.

Fortunately, there are a number of recently completed or currently underway efforts which address some of these issues. In the spirit of reuse, we are attempting to utilize the results of these efforts where feasible. Examples of these efforts are:

- The Navy's Next Generation Computing Resources (NGCR) Operating System Interface Specification [27]. The OSSWG has adopted the IEEE's Portable Operating System Interface Specification (POSIX) [22-26] as the operating system interface specification.
- The Domain Specific Software Architecture (DSSA) program [5].
- The Ada 9X effort [18].
- The SEI's Rate Monotonic Analysis (RMA) and Distributed Real-Time System Design Efforts [16, 17].
- The SEI's Generic Avionics System Specification effort [15].
- Software requirements and design methods like the Ada-based Design Approach for Real-Time Systems (ADARTS) [8].

Each of these efforts contribute some insight into what characteristics a system and software architecture should possess. A more detailed description of the use of these efforts by the SMR project will be presented in the methods definition section of the paper.

Although not directly applicable to the architecture of the target software (i.e., the operational flight programs), substantial effort has been expended in the Software Engineering Environment (SEE) area. This work is important because it addresses requirements and design processes, and the production of DoD-STD-2167A system and software documentation. For example, the DoD-STD-2167A Software Requirements Specification, and Software Design Documents are targeted by specific tools in a SEE. Some of the more notable SEE efforts include:

- The Joint Integrated Avionics Working Group (JIAWG) SEE, as exemplified by the F-22 program.
- The Software Technology for Adaptable, Reliable Systems (STARS) program.

- The Navy's Next Generation Computer Resources (NGCR) Program Support Environment Specification Working Group (PSESWG).
- The National Institute of Science and Technology (NIST) environment work.

These efforts have been defining reference models and standards in the SEE area. The SMR project will utilize these reference models and standards as integration requirements. The potential utilization of these efforts will be discussed in the process definition, and tool selection and creation sections of the paper.

1.3. Derived Requirements

The previous two subsections have briefly discussed the current and desired states of the F/A-18 system. In addition to this more technically oriented desirable properties view, there is also a programmatic and acquisition oriented view. This view demands, because of the need to continue to serve the fleet and because of declining budgets, that the transition from the current system state to the desired system state be evolutionary.

Evolution of the current state to the desired state will be a multi-year effort, with several incremental improvements to the current state. Evolution will occur in two areas:

- Product evolution. That is, the transition of the operational software from the current state to the desired state.
- Process evolution. That is, the transition of the procedures, methods, and tools used to create and maintain the operational system.

In the very near term, the perceived risk in converting any part of the OFP to Ada is very high. This is due to a number of factors. First, because it is too expensive to convert the entire OFP to Ada, only a part of it will be converted. The fact that only a part of the OFP will be converted means that the structure of the existing OFP must be very well understood so that the interfaces between the "old" code and the "new" code can be clearly specified designed, and implemented. Second, this initial conversion effort also involves moving from a 16-bit computer to a 32-bit computer.

To mitigate this programmatic risk, the SMR toolset will be utilized to aid in understanding and identifying a subset of the F-18 avionics Operational Flight Program (OFP) to be modified and adapted. Part of the OFP will continue to execute on an Advanced AYK-14 computer containing a VHSIC Processor Module (VPM), and part of the OFP will execute on an Intel 80960-based processor module. To ensure effective operational test and evaluation, all functional and performance testing will be conducted in the NAWC simulation facility.

In addition to migrating the software, requirements in the process also include the use of new development standards (e.g., DoD-STD-2167A), more advanced software techniques, and a more advanced SEE. These requirements mean that the information recovered by the SMR toolset must be consistent with the information requirements of the chosen techniques, and that the SMR toolset must be integrated with the chosen toolset.

In the medium term, two to five years, there is a desire to integrate new operational capabilities into the F/A-18, and to employ more COTS technologies. Both of these require thorough understanding of the existing capabilities, and clear specification, design and implementation of interfaces between the existing capabilities and the new capabilities.

The long term objective is to provide a comprehensive set of tools for orderly incremental and semi-automated migration of 16-bit CMS-2 and AYK-14 assembly operational flight programs (OFP) to a 32-bit RISC architecture with the Ada language. The SMR project will integrate existing applicable process, methods, and tools while developing missing elements for providing the additional capabilities and uniting the disparate pieces together.

Meeting the near, medium, and long term objectives and providing an integrated solution requires the SMR project to address the following topics:

- Business case - investigate the cost-effectiveness of re-engineering existing software versus new software development.
- Processes - analyze the procedures that concern not only migration and re-engineering but new functionality additions, overall system architecture modifications, and interaction with systems engineering, test, and software reuse.
- Methods - consider existing software understanding, inter-language translation issues, current design representation, new analysis and design model usage, and reuse.
- Tools - analyze software programs with capabilities of reverse engineering, design viewing, design re-engineering, and forward engineering.

The remaining sections of the paper address these four topics.

2. Business Case

As discussed in [1], there is a reengineering decision making process, which consists of four steps: 1) identify alternative strategies, 2) perform an economic assessment of the alternatives, 3) select the preferred strategy, and 4) implement the selected strategy.

2.1. Identification of Alternative Strategies

Several different reengineering "strategies" are identified by [1]. The strategies, along with their definitions, are:

- Restructuring. The engineering process of transforming the system from one representation form to another at the same relative abstraction level, while preserving the subject system's external functional behavior.
- Redocumentation. The process of analyzing the system to produce support documentation in various forms including users manuals, and reformatting the systems' source code listings.
- Reformat. The engineering process of reformatting the existing source code so that it is easier to understand and maintain, while preserving the subject system's external functional behavior.
- Reverse engineering. The engineering process of understanding, analyzing, and abstracting the system to a new form at a higher abstraction level.
- Forward engineering. Forward engineering is the set of engineering activities that consume the products and artifacts derived from legacy software and new requirements to produce a target system.
- Redevelop. Redevelopment refers to developing new software from scratch ignoring any existing software.

As part of the reengineering strategy selection process, a number of questions concerning product complexity, environmental risk, and system lifetime are used as a part of the decision making process. As one of the initial tasks in our reengineering effort, we applied this process and derived the following metrics:

- Average Product Complexity Value = medium (the actual score was 2.20).
- Environmental Risk = medium (the actual score was 2.05).
- Remaining System Life = Long.

Table 1 is the matrix from [1]. The Y axis of this table is product complexity, and the X axis is environmental risk. The highlighted area of the table shows the reengineering strategies suggested by the selection procedure.

Restructure or Redocument	Reverse then Forward	Redevelop	Redevelop
Restructure or Redocument	Restructure	Restructure	Reverse then Forward
Reformat	Reformat	Restructure	Restructure
Leave Alone	Leave Alone	Leave Alone	Restructure

Table 1. Strategy Selection Matrix (Long Lifetime Remaining)

Although, by strict application of this procedure, restructuring was identified as the advised strategy, it was rejected because of the desire to transition to Ada. Restructuring was viewed as the restructuring of the existing CMS-2 and AYK-14 assembler implementation.

In addition, NAWC's desire is to possess the current OFP software, currently written in CMS-2/AYK-14 assembler, rewritten in the Ada language. This conflicts with the accepted restructuring approach of being in the same relative abstraction level; both would be in software languages but of very disparate types.

The two chosen alternative strategies are reverse then forward, and redevelop.

2.2. Economic Assessment of the Alternatives

A very high level of economic analysis was performed using the COCOMO model with the 1988 Ada process model which is consistent with DoD-STD-2167A. A COCOMO model, with the size of the system estimated to be 50K of newly developed Ada software, to represent development of the system from scratch—this matches the "redevelop" strategy identified above. The effort multipliers were set according to the characteristics of the product (high reliability and complexity), people (high application experience and analyst capability), and process (high modern programming practices and tools). This baseline model yielded a schedule duration of 16.5 months, effort of 150.6 staff months, and productivity of 331.9 SLOCs per staff month.

In order to calculate the potential savings from a reverse then forward strategy, some assumptions about the effectiveness of the reverse engineering toolset. The assumptions were as follows:

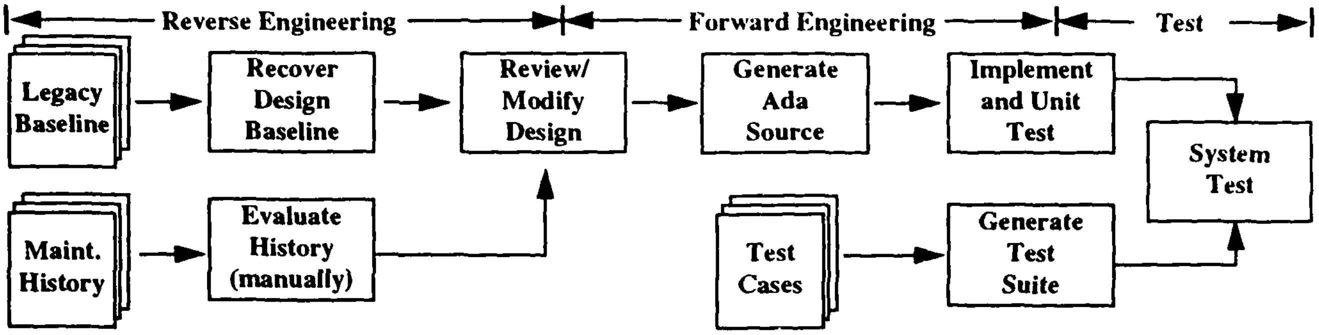


Figure 2. SMR Process Definition

- High Level Design Reuse on a scale of 0.0 (no reuse) to 1.0 (full reuse). Since the current design is based on functional decomposition, and an object based design is desired we set this factor to 0.25.
- Software Salvage on a scale of 0.0 (no salvage) to 1.0 (full salvage). This factor is an indication of the amount of low level design and existing code that can be recovered. This factor was set to 0.50.
- Translation Efficiency on a scale of 0.0 (no translation) to 1.0 (full translation). We arbitrarily set this to 0.75.

These assumptions lead to an *Adaptation Adjustment* Factor (AAF) equal to 0.704875. The original SLOC estimate of 50K lines is multiplied by the AAF and results in a SLOC estimate of 35244 which is fed back into the baseline COCOMO model. This COCOMO model yielded a schedule duration of 14.4 months, effort of 101.2 staff months, and productivity of 348.3 SLOCS per staff month. Since savings are usually measured in effort saved we have 150.6 - 101.2 = 49.4 staff months saved.

2.3. Select Preferred Strategy

We believe that the strategy followed will be a reverse then forward strategy. Even if a redevelop strategy was envisioned, the development staff will naturally go back to the existing system, and reverse engineer some if not most of the functionality. Without the SMR toolset this reverse engineering activity would be mostly manual.

2.4. Implement the Selected Strategy

Implementation of the reverse then forward strategy is very much like a new development supplemented by the reengineering environment. This includes integration of reengineering processes with the "normal" developmental processes; integration of reengineering methods with the chosen development methods; and integration of reengineering tools with normal developmental tools. The key is letting the people control the application of the processes, methods, and tools.

3. Process Definition

As with any other software system, an accurate process must be defined. As illustrated in Figure 2, SMR defines a well-structured process for migrating CMS-2/AYK-14 assembler to the Ada language. The following subsections describe the essential procedures for the SMR process definition and additional procedures that SMR takes into consideration.

3.1. Migration and Reengineering Procedures

Software migration provides a means for moving functionality representation from one software programming language to another. Re-engineering leverages the value of pre-existing system software assets to renovate or reconstitute it in a new form, while retaining and possibly augmenting its essential functionality.

Re-engineering encompasses certain aspects of both reverse and forward engineering. The reverse engineering process analyzes an existing system for generating its higher level abstractions [6]. The forward engineering process moves a high-level conceptual abstraction of a system to its logical design and final physical implementation (i.e., code).

The initial step identifies the collection of software that the software migration and re-engineering process will be implemented upon. Once identified, the next step involves parsing the identified software for recovering the baseline design into an alternate representation in a repository. Applying a level of language specific heuristics during parsing provides the ability for interpreting the entire set of code. Once in this alternate representation, the recovered information is better suited for supporting other migration and re-engineering procedures.

Because the SMR project does not want to conflict with the steps identified in DOD-STD-2167A, careful consideration involved this critical reverse engineering step. The project decided that reverse engineering would only go to the preliminary design level. This decision was primarily based upon project life and resources and system/software requirements recovery complexity.

The next step provides viewing and modification facilities of the recovered design. Static and dynamic analysis assists in determining the various components and their interrelationships. This captured information can be viewed through higher-level artifacts such as control-flow diagrams, module structures, program invocation graphs, stratified views, and variable access structures. Different perspectives of the recovered information facilitates the understanding of the system, its various components, and their interaction. Applying domain knowledge, external information, documentation, deduction, or fuzzy reasoning techniques to the recovered design information assists the user in attaining a better understanding of the system to be re-engineered.

While applying appropriate transformations the recovered design information can be augmented by re-design, structuring, modularization, and grouping. A level of semi-automation accelerates the re-engineering activity. At this time, available maintenance histories can be evaluated and applied to the re-engineered design. Design re-engineering provides the means for taking advantage of the target language constructs during code generation. Once the re-engineering of the design is complete, the Ada language representation is generated.

The code generation procedure must not only create fully compilable Ada code, it must also adhere to the current MIL-STD-1815A for the Ada programming language.

The next step implements the generated code through compilation, linking, and run-time execution facilities. The implementation will be performed hand-in-hand with unit test, the first step for testing DOD-STD-2167A software systems. As a parallel activity, test case generation allows for subsequent system level testing of the migrated and re-engineered application.

3.2. Overall System Architecture Modifications

Some programming languages require specific hardware considerations before implementation, especially assembly type languages. This dependency could have directed the original design and implementation of the application. Therefore, migration of applications from one programming language to another must consider the overall system architecture.

A function of the design recovery step will aid the identification of these system architecture dependencies. By recovering to an informative design state, the recovered articles are viewed for their original implementation and then re-engineered to the desired design state. This design state re-engineering allows for modification of the overall system architecture.

3.3. System Engineering Interaction

Any system must contain a definition of requirements and design. The logical steps for performing software development implements the DOD-STD-2167A methodology include system/segment requirements, system/segment design, software requirements, and software design. When integrating reverse engineering with forward engineering, the process must identify the logical insertion point into the DOD-STD-2167A methodology. In addition, successful reverse engineering promotes scalability with system engineering processes.

The migration and re-engineering process implements steps for reverse engineering of an assembly-based application, re-engineering of the recovered baseline design, and forward engineering for implementation onto a new hardware architecture. The current focus of the SMR project performs reverse engineering to an abstracted design state. This abstracted design state relates to the preliminary software design for DOD-STD-2167A software development. The next logical reverse engineering steps recovers software requirement and then system requirement/design information.

3.4. Maintenance Processes

Ideally, a software system will have a long lifetime, maximizing the benefit of the initial development. Redesigns, upgrades, and extensions require knowledge of the implemented system. Usually, this knowledge can be lost when the original implementors are no longer available. Therefore, maintenance should always be a contributing factor when defining a software system process.

The SMR project realized the necessity of a strong maintenance process because of the underlying purpose of reverse engineering a legacy system. SMR's identified process provides an increase of future maintenance efficiency by providing a scalable and adaptable design repository. Current trends in software engineering practices for software system development show programming evolving to designing. Once the design is complete and stored in a repository, a code generator can be utilized for the target software application, reducing the burden upon specific programming language knowledge.

3.5. Testing

Another integral part of a software migration and re-engineering process includes testing. Testing provides the feedback for securing the integrity of the developed software system. As described in section 5, the SMR toolset provides reverse engineering while the F/A-18 SEE provides forward engineering and testing. Therefore, the SMR toolset will not have any impact on the normal unit and system testing procedures.

3.6. Documentation

A comprehensive process must provide a means for sufficiently documenting the migrated and re-engineered software system. At each step of the SMR process, a path provides production of the necessary documentation. Good documentation aids in knowledge retention and future maintenance.

3.7. Reuse

The defined SMR process will hopefully further promote the notion of reuse. By including an object-oriented design repository, the SMR process promotes reuse of software design objects. In addition, creating the target application with the Ada language supports reuse of its software modules.

4. Methods Definition

Figure 3 depicts a highly simplified view of a re-engineering example based on the SMR problem. At this level, reengineering seems deceptively simple. It seems as though it should be possible to write computer programs that can understand the existing F-18 Operational Flight Program(s), which are written in a mixture of CMS-2 and AYK-14 assembly language, and generate some intermediate representation. From this intermediate representation one could then forward engineer to Ada source code. However, very few, if any, high-level design decisions can be captured by examining existing source code. It seems that this is true, independent of the implementation language.

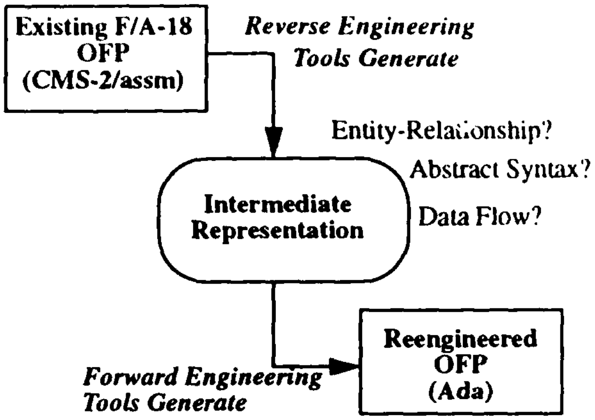

Existing F/A-18 OFP (CMS-2/assm)
Reverse Engineering Tools Generate
Entity-Relationship?
Abstract Syntax?
Data Flow?
Intermediate Representation
Forward Engineering Tools Generate
Reengineered OFP (Ada)

Figure 3. High-Level View of Reengineering

Although we may expect a significant amount of the intermediate representation can be automatically generated by the reverse engineering tools, we must recognize that there are key pieces of information that will be manually entered into the intermediate representation by the people working on the SMR project. Given the limited financial resources of the SMR project, it is not our desire to create a new intermediate representation. For at least these reasons, the choice of the intermediate representation must leverage mainstream commercial Computer Aided Systems or Software Engineering (CASSE) tools.

4.1. Software Understanding

In order to achieve the maximum "return on investment" from the existing system, there are a number of things we wish to abstract from the existing software. Among them are:

- Data items.
- Subroutines.
- Data reference patterns.
- Processes.

One of the first tasks in understanding the existing system is to identify what the major data items are; that is, what are the major variables. Initially, it is only important to discover the names of the variables and their relative size. At a later time—when more is understood about how the variables are used —decisions can be made as how to represent them in Ada; that is, what type to use.

Another task is to identify the subroutines that exist in the current implementation.

With these two pieces of information, an important ingredient is to identify what the reference patterns of subroutines to data items are. The importance of this type of information has been cited several times in the literature [4, 12], and appears to be important regardless of source and target language choices. This information is important for at least two reasons.

First, from the point of view of constructing an "object-based," or "object-oriented" design[1] it is important to understand what subroutines use what data and what kind of use it is (i.e., read or write). This will promote a modular design approach based on information hiding [14] concepts, rather than functional decomposition.

The second, more detailed, tidbit of understanding this enables has to do with how the subroutines use the data structures. It is hypothesized that this type of understanding will support the selection of types (e.g., records, arrays), and perhaps some representation issues (e.g., memory layout) as well.

Another piece of information which is important to know in order to understand is the existence of "processes." Many systems, both real-time and non real-time, are measuring physical parameters. Examples of this occur when the software is implementing things like differential equations, integrals, or kalman filters. These types of mathematically oriented applications have time-dependent, cyclic behaviors that are extremely important to recognize in order to more fully understand the existing software.

There may be several other more hardware oriented pieces of information that need to be recognized in order to create a more robust approach to software understanding. An example of this is addresses that correspond to memory-mapped I/O. The approach used for software understanding will need to remain flexible enough to incorporate these pieces of information.

4.2. Current Design Representation Issues

One of the more difficult choices is how to represent the information extracted from the current state. As indicated in Figure 3, the information could be represented as entity-relationship diagrams, abstract syntax trees, data flow diagrams, and so forth.

Many different approaches to the intermediate representation have been tried [10,11]. As we move to representations that attempt to capture more of the semantic content of the existing state; for example, problem domain oriented concepts, there seems to be a need to move towards knowledge-based representations. One the key ingredients in representation is flexibility.

4.3. Inter-Language Translation Issues

One of the principle arguments against inter-language translation approaches is that they do not take advantage of the power of the target language. Because the target language is Ada, the features discussed in the following paragraphs possess the "power" that the SMR design recovery approach will take advantage of:

Packages are one of the most powerful features of the Ada language, because they enforce many of the software engineering principles cited in section 1.3. Examples of the principles packages directly support are: modularity, information hiding, and standard interfaces.

Use of packages coupled with the use of strong typing will increase the reliability and readability of the operational software. In particular, private and limited private types will be key to the SMR design recovery approach. Effective use of these Ada features will facilitate data independence and localization of information about the implementation technology. Since implementation technology is changing rapidly, localization of information pertaining to the current implementation technology will be a key factor in adhering to the continuous change property.

Ada tasks are also an important feature since they allow the expression of concurrency. Concurrency in real-time systems has been a major source of concern over the past few years. The SMR design recovery approach will utilize Ada tasking and rate monotonic analysis concepts in order to facilitate recovery of the concurrent aspects of the existing design.

Finally, if the new software is expected to be reused in other applications, Ada generics may be an important feature to utilize. Some recent work has cited generics as being key to reusable Ada software.

4.4. New Analysis and Design Model Usage

When performing a transformation from a source form to some target form, it is important to understand both forms. Sections 4.1 and 4.2 dealt with the definition of the source. This section deals with the definition of the target form.

It is important to remember that in addition to supporting the reengineering and migration of the F/A-18 software from CMS-2M/AYK-14 assembly to Ada upgrading the documentation to DoD-STD-2167A is also desirable.

[1] It is not our desire to debate the merits of object-based versus object-oriented design. The purpose of this phrase is to let the reader know that we understand that there is a subtle difference between the two, and that Ada 83 is more object-based than it is object-oriented.

Fortunately, substantial work has already been done in this area.

Analysis and design methods and notations have been in existence for many years. Within the last ten years or so, many CASE tools have emerged to assist in the implementation of these methods and creation of the associated notations. There are a number of SEE efforts that are trying to create integrated collections of these tools.

A prime example of these efforts is the F-22 SEE. Although its definition may not be entirely complete, several leading CASE tools focusing on analysis and design are a part of that SEE. Examples include: Ascent Logic's RDD-100; I-Logix' Statemate; Cadre's Teamwork; IDE's Software through Pictures; and SES/Workbench.

Personnel from the SMR project have investigated each of these tools, either through previous experience or through current work at their respective organizations. Because of customer preferences, and the desire to reengineer the existing software to an Ada-based design representation, we have decided to concentrate on Cadre's Teamwork/Ada product.

Each CASE tool implementation is trying to support the automation of one (or more) technical methodologies. For example, there are different versions of Cadre's Teamwork. Teamwork/SART supports the Hatley/Pirbhai methodology [19], Teamwork/OOA supports for Object–Oriented Analysis [20], and Teamwork/Ada supports the Buhr notation [21]. There is also a version of Teamwork that supports the ADARTS methodology [8], which basically integrates concepts from Hatley/Pirbhai and Buhr (unfortunately we have been unable to evaluate this product).

Each methodology has a preferred style of representing information about requirements, design, or both. The methodology usually specifies a number of graphical symbols, the problem domain concept that the symbol stands for, what other information may be associated with each symbol, relationships among symbols, and rules for constructing representations that contain many symbols.

For example, the Hatley/Pirbhai methodology uses data and control flow diagrams as the primary representations, and includes state transition diagrams. Shlaer/Mellor uses a form of entity-relationship diagrams, supported by data and control flow diagrams.

The goal for SMR is to perform design recovery back to one of these representations. It may be possible to recover to several of these representations; however, due to funding limitations, the project team decided upon one option, the Buhr notation. This notation was chosen because of the inherent Ada structure graph qualities that the target software programming language.

5. Tool Selection and Creation

As with any other software system, the SMR project needs computer models for supporting the defined process and methods. Through software tools, the computer models provide the semi-automated means for assisting the user through the defined process. The following sub-sections provide a high-level description of the F/A-18 SEE and SMR's relationship with it, describe the process for tool selection, identify the additional software necessary, and present the SMR toolset solution.

5.1. F/A-18 Software Engineering Environment

The F/A-18 Software Engineering Environment (SEE) contains a collection of software engineering tools for system analysis and design, software analysis and design, code development, compiling, debugging, configuration management, documentation, requirements traceability, and project management. The F/A-18 SEE provides the foundation for future F/A-18 new software development. While being comprehensive for forward engineering (i.e., systems analysis through code generation and testing), the SEE lacks reverse engineering for information capture of existing applications.

The SMR project solves the reverse engineering dilemma by integrating the SMR toolset to the F/A-18 SEE. SMR provides the reverse engineering and design reengineering capabilities for existing CMS-2/AYK-14 assembly based applications. The solution interfaces the SMR design representation to the SEE equivalent, Cadre Teamwork/Ada. Cadre Teamwork/Ada is a design level software tool represented with an Ada structure graph notation. The SMR toolset provides a natural enhancement to the F/A-18 SEE.

As seen in figure 4, the F/A-18 SEE provides a reference model defining a foundational platform for presentation, process, control and data elements. The platform allows software tools to be configured and integrated within the SEE. The SMR toolset fills another slot in the reference model.

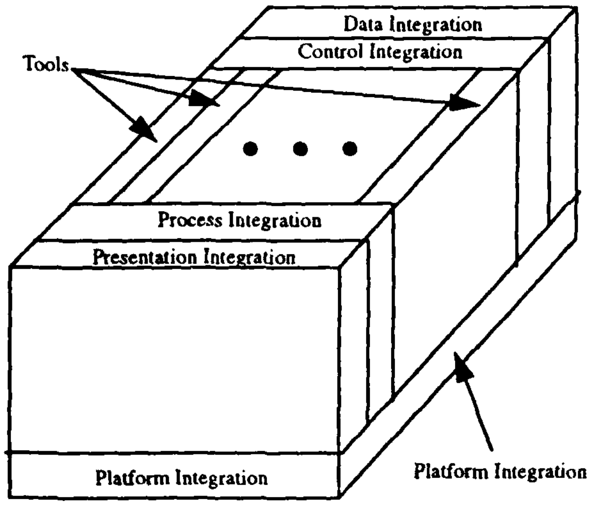


Figure 4. SEE Reference Model

5.2. Investigation of COTS Tools [28]

Software engineering tools provide an expanding variety of capabilities for automating manual software engineering process and methods. These software tools have evolved from prototypes to powerful, fully supported commercial tools for small and large scale development.In addition, tool vendors have modified their attitudes from single point solutions to integrated toolsets. Commercial tools are increasingly addressing previous gaps in coverage, especially those in program understanding and reverse engineering.

Collectively, over a hundred software companies produce hundreds of various software tools that relate to most of the SMR domain. A rigorous evaluation of currently available software tools is therefore required to identify those most applicable and effective for the SMR project. In addition, the evaluation survey assesses the current technology state, uncovers emerging standards, and provides proper information for proposing the solution.

In order to decrease the amount of evaluation time and increase evaluation efficiency, the SMR project defined an evaluation survey process:

1. Collect vendor/tool prospects - sources include inhouse knowledge, trade magazines, marketing advertisements, STSC list, Ada Letters, and CASE Product Guide,
2. Contact vendors - this step obtains the technical information needed for product evaluation,
3. Evaluate product information (first phase) - first phase evaluation provides SMR relevancy information in order to down-select,
4. Generate first-pass filter criteria - basic SMR scope level including programming languages (i.e. CMS-2 or Ada), reverse engineering, design viewing, design re-engineering, and forward engineering,
5. Perform more in-depth technical evaluation - hands-on usage, in-depth discussions with vendor's technical department, collaborative discussions,
6. Generate second-pass filter criteria - interfaces to other tools, tool scalability, platform and environment, collaboration openness, industry standards conformity, and
7. Propose tool(s) for SMR solution.

After the first two steps, it was apparent that much research and development in the software migration and re-engineering field has resulted in many supportive software tools. In any event, 87 software tools were identified that conceptually relate to the SMR project. Further investigation showed many of these tools aid in migrating and re-engineering software written in FORTRAN, COBOL, C, and Pascal to other languages. A few CMS-2 to Ada translators exist, but none are capable of handling the assembly language portions of the system.

After steps three and four, 29 tools did not require further evaluation due to the vendor's market focus (i.e., MVS/Cobol business application market). Another 40 tools were marked for contingency evaluation, such as a broadening in SMR scope or necessity for acquiring additional technical approach and knowledge. A group of 18 tools or toolsets were nominated for further evaluation. The following is a listing of those tools nominated for more in-depth technical evaluation:

Advanced Systems Technology Corp. - Astec's 2167A Tool Set is an extension to the CaMERA's semantic database repository capabilities for producing DOD-STD-2167A documentation.

Ascent Logic - Ascent Logic's RDD-100 tool provides full requirements analysis and traceability, behavioral modeling and simulation, and component allocation.

CACI - CACI's GenEleC generates complete reusable Ada packages from CASE component specification via data element specifications.

Cadre - Cadre's CASE tools contain the Teamwork series, providing graphical representation of software systems for analysis, design, and code generation. Cadre promotes the CDIF data repository protocol which allows interfacing to other CDIF compliant software tools.

Carleton University - TimeBench is a CAD tool for the design of real-time systems and supports an extended version of Buhr's "MachineCharts" diagramming notation.

DDC-I - DDC-I provides a CASE tool box for software development life-cycle support from requirements definition to maintenance.

Hewlett-Packard - HP's SoftBench3.0 framework is a tool integration platform which combines SoftBench program construction tools, encapsulated third-party development tools, and one's own custom development tools.

Hill AFB Software Technology Support Center - The STSC has the JOVIAL Re-engineering Tool Set (JRETS), which automatically analyzes JOVIAL source code and extracts design information.

i-Logix - i-Logix 's CASE tool, Statemate, is a systems engineering tool permitting the graphic modeling and the design of complex, reactive systems.

Interactive Development Environments (IDE) - IDE's CASE tools include Software through Pictures (StP), providing graphical representation of software systems for analysis and design. Integrated with IDE's Ada Development Environment, the user can generate Ada code.

McCabe & Associates - McCabe has the Battlemap Analysis Tool (BAT), a software reverse engineering and maintenance tool which analyzes system level source code and calculates McCabe complexity metrics.

Mark V Systems - Mark V Systems' CASE tools include ObjectMaker and Adagen. ObjectMaker is a user-tailorable CASE workbench, while Adagen provides the Ada OO component with Ada code generation.

MCC - Previous consortial research at MCC generated design information recovery (DESIRE) technology. DESIRE contains software for scanning and parsing programming languages, intermediate object base representation, graphical entity relationship modeling, design viewing, and object-oriented design representations.

Naval Surface Warfare Center - The NSWC has On-Line Tools (OLTools), a collection of report generation utilities for program designs involving CMS-2 source files, Target System Files, and AEGIS SYSBLD/7 specification files, amongst others.

Oregon State University (OSU) - OSU has a parser as a front-end to a compiler for CMS-2 and AYK-14 assembler code.

Purdue University - The Purdue Compiler-Construction Tool Set (PCCTS) provides a set of tools for constructing compilers and code-to-code translators.

Quantasm Corp. - Quantasm has the ASMFlow Professional, an assembly language flow charting and source code analysis tool for the IBM PC.

Reasoning Systems - Reasoning Systems supports and licenses two customizable software and maintenance and re-engineering tools, Software Refinery and REFINE. Software Refinery allows for building automated software processing tools, while REFINE provides source code analysis and re-engineering.

Performing steps five and six allowed the SMR project team to concentrate on three tools: Cadre Teamwork/Ada, MCC DESIRE, and OSU parser. In addition to these tools, other tool finalists include IDE Software through Pictures and Reasoning Systems Software Refinery.

5.3. The Integrated Toolset

Although COTS tools were selected for the SMR toolset, they do not provide a 100% solution to the process definition. In addition, the tools are not currently compatible and need to be integrated together. The SMR project intends to leverage previous research on software engi-

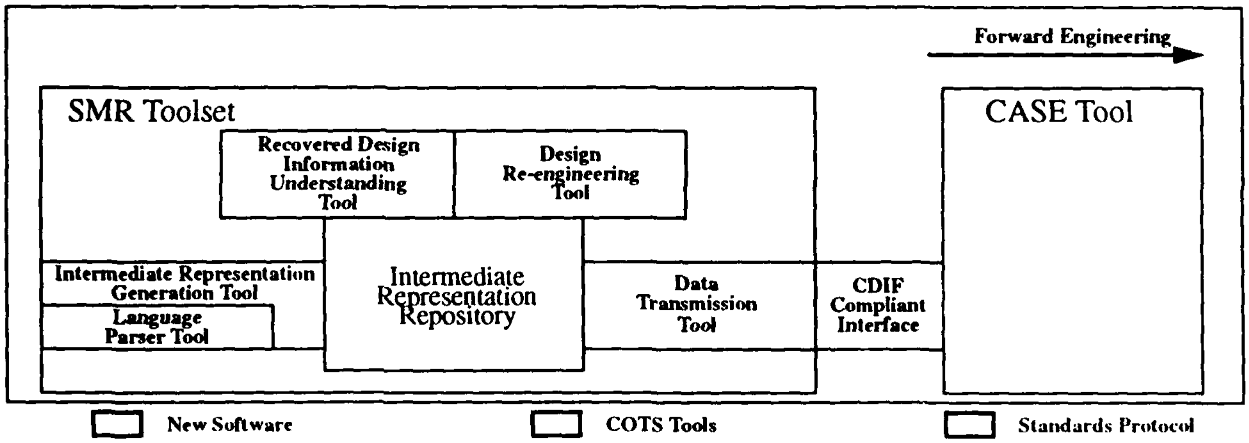

Forward Engineering
SMR Toolset
Recovered Design Information Understanding Tool
Design Re-engineering Tool
Intermediate Representation Generation Tool
Language Parser Tool
Intermediate Representation Repository
Data Transmission Tool
CDIF Compliant Interface
CASE Tool
New Software
COTS Tools
Standards Protocol

Figure 5. Integrated SMR Toolset

neering processes and methodologies with the three COTS tools to provide an integrated toolset.

The SMR toolset, illustrated in Figure 5, integrates currently available software and COTS tools with project developed software. The developed software includes intermediate representation generation, recovered design information understanding, design re-engineering (including object clustering), intermediate representation to CASE tool transmission, and interactive graphical user interface software.

Since the OSU parser is a front-end to a compiler, the software does not have capability for stand-alone execution. Therefore, the parser will be integrated with Intermediate Representation Generation Tool software that loads the recovered information into the Intermediate Representation Repository (IRR). The IRR represents the recovered information in an object base format and organized to reflect the lexical structure of the program source [29].

For information retrieval, the SMR project creates and exercises a Recovered Design Information Understanding Tool, software for understanding the recovered design information. This tool interprets the IRR and then generates graphical views and slices for program and design understanding.

Since the recovered information will not be conducive to object-oriented or Ada design notations, the SMR project provides a Design Re-engineering Tool. This tool, in conjunction with the understanding tool, provides a semi-automated method for clustering recovered information into the appropriate design notation. For example, the user would have the ability to select variables or program modules and associate them as objects. In addition, this tool would permit a functional re-design cf the recovered information.

The last tool developed for the SMR project is a Data Transmission Tool, software that interprets the intermediate representation repository and exports data to CASE tool via a CDIF interface protocol. The CASE tool provides code generation as a part of the F/A-18 SEE and is compliant with CDIF.

The SMR toolset is exercised through a highly interactive graphical user interface (GUI). The X-window system and Motif graphical widget set's inherent object capabilities enriches the human interaction with the toolset.

6. Conclusion

The SMR Pilot Project contributes many benefits to the F/A-18 community and in the software engineering domain. By integrating the SMR toolset into the F/A-18 SEE, the software development branch receives a cohesive solution. This integrated solution creates a mechanism to capture and take advantage of pre-existing engineering awareness and domain expertise for the current version of the avionics software. Automating the capture of this information provides the means for an increase in efficiency in migrating future avionics software.

In addition, the SMR project advances research in the software migration and re-engineering domain. Since much legacy code is assembly based, these legacy system owners and maintainers gain an automated means for capturing program knowledge. Interfacing the design information recovery with forward engineering offers a method for migrating legacy software to modem and future languages.

7. References

1. Joint Logistics Commanders, Joint Policy Coordination Group on Computer Resources Management. *First Software Reengineering Workshop, Santa Barbara I, "Back to the Future Through Reengineering,"* 21 September - 25 September 1992.

2. Feiler, P.H., *Reengineering: An Engineering Problem* (CMU/SEI-93-SR-5). Pittsburgh, PA.: Software Engineering Institute, Carnegie Mellon University, July 1993.

3. Department of Defense, *Reengineering Economics Handbook* (MIL-STD-REH), Draft, March 1, 1993 (appears as an addendum to reference 1).

4. Wilde, N., *Understanding Program Dependencies* (SEI-CM-26). Pittsburgh, PA.: Software Engineering Institute, Carnegie Mellon University, August 1990.

5. Mettala, E. and Graham, M.H. (editors), *The Domain-Specific Software Architecture Program* (CMU/SEI-92-SR-9). Pittsburgh, PA.: Software Engineering Institute, Carnegie Mellon University, June 1992.

6. Chikofsky, E.J., and Cross II, J.H., "Reverse Engineering and Design Recovery: A Taxonomy," IEEE Software, January, 1990, pp. 13-17.

7. Harrison, W., Magel, K., Kluczny, R., and DeKock, A., "Applying Software Complexity Metrics to Program Maintenance," IEEE Software, September 1982, pp. 65-79.

8. Gomaa, Hassan. *Software Design Methods for Concurrent and Real-Time Systems*, Addison Wesley, 1993.

9. National Research Council, Computer Science and Technology Board. *Scaling Up: A Research Agenda for Software Engineering*. National Academy Press. Washington, D.C., 1989.

10. Biggerstaff, T.J., Hoskins, J., and Webster, D., *DESIRE: A System for Design Recovery*. Microelectronics and Computer Technology Corporation (MCC), Software Technol-

ogy Prog.am, MCC Technical Report STP-081-89, April 1989.

11. Pettengill, R.C., *DOOD: DESIRE Object-Oriented Design, Language Independent Object-Oriented Design Using GERM Views.* MCC Software Technology Program, MCC Technical Report STP-193-90, September 19, 1993.

12. Hutchens, D.H, Basili, V.R., "System Structure Analysis: Clustering with Data Bindings," IEEE Transactions on Software Engineering, Vol. 11, No. 8, August 1985, pp. 749-757.

13. Goguen, J.A., "Reusing and Interconnecting Software Components," IEEE Computer, February 1986, pp. 16-28.

14. Parnas, D.L., "On the Criteria to be Used in Decomposing Systems into Modules," Communications of the ACM, Vol. 15, 1972.

15. Locke, C.D., Vogel, D.R., Lucas, L., and Goodenough, J.B., *Generic Avionics Software Specification* (CMU/SEI-90-TR-8). Pittsburgh, PA.: Software Engineering Institute, Carnegie Mellon University, December 1990.

16. Sha, L. and Goodenough, J.B., *Real-Time Scheduling Theory and Ada* (CMU/SEI-89-TR-14). Pittsburgh, PA.: Software Engineering Institute, Carnegie Mellon University, April 1989.

17. Sha, L. and Sathaye, S.S, *Distributed Real-Time System Design: Theoretical Concepts and Applications* (CMU/SEI-93-TR-2). Pittsburgh, PA.: Software Engineering Institute, Carnegie Mellon University, March 1993.

18. Ada 9X Project Report (Draft), Volumes I and II, March 1992, Intermetrics, Inc.

19. Hatley, D.J. and Pirbhai, I.A. *Strategies for Real-Time System Specification.* Dorset House, New York, NY,1987.

20. Shlaer, S. and Mellor, S.J. *Object Lifecycles: Modeling the World in States.* Yourdon Press Computing Series, Englewood Cliffs, NJ, 1992.

21. Buhr, R.J.A. *System Design with Ada.* Prentice-Hall, Englewood Cliffs, NJ, 1984.

22. IEEE POSIX, *Draft Standard for Information Technology—Portable Operating System Interface (POSIX)—The Open Systems Environment.* P1003.0/D16, Institute of Electrical and Electronic Engineers (IEEE), August 1993.

23. IEEE POSIX, *Draft Standard for Information Technology—Portable Operating System Interface (POSIX)—Part 1: System Application Interface (API) Amendment 1: Real-time Extensions.* P1003.4/D14, Institute of Electrical and Electronic Engineers (IEEE), May 1993.

24. IEEE POSIX, *Draft Standard for Information Technology—POSIX Ada Language Interfaces—Part 2: Binding for Real-time Extensions,* P1003.20/D2, Institute of Electrical and Electronic Engineers (IEEE), April 1993.

25. IEEE POSIX, *Draft Standard for Information Technology—Standardized Application Environment Profile—POSIX Real-time Application Support (AEP).* P1003.13/D5, Institute of Electrical and Electronic Engineers (IEEE), February 1992.

26. Meyers, C.B., *Interface Requirements for Real-Time Distributed Systems Communications* (SEI) Working Paper, Version 5.0, September 7, 1993.

27. Juttlestad, D.P., *Operational Concept Document for the Next Generation Computer Resources (NGCR) Operating System Interface Baseline*, Naval Underwater Systems Center (NUSC) Technical Document 6998, 1 April 1991.

28. Mackey, S.R., Mitbander, B., Meredith, L.M., Software Migration and Re-engineering, An Evaluation Survey of Processes, Methods, and Tools, Technical Report ASN-SMR-119-93(P), Microelectronics and Computer Technology Corporation, November 4, 1993.

29. Pettengill, R. C, The DESIRE 2 Intermediate Data Base, Technical Report STP-MT-450-91(P), Microelectronics and Computer Technology Corporation, December 31, 1991.

Reverse Engineering Complex Databases to Support Data Fusion

R. D. Semmel[1] and R. P. Winkler[2]

[1] The Johns Hopkins University
Applied Physics Laboratory
Laurel, MD 20723
rds@aplcomm.jhuapl.edu

[2] U. S. Army Research Laboratory
Adelphi, MD 20783
winkler@adelphi-assb01.army.mil

Abstract

Large information systems often require the fusion of multiple databases to achieve desired functionality. In this paper, we focus on how automated query formulation capabilities may be realized over a set of fused databases. Reverse engineering issues related to database design and fusion are discussed, and a query formulation and design system known as QUICK is described. A case study is presented in which the logical schemas for two independent U. S. Army databases are reverse engineered into conceptual schemas that are subsequently used for data fusion and automatic query generation. In addition, enhanced methods that employ meta-level conceptual constructs to support reverse engineering, data fusion, and query formulation are described.

1. Introduction

In the course of reengineering a large information system, it is often necessary to fuse multiple databases to provide desired functionality. From a systems engineering perspective, database fusion should be modeled explicitly to facilitate information system development and maintenance. As a minimum, database logical schemas should be modified to ensure consistency among attributes and to foster interaction among participating databases. In many cases, however, modification of existing databases is not feasible (e.g., operational or legal constraints preclude database design modification). In such cases, participating databases must be associated in a nonintrusive manner. To a great extent, such association can be accomplished by reverse engineering logical schemas into conceptual schemas and relating participating databases via appropriate meta-level representation constructs.

Even when database fusion is realized, however, it is often difficult for users to formulate queries over participating databases. In particular, users must be aware of logical-level query languages as well as underlying logical structures and fusion links. For complex systems, queries may be elaborate, and may take many hours (and iterations) to write. Query formulation problems are exacerbated when the participating databases are large, and multiple logical models are employed. Yet, by using reverse-engineered conceptual schemas, it is possible to abstract user interaction from the logical level to the conceptual level. Furthermore, with appropriate meta-level abstractions, the need for conceptual-level knowledge can be significantly reduced.

In this paper, we focus on how queries can be automatically generated from high-level requests posed over a set of fused databases. In the next section, a system known as QUICK (for "QUICK is a Universal Interface with Conceptual Knowledge") is described. QUICK supports database reverse engineering and automated query formulation by modeling global logical schemas at the conceptual level. In Section 3, reverse engineering issues are discussed and the implications of data fusion are considered. In Section 4, relevant portions of two U. S. Army databases are described, and corresponding portions of the reverse-engineered conceptual schemas are presented. Then, in Section 5, specific data fusion issues regarding the U. S. Army databases are described. Finally, in Section 6, some enhancements are proposed to improve the performance of QUICK for complex systems reengineering and development.

2. Query Formulation and Contexts

Formulating queries over a complex database is a cognitive activity that requires extensive knowledge of the syntax of a database query language and the structure of the underlying database. While query language details can be masked by high-level interfaces, requirements for knowledge of the structural associations are not so easily hidden. For example, most graphical user interfaces require that users specify e^xplicit navigational paths and join criteria [4,22]. However, for complex databases, there may be many ways to associate data, and only some of those ways may be semantically meaningful. In addition, graphical interfaces become difficult to use with large databases that require many screens for representing high-level aggregate database objects; in such cases, a user must navigate among screens as well as database objects.

An alternative to graphical interfaces is the universal relation (UR) interface approach [10,12,20]. With a UR interface, a user is given the impression that all database attributes are represented in a single relation. Thus, querying requires only the specification of attributes and high-level constraints, and the interface is responsible for determining semantically reasonable associations and inferring appropriate joins.

Several approaches have been developed for developing UR interfaces [9,13]. One of the more popular approaches is based on the notion of maximal objects [11], where associations among attributes are inferred based on specified functional dependencies. While based on sound principles, there are several problems with the approach. First, it fails to use the constructs with which a database designer is most familiar. Thus, a separate knowledge base is required that is orthogonal to that used for design. Second, the exponential behavior of the approach does not scale well as system complexity increases: this is particularly disturbing in a data fusion environment where the participating databases may be large. Finally, because of its low level of abstraction, the approach does not lend itself to reverse engineering.

A more promising approach to constructing UR interfaces is based on using the knowledge captured in a high-level semantic data model. In the QUICK system, an Extended Entity-Relationship (EER) model is used [3,17,19] for formulating queries. In particular, QUICK segments the EER conceptual schema into overlapping subgraphs of strongly associated conceptual schema objects, and then selects appropriate subgraphs of the EER conceptual schema that are mapped to logical-level constructs.

By operating at the EER level, QUICK overcomes many of the limitations that hamper the maximal object approach. First, by using EER constructs, conceptual database design knowledge can be used directly. Second, though certain aspects of behavior may still be exponential, analysis of the conceptual schema is performed over aggregate objects and conceptual-level constraints are applied to reduce the time it takes to identify strongly associated sets of objects. Finally, the approach can be used effectively in a reverse engineering environment. In particular, additional EER constructs have been introduced in QUICK that facilitate reverse engineering a conceptual schema from a logical schema.

QUICK is able to generate queries efficiently by preprocessing an EER conceptual schema into maximal acyclic subgraphs of strongly associated objects. These maximal subgraphs are referred to as *contexts* [15,16]. The theory underlying contexts assumes that a single database is being modeled. Yet, it is often the case that information systems are composed from multiple databases. Thus, the notion of contexts must be extended to take into account possible data fusion requirements. In many cases (e.g., federated database systems [18]), it is not feasible to develop a unified global conceptual schema, as the individual databases continue to exist and be queried. Thus, context regeneration over a global schema may not be appropriate. In such cases, a designer will have to identify explicit gateways that can be used to fuse the participating databases and thus allow global querying. These gateways often can be identified when the participating logical schemas are reverse engineered into richer conceptual schemas.

3. Reverse Engineering from a Logical Schema

As information systems incorporate multiple databases, the need for effective data fusion becomes essential. However, fusing databases is fraught with problems relatec to disparate logical models, data redundancy, and potential inconsistencies. To facilitate communication among system designers, a high-level semantic data model [7] often is chosen into which logical models are reverse engineered. From a database perspective, the EER model is often the target model. The EER model is independent of a logical implementation and supports mappings to a variety of logical models (e.g., relational, network, hierarchical, and object-oriented). Moreover, when the reverse-engineered conceptual design is completed, it can be enriched with additional knowledge (e.g., constraints that can be used for automated query formulation).

Many techniques have been developed for reverse engineering a logical-level schema into an EER conceptual schema [2,5,8,14]. For example, Batini et al. [1] describe a process for converting a relational database schema that entails identifying primary relations (i.e., relations with primary keys that do not contain keys of other relations), weak primary relations (i.e., relations with primary keys that contain primary keys of other relations), and secondary relations (i.e., relations with primary keys that are the concatenation of primary keys of multiple relations). Once the relations are classified, they can be mapped into EER objects. For example, primary relations are mapped into entity types, weak primary relations are mapped into weak entity types, and secondary relations are mapped into relationship types. Then, the designer can make decisions regarding inheritance associations among created EER objects, and can further customize the design to satisfy additional system constraints that may exist.

Because of the sparse representation at the logical level, the process of reverse engineering from a logical schema into a conceptual schema cannot be fully automated. For example, the classification of relations may require renaming of candidate and foreign keys so that syntactic matching may be performed. Similarly, without explicit inclusion dependency constraints, it is not possible to identify inheritance lattices. Structural constraints on relationship types (e.g., cardinality ratio and participation constraints) can sometimes be inferred, but must often be provided by the designer. These problems are exacerbated if the logical-level design is not in an appropriate form (e.g., relations should be in at least third normal form), as it is difficult to infer nontrivial dependencies within an object.

When multiple databases must be fused, the reverse engineering process is further complicated. Issues regarding data redundancy and consistency across databases must be resolved. Furthermore, decisions must be made regarding changes to existing databases. In many cases, changes may be restricted due to operational constraints. In such cases, the reverse-engineered model must preserve the mapping to the logical level while representing at the conceptual level such logical-level limitations. From an automated query formulation perspective, this means that attribute disambiguation must occur across databases as well as within a single database.

QUICK provides substantial support for reverse engineering. For example, an optimization relationship type has been introduced that enables logical-level denormalization decisions to be represented at the conceptual level. In particular, a long path of EER objects may be circumvented by a short path that can produce the same result. Given a high-level request, the shorter path is used in lieu of the long path if all of the requested attributes are covered. On the other hand, if intermediate objects are needed, the long path is used. Note that if an optimization relationship type were modeled as a basic relationship type, then, syntactically, the conceptual schema would imply that distinct paths connected two or more entities. Consequently, given a request involving attributes of the connected entities, a query formulator might consider a union of subqueries to be appropriate or it might determine that interaction with the user is needed to refine the request before the final query can be generated. However, given the semantics of an optimization relationship type, the query formulator can infer that such actions would be superfluous.

QUICK also supports meta-level constructs. For example, a meta-generalization construct is provided that allows an entity type instance to be a child of multiple instances of a parent entity type. For example, a parent instance concerning some type of target may contain information uniquely identified by the key attributes TARGET_IDENTIFICATION, DATE, and TIME. In turn, some of those parent instances may be further described by a child instance that depends only on TARGET_IDENTIFICATION. Note that weak entity types could not be used in this case because the actual child would have to be considered a parent in the weak entity type association.

4. Description of U. S. Army Databases

The U. S. Army Research Laboratory (ARL) is directing a program designed to integrate heterogeneous databases so that high-level requests may be posed by users unfamiliar with the underlying conceptual or logical designs of the participating systems. In the final system, a wide variety of logical models will be employed (e.g., relational, inverted list, semantic network, and object-oriented). However, for illustrative purposes, only portions of a relational database and an inverted list database are described below. To further simplify discussion, only basic EER model constructs are used.

Figure 1 shows a portion (i.e., approximately 10 percent) of the EER conceptual schema for the ARL Electronic Intelligence (ARL-ELINT) relational database. To avoid cluttering the diagram, attributes are not shown, though it should be noted that both entity types and relationship types may have attributes. ARL-ELINT is used for gathering and correlating low-level electronic and imaging intelligence information (e.g. tracking a mobile radar unit via characteristics such as pulse width and operating frequency). High-level tactical information is not represented. As the ARL-ELINT logical schema was well-designed, it was straightforward to reverse engineer it into an EER conceptual schema. The process described in Section 3 was used, and only minimal changes to the logical schema were necessary to produce the schema shown in Figure 1.

Automating query formulation for ARL-ELINT is straightforward. First, contexts are produced, and then high-level requests are posed. In this particular case, the

EER conceptual schema is acyclic, and there exists only a single context. Thus, formulating a query entails pruning the leaves of the context (i.e., the conceptual schema shown in Figure 1) until all leaves cover requested attributes. Then, the resulting subgraph is mapped to a set of relational schemas, natural joins are inferred, and the final query is produced.

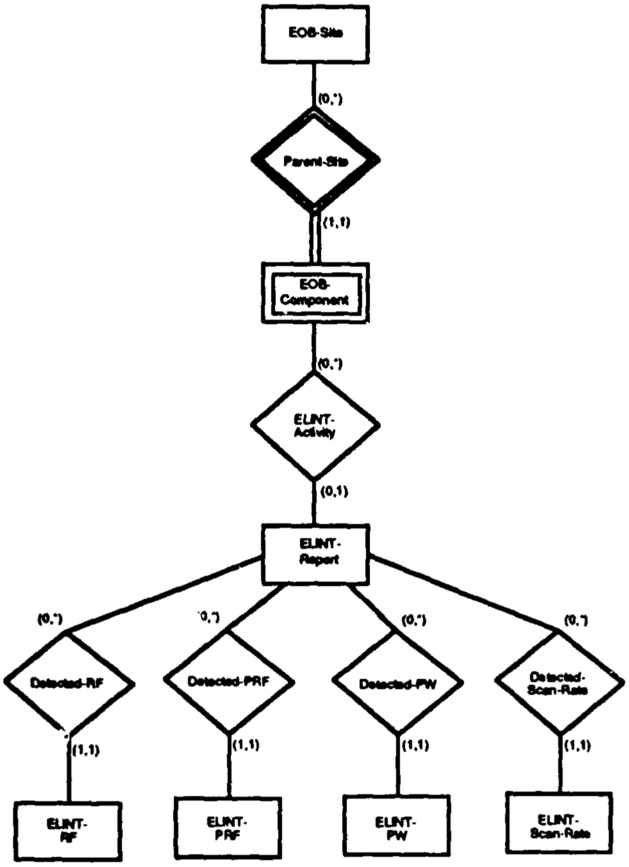


Figure 1. Portion of ARL-ELINT database.

As an example, suppose that a request were made to list the known operating characteristics (i.e., the pulse repetition frequency [PRF], the radio frequency [RF], and the pulse width [PW]) of electronic device ABC1234. Using QUICK directly (i.e., without a higher level interface), this request could be expressed as follows:

```
Select  prf,
        rf,
        pw
Where   elnot_id = "ABC1234"
```

The notation above is referred to as USQL because it resembles SQL, but assumes a universal relation view of the database (i.e., it is not necessary to qualify attributes with relation information, specify a FROM clause, or include natural join criteria).

Upon receiving the request, QUICK identifies the contexts in which the requested attributes reside. As Figure 1 contains a single context (i.e., the complete conceptual schema), QUICK prunes the context of extraneous leaves. In particular, the following EER objects are pruned: EOB-Site, Parent-Site, Detected-Scan-Rate, and ELINT-Scan-Rate. Note that while attributes are not shown in Figure 1, each leaf in the pruned context covers at least one requested attribute. After mapping the EER objects in the pruned context to logical schema objects and determining a natural join order based on explicit EER associations, the following sequence of relation schemas is produced:

```
(elint_report,elint_rf,elint_prf,elint_pw)
```

At this point, attributes are qualified with relation and database information, natural join criteria are inferred, and the final query is generated:

```
SELECT
    arl_elint..elint_prf.prf,
    arl_elint..elint_rf.rf,
    arl_elint..elint_pw.pw
FROM
    arl_elint..elint_report,
    arl_elint..elint_rf,
    arl_elint..elint_prf,
    arl_elint..elint_pw
WHERE
    arl_elint..elint_report.elnot_id =
      "ABC1234"  AND
    arl_elint..elint_rf.elint_report_db_id =
      arl_elint..elint_report.
      elint_report_db_id  AND
    arl_elint..elint_prf.
      elint_report_db_id =
    arl_elint..elint_report.
      elint_report_db_id  AND
    arl_elint..elint_pw.elint_report_db_id =
      arl_elint..elint_report.
      elint_report_db_id
```

The query above is more complicated than its corresponding USQL request. In fact, in complex databases, it is not unusual to generate SQL queries that are more than 100 lines in length from USQL requests that are only a few lines in length. It should also be noted that USQL was not designed for direct use in an interface. Instead, it was designed to facilitate the construction of higher level interfaces, such as those based on direct manipulation or natural language. Such interfaces accept requests expressed in a user-friendly form, convert them to USQL, and pass the converted requests to a system such as QUICK for query generation. Significantly, as a result of the capabilities provided by automated query formulation systems, interface designers can focus on user interaction instead of on structural aspects of the underlying databases.

Figure 2 shows a portion of the Military Intelligence Integrated Data System and Integrated Database (MIIDS/IDB), which is an inverted file database that integrates data contained in the Automated Intelligence File and the Defense Intelligence Order of Battle System. While all three systems contain data to support command, control, and communications, MIIDS/IDB also is used to support war planning and fighting at multiple levels (i.e., national, theater, tactical, and operational levels).

The heavy line in Figure 2 corresponds to one of the five (overlapping) contexts that comprise the MIIDS/IDB EER conceptual schema. Two rules were used to include EER objects in the context shown. First, any entity type that can be functionally determined (via N:1 and 1:1 relationship types) from EQUIPMENT-TYPE and that does not introduce a cycle at the EER level is included with its associating relationship type. This follows from a theorem in relational database theory which says that if one relation functionally determines the attributes in another, then the two relations can be natural joined in a lossless manner [21]. Consequently, a strong association is implied and context inclusion is justified. Second, the recursive relationship types are included because their participating entity types are included. This follows from the fact that any recursive relationship type relation can be joined in a lossless manner with its participating entity type relation, and thus is strongly associated with that entity type.

Due to the violation of various relational design guidelines, reverse engineering the conceptual schema of MIIDS/IDB using the techniques described in Section 3 was not straightforward. For example, attributes were inappropriately propagated from entities to relationships, thus complicating database updates and violating normal form guidelines based on functional dependency specifications [6,21]. In addition, some relationships actually represented sets of relationships with variant primary keys. Thus, entity integrity [6] was violated as there was no standard primary key that did not contain null values for some tuple.

Because of the problems associated with the logical schema, a decision was made by ARL designers to abandon the logical schema and to create instead a more idealized conceptual schema. However, to the extent possible, an attempt was made to preserve the mappings from the EER level to the logical level. The revised conceptual schema was then provided as input to QUICK, which performed context analysis and generated queries in response to high-level requests.

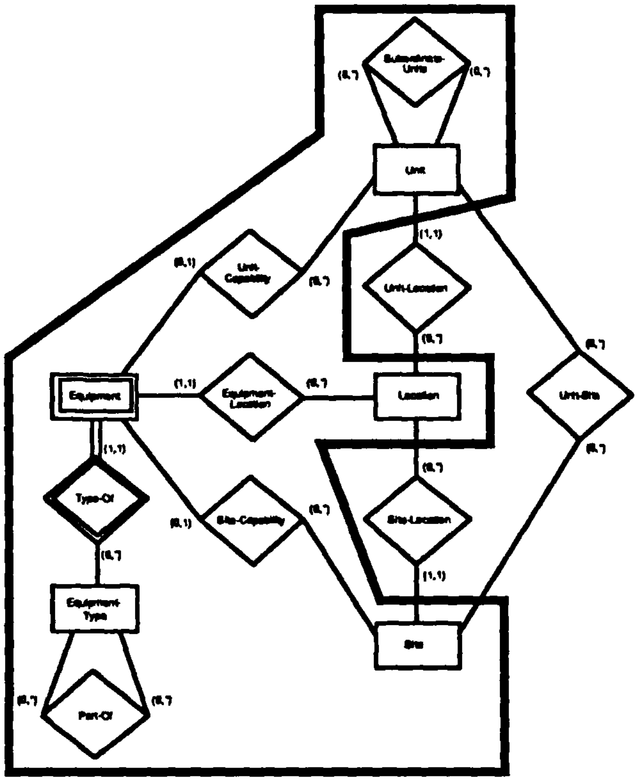


Figure 2. Portion of MIIDS/IDB database with one context shown.

It was after several queries were generated that the magnitude of the problems associated with the MIIDS/IDB design were revealed. In particular, the formulated queries did not correspond to the likely intent of a typical user. However, when analyzed individually, the rationale for QUICK's behavior was apparent. Consequently, a decision had to be made as to whether the generated set of contexts should be discarded in favor of a manually created set or whether further redesign was justified. Upon careful analysis, it became apparent that the EER conceptual schema was flawed and should be revised. After several iterations of revision, the generated queries were deemed reasonable, implying that the new contexts effectively represented sets of strongly associated EER objects.

The iterative process involved in the redesign of MIIDS/IDB demonstrated that QUICK can play a valuable role in validating either a new design or a reverse-engineered design. In particular, an EER conceptual schema can be given to QUICK for analysis. Then, typical high-level requests can be posed, and the generated queries can be analyzed. If the queries are deemed inappropriate, then the conceptual schema is evaluated by designers and modified accordingly. The process iterates until no more changes are necessary or until the designers feel that no more changes are justified. In the latter case, the generated contexts can be manually modified so that the resultant queries are consistent with the likely intent of typical users. In practice, manual context modification has not been necessary for well-designed EER conceptual schemas.

5. Fusing the U. S. Army Databases

Though logically disjoint, the ARL-ELINT and MIIDS/IDB databases are conceptually associated. As a result, decision makers would like to pose requests that integrate data from the two systems. Achieving the appropriate level of data fusion between the databases was straightforward. Specifically, a gateway relationship type was introduced that explicitly linked SITE in MIIDS/IDB with EOB-SITE in ARL-ELINT. Figure 3 illustrates this relationship, and identifies the explicit natural join specification between the corresponding entity types. Note that the ELINT-MIIDS-SITE relationship type is virtual in the sense that its only purpose is to serve as a gateway for database fusion. It does not map to a logical-level structure.

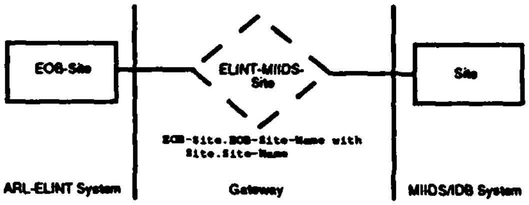


Figure 3. Gateway relationship type between ARL-ELINT and MIIDS/IDB systems.

With the gateway relationship type in place, contexts were regenerated, and queries were formulated in response to high-level requests. In particular, it became possible to generate queries that required data fusion between low-level electronic intelligence information and high-level tactical information. For example, suppose that a request were made concerning the last-known activities, mobility status, and equipment types and quantities of any known enemy unit associated with an EOB site where a pulse repetition frequency (PRF) of 50 Hz has been detected. In USQL, the request could be expressed as follows:

```
Select  site_name,
        unit_name,
        unit_role,
        echelon,
        activity_name,
        mobility_status,
        equip_code,
        equip_quantity
Where   allegiance = "ENEMY"  And
        prf = 50
```

In turn, the generated query is as follows:

```
SELECT
    miids..site.site_name,
    miids..unit.unit_name,
    miids..unit.unit_role,
    miids..unit.echelon,
    miids..unit.activity_name,
    miids..unit.mobility_status,
    miids..equipment_type.equip_code,
    miids..equipment.equip_quantity
FROM
    miids..equipment,
    miids..unit,
    miids..site,
    arl_elint..eob_site,
    arl_elint..eob_component,
    arl_elint..elint_report,
    arl_elint..elint_prf,
    miids..equipment_type
WHERE
    (miids..unit.allegiance =
       "ENEMY"  AND
     arl_elint..elint_prf.prf =
       50)  AND
    miids..equipment.unit_name =
      miids..unit.unit_name  AND
    miids..equipment.unit_role =
      miids..unit.unit_role  AND
    miids..equipment.echelon =
      miids..unit.echelon  AND
    miids..equipment.site_name =
      miids..site.site_name  AND
    arl_elint..eob_site.eob_site_name =
      miids..site.site_name  AND
    arl_elint..eob_component.
      eob_site_db_id =
      arl_elint..eob_site.
      eob_site_db_id  AND
    arl_elint..elint_report.elnot_id =
      arl_elint..eob_component.elnot_id  AND
    arl_elint..elint_prf.
       elint_report_db_id =
       arl_elint..elint_report.
       elint_report_db_id  AND
    miids..equipment.equip_code =
          miids..equipment_type.equip_code
```

The resultant query is quite a bit more complicated than its corresponding USQL request, and demonstrates the seamless fusion capabilities supported by QUICK. Conceptually, the query was formulated by finding the appropriate subgraphs in each of the corresponding databases (i.e., the pruned conceptual schema of Figure 1 and the pruned context shown in Figure 2), explicitly connecting the found subgraphs via the gateway relationship type, mapping to the logical level, inferring join criteria, and generating the final query. As desired, the user did not have to be concerned with the underlying structures of the individual systems, and did not have to be aware that multiple systems were required to respond to the request.

6. Enhancing the Approach

When the information contained in distinct databases is disjoint or almost disjoint (e.g., as was the case with the ARL-ELINT database and the MIIDS/IDB database), the process of query generation via explicit data fusion specification is relatively straightforward. However, in cases where a large number of attributes are duplicated or where the participating databases may be fused via multiple gateways, alternative approaches must be considered.

When a large number of duplicate attributes exist, actions must be taken to resolve potentially conflicting information and to ensure that consistent results will be produced. As with the UR approach, QUICK does not allow duplicate attributes, and warns designers if such attributes are found when multiple database schemas are analyzed. In turn, designers are given the option of renaming attributes at the conceptual level while retaining duplicate logical-level names. Consequently, high-level requests that specify the appropriate conceptual-level attribute will result in a semantically reasonable logical-level query.

Requiring knowledge of different attribute instances calls for some sophistication on the part of the user. This stems from the fact that it is difficult to infer an appropriate instance based on, for example, measures of nearness. A promising compromise entails listing the ambiguous attributes with brief descriptions so that the user can make appropriate selections. However, the approach requires a sophisticated interface that is able to interact intelligently with users about the meanings of different attributes. A somewhat simpler approach has been successfully used in the StarView intelligent interface developed by the Space Telescope Science Institute [17]. StarView lists all attributes associated with relations, and allows users to select the desired set of attributes. Based on the attributes chosen, specific attribute instances are identified. Given this set of instances, QUICK is able to select contexts unambiguously and generate the final query.

When a large number of gateways exist, cycles may be introduced between participating databases. As a result, context regeneration may be inappropriate, as context inclusion rules involving cycles can change the sets of strongly associated objects in an individual database. Thus, in the general case, a different approach is needed to associate contexts among databases. One possibility being considered involves introducing meta-level constructs that enable explicit specification of gateway relationship types, but that do not affect individual database contexts. A significant advantage to this approach is the savings in processing time required for context generation. The approach is also intuitively appealing as it retains the strong associations found within individual databases. However, issues regarding query disambiguation arising from the multiple paths among the fused databases must still be resolved.

7. Summary and Conclusions

As information systems grow in complexity, the need to integrate multiple databases is often required and effective data fusion becomes essential. In this paper, we have focused on how retrieving data across databases is complicated by potential inconsistencies among similar data elements, as well as by the burden placed on users to be familiar with the underlying logical schemas of participating databases. We also presented a system known as QUICK, which has been designed to counteract some of the problems discussed.

QUICK facilitates reverse engineering of logical schemas into knowledge-rich EER conceptual schemas and supports automated query formulation. By using an EER model, QUICK is not dependent on a particular logical model, and can be used for unified conceptual design. Furthermore, QUICK's ability to formulate queries efficiently provides a mechanism for validating both new and reverse-engineered conceptual schemas. QUICK's capabilities were demonstrated with portions of two independent U. S. Army databases, and it was shown how logically disjoint, but conceptually related database systems could be fused and queried. In addition, enhanced methods that employ meta-level conceptual constructs to support reverse engineering, data fusion, and query formulation were discussed.

Currently, QUICK is being extended to handle additional meta-level constructs. For example, consideration is being given to supporting an abstract generalization type that realizes its existence through the existence of its children. Such a capability would allow a designer to specify a set of related entity types that share the same relational structure, but that correspond to different conceptual entity types. Semantic query optimization techniques are also being developed to ensure that the queries generated by QUICK will execute efficiently. Finally, extended gateway relationship types are being explored.

8. References

[1] Batini, C., Ceri, S., and Navathe, S. B. 1992. *Conceptual Database Design: An Entity-Relationship Approach.* Benjamin/Cummings, Redwood City, CA.

[2] Briand, H., Habrias, H., Hue, J. F., and Simon, Y. 1985. Expert System for Translating an ER Diagram into Databases. In *Proceedings of the Fourth International Conference on the Entity-Relationship Approach*, Lin, J., Ed. IEEE Computer Society, Chicago.

[3] Chen, P. P. 1976. The Entity-Relationship Model - Toward a Unified View of Data. *ACM Transactions on Database Systems 1*, 1, 9-36.

[4] Czejdo, B., Elmasri, R., Rusinkiewicz, M., and Embley, D. W. 1990. A Graphical Data Manipulation Language for an Extended Entity-Relationship Model. *Computer 23*, 3, 26-36.

[5] Dumpala, S. R., and Arora, S. K. 1983. Schema Translation Using the Entity-Relationship Approach. In *Entity-Relationship Approach to Information Modeling and Analysis*, Chen, P.P., Ed. North-Holland.

[6] Elmasri, R., and Navathe, S. B. 1989. *Fundamentals of Database Systems.* Addison-Wesley, Reading, MA.

[7] Hull, R., and King, R. 1987. Semantic Database Modeling: Survey, Applications, and Research Issues. *ACM Computing Surveys 19*, 3, 201-260.

[8] Johannesson, P., and Kalman, K. 1988. A Method for Translating Relational Schemas into Conceptual Schemas. In *Proceedings of the Seventh International Conference on the Entity-Relationship Approach*, Batini, C., Ed. North-Holland, pp. 279-294.

[9] Korth, H. F., Kuper, G. M., Feigenbaum, J., Van Gelder, A., and Ullman, J. D. 1984. System/U: A Database System Based on the Universal Relation Assumption. *ACM Transactions on Database Systems 9*, 3, 331-347.

[10] Leymann, F. 1989. A Survey of the Universal Relation Model. *Data & Knowledge Engineering 4*, 4, 305-320.

[11] Maier, D., and Ullman, J. D. 1983. Maximal Objects and the Semantics of Universal Relation Databases. *ACM Transactions on Database Systems 8*, 1, 1-14.

[12] Maier, D., Ullman, J. D., and Vardi, M. Y. 1984. On the Foundations of the Universal Relation Model. *ACM Transactions on Database Systems 9*, 2, 283-308.

[13] Markowitz, V. M., and Shoshani, A. 1990. Abbreviated Query Interpretation in Extended Entity-Relationship Oriented Databases. In *Entity-Relationship Approach to Database Design and Querying*, Lochovsky, F. H., Ed. North-Holland, Amsterdam, pp. 325-343.

[14] Navathe, S. B., and Awong, A. M. 1987. Abstracting Relational and Hierarchical Data with a Semantic Data Model. In *Proceedings of the Sixth International Conference on the Entity-Relationship Approach*, March, S., Ed. North-Holland

[15] Semmel, R. D. 1992. QUICK: A System that Uses Conceptual Design Knowledge for Query Formulation. In *Proceedings of the Fourth International Conference on Tools with Artificial Intelligence.* IEEE Computer Society Press, Los Alamitos, CA, pp. 214-221.

[16] Semmel, R. D. 1993. Discovering Context in an Entity-Relationship Conceptual Schema. *Journal of Computer and Software Engineering* (in press).

[17] Semmel, R. D., and Silberberg, D. P. 1993. An Extended Entity-Relationship Model for Automatic Query Generation. *Telematics and Informatics 10*, 3, 301-317.

[18] Sheth, A. P., and Larson, J. A. 1990. Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases. *ACM Computing Surveys 22*, 3, 183-236.

[19] Teory, T. J., Yang, D., and Fry, J. P. 1986. A Logical Design Methodology for Relational Databases Using the Extended Entity-Relationship Model. *ACM Computing Surveys 18*, 2, 197-222.

[20] Ullman, J. D. 1983. Universal Relation Interfaces for Database Systems. In *Proceedings of the IFIP 9th World Computer Congress*, pp. 243-252.

[21] Ullman, J. D. 1988. *Principles of Database and Knowledge-Base Systems*, Vol. 1. Computer Science Press, Rockville, MD.

[22] Zhang, Z., and Mendelzon, A. O. 1983. A Graphical Query Language for Entity-Relationship Databases. In *Proceedings of the 3rd International Conference on Entity-Relationship Approach*, pp. 441-448.

VHDL Board-Level Modeling to Expedite Redesign

L.J. Ceder
Naval Research Laboratory, Washington, DC 20375-5320

Charles Rogers, Louie Kitcoff, James Michaud,
David Broadhead, Lindsay Skidmore
Naval Air Warfare Center - Aircraft Division (NAWC - AD)
Indianapolis, IN

John Miles, Gary Hout, Ed Woods, Darin York
Naval Surface Warfare Center (NSWC), Crane, IN 47522

Peter Everitt
CACI, Inc. - Federal
Advanced Manufacturing Technologies
222 W. Coleman Blvd.
Mt. Pleasant, SC 29464

Obsolescence of electronic components in military systems is becoming one of the most expensive problems of the Department of Defense (DoD), costing millions of dollars a year in component reengineering, special orders, volume buys when only single products are required, and re-design costs. For this reason, the need for new techniques to reduce the maintenance cost of older systems is especially important as funding is reduced and weapon system life cycles are increased. The Standard Hardware Acquisition and Reliability Program (SHARP) and the Flexible Computer Integrated Manufacturing (FCIM) program office put together the Technology Independent Representation of Electronic Products (TIREP) program to address these problems. The main purpose of the TIREP program is to develop and demonstrate a method to recreate an equivalent physical manifestation of an electronic circuit assembly from a digital functional behavioral description such as the Very High Speed Integrated Circuit (VHSIC) Hardware Description Language (VHDL).

In this paper we will describe the efforts of the TIREP program in recreating form-fit, electronic equivalents of several Standard Electronic Modules (SEMs). Through this effort we intend to develop a methodology for redesigning obsolete components and subsystems quickly and cost effectively, using VHDL and related standards.

Objectives

The DoD has set as its goals the movement from physical inventory to design-to-shelf products. The concept of design-to-shelf is that new technology innovation can be developed and prototyped and then held in a low or pre-production state until the need is established. The availability of programmable logic components, such as Programmable Array Logics (PALs), Programmable Logic Arrays (PLAs) and Field Programable Gate Arrays (FPGAs), provides an attractive vehicle to implement this methodology. If a standard module based on this technology were available, inventories would only require a small number of different part types, which would be personalized for specific functions as needed. Also, commercially available standard components which provide equivalent functionality could be procured at a fraction of the cost of special component buys. The saving would be both economical and compress the time to customer for delivery of electronics. This concept would have obvious life cycle cost advantages.

Modeling the subsystem using VHDL, we can develop a technology independent, functional representation of it. From this model, we can then synthesize a functionally equivalent circuit that is "plug compatible" with the rest of the system, using whatever technology is currently available. If a programmable logic device (PLD) is to be used, the VHDL model can even generate the bit-stream to program the PLD.

Ms. Ceder's work was supported by Dr. Ingham Mack, Office of Naval Research
The work of personel from NSWC and NAWC was funded by Dave Fisher, SHARP Program office.

A major objective of the TIREP Program is the documentation of the business rules and processes required to share the information between multiple organizations. While the VHDL standard defines the representation of the behavioral and structural information, actual implementation variation or styles can impede successful use of this standard information. The development of a User Guide which specifies the business rules, using and structuring the various information, will be a deliverable of the TIREP Program. This User Guide will assist organizations in developing VHDL compliant datasets that will be supported through the life cycle of the product.

Another important requirement of the TIREP Program is the development of a VHDL-to-manufacturing interface. The design or storage of VHDL models without the ability to efficiently transfer the information to the manufacturing facility is non-productive. It is essential that the functional definition and the physical constraints can accurately be transferred to the production facility. As part of the development of this interface, the TIREP Program includes the actual production of "format A" Standard Electronic Modules (SEM "A"s). The Rapid Acquisition of Manufactured Parts (RAMP) Printed Wiring Assembly (PWA) facility at NAWC - AD will be the primary production site.

There are other objectives defined for the TIREP Program that will assist in the development of a methodology for the representation and storage of technology independent electronic products. These objectives include:

* Documenting the modeling process
* Recommendations to the Repository of Electronic Components Designs
* Virtual prototyping of Electronic Designs
* Improved representation of timing values and test vectors
* Dual-use technology transfer
* Integration and cooperation with other programs such as GEM.

Approach

The TIREP Program selected thirteen SEM "A" modules of medium to high obsolescence. These modules were chosen because of the urgency of the need for replacement parts and also for their simplicity. They are purely digital and range in complexity from 2-input NAND gates to basic functional units such as multiplexers and ALUs. After a methodology for redesign is developed and proved by the manufacture and test of these smaller parts, the same procedures can be applied to other larger format SEMs and commercial format modules.

The TIREP Program emphasizes useability through the use of industry standards such as VHDL (IEEE Std. 1076), Waveform and Vector Exchange Specification (WAVES, IEEE Std. 1029.1), EIA 567 Commercial Component Specification, the VHDL Data Item Description (DID, DI-EGDS-80811), and the multi-value logic system interface package (IEEE Std. 1164). If a standard set of rules by which to model the components is not established, the specification is open to a variety of interpretations and implementation techniques. By defining a standardized modeling approach and testing the implementation by exchange of the models, these models can be used again in the future to synthesize new components and subsystems.

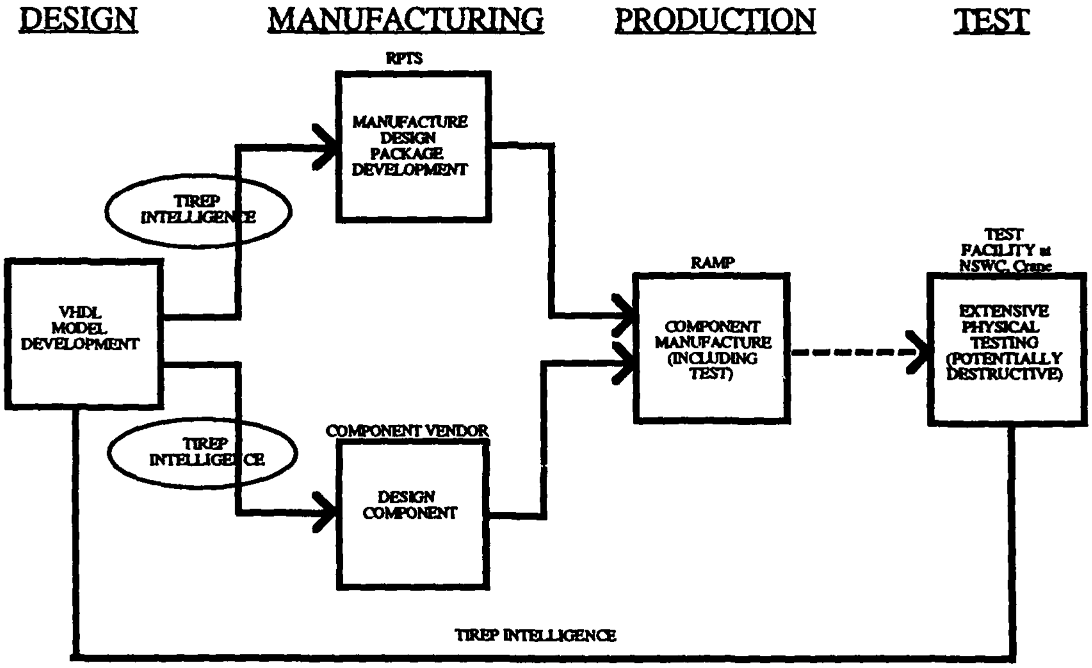
In order to maintain a technology independent representation of the SEM, the models were developed along the guidelines of EIA 567. This implementation requires a functional "core" model with electrical, physical, and timing characteristics defined in VHDL packages. These packages are wrapped around the core model and values are passed down from the top "board level" as generics. Therefore, the "core" model remains purely functional and can be used in the synthesis of subsequent components. The EIA 567 standard packages contain definitions of object types to facilitate assigning DC parametrics and timing information. The logic levels are defined in IEEE Std. 1164, the nine-level logic package, and the test bench and test vectors are written in IEEE Std. 1029.1, the WAVES dataset packages.

After the VHDL models were written, they were exchanged among members of the group for review, simulation, and evaluation. This was done to verify the information was modeled correctly, portability between different VHDL toolsets was upheld, and to evaluate any differences in the way component modeling was interpreted or approached. Each model was evaluated using a common "measuring stick" which was developed using criteria required by the DID (DI-EGDS-80811) and EIA 567. Points of discrepancy and misunderstanding with the models were raised as issues and discussed in order to determine the procedure the "standardized" models are going to use. This paper will address those issues later.

Once the VHDL models are revised to reflect the standard methodology by which to model components, the "core" model will be used to synthesize the new component. This synthesis may result in a new module which implements a one-for-one replacement of the IC's, if ICs can be found that implement the functionality, timing, and DC parametrics of the obsolete component. One of the major problems with redesigning a component into a new technology, however, is matching the timing constraints. If this is a concern, then the designer may want to consider a PLA, PLD, or FPGA that can implement the correct timing delays.

The proof of our efforts in developing a methodology for redesign of obsolete parts is going to lie in the actual production, and then test, of the new parts. Upon completion of the validation effort, the models will be provided to the Rapid Acquisition of Manufactured Parts (RAMP) facility at NAWC, Indianapolis for manufacture. We have only begun to investigate the design-to-manufacture interface. At this point we have provided the RAMP facility with a complete parts list included on the new module. In addition to the list of new components, this includes the frame and connector numbers from the old part in order to maintain the same form-fit as the old part.
A summary of this process is shown in figure 1.

A standard convention for VHDL file names and directory structures needed to be established in order to interface with the RAMP Product Data Translation System (RPTS). The following are the VHDL file conventions which are recommended for use with the subject databases.
* All VHDL files shall reside in the "VHDL" subdirectory of a specified job. It is recommended that there be no additional directory structure under "VHDL", although this is not prohibited. It has been noted that directory structures can complicate file transfers between computer systems.
* There shall be no constraint on the number of files present in the "VHDL" directory.
* A VHDL model file shall be named according to the following convention. The <filename> shall be generated by the design engineer and should be consistent with the naming conventions for VHDL identifiers. The <filename> should not exceed 8 characters in length. The <extension> is a 3 character file extension as defined below.
<filename>.<extension>
* All files contained in the "VHDL" directory shall have one of the following file extensions:


DESIGN
MANUFACTURING
PRODUCTION
TEST
RPTS
MANUFACTURE DESIGN PACKAGE DEVELOPMENT
TIREP INTELLIGENCE
VHDL MODEL DEVELOPMENT
RAMP
COMPONENT MANUFACTURE (INCLUDING TEST)
TEST FACILITY at NSWC, Crane
EXTENSIVE PHYSICAL TESTING (POTENTIALLY DESTRUCTIVE)
TIREP INTELLIGENCE
COMPONENT VENDOR
DESIGN COMPONENT
TIREP INTELLIGENCE

Summary of Design-to-Manufacture Process
for TIREP

Figure 1

Extension	File Type/Description
.VHD	VHDL source file
.WAV	VHDL WAVES source file. It is not require that this extension be used for WAVES files. It is acceptable practice to use .VHD for these files.
.DAT	model specific data file. Examples of data files include the vetor file for the WAVES testbench, component programming file, etc.
.TXT	An ASCII text file which contains information about VHDL model, it's use and application.

* All VHDL model files shall contain only ASCII text.

* DI-EGDS-80811, paragraph 10.3 dictates the presence of 7 or more text files (.TXT extension) which contain model content information. These files shall be included with all VHDL models. It is recommended that these files be named as follows:
FILE_1.TXT
FILE_2.TXT
FILE_3.TXT
etc.

Issues Encountered During VHDL Model Evaluations

Data Sheet Requirements

One of the modeling issues that was addressed during evaluations of the VHDL models was the way DC parametrics and other Mil-Spec Data Sheet requirements are applied to the functional model. We found limitations with the way VHDL can model these requirements, thus making it impossible to write a true electronic data sheet (EDS) for the SEM "A" boards using VHDL. It was decided that DC parametrics would be captured using the EIA 567 electrical view approach. This will provide a standardized modeling approach which may be used in the development of customized software directed at the generation of DC parametric test and evaluation. Also, in accordance with EIA 567, voltage and temperature levels out of range will be flagged with assertion statements. Other requirements that can't be modeled in VHDL, such as isolation and output crosstalk, will be documented in the VHDL software with comments.

Power and Ground

The representation of power and ground is another problem we addressed. It was agreed that power and ground need to be represented for netlisters and for accurate documentation of the part. Synthesis tools, however, have different methods of implementing power and ground. Since one of our goals is the VHDL-to-manufacturing interface, which includes synthesis, we decided not to include power and ground in the synthesizable "core" model. Power and ground may be implemented in the component model at the discretion of the design engineer if functionality of the component depends on it. In this case, however, power and ground should still be represented as a structural shell built around the synthesizable core model. In all other cases, power and ground should be included as nets on the printed circuit assembly (PCA) model and, when appropriate, as pins on the PCA connector. Implementation of power and ground in the WAVES testbench has yet to be determined and may rely on software development to generate test programs from the testbench.

Decomposition of Component for Modeling

The architecture of the "core" VHDL model should be logically modeled as a decomposition of functional blocks, in order to maintain technology independence. In some cases, this decomposition may result in an actual correlation to the decomposition of components on the board and component level models may be used. The designer, however, is not constrained by component boundaries when developing the model.

If, at the discretion of the designer, the component or subsystem is modeled using a structural architecture that breaks down to models of actual, technology dependent, physical components, this model must be used to establish a baseline against which the redesign will be evaluated. A VHDL model that documents the new design must also be developed and should be technology independent, whenever possible, to facilitate synthesis into future technologies. When the model of the original design is used only as a development tool and may not be representative of the redesigned circuit, it is not necessary that this model be compliant with the DID, DI-EGDS-80811. The new, potentially synthesizable VHDL model that is developed, that also serves as documentation of the redesigned part, should follow all the standards and guidelines we have listed, including DI-EGDS-80811, as closely as possible.

Each component or subsystem that is modeled in VHDL should be accompanied by a high level behavorial model for future redesign efforts. However, limited documentation or component complexity may limit the design engineer's ability to develop high level behavorial models within a reasonable amount of time and effort. Therefore, it is decided that development of a high level behavorial model should be addressed, by the design engineer, on a case by c ..e basis.

Testbenches and Test Vector Sets

Test benches and test vector sets were developed using the WAVES standard. Test benches should be provided with at least two test vector files. The first one should validate a good model that is error free. The second vector file should be seeded with errors to test that the model flags the proper assertion statement upon receiving an error and to test for proper functional operation. The "error seeded" vector file should be adequately documented with comments to indicate which error conditions it is testing.

Unresolved Issues

During the evaluation of the VHDL models, several issues were brought up that remain unresolved.

* The implementation of power and ground in the WAVES testbench has yet to be determined and may rely on software development to generate test programs from the testbench.

* Guidelines must be established to detail the manner in which multiple timing constraints on a single pin will be handled.

* In order to meet the requirements of DI-EGDS-80811, nominal component delay values may be modeled as the average of the minimum and maximum delay values, when known. In those instances where only the maximum delay is specified, it will be necessary to generate a recommended approach for determining the minimum and nominal delays.

Conclusion

TIREP was established to address the costly issue of component obsolescence in military systems. By using industry standards such as VHDL and WAVES in the redesign of obsolete components we not only make full use of synthesis tools that are available today, but we also address problems of future obsolescence. The efforts of TIREP in developing a set of business rules and processes with which to represent "standardized" behavioral and structural information and develop a design-to-manufacture interface is not complete. VHDL modeling guidelines are currently in the process of being developed and documented. With the interfaces that exist between VHDL and PLAs, PLDs, and FPGAs, systems can be redesigned with greater part redundancy, resulting in inventories with smaller number of part types. Eventually with the shift to programmable logic components, we may even see component consolidation in systems which would not only greatly reduce redesign-for-obsolescense costs, but may as well reduce system failure rates as the number of fallible components decreases.

Using the CIM Software Systems Reengineering Process Model in Navy Reengineering Efforts

Tamra K. Moore
Defense Information Systems Agency
Joint Interoperability Engineering Organization
Center for Information Management
Arlington VA 22204-2199.

This paper introduces the Center for Information Management (CIM) Software Systems Reengineering Process Model [8] and explores its use in planning and implementing Navy reengineering efforts. The Model provides guidance for reengineering automated information systems within the Department of Defense (DoD). This Model addresses many issues concerning large-scale, complex systems which impact Navy real-time system modernization efforts. The intended audience for the Model is any organization within DoD tasked to reengineer information systems, however, Navy efforts to define a reengineering process model for real-time embedded systems may find the Model a useful starting point.

Background

The purpose of the CIM Software Systems Reengineering Process Model is to capture the essence of software reengineering as it applies in the DoD Information Management (IM) community. Two broad concepts guide software reengineering for DoD IM. The first concept is the prevention of duplication by joint use of personnel, information systems, facilities, and services across DoD. For the IM community, multiple software systems often provide similar services which may be consolidated and reduced to a single system through software reengineering. This holistic view of reuse does not apply to Navy embedded systems for which a reduction in duplication at the component level drives reuse activities. Utilization of common low-level system components minimizes redundancy in the embedded, real-time system domain and provides cost-benefits in terms of maintenance and system development. Reuse and domain analysis programs within various Navy organizations are addressing these issues. The second concept guiding software reengineering is conformance to new regulations, policies, standards, and guidelines for software acquisition and support. This concept also drives most Navy modernization efforts today. Such standards include using a common programming language (MIL-STD-1815A Ada Programming Language), moving towards open software systems (FIPS 146-2 Government Open-Systems Interconnection Protocol), and maintaining compliance with a common operating system interface (FIPS 151-1 Portable Operating System Interface Exchange, POSIX). Guidelines include integrating Commercial Off-the-Shelf (COTS) products where possible, including Computer-Aided Software Engineering (CASE) tools.

Model Overview

The CIM Software Systems Reengineering Process Model is represented using the $IDEF_0$ method [3, p58]. $IDEF_0$ is used to produce a structured representation of activities or functions and the relationship between those activities. $IDEF_0$ models are composed of activities and interfaces, including inputs, controls, outputs, and mechanisms (Fig. 1).

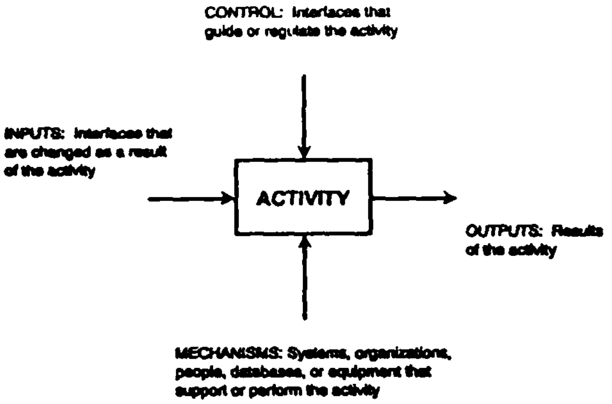

CONTROL: Interfaces that guide or regulate the activity
INPUTS: Interfaces that are changed as a result of the activity
ACTIVITY
OUTPUTS: Results of the activity
MECHANISMS: Systems, organizations, people, databases, or equipment that support or perform the activity

Fig. 1 IDEF Activity Model

Activities are represented as boxes and the interfaces are depicted as arrows, entering and leaving the boxes. Inputs enter from the left and outputs leave from the right of the box; the activity transforms inputs into outputs. Controls enter at the top of the box; they provide direction and constraint. Mechanisms, representing the means used to perform the activity, enter from the bottom. Activities and their relationships are not related to or limited by time.

Software Systems Reengineering Process

The CIM Software Systems Reengineering Process Model describes the activities supporting the development and maintenance of automated information systems based on the examination and utilization of existing software system resources. The guidance provided by the Model for reengineering information systems is applicable to Navy reengineering efforts. Two differences between reengineering automated information systems and Navy embedded, real-time applications are (1) the level of effort spent in each of the reengineering activities, and (2) the methodologies and tools supporting these activities. Since the CIM Model does not specify levels of effort nor identify specific methodologies and tools, this model may be useful in planning the reengineering of embedded, real-time applications. The CIM Model provides a framework upon which information on specific methodologies, tools, and case studies presented during the Systems Reengineering Technology Conference can be integrated.

The Model's context diagram provides a framework for reengineering by identifying the interfaces to external activities (Fig. 2). The names of these interfaces and their sponsoring activities reflect the Model's intended use in the IM community. Comparable activities constrain and guide the reengineering process for the Navy. In this paper, the names of these interfaces have been modified to facilitate the Model's use in Navy applications. The original IM names are listed in parenthesis immediately following the modified names for reference when applicable.

Inputs

The inputs to the reengineering activity include the New Requirements (Business Requirements), Existing System (Automated Information System), Feasibility Analysis Results, and Reusable Assets. The New Requirements are optional and specify new functionality for the known existing system. The Existing System will be reengineered. Feasibility Analysis Results summarize preliminary studies which may have been performed, including a cost/benefit analysis, technical feasibility study, and a risk analysis. Reusable Assets are software work products, including source code, documentation, designs, test data, and specifications. Reuse components for Navy are stored in a DISA-sponsored repository for Navy efforts in Washington D.C. supported by NAVCOMTELCOM. Reusable assets are also available from NUWC, NAVSEA, and FCDSSA, Dam Neck VA.

Controls

Controls on the reengineering process include the DoD Enterprise Model [4]; Functional Area Models, Regulations, Policy, Standards, and Guidelines; Resource Limitations; Technical Architectures; and Available Reengineering Technology. Many of these controls primarily impact automated information systems. However, similar guidelines may exist for embedded, real-time systems. The DoD Enterprise Model provides the high-level vision of the mission area for the reengineered system, while Functional Area Models govern the business domain in which this system will operate. For embedded, real-time applications functional area models would define the rules governing a class systems. Regulations, Policy, Standards, and Guidelines are the documents containing the principle rules designed for governing and influencing decisions and actions during software engineering activities included in reengineering. Resource Limitations scope the reengineering project by estimating the manpower, funding, schedule deadlines, and computer resources available. These limitations are usually imposed on the organization performing the reengineering by an external sponsoring organization. Technical Architectures describe the computing and communications environment in which the Reengineered System must execute. For the Navy, this control would be synonymous with a description of the target environment for the system.

Available Reengineering Technology identifies proposed methodologies and tools available for reengineering. The technology available for reengineering automated information system is well-publicized and commercially available. For Navy applications, the available reengineering technology is often program-specific and difficult to identify. Naval Sea Systems Command contains a database of available tools for supporting CMS-2 and AN\UYK series computers. Reengineering technology development programs in the Navy, include the Engineering of Complex Systems program at NSWC. Others include China Lake (assembler issues), NUWC (reuse projects), NRaD, SBIRs, NAVSEA, FCDSSA, and the Naval Information System Management Center (NISMC).

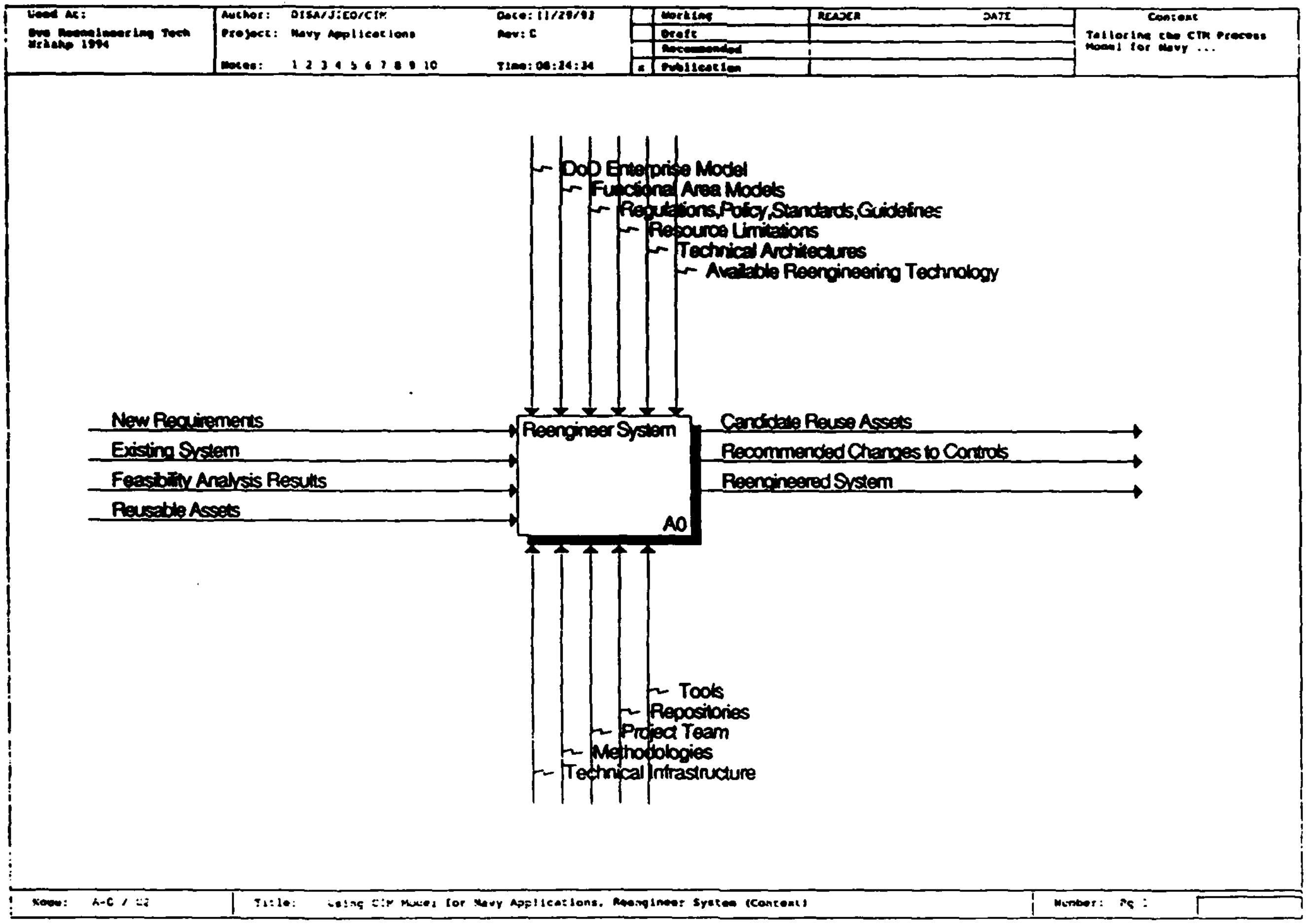


Fig. 2 Reengineering Context Diagram

Outputs

The reengineering effort produces Candidate Reuse Assets, Recommended Changes to Controls, and the Reengineered System. Candidate Reuse Assets are produced and sent to a reuse certification program. These assets include design models, system specifications, metric data, data models, process models, design decisions, and test procedures. Other assets include reengineering strategies, improved maintenance procedures, and new business practices. As experience in reengineering increases, the lessons learned will improve the process. The impact of integrating new technical architectures during the reengineering activity and adhering to new regulations, policy, standards, and guidelines during reengineering may promote new reengineering strategies. The Reengineered System, generated from activities described within the Model, consists of the application software, data, technical infrastructure, and all associated documentation.

Mechanisms

Reengineering is supported by Tools, Repositories, the Project Team, Methodologies, and the Technical Infrastructure (Computing and Communications Infrastructure). Tools are automated mechanisms used to improve productivity in software reengineering. The types of tools needed for software reengineering include project management, restructuring, reverse engineering, and forward engineering tools. Reverse engineering tools include source code analyzers, design recovery and redocumentation tools. Forward engineering tools include code generators, requirements analysis, design support tools, test case generators, and integration support tools. The tools available for supporting automated information systems and Navy embedded systems are very different due to the programming languages and hardware platforms upon which these tools operate. The IM domain has many tools which are commercially available to support the predominant languages and hardware platforms for both development and execution of automated information systems. There are very few commercially available tools which support reengineering Navy embedded, real-time systems. Many tools have been developed for specific Navy projects and identifying them, their capabilities, and acquiring them for use on Navy systems other than the ones for which they were developed can be very difficult. Reengineering support for Navy embedded real-time systems is primarily a manual effort, integrated with CASE tools. CMS-2 to Ada translators include those developed by APL, CCCC, MITRE, and NRaD.

Repositories are mechanisms for storing and retrieving information or reusable assets. Examples of Tool Repositories include those maintained by ECS, NAWC-China Lake, SPC, and STSC. Repository-based technology may also be used to store and retrieve information generated during reengineering, including system components, Reverse Engineered Products, components of the Reengineered System, reports defining Available Reengineering Technology, and Reusable Assets.

The members of the Project Team for IM include reengineering experts, maintainers, functional personnel, and the users of the system. For the Navy, the Project Team would primarily consist of the maintainers and the users of the systems. The Project Team should include experts in the following areas: software/system engineering, technical infrastructure, function/mission of the system domain, application software usage, and reengineering technology. Matching skills with the activities described in this model insures productivity and minimizes risk. The users of the system should be involved with the Project Team throughout the reengineering effort. Methodologies are systems of principles, procedures, and practices applied to the development, operation, reengineering and support of a software system.

Methodologies for reengineering include reverse and forward engineering methodologies. These Methodologies support a variety of software engineering activities, which should be investigated to minimize impact on the sponsoring organization's SEE. Reengineering methodologies for Navy embedded real-time systems are primarily informal, project-dependent, and devised by the prime contractor.

The Technical Infrastructure is the environment in which the reengineered system operates. The IM infrastructure is a service utility that provides common shared computing and communications capabilities, including data bases, common networks, electronic messaging, and computing platforms. For Navy systems, testbed facilities are used to insure that the system will operate when deployed.

Reengineering Process Activities

Software reengineering is composed of three major activities: Define Project, Reverse Engineer, and Forward Engineer (Fig. 3). The Model diagrams which refine these activities are not included in this paper, but are in the CIM Software Systems Reengineering Process Model [8].

Define Project

The Project Team defines the Reengineering Project Plan which serves as the controlling document for the reengineering project. The Project Plan should be flexible enough to handle modifications based on results of both the reverse and forward engineering activities. Analysis Deliverables from forward engineering may provide information about the New Requirements which impact the Project Plan. Reverse engineering provides a complete understanding of the existing system which may also impact the reengineering effort. Define Project is composed of the following activities: Define Objectives, Identify Baseline, and Define Reengineering Project Plan.

The Objectives identify the organizational goals of the reengineering effort, including objectives for using the system, supporting the system, and applying reengineering technology. The objectives for using the system include performance issues and user interface requirements. Improvements in the maintenance process and extending the life expectancy of existing systems are typical objectives for supporting the system. The objectives of applying reengineering technology include proofs-of-concept, proofs-of-utility, and identification of risks which might impede the reengineering process.

The Objectives may change based on the scope of the reengineering effort. Recommended Changes to Objectives may result from the Available Reengineering Technology, Resource Limitations, and New Requirements. Development of concrete measurable Objectives is an essential step in establishing the Reengineering Project Plan.

The Project Team will identify the configuration items which comprise the current System as the Baselined System. These items include the application software, data, technical infrastructure, and all associated documentation. The Baselined System will not undergo any modifications outside the scope of the reengineering project. The activity of identifying the baseline does not include the analysis of any configuration items, but simply identifies the system upon which the reengineering activities will be performed. The Objectives may control the identification of the Baselined System by requiring the reengineering of a specific version or the consolidation of multiple versions of the same system.

The Define Reengineering Project Plan activity is performed in four parts, including *Develop Reengineering Strategy*, *Identify Methodologies and Tools*, *Allocate Resources*, and *Develop Reengineering Project Plan*.

The reengineering strategy identifies reengineering alternatives for incorporating new technology and approaches, and the use of methodologies and tools. Possible alternatives include restructuring and redocumentation. The alternatives are evaluated with respect to objectives, risks, impacts, and requirements. The project strategy drives the identification and utilization of methodologies and tools for software reengineering. The strategy also identifies and describes the structure of the products expected from the reengineering effort.

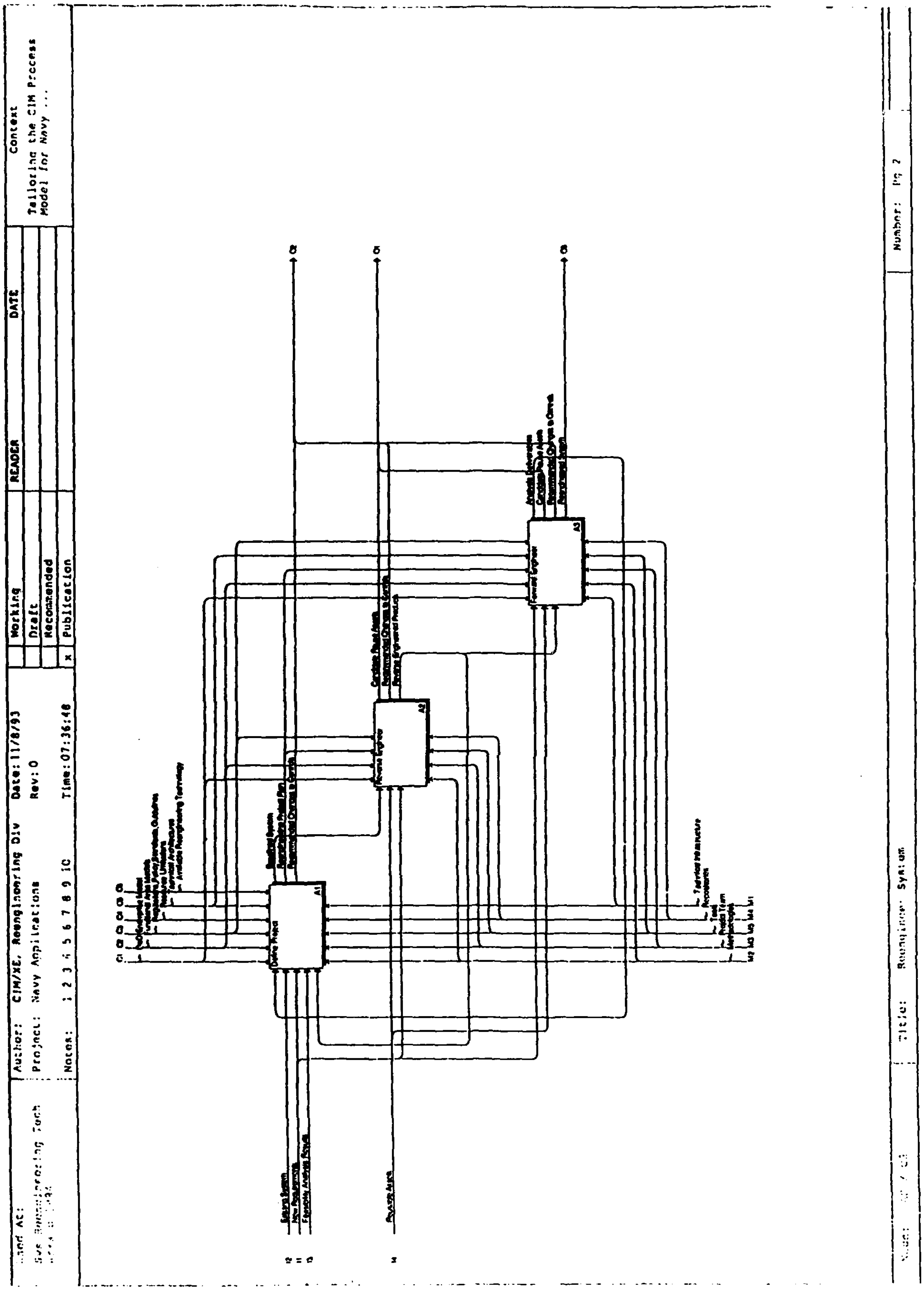


Fig. 3 Reengineering System

Proposed methodologies and tools are identified through an analysis of Available Reengineering Technology. The proposed methodologies and tools must integrate into the sponsoring organization's SEE and support the activities defined in the project strategy. The Allocate Resources and Develop Reengineering Project Plan activities may require revisions to the proposed methodologies and tools to comply with the controls and New Requirements.

The project resources are allocated for performing the reengineering project. Project resources include personnel, computer resources, tools, and the necessary training. These resources must remain within the constraint of the Resource Limitations.

A structured plan is developed for accomplishing the reengineering under the direction of Regulations, Policy, Standards, and Guidelines. The plan includes the project strategy, project resources, and selected tools and methodologies for implementing the reengineering. Revisions to the project resources are based on constraints from Regulations, Policy, Standards, and Guidelines, or inconsistencies between the project strategy, methodologies, and tools.

Reverse Engineer

The Project Team analyzes the documentation, application software, data, and the technical infrastructure of the baselined system. This analysis is performed to identify the system components and their interrelationships, and to capture these components in representations that provide a better understanding of the system. Reusable Assets should be used to compose these representations when possible. Any new representations of these components should be designed for reuse as Candidate Reuse Assets. Reverse Engineered Products must be manageable and usable in the subsequent forward engineering activities. Reverse Engineer is composed of the activities called: Analyze Documentation, Analyze Application Software, Analyze Data, Analyze Technical Infrastructure, and Integrate Extracted Products.

The Analyze Documentation activity analyzes existing documentation to extract a system specification, technical infrastructure capabilities, and system design decisions.

The Analyze Application Software activity analyzes the existing application software to extract the software specification, functional requirements, metric data, data models, process models, and software design decisions.

The Analyze Data activity analyzes the existing data to extract data products that are used to define the design model, system specification, functional requirements, metric data, data models, and data design decisions.

The Analyze Technical Infrastructure activity analyzes the existing technical infrastructure to extract technical infrastructure products that define its capabilities and related design decisions. A good understanding of the existing technical infrastructure is imperative, since many reengineering efforts integrate new operating systems and hardware platforms. The current infrastructure provides capabilities to the existing system which must be captured during reverse engineering in order to have a complete understanding of the current system. These capabilities may still be supported in the Reengineered System.

The Integrate Extracted Products activity integrates the information from the extracted products to form the Reverse Engineered Products which are forward engineered. Reverse engineering can be used to identify whether any New Requirements are supported in the existing system. Recommended Product Revisions are generated when an inconsistency is detected between one or more of the extracted products. These inconsistencies must be reconciled as part of the reverse engineering process.

Forward Engineer

Within the context of reengineering, forward engineering is the software engineering activities that consume the products of reengineering activities (primarily reverse engineering) and reuse, along with new system requirements to produce a target system. Forward Engineering is composed of activities called Analysis, Design, Build, Integrate, and Test.

Reverse Engineered Products are input to the Analyze and Design activities. Reusable Assets should be used throughout the Forward Engineer activity when possible. The Reengineering Project Plan; and Regulations, Policy, Standards, and Guidelines concerning application software development guide this activity by defining the structure of the expected components and results. Candidate Reuse Assets may be generated during the Analyze, Design, Build, and Test activities. Each activity produces the documentation required by applicable standards, including DoD-STD-2167A, DoD-STD-7935A, and the proposed MIL-STD-SDD. The DoD Enterprise Model and Function Area Models are employed. The life-cycle management is guided by DoD 7920.1 for the Life-cycle Management of Automated Information Systems.

In the Analyze activity, the New Requirements and the Reverse Engineered Products are analyzed during this activity to generate the Analysis Deliverables. The Analysis Deliverables include requirements for the Test activity and a formal specification of the analyzed New Requirements addressed in the existing system. The principal Analysis Deliverables are the business rules, system specifications, design decisions, and test procedures.

In the Design activity, the Analysis Deliverables and the Reverse Engineered Products concerning Design generate the Design Components during this activity. Design Components are modules representing a design of the system parts to be constructed during the Build activity, including the required documentation summarizing the results of the design activity. Design Results confirm that the Design Components have been constructed according to specification or they request clarification of an analysis issue needed to complete the Design activity. The principal Design Components are the design model, data models, process models, and design decisions.

In the Build activity, the Design Components are used to generate the Build Components. Build Results confirm that the Build Components have been constructed or request clarification on a design issue that is preventing the completion of the Build activity. The Build Components are the constructed system parts to be interfaced during the Integrate activity.

In the Integrate activity, any number of Build Components are combined to form Integrated Components. The Integrate activity insures the interfaces between Build Components are correct and complete. Integration Results confirm that the Build Components have been interfaced successfully or request clarification on an interface or build issue that is preventing the completion of the Integrate activity. The Integrated Components are the interfaced Build Components representing part or all of the system.

In the Test activity, the Integrated Components are verified using a test plan developed from the requirements for testing defined in the Analysis Deliverables. Every Build Component is tested according to the individual component specification. The Test Results confirm that the tests were successful and describe the test procedure. The principal products from this activity are the tested Reengineered system and the Test Results.

Conclusion

Ultimately, the process of software reengineering must support the high-level goals of any organization: (1) eliminating non-essential products and processes; (2) increasing the value of those remaining; and (3) increasing the efficiency through streamlining, simplification and/or automation [9]. Software reengineering technology provides a myriad of capabilities which support a variety of software engineering activities.

Future plans are to validate the CIM Software Systems Reengineering Process Model by applying it in software reengineering efforts. Previous efforts served as the framework for the development of the Model and subsequent efforts will prove its effectiveness. The Model represents the software reengineering process from the software engineer's viewpoint; additional models currently under development at CIM which represent software reengineering from alternative viewpoints include software management and acquisition.

Reengineering emerges as a strategy for bringing the cost of developing and maintaining software under control. The need for a comprehensive plan to apply reengineering technology is the driving force in the CIM Software Systems Reengineering Program. The CIM Software Reengineering Process Model will assist program managers facing this situation.

Acknowledgement

The author would like to thank G. Russomano, C. Wright, J. Smith, and M. Gross of the Center for Information Management for their contribution to the development of the CIM Software Systems Reengineering Process Model.

References

[1] E. J. Byrne and D. A. Gustafson, "A Software Reengineering Process Model," Conference on Computer Software and Applications (COMPSAC), Sep 1992, Chicago, IL.

[2] E. J. Chikofsky and J. H. Cross, "Reverse Engineering and Design Recovery: A Taxonomy," IEEE Software, pp. 13-17, Jan 1990.

[3] Functional Management Process for Implementing the Information Management Program of the Department of Defense, DoD 8020.1-M (Draft), Aug 1992.

[4] The DoD Enterprise Model: A White Paper, February 1993, OASD(C3I)/DDI, 1225 Jefferson Davis Highway, Suite 910, Arlington VA 22202.

[5] R. L. Hobbs, J.R. Mitchell, and G.E. Racine, System Re-engineering Project Executive Summary, ASQB-GI-92-003, Nov 1991.

[6] "Lessons Learned: Re-engineering the Weighted Airman Promotion System for the CIM Environment," MITRE Corporation, Reston VA, Sept 1992.

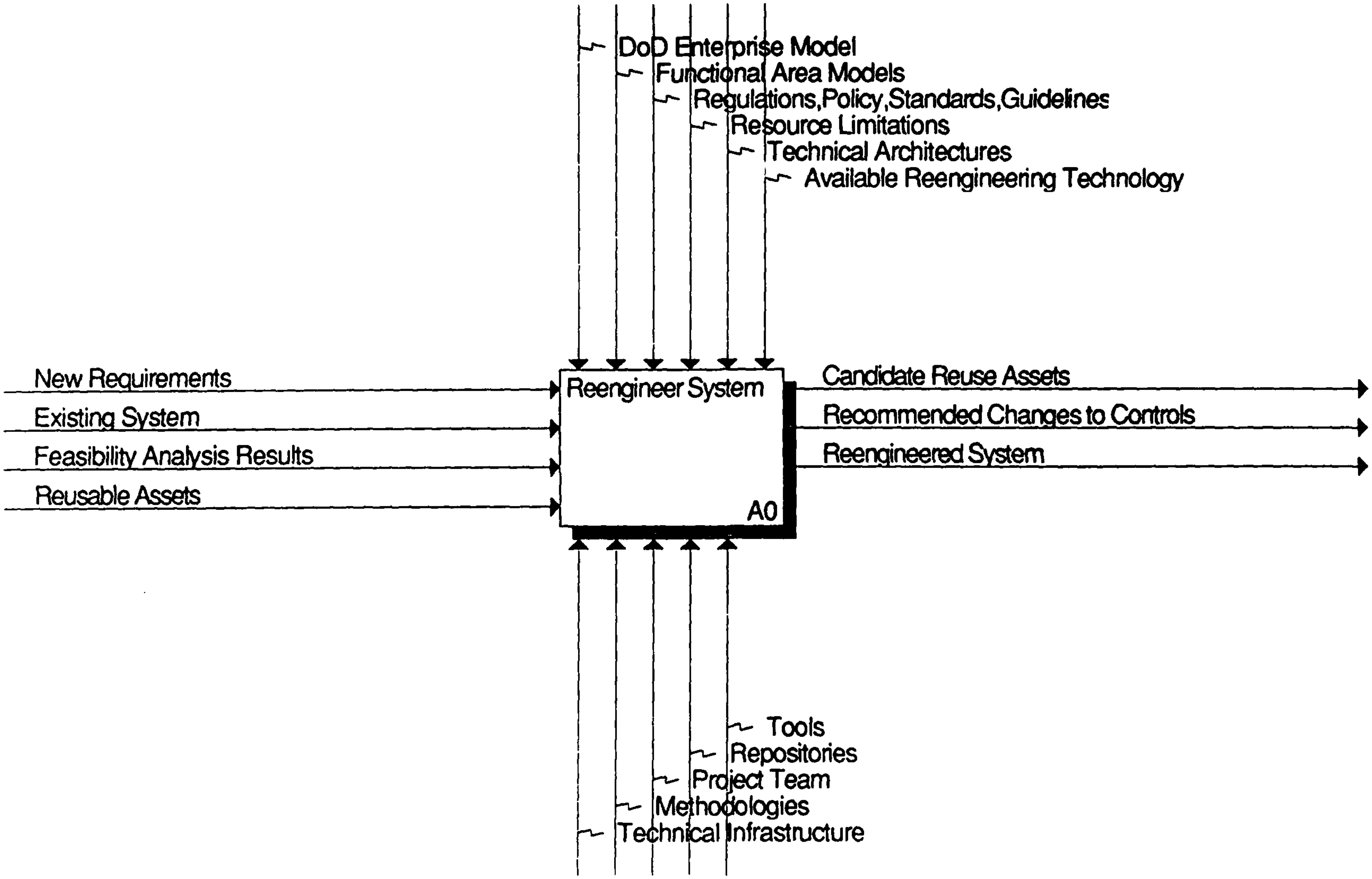
[7] T. K. Moore, Information Systems Criteria for Applying Software Reengineering, Technical Report, Center for Information Management, Arlington VA 22204, Jan 1993.

[8] T. K. Moore, CIM Software Systems Reengineering Process Model, Version 1.0, Technical Report, Center for Information Management, Arlington VA 22204, Aug 1993.

[9] L. Roomets, "Integrating CASE with Business Process Re-Engineering," Proceedings CASE WORLD, Sept 30 - Oct 2, 1992, pD9-1 to D9-13.

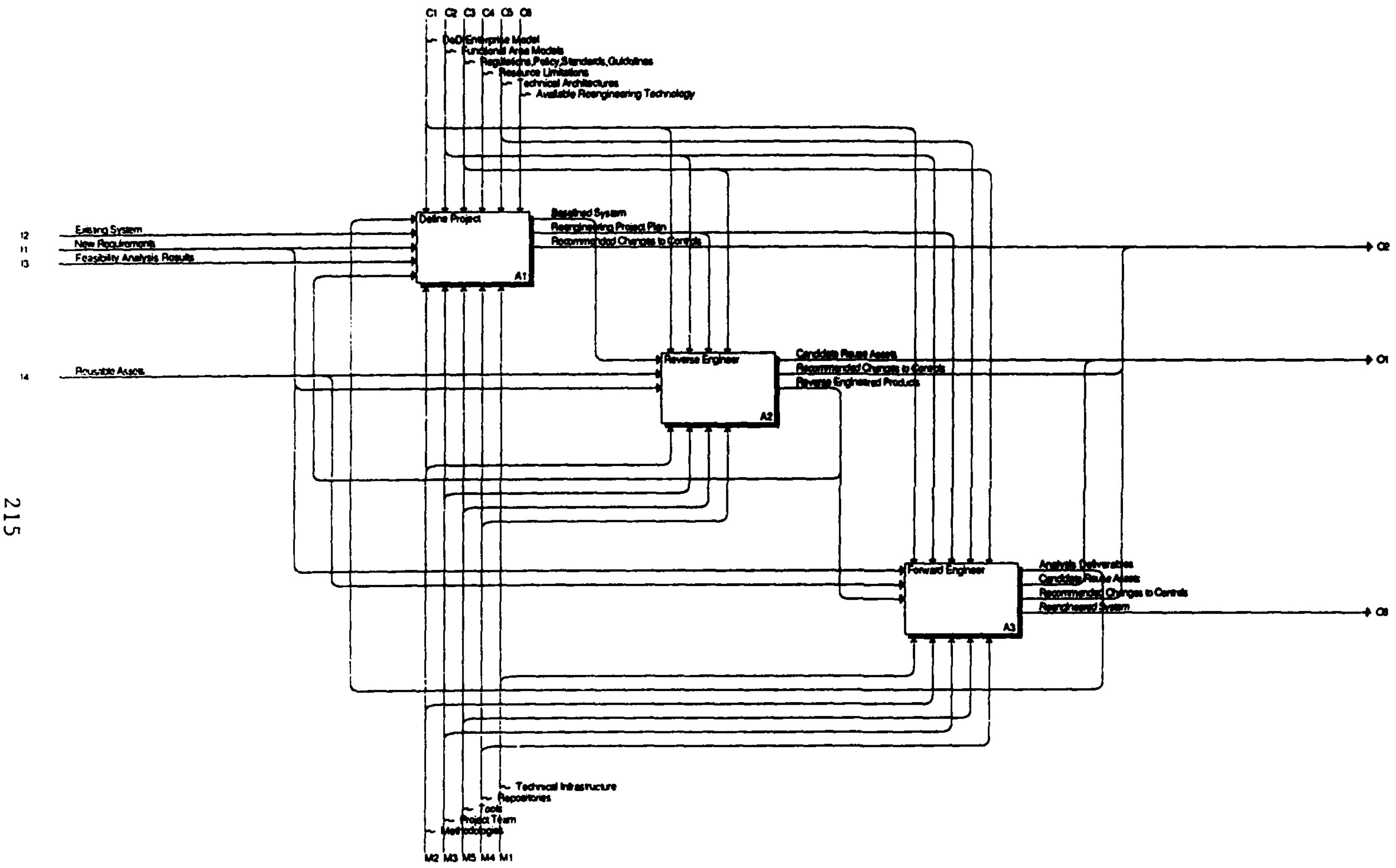
[10] M. K. Ruhl and M. T. Gunn, "Software Reengineering: A Case Study and Lessons Learned," NIST Special Publication 500-193, National Institute of Standards and Technology, Sept 1991.

Used At: Sys Reengineering Tech Wrksrp 1994	Author: DISA/JIEO/CIM	Date: 11/29/93	Working	READER DATE	Context
	Project: Navy Applications	Rev: C	Draft		Tailoring the CIM Process Model for Navy ...
			Recommended		
	Notes: 1 2 3 4 5 6 7 8 9 10	Time: 08:24:34	x Publication		



Node: A-0 / 07	Title: Using CIM Model for Navy Applications, Reengineer System (Context)	Number: Pg 1

Used At:	Author: CIM/XE, Reengineering Div	Date: 11/8/93		Working	READER DATE	Context
Sys Reengineering Tech	Project: Navy Applications	Rev: 0		Draft		Tailoring the CIM Process Model for Navy ...
[illegible] 1994				Recommended		
	Notes: 1 2 3 4 5 6 7 8 9 10	Time: 07:36:48	x	Publication		



[illegible] | Title: Reengineer System | Number: Pg 2

Software Reengineering Assessment Handbook (MIL-HDBK-SRAH)

John Clark, COMPTEK Federal Systems, Inc.
John Donald, Air Force Cost Analysis Agency
Barry Stevens, COMPTEK Federal Systems, Inc.
Sherry Stukes, Management Consulting & Research, Inc.

February 1994

ABSTRACT

Legacy software is a valuable DoD asset which should be leveraged to the greatest degree possible. There is an immediate need for DoD guidance for conducting technical, economic, and management analyses to determine when reengineering techniques are beneficial to conduct. Such guidance has been developed under the auspices of the Joint Logistics Command and the U.S. Air Force. This paper introduces the key concepts of the new draft Software Reengineering Assessment Handbook (MIL-HDBK-SRAH) which defines a process for conducting an effective technical, economic, and management assessment to determine whether and how to reengineer legacy software.

INTRODUCTION

The decision to reengineer legacy software is based on a number of technical, economic, and management factors. The new draft Software Reengineering Assessment Handbook (MIL-HDBK-SRAH) organizes these factors into a decision-making process which is targeted for the following two cases:

- A single software program needs to be assessed to determine if, and how, it should be reengineered. The SRAH process develops a recommended reengineering strategy for that program, if warranted, and an estimated return on investment and breakeven point for that strategy.

- A set of software programs within the organization needs to be assessed to determine which, if any, should be reengineered and which reengineering strategy(ies) should be used. The SRAH process develops a recommended strategy for each program, if warranted, and prioritizes the list of programs based on the need to reengineer, estimated return on investment, the estimated breakeven point, and management considerations.

The SRAH process is depicted in Figure 1 and consists of technical, economic, and management decision processes. The SRAH process begins with the identification of programs within the organization and the application of a quick screening filter to remove programs from consideration which are least likely to show a return on investment through reengineering. The resulting list of candidate programs (or, in the first case above, the single program) is subjected to a technical assessment process including a set of technical questions designed to disclose the need to reengineer. The responses to the question set are used to determine the need to reengineer and to develop a set of candidate strategies for that program.

An economic assessment process is used to determine the preferred strategies for each program. For each strategy, comparative total life cycle cost estimates are developed. Parametric cost estimating models may be used to estimate reengineering, redevelopment, and continued maintenance/support costs. Cost risk and parameter sensitivity assessments should be considered.

A management decision process is used to select the single recommended strategy for each program and to prioritize the list of programs. Selection and prioritization are based on the composite results of the technical and economic assessments and management considerations. Evaluation of the need to reengineer, return on investment, breakeven point, and management objectives and constraints results in the recommended course of action.

IDENTIFY POTENTIAL PROGRAMS FOR REENGINEERING

When evaluating a set of programs within the organization for reengineering, the MIL-HDBK-SRAH process begins with listing the potential programs for reengineering. This list could contain

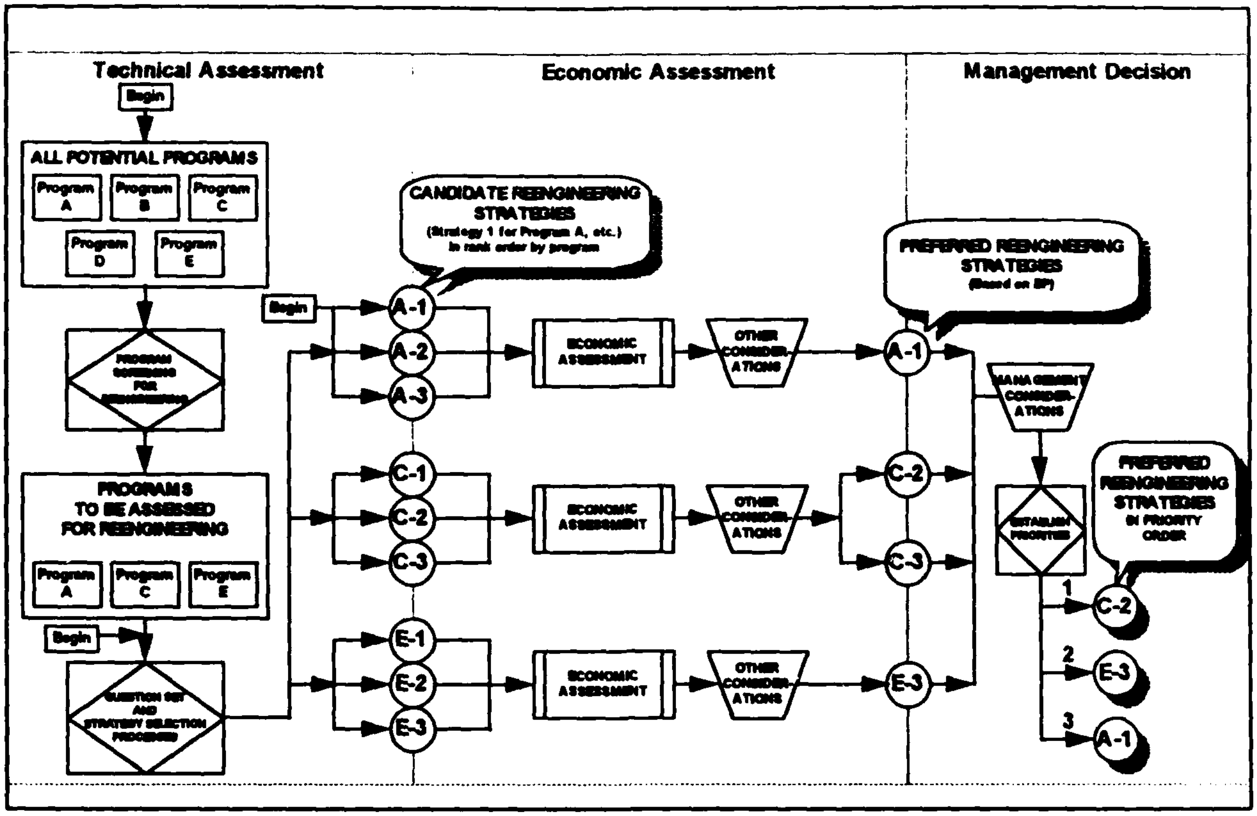


Figure 1 - MIL-HDBK-SRAH Process

all the programs in the organization, but should at least contain any program which is perceived to be a "problem." Programs over five years old, programs which require over two manyears/year to maintain, and programs whose remaining life is over three years should be on the list.

SCREEN PROGRAMS FOR REENGINEERING

Not all of the programs on the list need to be subjected to the MIL-HDBK-SRAH process. A quick screening filter is suggested as a means to remove programs from consideration which are least likely to show a return on investment through reengineering. Programs which meet any of the following criteria may be eliminated from the list of candidates:

(1) If the program age is less than five years since Initial Operational Capability (IOC) or program deployment;

(2) If the annual maintenance budget for software is less than $250,000 per year or less than two manyears per year; or

(3) If the remaining life is less than three years.

Programs which meet criterion (1), (2), or (3) may be retained on the list if desired. Each organization may desire to develop its own criteria

REENGINEERING QUESTIONNAIRE

The need to reengineer is viewed as a function of the following system and organizational factors:

A. Size (Source Lines of Code or Function Points)
B. Complexity (Logical and Data Complexity)
C. Language Type and Portability
D. Development Process Used
E. Program Quality Trend
F. System Age
G. Level of Maintenance Activity
H. Documentation State
I. Impact of System Failure
J. Personnel Factors.

Twenty-five questions are answered for each candidate program. The questions are designed to disclose the need to reengineer. Figure 2 shows the

SYSTEM FACTORS

Factor A - Size

1. *SLOC*. How many Source Lines of Code (SLOC) exist in the candidate software?
   2-Less than 15K   4-Between 15K and 100K   6-More than 100K
2. *Function Points*. How many function points exist in the candidate software?
   2-Less than 500   4-500 to 2500   6-More than 2500

Factor B - Complexity

3. *Average SLOC/CSU*. What is the average number of Source Lines of Code per Computer Software Unit (CSU) in the candidate software?
   3-Less than 50   6-500 to 200   9-More than 200
4. *Average Cyclomatic Complexity*. What is the statistical average of the Cyclomatic complexity per module?
   3-Ten or less   6-Between 10 and 20   9-More than 20
5. *Average Essential Complexity*. What is the statistical average of the Essential complexity per module?
   3-Five or less   6-Between 5 and 10   9-More than 10
6. *Data Complexity*. Which response best characterizes the state of system data?
   3-Data is managed by relational database(s) which is/are in at least 3rd Normal Form or an object-oriented database. The data dictionary is current. Data relationships are clearly documented. A high degree of data name rationalization exists.
   6-Data is managed by relational database(s) in less than 3rd Normal Form. The data dictionary and data relationship documentation (such as Entity-Relation diagrams) are mostly current; some catch-up may be required to support major enhancement. Some cryptic names for relation tables or columns exist.
   9-Data is managed as flat files. Some data is no longer used by the program, although some effort would be required to determine which. Usable documentation of data definitions and logical relationships do not exist.

Factor C - Language

7. *4GL/3GL/Assembler*. What is the system's principal language level?
   2-4GL   4-3GL   6-2GL/Assembly Language
8. *Number of Languages*. How many programming languages does the system use?
   2-One   4-Two   6-More than two
9. *Language Portability*. Which situation best characterizes the need to change source languages?
   2-No need because: An approved HOL is being used, adequate software support tools (compilers, etc.) exist, and the implementation language permits adequate selection from a number of host processors.
   4-Some need. One of the conditions for answer #1 is not true. There is some resulting motivation to change languages.
   6-Strong need. More than one of the conditions for #1 is not true. Continued software evolution is being constrained by the current language.

Factor D - Development Strategy Discipline

10. *Development Process Followed?* The system was created using a development process that was:
    2-Rigidly followed   4-Sometimes followed   6-Not followed
11. *Change Control Discipline Followed?* The change control procedure for the system has been:
    2-Strictly enforced   4-Loosely enforced   6-Ad hoc or does not exist

Figure 2 - MIL-HDBK-SRAH Question Set

Factor E - Quality Trend

12. *Number of Errors Trend*. Over the last 6 months, has the number of errors:
    3-Decreased   6-Leveled off   9-Increased
13. *Maintenance Backlog*. Does the system have a maintenance backlog?
    3-No   6-Yes, but steady or decreasing   9-Yes and increasing
14. *Perceived Quality Trend by Customer/User*. Do the system's users think the system quality is:
    3-Improving   6-Remaining the same   9-Declining

Factor F - System Age

15. *Age Since First Release*. What is the system's age as measured from the first release?
    1-Less than 2 years   2-2 to 7 years   3-More than 7 years

Factor G - Level of Maintenance Activity

16. *Annual Change Traffic (Past 12 Months)*. What is the annual change traffic of the system within the past 12 months?
    2-Less than 5%   4-5 to 10%   6-More than 10%
17. *Number of Change Releases (Past 5 Years)*. How many change releases have been released within the last 5 years or since the last reengineering effort, whichever is less?
    2-Less than 5   4-5 to 10   6-More than 10

Factor H - Documentation

18. *State of Documentation*. The system's documentation is best characterized as:
    3-Complete and current
    6-Mostly complete and mostly current
    9-Non-existent or inaccurate

ORGANIZATIONAL FACTORS

Factor I - Impact of System Failure

19. *Effect of System Failure*. If the system failed what would be the effect?
    3-Little or no damage   6-Significant damage   9-Permanent damage, major financial loss, or potential loss of life
20. *Contingency Plan*. Is there a contingency plan (current, recently tested, and ready at a moment's notice) which could be used if the system fails?
    3-Yes or not needed
    6-Yes, but with some difficulty and significant loss of efficiency
    9-No
21. *Contribution to User's Mission*. How much does the candidate software contribute to the using organization's mission?
    3-Not at all   6-Small percentage   9-Significant percentage

Factor J - Personnel

22. *Percent of Maintenance Personnel With In-depth System Knowledge*. For the staff currently maintaining the system: What percentage of the maintenance personnel have in-depth experience with the system?
    3-More than 30%   6-5 to 30%   9-Less than 5%
23. *Maintainer Experience*. What is the average number of years experience as a maintenance programmer for those who maintain the existing system?
    3-More than 5 years   6-2 to 5 years   9-Less than 2 years

Figure 2 - MIL-HDBK-SRAH Question Set (Cont.)

24. *Maintainer Turnover.* What is the percentage of maintenance personnel turnover per year?
3-Less than 5% 6-Between 5 and 30% 9-More than 30%
25. *Original Developers Available?* Are the original developers available for consultation?
3-Yes
6-Yes, but the system is over 5 years old, or the original developers are not easily accessible
9-No

Figure 2 - MIL-HDBK-SRAH Question Set (Cont.)

question set. In MIL-HDBK-SRAH, a questionnaire form is provided for marking responses for each question. The average response for each factor is recorded. The resulting points are totaled to provide a quantitative result representing the need to reengineer. Programs are then rank-ordered by their need to reengineer.

The question set grew out of efforts at the Software Technology Support Center. In [STSC92], Chris Sittenauer and Mike Olsem introduced a set of questions to assist in determining if a program needed to be reengineered. Minor modifications occurred during the process of gaining technical consensus among panel members during [SB-I] and in the following months. As with the entire SRAH process, a wider consensus on the question set is now being sought.

IDENTIFY REENGINEERING STRATEGIES

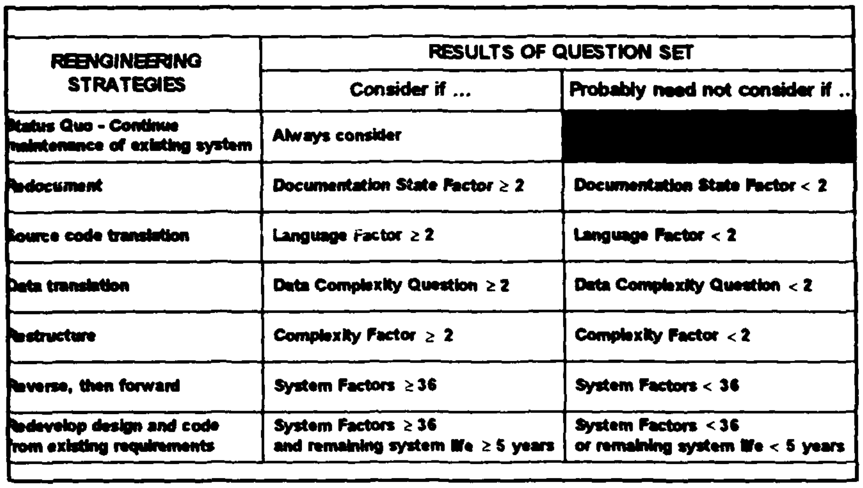
Now that the candidate programs with the greatest need to be reengineered have been identified, candidate reengineering strategies can be generated for those programs. The Reengineering Strategy Selection Matrix, shown in Figure 3, identifies candidate strategies based on the responses made to the question set in the previous section.

ECONOMIC ASSESSMENT

At this point in the MIL-HDBK-SRAH process, a set of candidate reengineering strategies has been identified for each program being considered. The remainder of the process is designed to perform an economic assessment of the strategies and to execute a management decision process to select one strategy per program and prioritize the list of programs.

The purpose of the economic assessment process is to provide accurate, traceable, and credible comparative cost information, in a consistent format for each program, to allow rank ordering of the candidate strategies according to breakeven point (BP) and return on investment (ROI).

The economic assessment approach is direct, repeatable, and logical. Each program is analyzed separately and independently. For each program, the candidate strategies are evaluated by estimating total remaining life cost (RLC), which is the sum of investment (development) and support costs for the



REENGINEERING STRATEGIES	RESULTS OF QUESTION SET	
	Consider if ...	Probably need not consider if ..
tatus Quo - Continue naintenance of existing system	Always consider	
edocument	Documentation State Factor ≥ 2	Documentation State Factor < 2
ource code translation	Language Factor ≥ 2	Language Factor < 2
ata translation	Data Complexity Question ≥ 2	Data Complexity Question < 2
estructure	Complexity Factor ≥ 2	Complexity Factor < 2
everse, then forward	System Factors ≥ 36	System Factors < 36
edevelop design and code rom existing requirements	System Factors ≥ 36 and remaining system life ≥ 5 years	System Factors < 36 or remaining system life < 5 years

Figure 3 - Reengineering Strategy Selection Matrix

defined remaining lifetime of the software. For reengineering strategies, RLC also includes the cost of supporting the legacy software during the reengineering development time. All estimates for each program will be based on the same parametric model and the same general assumptions to allow the results to be compared. The preferred strategies are defined as those strategies whose breakeven point is less than the remaining life of the legacy software.

COST ELEMENT STRUCTURE (CES)

Each economic assessment will first establish a CES to be used for the analysis. The CES may be similar to that shown in Figure 4 or may be tailored to satisfy any of the following requirements:

(1) To match the CES from a useful and earlier estimate for this program,

(2) To match the CES output from the parametric cost estimating model used (the situation experienced in the above example), or

(3) To highlight a particular cost sensitivity, e.g., to provide more granularity into the maintenance estimate.

Since the economic assessment is a comparative process, it is not necessary to establish an exhaustive CES, particularly where the cost of the elements would be the same or similar for all strategies (facilities or utilities, for example). Costs may be estimated at the CSU, CSC, or CSCI level and aggregated to program level.

MIL-HDBK-SRAH leads the analyst through a process to consider applicable ground rules and assumptions which need to be documented with the results of the assessment.

MODEL SELECTION AND DATA GATHERING

Software estimates are normally made by level of effort, analogy, or parametric methods. For the handbook, only parametric methods (cost models) are considered. Examples using several parametric models are provided as Appendices in MIL-HDBK-SRAH. Reasons for selecting a particular model may be familiarity, availability of the model or particular input data, suitability of the model CES, or a desire to be comparable with an earlier estimate. In any case, the same model should be used for all estimates to be compared. General information on model estimating and model calibration may be found in source documents listed in SRAH.

The following list identifies the models which have been identified for use in the SRAH economic assessment process:

- COCOMO
- PRICE S
- REVIC
- SEER-SEM
- SLIM
- SOFTCOST

A *Reengineering Size Adjustment* (RESIZE) model is included in SRAH as an Appendix and describes a method of adjusting source lines of code for a reengineering project for input to a cost model.

ECONOMIC INDICATORS

The economic assessment of candidate strategies for each program depends on estimating and comparing economic indicators (RLC, RLC Savings, ROI, and BP) for each strategy, using the same estimating model. Figure 5 is an illustrative example of a summary of the economic assessment of five strategies for a single program. Strategy #1,

1.0 Investment (Development)
  1.1 Software Development
    1.1.1 Requirements Analysis
    1.1.2 Preliminary Design
    1.1.3 Detailed Design
    1.1.4 CSU Code and Test
    1.1.5 CSC Integration and Test
    1.1.6 CSCI Testing
    1.1.7 System Integration and Test
    1.1.8 Operational Test & Evaluation
    1.1.9 Site preparation
    1.1.10 Development Tools
    1.1.11 Program Management
    1.1.12 Documentation
  1.2 Site Preparation
  1.3 Training
  1.4 Development Tools
  1.5 Hardware Development
2.0 Support
  2.1 Reengineered Software
    2.1.1 Software Maintenance
    2.1.2 System Operations
    2.1.3 Hardware Maintenance
    2.1.4 Training
  2.2 Legacy Software
    2.2.1 Software Maintenance
    2.2.2 System Operations
    2.2.3 Hardware Maintenance
    2.2.4 Training

Figure 4 - Generic CES

continued maintenance (Maintain Status Quo) of the legacy software, is the baseline against which the other strategies are compared. Each of the three reengineering strategies (#2, #3, #4) would require an investment (development cost) to achieve the savings shown. The final strategy (#5) would require full replacement of the legacy software and would incur the greatest investment of all candidate strategies.

The RLC Savings is defined as the RLC of Strategy #1 minus the RLC of the strategy under consideration. The breakeven point is defined as that point in time when the RLC of a strategy equals the RLC of Strategy #1, i.e., the cost of reengineering equals the cost of continuing to maintain status quo. A detailed version of the worksheet is contained in MIL-HDBK-SRAH.

In this example, the Cost Element Structure (CES) was summarized (rolled-up). Risk and sensitivity assessments were excluded from this example for simplicity. The SRAH process provides a detailed method for including risk and sensitivity assessments into the economic analysis.

From the illustrative example in Figure 5, the following conclusions can be drawn:

- Strategy #1 (Maintain Status Quo) is a low risk candidate (no investment), but its ROI (arbitrarily set at zero) places it lower than the preferred candidates. All other strategy RLCs will be compared to its RLC.

- Strategy #2 (Redocument) has the lowest investment of all strategies, but only moderate RLC Savings (as compared to #1) over the remaining life. The ROI ranking is the same as the BP ranking.

- Strategy #3 (Translate) is a poor choice, demonstrating a higher RLC than #1, the highest investment, and negative ROI values. Note also that the BP (12.8) years exceeds the remaining life (12 years). This is not a preferred strategy.

- Strategy #4 (Restructure) is clearly the preferred choice, showing the best ROI and the earliest BP.

Program: Illustrative Case FY94 $K	Candidate Strategies				
	Maintain Status Quo	Redocument	Translate Source Code	Restructure	Redevelop
Parameters					
SLOC	31,574	31,574	31,574	31,574	31,574
Design Modified (DM)	0%	0%	0%	10%	50%
Code Modified (CM)	0%	0%	100%	20%	80%
Integ & Test Modified (IM)	0%	0%	100%	100%	100%
New Documentation (ND)	0%	100%	20%	50%	80%
Equiv SLOC (ESLOC)	0	6,946	19,386	14,556	24,975
Annual Chg Traffic (ACT)	20%	19%	18%	16%	15%
Remaining Years (Y)	12.0	12.0	12.0	12.0	12.0
Reengineering Years (YR)	0.0	0.6	0.9	0.8	1
Support Years (YS)	12.0	11.4	11.1	11.2	11.0
Cost by CES					
1.0 Investment	$0	$344	$1,179	$836	$1,597
2.0 Support	$4,232	$3,650	$3,136	$2,496	$2,098
2.1 Reengineered Software	$0	$3,435	$2,818	$2,210	$1,746
2.2 Legacy Software	$4,232	$215	$319	$286	$351
Economic Indicators					
1. Remaining Life Cost (RLC)	$4,232	$3,994	$4,315	$3,331	$3,695
2. RLC Savings vs #1	$0	$239	($83)	$901	$537
3. Return On Investment (ROI)					
a. Investment ROI (ROI$_I$)	0	0.69	(0.07)	1.08	0.34
Rank Order	4	2	5	1	3
b. Total ROI (ROI$_T$)	0	0.06	(0.02)	0.27	0.15
Rank Order	4	3	5	1	2
4. Annual Support (AS)	$353	$301	$254	$197	$159
5. Annual Support Savings	$0	$52	$99	$156	$194
6. Breakeven Point (BP) in Years	12.0	7.2	12.8	6.2	8.2
Rank Order	4	2	5	1	3
7. Preferred Reengineering Strategy	N/A	Yes	No	Yes	Yes

Figure 5 - Summary Comparison Worksheet for a Single Program (Example)

- Strategy #5 (Redevelop) is clearly the greatest investment, provides only modest RLC Savings, but still displays a BP within the remaining life. It may also incur the greatest cost and schedule risk, something that should be investigated in accordance with procedures described further in MIL-HDBK-SRAH.

Ranking of strategies is assigned in ascending order of BP or ROI, with the preferred strategies being #4, #5, #2, and #1 in that order.

COST RISK AND SENSITIVITY

MIL-HDBK-SRAH includes a process for reevaluating the results of the economic assessment and assessing the associated cost risks and parameter sensitivities. Guidance for documenting the economic assessment process is provided.

MANAGEMENT DECISION PROCESS

At this point in the MIL-HDBK-SRAH process, a set of preferred strategies has been identified for each program being considered. The remainder of the process is to select one strategy per program and prioritize the list of programs. Selection and prioritization are based on the composite results of the technical and economic assessments and management considerations. Evaluation of the need to reengineer, economic indicators, and management objectives and constraints results in the recommended course of action.

While the technical and economic assessment processes are well defined in SRAH, the managemment decision process is more sensitive to less well defined and less tangible program considerations and is based primarily on judgement. Non-quantifyable program considerations include perceived risk, resource availability, requirements realism, estimate credibility, and schedule uncertainty.

The primary objectives of the management decision process are to reduce software support cost, improve software quality, and meet other management or organizational objectives.

The ideal choice for the highest priority program would be if all of the following were true:

- Greatest technical need to reengineer
- Earliest BP within the remaining life
- Highest ROI
- Investment cost within budget
- High confidence in estimate and schedule
- Remaining life confirmed
- Legacy software has the highest support cost
- High probablilty of project success.

HISTORY OF MIL-HDBK-SRAH

The Joint Logistics Commanders (JLC) Joint Policy and Coordinating Group (JPCG) on Computer Resources Management (CRM) initiated the first draft of this handbook at the First Software Reengineering Workshop [SB-I] in September 1992. The handbook was developed by the members of the Reengineering Economics Panel at the workshop and was entitled *Reengineering Economics Handbook (MIL-HDBK-REH)*. In April 1993, Comptek Federal Systems, Inc. and Management Consulting & Research, Inc., refined and enhanced the handbook under contract to the Air Force Cost Analysis Agency. Refinement and use of MIL-HDBK-SRAH continues under Joint Logistics Command and Air Force Cost Analysis Agency direction.

FOR FURTHER INFORMATION

Further information regarding MIL-HDBK-SRAH may be obtained through the Air Force Cost Analysis Agency (AFCAA) from Mr. John B. Donald, AFCAA, 1111 Jefferson Davis Hwy, Suite 403, Arlington, VA 22202; phone: (703)746-5865 or (703)692-0006; MILNET: donald@afcost.af.mil. Copies of SRAH may be obtained from the Air Force Software Technology Support Center (STSC), Hill AFB, Utah; phone: (801)777-8045; or Mr. Chris Sittenauer at the STSC; phone: (801)777-9730. Comments regarding the handbook are particularly solicited and should be sent to Mr. John Clark, COMPTEK Federal Systems, Inc., 2877 Guardian Lane, Va Beach, VA, 23452; phone: (804)463-8500; clark@comptek.mhs.compuserve.com..

REFERENCES

[SB-I] *Workshop Proceedings*, JLC-JPCG-CRM First Software Reengineering Workshop, Santa Barbara I, 21-25 September 1992

[STSC92] Chris Sittenauer and Mike Olsem, "Time to Reengineer?" *CrossTalk*, Issue 32, March 1992.

System Reengineering Evaluation:
A Design Dependent Parameter Approach

Wolter J. Fabrycky, Ph.D., P.E.
Lawrence Professor of Industrial and Systems Engineering
Virginia Polytechnic Institute and State University
Blacksburg, Virginia 24061

Abstract

Mutually exclusive reengineering design alternatives for modifying or extending complex systems must be evaluated rigorously before the most attractive option can be identified for implementation. This paper presents a Design Dependent Darameter (DDP) approach utilizing a general Design Evaluation Function (DEF) to guide the search through the redesign space. Selection of the best reengineering alternative is aided by the use of a Design Evaluation Display (DED) incorporating multiple criteria. Both the DEF and the DED will be explained and illustrated for hypothetical systems of deployed repairable equipment.

I. The system reengineering process*

A reengineering design effort is needed whenever a deficiency is recognized in an existing system, or when the system fails to perform its intended mission satisfactorily. Deficiencies occur and may be revealed for several reasons:

1) The actual observed performance of the system after deployment does not fully meet the performance requirements for which it was designed.
2) The performance requirements change, or are expanded, after the system has come into being and the system does not meet these new requirements.
3) The operating environment for which the system was originally designed has changed, making the existing system inadequate for its intended mission.

System reengineering evaluation requires the systematic application of a rigorous procedure that can guide redesign efforts toward a preferred solution. This evaluation procedure can also be useful in identifying systems that need to be reengineered. A systematic reengineering procedure is illustrated in Figure 1.

Block 1 in Figure 1 shows the deficiency in the system to be the genesis of the reengineering process. This deficiency in the existing system should be the stimulus that begins the process of reengineering. The deficiency must be clearly identified so that the reengineering effort can truly focus on the need.

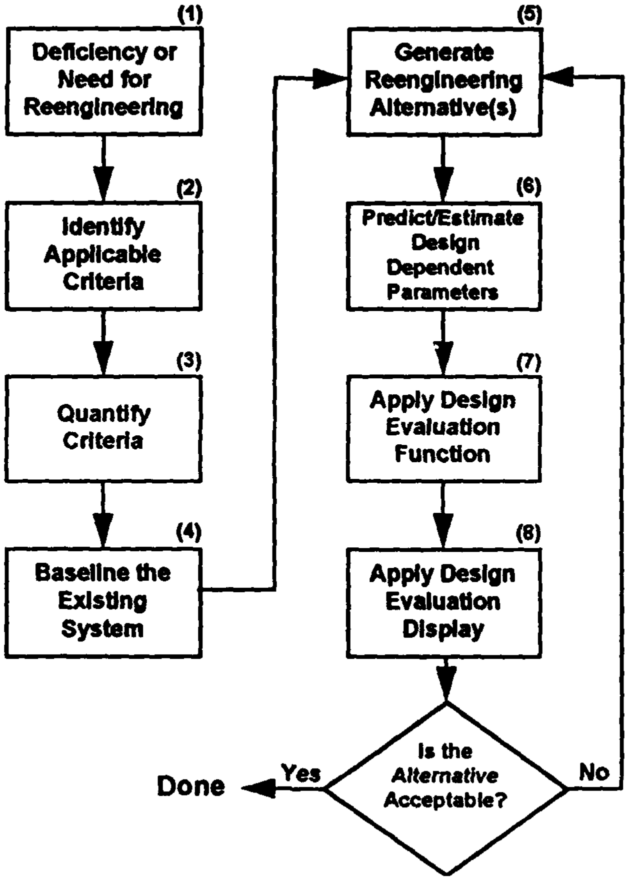

(1)
Deficiency or Need for Reengineering
(2)
Identify Applicable Criteria
(3)
Quantify Criteria
(4)
Baseline the Existing System
(5)
Generate Reengineering Alternative(s)
(6)
Predict/Estimate Design Dependent Parameters
(7)
Apply Design Evaluation Function
(8)
Apply Design Evaluation Display
Is the Alternative Acceptable?
Yes
No
Done

Figure 1 - A Systematic Reengineering Procedure

*System reengineering should be pursued as an integral part of the system engineering process. A system engineering process model is being developed at Virginia Tech by Blanchard, Bowen, Fabrycky, Nance, and Verma under NSWC Contract #N60921-89-A239-0027.

This deficiency, or need for reengineering, arises because the system does not satisfy specific mission or design criteria. These criteria must be identified as indicated in Block 2 of Figure 1. Criteria will usually be both economic and noneconomic.

Multiple criteria considerations in life-cycle reengineering analyses arise when both cost and effectiveness elements are present in the evaluation as shown in Figure 2. Effectiveness is a measure of mission fulfillment for a system in terms of the stated operational need. Mission fulfillment may be expressed by one or more figures of merit, depending on the type of system and mission objectives.

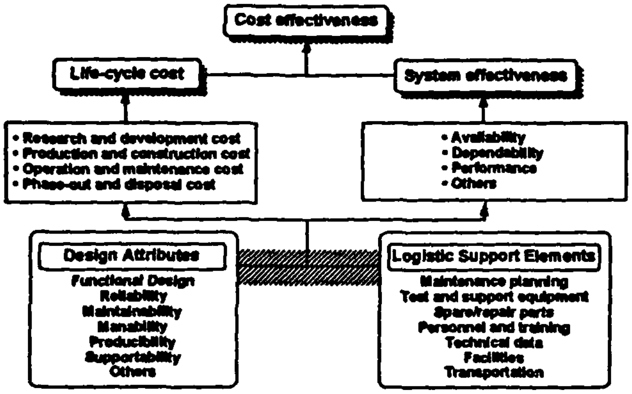


Figure 2 - Elements for Cost-Effectiveness

Next, multiple criteria must be expressed as quantifiable goals for the reengineered system. This important process is identified in Block 3 of Figure 1 and should embrace the elements in Figure 2.

Quantification is a not an easy task. Most systems have multiple purposes that significantly complicate the measurement. Criteria such as availability, reliability, maintainability, etc. are usually used since they are more easily quantified than utility, merit, gain, etc. Also, it is appropriate to establish the means with which to quantify criteria for each alternative. Indirect experimentation techniques utilizing mathematical and simulation models come into play here.

As indicated by Block 4 of Figure 1, the existing system must be baselined against established design criteria. This baselining should illuminate the deficiency in the system. Baselining is used as a basis for comparison with proposed reengineering design alternatives. In the case of an existing system, design dependent parameters such as reliability, design life, weight, etc. are fixed by the current design. If the system deficiency is caused by an operating environment change (e.g., higher operating costs, mission change, higher shortage costs), the baselining of the system must incorporate these factors in order for the system deficiency to be accurately identified.

After baselining of the existing system is complete, the process of reengineering design may begin. The effectiveness of these designs must be evaluated so that comparison with competing design alternatives and the baseline is possible. Only when each reengineering design has been evaluated can the best option be selected for implementation.

II. Reengineering design evaluation

Reengineering design evaluation is part of the system reengineering process of Figure 1. Blocks 5 through 8 in Figure 1 illustrate the remaining steps.

After deficiencies in the existing system have been identified, changes (reengineering design options) must be developed and presented as mutually exclusive alternatives. Cost and effectiveness measures for these alternatives are generated using the established criteria. Each alternative is then displayed beside the baselined existing system, and the lowest cost alternative that most closely meets other criteria is selected as the best reengineered system design.

Figure 3 illustrates a concurrent approach to the redesign generation and evaluation portion of the system reengineering process. In Blocks 1 and 2 of Figure 3, human designers (using CAD/CAE tools) develop reengineering design alternatives. This activity corresponds to Block 5 in Figure 1.

Block 6 in Figure 1 identifies the activity of estimating and predicting Design Dependent Parameter values for each alternative (see also Block 3 in Figure 3). These parameter values provide a basis for comparison with established design criteria to determine the merit of each alternative. They may be passed to Block 1 in Figure 3 as appropriate.

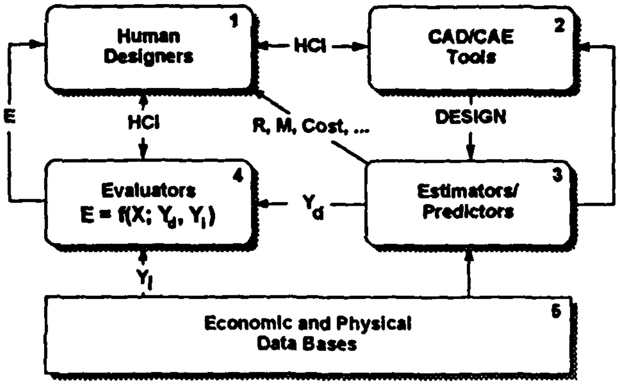


Figure 3 - Concurrent Engineering Morphology

Redesign alternatives must also be judged from a system life-cycle perspective. In Block 7 of Figure 1, the cost-effectiveness of each alternative is determined by using a life-cycle oriented evaluation function (also see Block 4 in Figure 3). Only after each alternative has been rigorously evaluated can the least costly be selected which satisfies all other criteria.

The evaluated alternative is compared to the baseline (or to the best alternative at this point) as shown in Block 8 of Figure 1 (also see the link between Block 4 and Block 1 in Figure 3). In these situations, decision evaluation is facilitated by the use of a decision evaluation aid visually exhibiting both cost and effectiveness measures. Life-cycle cost and one or more effectiveness measures may be displayed simultaneously as an aid in decision making. A Decision Evaluation Display (DED), as shown in Figure 4, is one way of doing this.

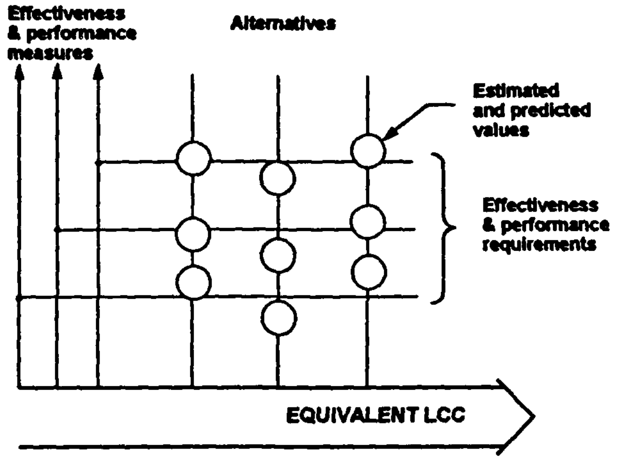

Effectiveness & performance measures
Alternatives
Estimated and predicted values
Effectiveness & performance requirements
EQUIVALENT LCC

Figure 4 - The Decision Evaluation Display

Note that effectiveness requirements (or thresholds) are shown on the DED. These are useful to the decision maker in assessing (subjectively) the degree to which each alternative satisfies effectiveness criteria. Each alternative is displayed with its effectiveness measures shown in relation to the established criteria. Life-cycle cost, shown on the horizontal axis, is an objective measure. The goal is to select the alternative with the lowest life-cycle cost that best satisfies important performance and effectiveness measures.

III. Reengineering evaluation theory

Models and the process of indirect experimentation provide a convenient means for obtaining factual information about a new system being designed, or an existing system which needs to be improved. In most design and operational situations, the objective sought is the optimization of effectiveness measures economically. Rarely, if ever, can this be done by direct experimentation with a system under development or with one that already exists. The primary use of simulation in any system reengineering effort is to explore the effects of alternative system characteristics on system performance and effectiveness, without actually producing and testing each candidate system.

In system reengineering, mathematical models may be used during design evaluation of the baseline (existing system). Then, these models can be invoked for each redesign alternative. Design Dependent Parameters (DDP) are the key. These parameters are design characteristics inherent in the physical equipment which are subject to manipulation by the designer during the process of seeking the best design. When imbedded in models, these parameters are the key to indirect experimentation during the system reengineering process.

Design evaluation in terms of life-cycle cost and system effectiveness can be facilitated by utilizing the Design Evaluation Function (DEF) shown in Figure 5. This function is a mathematical way to link design actions with operational outcomes. It incorporates Design Dependent Darameters. From the following definitions of terms in the DEF, the structure for system redesign optimization should be evident:

E = a life-cycle complete evaluation measure (usually equivalent life-cycle cost)

X = design variables (e.g., number of deployed units, armor thickness, retirement age, repair capacity, rated thrust, etc.)

Y_d = design dependent parameters (e.g., weight, reliability, design life, load capacity, producibility, maintainability, etc.)

Y_i = design independent parameters (e.g., cost of money, labor rates, material cost per unit, shortage cost penalty, etc.)

To Optimally Achieve

ACQUISITION / UTILIZATION

Mathematically Link

DESIGN / OPERATIONS

Utilizing The

DESIGN EVALUATION FUNCTION (DEF)

For Choosing Among Design Alternatives

$$E = f(X; Y_d, Y_i)$$

Figure 5 - The Design Evaluation Function

The Design Evaluation Function, with its Design Dependent Parameters and design independent parameters, facilitates optimization within each alternative. It provides the basis for a classification of the true difference between redesign alternatives (a design-based choice) and optimization (a search-based choice).

Reengineering of the system can take many forms. The system might be reengineered by reoptimizing the current design's logistics support system. Increasing the number of units deployed, or the number of repair channels, might provide the needed change to overcome the deficiency. However, the examples presented here are for cases where reengineering consists of redesign of the physical equipment to change Design Dependent Parameter values.

Two cases are considered. The first example is a situation where the reengineering problem arises because actual observed performance measures of a system do not meet established criteria. As shown in Figure 6, the baseline's estimated performance does not accurately predict the actual effectiveness and performance of the system and its life-cycle cost (Case A).

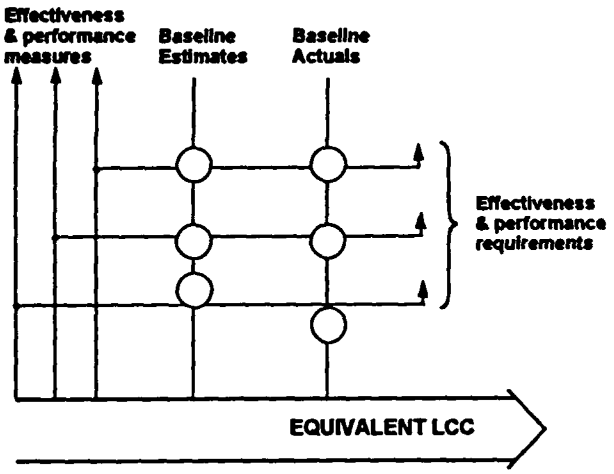

Effectiveness & performance measures
Baseline Estimates
Baseline Actuals
Effectiveness & performance requirements
EQUIVALENT LCC

Figure 6 - DED for Redesign Case A

For this reengineering case, design requirements are assumed not to change since the original design effort, but the actual deployed system fails to meet one or more of the requirements. Thus, reengineering of the system is indicated. Redesigned system alternatives should be offered to compete with the baseline (the existing system).

In this situation, the methods of predicting performance measures used during design need to be redeveloped. Since the system failed to perform as predicted, the prediction methods are probably deficient. These methods must be corrected to account for errors in the original design predictions and estimates. The methods used need to be refined so that estimated performance matches actual performance.

The second example is a situation where the system requirements change (tougher performance criteria or new criteria) from the requirements used in the original design. This situation is shown in Figure 7, where actual values of the baseline's measures no longer meet all of the new criteria (Case B). As represented in Figure 7 by a movement down of one of the horizontal lines, one of the design specifications changes. This will cause the system, which may have met all original requirements, not to meet all new requirements.

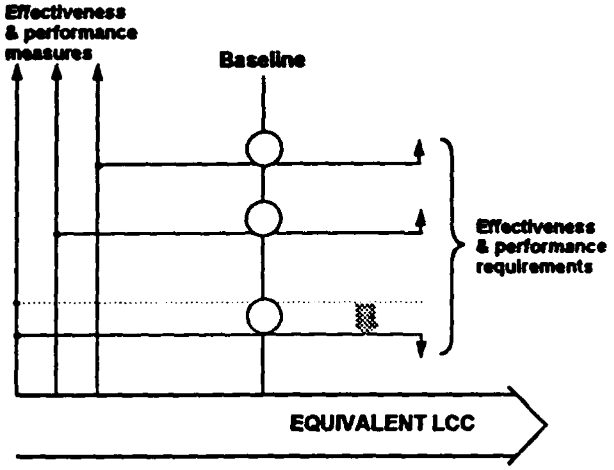

Effectiveness & performance measures
Baseline
Effectiveness & performance requirements
EQUIVALENT LCC

Figure 7 - DED for Redesign Case B

Both cases outlined above will be illustrated by a hypothetical situation involving a population of repairable equipment which is deployed to meet a demand. This hypothetical example situation is called the Repaiarble Equipment Population System (REPS).

IV. Hypothetical reengineering examples

Two hypothetical, but realistic, REPS examples will be presented in this section. These were fashioned to illustrate how modeling and indirect experimentation for system optimization applies to system reengineering design evaluation for both a Case A and a Case B situation.

REPS example overview

Consider the following situation: A finite population of repairable equipment is maintained in operation to meet a demand. As equipment units fail or become unserviceable, they are repaired and returned to service. As they age, the older units are removed from the system and replaced with new units. The problem is to determine the population size, the replacement age of units, and the number of repair channels for a given design alternative, or set of design dependent parameter values, so that requirements will be met at a minimum life-cycle cost.

A general schematic of REPS is shown in Figure 8. Repairable equipment systems exist in many operational settings. Both the military and the airlines operate and maintain aircraft with these system characteristics. In ground transit, vehicles such as rental automobiles, taxis, and commercial trucks constitute repairable item systems. Production equipment types such as autoclaves, drill presses, and weaving looms, are populations of equipment which fit the repairable classification. Additionally, both military and commercial organizations maintain populations of repairable system components such as motors, pumps, valves, etc.

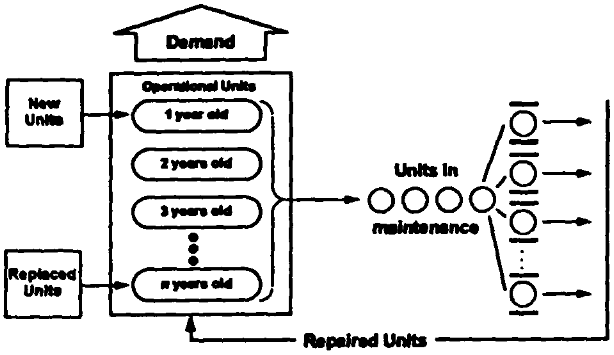


Figure 8 - Repairable Equipment Population System

Table 1 summarizes the design variables and system parameters for REPS. The Design Dependent Parameters of unit cost (C_u), reliability (MTBF), and maintainability (MTTR), are central to the system design problem. DDP values are inherent in the design of the equipment itself. Design independent parameter values depend solely on the operating environment of the equipment. They include demand (D), shortage cost (C_s), interest rate (i), etc. Design variables, on the other hand, are those factors that are adjustable and used to optimize within each design alternative to find the lowest life-cycle cost.

Table 1 - Design Variables and System Parameters

Variables/Parameters	Design Var.	Design Dep.	Design Ind.
D = demand in units			X
N = number of units to deploy	X		
M = number of maintenance channels	X		
n = retirement age	X		
C_u = annual equivalent units cost per unit		X	
C_r = annual equivalent channel cost per channel			X
A_r = annual overhead cost per channel			X
C_s = shortage cost per unit short per year			X
MTBF = mean time between failure		X	
MTTR = mean time to repair a unit		X	

Figure 9 illustrates, graphically, the optimization process based on three REPS design variables: number of deployed units (N), retirement age (n), and number of repair channels (M). Various combinations of values for these system design variables are searched until the lowest life-cycle cost for the alternative is found. This life-cycle cost results from the optimal mix of values for the design variables for this alternative.

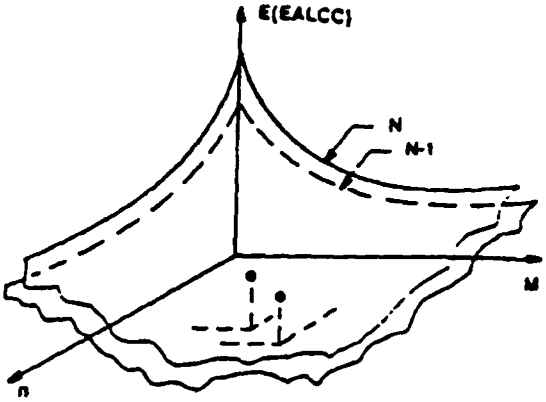


Figure 9 - REPS Alternative Optimization

Indirect experimentation with REPS may proceed on the basis of the Design Evaluation Function of Figure 5, E = f(X; Y_d, Y_i). For REPS, annual equivalent life-cycle cost (AELCC) will be used as the *economic metric*. AELCC is the total of population annual equivalent cost (PC), repair facility annual cost (RC), annual operating cost (OC), and annual shortage cost (SC), expressed as:

$$AELCC = PC + RC + OC + SC.$$

Population Annual Equivalent Cost is PC = C_u(N) where C_u = (P-B)(A/P, i, n) + B(i). This annual equivalency formula uses first cost per unit (P) and its book value at retirement (B). In these examples, book value is determined using salvage value and straight-line depreciation. Repair Facility Annual Cost is RC = C_r(M). Annual Operating Cost is OC = (EC + LC + PMC + Other)(N); energy costs, labor costs, preventive maintenance costs, and other. Annual Shortage Cost is SC = C_s[E(S)] where E(S) is the expected number of units short. E(S) is found by multiplying the number of units short by the probability of that occurrence and summing across all instances. Refer to the Appendix for detailed derivation of elements comprising AELCC.

Calculations are done repeatedly with various combinations of N, n, and M until an optimum AELCC is achieved for each alternative (each instance of DDP values). Each alternative can then be compared to the current best to determine if all criteria have been met with AELCC reduced as far as possible.

REPS redesign example (Case A)

Consider a deployed REPS for which the system attributes are as listed in Table 2. Suppose that these are the actual performance measures experienced by REPS after deployment. Also, assume that the criteria for this system during its original design are still in effect. They are as follows:

1) Design to cost - the deployed population shall have a first cost not exceeding $1,000K.
2) Probability of shortages - the probability of one or more equipment units short of demand shall not exceed 0.38.
3) Equipment reliability - the mean time between failure for equipment units shall not be less than 0.25 years.

Table 2 - Attributes for Baseline Design

Attributes	Baseline Actuals
Design Variable Values:	
Population, N	20
Repair channels, M	4
Retirement age, n	4
Independent Parameters:	
Demand	15
Shortage cost per year	$73,000
Interest rate	10%
Repair channel cost	$45,000
Dependent Parameters:	
Design life in years	$43,000
Salvage value	$5,000
Design life in years	6
Operating costs	$2,300
Measured/Calculated:	
Prob. one or more short	0.27
AELCC	$468K
Average MTBF	0.22*
Average MTTR	0.045

*Violation of criteria

The actual reliability does not meet the specified criteria. During the original design effort, the system performance estimates met all the requirements. However, after deployment, a deficiency was discovered in the MTBF prediction. The equipment did not perform as well as predicted (a Case A situation). It is experiencing a MTBF of only 0.22 years.

A redesign alternative must be offered to compete with the baseline to correct for the deficiency. Table 3 shows the alternative's redesign dependent parameters. The independent parameters are not exhibited, since they are the same for both the baseline and the alternative.

Optimization of the REPS system for the alternative set of design dependent parameters follows. The needed calculations are given in the Appendix for detailed consideration and study.

Table 3 - Parameter Values for Alternative

Parameter	Value	
Cost of equipment unit	$52,000	
Salvage value	$7,000	
Design life in years	6	
Operating costs	$1,750	
Age cohorts	MTBF	MTTR
0-1	0.20	0.03
1-2	0.24	0.04
2-3	0.29	0.05
3-4	0.29	0.05
4-5	0.26	0.06
5-6	0.22	0.07

Table 4 - Points in the Optimum Region

Retirement age, *n*	Number of units, N	Number of repair channels, M		
		2	3	4
3	19	$598K	$466K	$469K
	18	$593K	$465K	$466K
4	19	$600K	$463K	$464K
	20	$611K	$467K	$469K
5	19	$643K	$480K	$467K

Table 4 shows points around the optimum region. As shown, the point for AELCC = $463K is optimum for this alternative, since all others are higher. A summary of the calculated and optimal design variables for the alternative design is shown in Table 5 (also see the Appendix).

This REPS alternative meets all established criteria according to the estimating methods used. Care must be taken, though, to investigate the source of the original error in predicting the MTBF. If this prediction error is

Table 5 - Summary Outputs for Redesign

Output Item	Value
Population first cost	$988,000
Annual operating cost	$33,250
Annual repair facility cost	$135,000
Annual shortage penalty cost	$73,484
Expected (AELCC)	$463,350
Mean MTBF	0.26
Probability of one or more short	0.38
Deployed units, N	19
Repair channels, M	3
Retirement age, *n*	4

not corrected for use in the reengineering effort, the alternative may perform no better than is being experienced with the existing design.

Comparison of the alternative design and the baseline is now possible. It is facilitated by a Design Evaluation Display (DED). As the DED shown in Figure 10 illustrates, the existing system (baseline) does not satisfy the established criteria. However, the reengineered design alternative does meet the criteria. Although the initial procurement cost increased by $128K, the AELCC decreased by $5K per year.

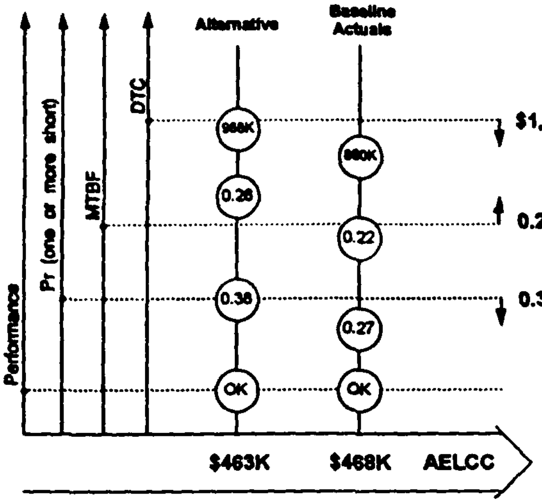


Figure 10 - Design Evaluation Display for Case A

REPS redesign example (Case B)

The system requirements for this example are as follows:

1) Design to cost - the deployed population shall have a first cost not exceeding $800K.
2) Probability of shortages - the probability of one or more equipment units short of demand shall not exceed 0.30.
3) Equipment reliability - the mean time between failure for equipment units shall not be less than 0.25 years.

In this case, suppose that the existing system must be reengineered because one of its mission requirements changed from a MTBF of at least 0.2 years to a MTBF of at least 0.25 years. Since the baseline system has an MTBF of 0.2 years, the current deployed system does not now meet all effectiveness criteria (a Case B situation).

The system attributes for the baseline design, and candidate alternatives, are given in Table 6. The baseline's MTBF is shown to be 0.20 years, which violates the new requirement. It must be at least 0.25 years. The alternatives are offered for comparison with the baseline, since both redesigns improve the system MTBF. Design variables were optimized to find the lowest life-cycle cost for each alternative. Table 7 shows the optimized effectiveness measures and calculated values for both alternatives.

Table 6 - System Attributes for Case B

System Parameters	Baseline Actuals	Alt. 1	Alt. 2
Independent Parameters:			
Demand	10	10	10
Shortage cost per day	$50,000	$50,000	$50,000
Interest rate	10%	10%	10%
Repair channel cost	$10,000	$10,000	$10,000
Dependent Parameters:			
Design life in years	5	5	5
Salvage value	$7,000	$6,500	$7,500
Cost of equipment unit	$60,000	$68,000	$72,000
Operating costs	$1,500	$1,500	$1,200
Average MTBF	0.200	0.285	0.330
Average MTTR	0.032	0.037	0.032

Table 7 - Optimized Outputs for Alternatives

Output Item	Alternative 1	Alternative 2
Population cost	$748,000	$860,000
Repair facility cost	$30,000	$20,000
Shortage penalty cost	$29,000	$24,000
Expected (AELCC)	$260,000	$253,000
Probability of one or more short	0.376	0.297
Mean MTBF	0.38	0.30
Deployed units, N	11	11
Repair channels, M	3	2
Retirement age, *n*	5	5

A comparison of the two alternatives with the baseline is shown in Figure 11. As this DED shows, the baseline design does not meet the changed MTBF criteria. Alternative 1, while meeung the MTBF criteria, fails to achieve a low enough probability of one or more units short. Alternative 2, however, meets all criteria and should be selected as the best redesign alternative.

V. Other reengineering issues

This paper deals with those aspects of system reengineering where Design Dependent Parameter values of deployed equipment is the main focus. However, reengineering efforts can concentrate on other system factors, such as system design variable values, so as to optimize the system in operation.

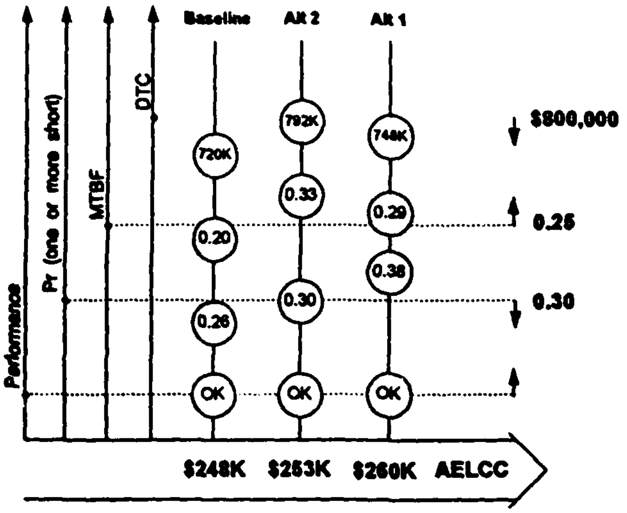


Figure 11 - Design Evaluation Display for Case B

Reengineering of a system because of a deficiency may not always mean redesigning the physical equipment itself. For example, if a system is found to need reengineering, one approach may be to deploy more units, add to the repair capability, or otherwise change the support environment so that mission requirements are more closely met. It would be desirable (and probably less costly) for a system deficiency to be resolved by reoptimizing the support subsystem. However, this is not feasible if the deficiency in the system is one traceable to a major flaw in design, or to a mission change.

Also of consideration, but not discussed here, is the extent to which the existing system is deployed. It must be determined what costs for the existing system are sunk and which ones are not. All sunk costs cannot be considered when comparing the baseline to competing alternatives. Likewise, the costs of the alternatives may not include a total initial deployment cost if the alternative is a retrofit of the existing system. For the case of a total redeployment of equipment as the alternative, the salvage value of the existing equipment can be taken as the first cost of the new equipment to help determine the cost of the redesigned alternative.

VI. Summary and Conclusions

In these times of tight defense budgets, the military services are searching for ways to gain a sustainable utilization advantage for existing systems. System reengineering design, embracing system optimization by indirect experimentation, has an excellent chance of enhancing system acceptability through the integration of important design consideration (performance, cost, and quality). But these desiderata may not be attainable unless the importance of system evaluation involving optimization is recognized and implemented as part of the reengineering process.

Optimization may be formalized for redesign by the identification of Design Dependent Parameters and the use of the Design Evaluation Function. By incorporating life-cycle factors into this function, reengineering alternatives can be compared equivalently. System improvements through redesign may be identified for implementation by use of the Design Evaluation Display.

References

1. Blanchard, B. S. and W. J. Fabrycky, **Systems Engineering and Analysis**, Second Edition, Prentice-Hall, Inc., 1990.
2. Fabrycky, W. J. and B. S. Blanchard, **Life-Cycle Cost and Economic Analysis**, Pretice-Hall, Inc., 1991.
3. Fabrycky, W. J., "Indirect Experimentation for System Optimization: A Paradigm Based on Design Dependent Parameters," **Proceedings, Second Annual International Symposium**, National Council on Systems Engineering, Seattle, Washington, July 1992.
4. Olsen, M. R. and C. Sittenauer, "Reengineering Technology Report", STSC, Hill Air Force Base, Utah, August 1993.

Credits

The author wishes to thank Mr. Shawn Looney, an undergraduate student in Aerospace Engineering, for helping with the examples and preparation of this paper. Credit for the PC software behind the REPS reengineering examples is due the Systems Engineering Design Laboratory. Copies of the REPS software are available for educational use, without charge, from:

Systems Engineering Design Laboratory
ISE - 146 Whittemore Hall
Virginia Polytechnic Institute and State University
Blacksburg, Virginia 24061

APPENDIX

This Appendix details a sample calculation of Annual Equivalent Life-Cycle Cost (AELCC) for the alternative in the hypothetical example (Case A) of Section IV. The AELCC calculation will utilize independent parameters from Table 2, the dependent parameters from Table 3, and optimum values for design variables from Table 4.

Recall that Annual Equivalent Life-Cycle Cost is:

$$AELCC = PC + OC + RC + SC$$

Each component of AELCC is derived next.

Annual Equivalent Population Cost:

Compute the book value, B, of the units at a 4 year retirement age as an input to C_u.

$$B = \$52,000 - 4\left(\frac{\$52,000 - \$7,000}{6}\right) = \$22,000$$

$$C_u = (P-B)(A/P, i, n) + B(i)$$
$$C_u = (\$52,000 - \$22,000)(0.3155) + \$22,000(0.10)$$
$$= \$16,404 - \$4,740 = \$11,664$$

$$PC = C_u(N)$$
$$PC = \$11,664(19) = \$221,616$$

Annual Operating Cost:

$$OC = (EC + LC + PMC + Other)(N)$$
$$OC = \$1,750(19) = \$33,250$$

Annual Repair Facility Cost:

$$RC = C_r(M)$$
$$RC = \$45,000(3) = \$135,000$$

Annual Shortage Cost:

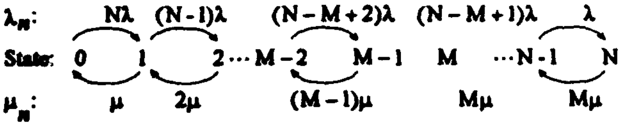
Shortage Cost is SC = $C_s[E(S)]$, where E(S) is the expected number of units short. The expected number of units short can be found from the probability distribution of *n* units short, P_n. Let

N = number of units in the population
M = number of service channels in the repair facility
λ = failure rate of a unit, 1/MTBF
μ = repair rate of a repair channel, 1/MTTR
n = number of failed units
P_n = steady-state probability of *n* failed units
P_0 = probability that no untis failed
$M\mu$ = maximum possible repair rate
λ_n = failure rate when *n* units already failed
μ_n = repair rate when *n* units already failed

The failure rate of a unit is expressed as λ = 1/MTBF and the failure rate of the entire population when *n* units already failed can be expressed as $\lambda_n = (N - n)\lambda$, where N - *n* is the number of operational units, each of which fails at a rate of λ. Similarly, the repair rate of a repair channel is expressed as μ = 1/MTTR, and the repair rate of the entire repair facility when *n* units have already failed can be expressed as

$$\mu_n = \begin{cases} n\mu & \text{if } n \in 1, 2, \dots, M-1 \\ M\mu & \text{if } n \in M, M+1, \dots, N \end{cases}$$

An analysis using a birth-death processes is employed to determine the probability distribution, P_n, for the number of failed units. In the birth-death process, the state of the system is the number of failed units (state = 0, 1, 2,..., N). The rates of change between the states are the breakdown rate, λ_n, and the repair rate, μ_n. This gives



Assuming steady-state operation of the system, this yields

$$N\lambda P_0 = \mu P_1$$
$$N\lambda P_0 + 2\mu P_2 = [\mu + (N-1)\lambda]P_1$$
$$(N-1)\lambda P_1 + 3\mu P_3 = [2\mu + (N-2)\lambda]P_2$$
$$\vdots$$
$$(N-M+2)\lambda P_{M-2} + M\mu P_M = [(M-1)\mu + (N-M+1)\lambda]P_{M-1}$$
$$\vdots$$
$$2\lambda P_{N-2} + M\mu P_N = (M\mu + \lambda)P_{N-1}$$
$$\lambda P_{N-1} = M\mu P_N$$

Additionally,

$$\sum_{n=0}^{N} P_n = 1$$

Solving these balance equations gives

$$P_0 = \left(\sum_{n=0}^{N} C_n\right)^{-1}$$

where

$$C_n = \begin{cases} \dfrac{N!}{(N-n)!n!}\left(\dfrac{\lambda}{\mu}\right)^n & \text{if } n = 0,1,2,\dots,M \\ \dfrac{N!}{(N-n)!M!M^{n-M}}\left(\dfrac{\lambda}{\mu}\right)^n & \text{if } n = M+1, M+2, \dots, N \end{cases}$$

These can now be used to find the steady-state probability of *n* failed units as $P_n = P_0C_n$ for *n* = 0, 1, 2,..., N.

Define the quantity N - D as the number of extra units to be held in the population. For n = 0, 1, 2,..., N - D there is no shortage of units. However, when

$n = N - D + 1$ a shortage of 1 unit exists
$n = N - D + 2$ a shortage of 2 units exists
$n = N - D + 3$ a shortage of 3 units exists
$\vdots$
$n = N$ a shortage of D units exists

The expected number of units short, E(S), can be found by multiplying the number of units short by the probability of that occurence and summing across all instances as

$$E(S) = \sum_{j=1}^{D} jP_{(N-D+j)}$$

The calculation of the shortage cost for the alternative in the Case A example is based on the MTBF and MTTR values in Table 3 for years 1 to 4. From these values, the average MTBF and MTTR for the population can be computed as

$$\text{MTBF} = (0.20 + 0.24 + 0.29 + 0.29)/4 = 0.2550$$

$$\text{MTTR} = (0.03 + 0.04 + 0.05 + 0.05)/4 = 0.0425$$

The failure rate of a unit and the repair rate at a repair channel are given by

$$\lambda = \frac{1}{\text{MTBF}} = \frac{1}{0.2550} = 3.9215$$

$$\mu = \frac{1}{\text{MTTR}} = \frac{1}{0.0425} = 23.5294$$

yielding $\lambda/\mu = 1/6$.

Then, compute C_n for $n = 0, 1,\ldots,3$ as

$$C_0 = \frac{19!(1/6)^0}{19!0!} = 1$$

$$C_1 = \frac{19!(1/6)^1}{18!1!} = 3.1665$$

$$C_2 = \frac{19!(1/6)^2}{17!2!} = 4.7496$$

$$C_3 = \frac{19!(1/6)^3}{16!3!} = 4.4856$$

Next, compute C_n for $n = 4, 5,\ldots,19$ as

$$C_4 = \frac{19!(1/6)^4}{15!3!3^1} = 3.9813$$

$$C_5 = \frac{19!(1/6)^5}{14!3!3^2} = 3.3224$$

$$C_6 = \frac{19!(1/6)^6}{13!3!3^3} = 2.5840$$

$$\vdots$$

$$C_{18} = \frac{19!(1/6)^{18}}{1!3!3^{15}} = 0.0000$$

$$C_{19} = \frac{19!(1/6)^{19}}{0!3!3^{16}} = 0.0000$$

Now $$\sum_{n=0}^{19} C_n = 27.9390$$

and $$P_0 = \frac{1}{\sum_{n=0}^{19} C_n} = \frac{1}{27.9390} = 0.0358$$

P_n for $n = 1, 2,\ldots,$ N can now be computed from $P_n = P_0C_n = (0.0358)C_n$ as

$$P_0 = 0.0358 \times 1 = 0.0358$$

$$P_1 = 0.0358 \times 3.1665 = 0.1134$$

$$\vdots$$

$$P_5 = 0.0358 \times 3.3224 = 0.1189$$

$$P_6 = 0.0358 \times 2.5840 = 0.0925$$

$$\vdots$$

$$P_{18} = 0.0358 \times 0.0000 = 0.0000$$

$$P_{19} = 0.0358 \times 0.0000 = 0.0000$$

The expected number of units short can be calculated as

$$\begin{aligned} E(S) &= \sum_{j=1}^{D} jP_{(N-D+j)} = \sum_{j=1}^{15} jP_{(4+j)} \\ &= 1(0.1189) + 2(0.0925) + \cdots + 15(0.0000) \\ &= 1.00663 \end{aligned}$$

From which Annual Shortage Cost is

$$SC = Cs[E(S)]$$

$$SC = \$73{,}000(1.00663) = \$73{,}484$$

The Annual Equivalent Life-Cycle Cost is now

$$\$221{,}616 + \$33{,}250 + \$135{,}000 + \$73{,}484 = \$463{,}350$$

The shortage probability distribution can be calculated from the P_n values and plotted as a histogram of Pr(S = s) = N - D + s. In this example, $\Pr(S = s) = P_{4+s}$.

$$\Pr(S = 0) = 0.622$$
$$\Pr(S = 1) = 0.119$$
$$\Pr(S = 2) = 0.093$$
$$\Pr(S = 3) = 0.067$$
$$\Pr(S = 4) = 0.046$$
$$\Pr(S = 5) = 0.027$$
$$\Pr(S = 6) = 0.015$$
$$\Pr(S = 7) = 0.008$$
$$\Pr(S = 8) = 0.003$$

The probability of one or more units short can be found to be

$$1 - \Pr(S = 0) = 1 - 0.62 = 0.38$$

as was shown in Table 5.

METRICS FOR REENGINEERING OF SOFTWARE SYSTEMS

by
Annette R. Ashton (K52) & William H. Farr (B10)
Naval Surface Warfare Center, Dahlgren Division
Dahlgren, VA 22448-5000

INTRODUCTION

With the impending budget cuts hanging over us many federal agencies, especially in the Defense Department, are evaluating the efficiency of their development process. Reengineering of all or part of a current software system is becoming a way of life. Billions of dollars have been invested in the development of systems that may need to be modified and extended to respond to changing requirements. Reengineering technology is necessary if we are to benefit from these extensive investments.

Robert Grady in his book on software metrics for project and process improvement, [Gra92], states, "Because we cannot economically replace all our old software, we must find better ways to manage needed changes. Until we do, software maintenance will continue to represent a large investment and software quality will *not* improve." Grady further states that we spend 2 to 3 times as much effort maintaining and enhancing software as we spend creating new software. In order to achieve feats of reusing current software systems one must better define his software development process and an integral part of this process improvement is the collection, analysis, and feedback of software metrics.

"Most of the total change to the software system will be located in only a few modules" was the finding of Warren Harrison. He further stated that "we were surprised to find that 10% of the modules that were changed accounted for over 60% of the maintenance activity." [Har90] This is a good example of where collecting simple metrics gave the developer an insight he might not overwise have had.

"Management can use metrics to understand and control the maintenance process. As changes are requested, measurements can be made, impact assessed, and implementation decisions made. The more we understand the impact, the less risk we take when making each change and the better that we can control software degradation resulting from change." [Pfl90]

This paper will give some basic definitions of software metrics and their role in the software development lifecycle with special emphasis on how they may be used in making decisions related to reengineering principles. We will further give definitions, tips on using the metrics in your day-to-day tasks, and graphical representations of the metrics to help evaluate your software development process. In addition, the paper will provide some metrics that are useful for determining when an existing software system has deteriorated to the point when reengineering must be considered.

To set a framework for our metrics discussion we will break the software development process into the five traditional system development phases:

1. requirements definition and analysis
2. design development
3. code and unit testing
4. system integration and testing
5. verification and validation
6. delivery and operational use

Questions that need to be addressed for each phase will be given along with the associated metrics to help answer those questions. More detailed explanations of the metrics as well as examples of how they can be used to evaluate the software system and/or its development process are provided in the appendix. This paper is not intended to provide an all inclusive list of metrics, but ones that we have found useful within our software development environment. The important consideration is to use those metrics that are most appropriate to your environment and the development process it follows. The reader can select any or none of the metrics considered here. For those selected, the reader is free to adapt the definitions and implementation to their own process. The issue is to get a handle on the development process and the products produced, otherwise demonstrated improvement is not possible. Measurement provides a basis of showing you where you are, where you've come from, and it provides a road map to lead you to where you want to go!

REQUIREMENT'S DEFINITION AND ANALYSIS PHASE

"Defective requirements are a dominant cause of cost and schedule overrun in defense and aerospace programs." [Hal93] Using measurements to track progress and quality of your requirements may help answer some of the following questions:

1. "Can we include all these new maintenance requirements and still meet our schedule with our reduced staff?"
2. "What percentage of the software modules can be reused if only these requirements modifications are made?"
3. "If our goal is to reuse 40% of the system's resources, can we accomplish all the needed modifications to meet the requirements?"

Some specific requirement metrics that could help answer these questions are:

a. Requirements Size (REQSIZE) - the number of requirements for a given Computer Software Configuration Item (CSCI), and
b. Requirements Changes (REQCHG) - the percentage of requirements of a specific CSCI that have changed (modified, added, or deleted) from one baseline/revision to another.

Using these metrics one could determine the magnitude of the change and on the basis of that determine resource allocations for design, coding, and testing. To aid in this determination one should establish a metrics database to store these types of metrics so that better estimates can be obtained using past similar development projects. The Appendix contains a detailed description of these metrics and illustrates in more detail how they can be employed.

DESIGN DEVELOPMENT PHASE

The following questions that might arise during the design development phase of the lifecycle:

1. "Given this set of requirements changing, can we complete our design within our given schedule with our reduced staff?"
2. "What percentage of the software design can be reused given the changed requirements?"
3. "If we achieved 40% reuse last year, what can we do to increase it this year?"

Metrics that could help answer these questions are:

a. Design Size (DESSIZE) - the number of design units of a CSCI. By a design unit we might mean the number of lines of Program Design Language (PDL) or the number of bubbles/arcs within a Data Flow Diagram for the proposed changes. You need some type of size measurement along these lines to again determine the magnitude of the changes.
b. Design Changes (DESCHG) - the percent of the original design that has changed (modified, added, or deleted) from one baseline/revision to another. The same design units considered for DESSIZE are employed here.

Again these metrics allow one to assess the impact of the changes for staffing, schedule, and cost determination. (See the Appendix for further elaboration.)

It is at this stage that performance analysis may also be addressed in the reengineered system through the use of rapid prototyping and simulation. A number of new tools are under development (NSWCDD's DESTINATION, Advanced System Technologies' QUEST and Conceptual Software Systems' ExpeR/T among others) that will take a design and generate a simulation representation of the system for analysis purposes.

CODE/UNIT TEST AND SYSTEM INTEGRATION/TEST PHASES

With the design in place, unit coding and testing begins followed by the system's integration and its associated testing. Using reengineering technology brings the challenge of pulling together old and new code into one software product. Keeping track of which modules are reused and which are updated is one way toward raising the awareness of the reuse issue. An awareness of the current development status and your well-defined goals, will often show you what reengineering steps must be taken to achieve those goals.

Metrics that could help here are:

a. The software program's size. Some specific examples that we use are: KSLOC - the number of executable lines of code divided by 1000 and KTLOC - the total number of lines of code, (including blanks and comments) divided by 1000. Many different measurements can be used for sizing other than the ones presented. The important point is to decide on a common definition and then use it consistently within the organization so that a common frame of refe.ence is obtained. These measurements can also be further broken down to whatever level is appropriate for the software system. For example, the Computer Software Unit, (CSU), the smallest compilable set of code, may be appropriate.
b. The number of modules or CSU's, (NUMMOD).
c. The percentage of CSU's changing (MODCHG). This metric is especially important in the reengineering process if reusability is emphasized.
d. The percentage of CSU's tested and the percentage of requirements tested. These metrics can be gathered at the unit test level or at system's integration testing to insure that all requirements have been incorporated into the code and adequately tested. Every test case must cover at least one requirement and vise-versa. These metrics are plotted against time to determine development progress. At the conclusion of this lifecycle phase, both metrics should be at or near 100%.
e. The Size of documentation (DOCSIZE)
f. The Percentage of documents that have changed (DOCCHG)

All of these metrics are addressing questions related to assessing the impact of the change. Whether the impact is viewed from a cost, schedule, staffing, testing, or reuse perspective. These metrics all provide the system's developer an idea of the size and complexity of the changes being made. If only 10% of the CSU's are being reused from the previous version, the management decisions made relating to the effort will be much different than the ones if 90% were being reused.

VERIFICATION AND VALIDATION PHASE

HP reported that "you will find 1 defect post-release for every 10 you find pre-release during test." [Gra92] By collecting and analyzing a simple metric such as number of defects in HP's system, a very valuable statistic was established. During verification and validation (V&V) is a good point in the development lifecycle to formally keep track of software defects.

In preparation for the V&V lifecycle phase, questions relating to staffing and the level of testing are often asked. Among the questions are:

1. "How many people should be assigned to this software sysiem V&V since only 10% of the requirements changed and 25% of the design changed?"
2. "How much testing needs to be done to insure the code matches the requirements?"

Beside employing the metrics described in the code and intøgration testing, since the system is now under some type of configuration management control one could collec' metrics relating to quality. These may includ‹ ,uch metrics as:

a. Number of defects (NUMPR) per baseline/revision.
b. A metric that rates the criticality of the defect (ERROR SEVERITY) (An example rating scheme is enclosed in the Appendix where we employ a *1 to 6 level rating with 1 being the worst*.)

c. Some type of metric that characterizes the defect type (ERROR CLASSIFICATION). This allows the user to better understand the development process and the kinds of defects that are being made. If one particular type is predominating, the developer could specifically target process improvement steps in this area. (A sample classification scheme that we employ for software systems is enclosed in the Appendix.)

d. Number of defects per KSLOC (DEFECT DENSITY). This metric can be used to determine if the system is ready for release. Too high a value may indicate problems.

These are just a sampling of the types of metrics that can be collected. In addition, the metrics can be cross tabulated against one another providing the user an additional perspective. An example would be number of defects broken down by error severity.

DELIVERY AND OPERATIONAL PHASE

The delivery/operational phase of the lifecycle will be the time to collect the final set of metrics on the product. They will be collected for both before it goes to the end user (delivery) and during its operational use. This type of information is used to compare back to the metrics collected earlier in the lifecycle and during its use to monitor changes. For example, at delivery one might collect design size and compare it with the size at the completion of the design phase. Has the design size changed? It shouldn't have but many times changes are made further down in the lifecycle then they should be. Both a process and product problem are indicated if the size change is substantial. After delivery the quality of the product can be monitored. If the quality begins to go down to the point that further maintenance is not advisable, reengineering of the entire system may be warranted. As a system is updated in the maintenance phase with new enhancements and corrections to existing faults, the risk of error introduction and greater demands for resources to make the updates increase dramatically.

Questions that are often asked at this point in the development process are:

1. "Has the quality of the system degenerated to the point that an entire new development effort is indicated?
2. "How fast are changes being made to the system?"
3. "What level of support is required to maintain the existing system?"

Metrics to address these questions would include all of the quality metrics discussed in the verification and validation phase plus some of the requirements through design metrics. This latter set would cover any maintenance activities on the operational software. In addition some other metrics that can be used include:

a. System Reliability - The probability the system will operate without failure for a specified time under a specified environment.

b. Mean Time To Failure (MTTF) - The expected time to the next failure of the system.

c. Availability - The probability the system will be operational during a specified time and in a specified environment.

These various metrics can be plotted against time during the operational use to determine if quality is going down. For example, if the operational reliability of the system has reached a point that is no longer acceptable, a decision will need to be made if continued upgrading is feasible or whether it is more cost effective to simply reengineer the entire system.

An interactive computer program called the Statistical Modeling and Estimation of Reliability Functions for Software (SMERFS), [Far93], is available to estimate both Reliability and MTTF, among others, for the software components of the system. The program is completely machine transportable and incorporates eleven of the most well known software reliability models appearing in the literature. It allows the user to do a complete reliability analysis using either the number of software faults per unit of time or the elapsed time between fault occurrences. Using either of these two data types, the user can input the data via keyboard or file, edit it if necessary, transform it, plot it, do a preliminary analysis of the most appropriate model, model the data, analyze the resulting model fit, and then do reliability predictions and trade-off analyses. A sample plot is shown to illustrate how this program could be used in a reengineering framework.

Figure 1 is a plot of the predicted operational reliability of a program over a 1 year period. As one can see the reliability, because of the changes that have been made to the software over this time period, has degraded to the point that a rework of the entire system may be necessary.

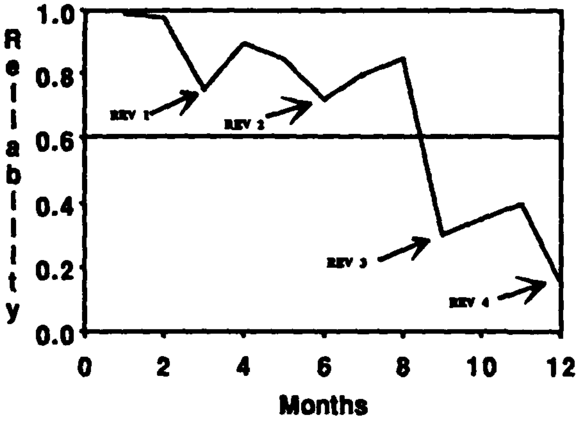


Figure 1 Reliability

Some additional metrics that can be collected over all phases include:

a. Engineering months (NUMMON) - the actual number of months the project took.
b. Staffing time (NUMPEO) - the number of full time people engaged in the development effort. This includes support personnel as well as management time.
c. Average Staffing (AVSTAFF) - NUMPEO divided by NUMMON

The above metrics can be broken down by phase of the lifecycle and used to determine scheduling and staffing requirements for similar reengineering efforts.

CONCLUSION

This paper has provided suggested metrics for the various phases of a reengineering effort. This list is not all inclusive. It is intended to be illustrative of the type measurements that one would be interested in. The metrics selected in this paper are illustrative of the ones collected in our development process. Little metrics have been given relating to cost, security issues, nor performance. The emphasis has been on quality, scheduling, and staffing issues. These areas are deemed important within our environment. The important consideration is to define a set of objectives for your development effort and then to select a set of metrics that will provide quantitative measures of how well those objectives are being met.

REFERENCES

[Far93] W. Farr,"Statistical Modeling and Estimation of Reliability functions for Software (SMERFS) User's Guide",NSWCDD TR 84-373, September 1993.

[Gra92] R. Grady, *Practical Software Metrics For Project Management and Process Improvement*, P T R Prentice Hall,Inc., 1992.

[Hal93] J. Halligan, "Requirements Metrics: The Basis of Informed Requirements Engineering Management", Proceedings of the 1993 Complex Systems Engineering Synthesis and Assessment Technology Workshop, July 1993.

[Har90] "Insights on Improving the Maintenance Process Through Software Measurement", Proceedings from Conference on Software Maintenance, November 1990.

[Pfl90] S. Pfleeger, S Bohner, "A Framework for Software Maintenance Metrics", Proceedings from Conference on Software Maintenance, November 1990.

APPENDIX: METRICS

This section will elaborate some of the metrics discussed earlier by illustrating how they can be used in the re-engineering process. In this section we will provide the metric name, its definition as it relates to our development environment, and how it can be used to relate back to the product and/or process. An example is provided where appropriate to illustrate the usage.

Metric: AVSTAFF

Description:

The metric AVSTAFF is the average effort (person-month) expended per month for a program. It is calculated from two other metrics, NUMPEO and NUMMON. It is useful for determining the level of staffing required for future development efforts and for tracking current efforts.

Feedback/analysis:

AVSTAFF would be used by project leaders to determine how well actual staffing meets the original projection. Previous values can also be used as data points to create scheduling and staffing estimates for future endeavors. Figure 3-2 shows how the average staffing effort for program X has increased over its development history, peaking in the critical design phase. The plotting of this metric could represent a particular phase (e.g. coding and unit testing) or the entire life cycle history of a program.

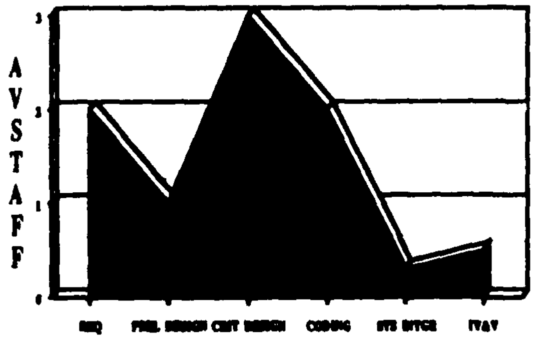


Figure A-1: AVSTAFF SIZE AS A FUNCTION OF TIME FOR PROGRAM X

Metric: DEFECT DENSITY

Description:

Defect density indicates the number of software errors (where error severity is 1 - 6) per KSLOC per program. This metric is a measure of code quality and testing adequacy.

Supporting Definitions:

KSLOC - the number of executable lines of code (excluding blanks or comments) of a program divided by 1,000.
Error Severity - a seven level rating system for evaluating the severity of software errors.

Feedback/analysis:

Defect density provides an idea of the complexity of the code or the reliability of the code. It also provides an indication of the level of Quality Assurance testing that was performed. A high defect density may indicate poor code quality or a low density might indicate inadequate testing was performed. Management can use this measurement to evaluate the quality of the software and decide on a future course of action. Figure A-2 illustrates how the defect density can be tracked over time for a given program, to determine how the quality is changing over time.

This metric can be plotted against a baseline (revision or yearly) for a given program to see if the quality of the program is going up or down. The following figure shows what a report of this nature might look like:

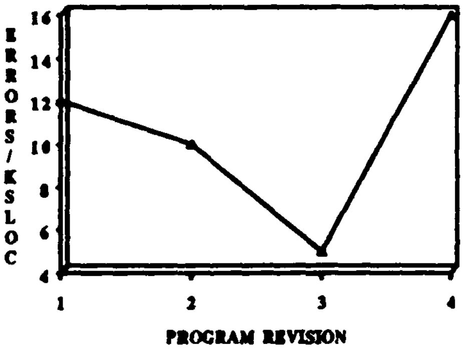


Figure A-2: DEFECT DENSITY FOR PROGRAM X

From this plot one can see that the defect density of Program X was steadily decreasing over the first 3 baselines (revisions). It then suddenly went up on baseline 4. Management would then need to investigate what led to this sudden change (e.g., extensive new capabilities added, more testing performed in contrast to earlier versions). If this trend persisted, more detailed tracking of this program would be required and appropriate corrective action taken.

Metric: DESCHG

Description:

Percent of software design that has changed (modified, added or deleted) from one baseline/revision to another, e.g. Program Design Language (PDL), Data Flow Diagrams (DFD).

$$\text{DESCHG} = \frac{\text{\# OF CHANGES}}{\text{DESSIZE}} \times 100$$

Feedback/analysis:

DESCHG may be used to evaluate program stability and to estimate resources needed by knowing how much of the code must be changed. It will be used in conjunction with MODCHG and REGCHG to estimate how many modules in a program will be changing in the future. Caution: Don't try to compare DESCHG across different software. Programmers design software in many different styles. The level of detail the programmer chooses will often vary. Also, don't assume that because the percent of requirements change (REGCHG) is a small number that the DESCHG will also be small. Figure A-3 illustrates how DESCHG might be used with NUMPEO.

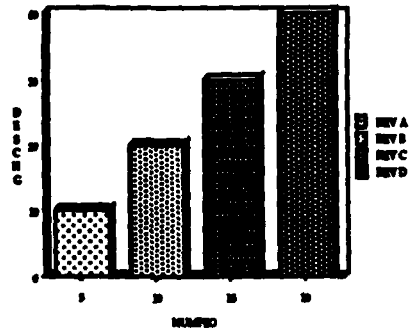


FIGURE A-3: DESCHG
The figure displays historical information about previous maintenance efforts for a piece of software. Using this figure the developer could predict that for DESCHG values < 30, NUMPEO is roughly half the DESCHG value. For values > 30, NUMPEO is closer to 40% of the DESCHG value.

Metric: DESSIZE

Description:

DESSIZE: Number of design units (e.g., PDL=lines, DFD=bubbles/arcs) of a CSCI.

Feedback/analysis:

DESSIZE is used to calculate DESCHG. Knowing the relative sizes of the software design is important when used in conjunction with other metrics to make an overall assessment of the software process/product. Figure A-4 illustrates how DESSIZE might be used with NUMPEO.

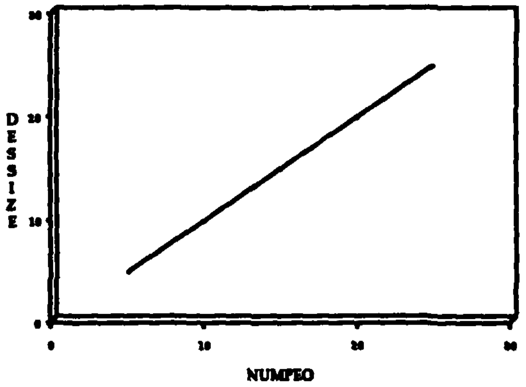


FIGURE A-4: DESSIZE

The figure displays historical information about pervious baseline efforts for similar pieces of software. Using this figure the developer could predict an estimated NUMPEO basing the estimate on DESSIZE.

Metric: DOCCHG

Description:

DOCCHG: Percent of pages of a specific document that has changed (modified, added, or deleted) from one baseline/revision to another.

$$\text{DOCCHG} = \frac{\text{\# OF CHANGE PAGES}}{\text{DOCSIZE}} \times 100$$

Supporting Definitions:

DOCSIZE: Number of pages of a specific document.

Feedback/analysis:

DOCCHG will be used to evaluate software stability and to estimate resources needed by knowing how much of the documentation has been changed. FigureA-5 illustrates how DOCCHG might be used.

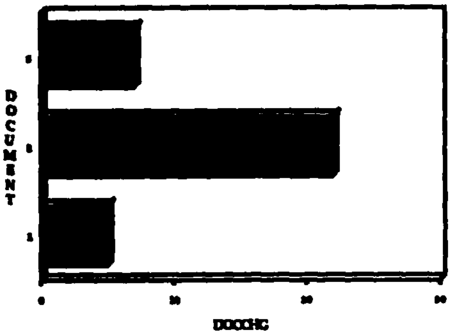


FIGURE A-5: DOCCHG FOR PROGRAM X

The figure displays the percentages that three documents describing Program X have changed. Using this figure the tester could conclude that more time should be set aside for reviewing document two since DOCCHG is higher for that document.

Metric: DOCSIZE

Description:

DOCSIZE: Number of pages of a specified document.

Feedback/analysis:

DOCSIZE is used to calculate DOCCHG. Knowing the relative size of a document is important when used in conjunction with other metrics to make an overall assessment of the software process/product. When used independently, DOCSIZE could indicate when there is insufficient documentation for a software process/product. Figure A-6 illustrates how DOCSIZE might be used.

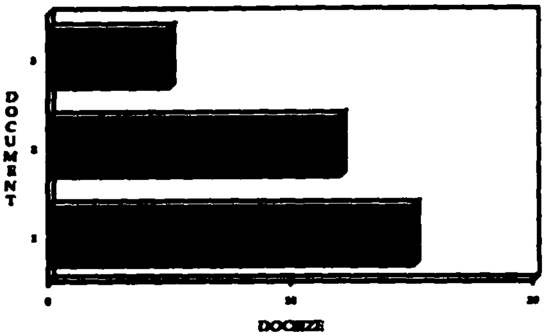


FIGURE A-6: DOCSIZE

The figure displays information about three documents. Using this figure the tester might conclude that more time should be set aside for reviewing document one since DOCSIZE is higher for that document. If the programs are comparable, it may also indicate that program one is better documented than program three.

Metric: ERROR CLASSIFICATION

Description:

Error Classification is a code that categorizes the type of software error that was made. The code will be made up of five major fields. Each code conveys information about the error. The scheme is as follows:

How did the error manifest itself to the user?

01 abort
02 program produced wrong answer
03 code reading
04 erroneous error message produced
05 using supporting documentation
06 using an analysis tool
07 other

Type of error

01 incorrect logic
02 incorrect equation
03 invalid subscript or counter used
04 data read in incorrectly
05 data written incorrectly
06 variable not initialized properly
07 incorrect variable type
08 module called with incorrect parameters
09 standard violation
10 database

11 documentation
12 generation problem
13 violated system constraints

Found by whom?

01 associated development team
02 associated IV&V team
03 NSWCDD user
04 non-NSWCDD user

How found?

01 operational use
02 product inspection
03 by converting/executing program for/on another platform
04 testing

Test Case Classification (use only if 04 is selected)

01 Operational - test that uses real data, and real sequences of events
02 Stress - test that simulates a program's full load capability
03 Boundary - test that uses maximum or minimum boundary data
04 Randomized - test that uses randomly chosen real data
05 Robustness - test that uses invalid data or procedures

Error was introduced at what point in life cycle?

01 requirements definition
02 design
03 coding
04 maintenance

Feedback/analysis:

Error Classification can be used to determine whether the particular project requires more or less people and/or time resources. It can also be used by the configuration management board and the Software Process Improvement team to identify areas where process improvements would be beneficial. This metric can be used to determine if new initiatives aimed at eliminating certain errors are successful. For example, this information can be used to determine what types of diagnostic tools might be helpful, and to evaluate the effectiveness of these tools. Additionally, this metric can be used to determine what types of testing are yielding what types of errors.

Figure A-7, a pie chart, is an example of how the distribution of error types might be reported for Program X. Similar charts can be constructed for each of the other 5 subgroups (error manifestation, found by, how found, test case classification, time error introduced).

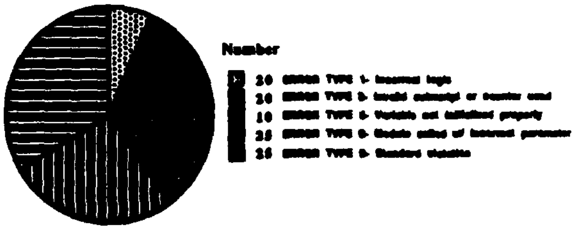


Figure A-7: ERROR CLASSIFICATION by TYPE for Program X

Each of these may be reported separately or in combinations. They can be reported either in tabular or plot form. For example, the plotting of the "Error was introduced at what point in the life cycle" against phase of the life cycle would be especially revealing. This would show where the errors were being introduced so that appropriate corrective action to the process could be applied. Also, the information from the various reports can be used in conjunction with one another. An example would be a high number of errors introduced in the requirements phase and were also primarily logic type errors. This information might be useful in determining the type of corrective action to take. Cross tabulations of the above categories would therefore provide additional information.

Further information can be derived by breaking this information down by each program. If a particular program was experiencing difficulty relating to quality or meeting deadlines, analysis of the above error classification type might help pinpoint the problem.

Metric: ERROR SEVERITY

Description:

Error Severity is a seven level rating system for evaluating the severity of software errors.

1. *Critical* - Program cannot be used until error is corrected. Program aborts abnormally, or produces unreliable results.

2. *Serious* - Some necessary portion of program

cannot be used until error is corrected. It is still possible to use other portions of the program.

3. ***Non-critical*** - Program does not perform correctly for non-typical test cases. Some nonessential feature of the program cannot be used. These types of errors can be worked around and are typically placed on a shopping list or corrected as time allows, rather than immediately. Program is still usable.

4. ***Standards violation*** - unjustified violation of standards. Violations that are intentionally introduced for a reason are not errors. (Intentional violations must be documented, giving the reason why they are necessary.) These errors are not visible to the end user.

5. ***Clerical*** - typographical errors, misspellings, grammatical errors in the code that do not prevent the reader from understanding what is written. If the user is misled by the error, then it gets a higher severity rating. Unnecessary code and variables would be included in this category.

6. ***Documentation*** - any error, (typographical errors, misspellings, etc.) that appears in the supporting documentation, (Software Design Document, User's Guide, etc.), for a program. If the user is seriously misled by the documentation error, then it should get a higher severity rating.

0. ***No error*** - problem reports submitted because of incorrect input, user misunderstandings, etc.

Note: Errors of severity level 1, 2 or 3 must be submitted one per problem report. For level 4, 5, or 6 errors, multiple problems may be submitted as a single problem report.

Feedback/analysis:

Error Severity can be used by a manager or group leader when scheduling tasks. This metric indicates whether a particular project requires more or less people and/or time resources. It can also be used by the IV&V team for the same reason. The Software Process Improvement Team and the CM board will use it to identify areas where process improvements would be beneficial. This might be used to determine if new initiatives aimed at eliminating certain errors are successful.

Figure A-8 is an example of a graphical report of this metrics for Program XX. Ideally the graph should have no errors of severity 1 or 2. For Program XX there are a total of 11 such errors. Management may question the reasons for this count.

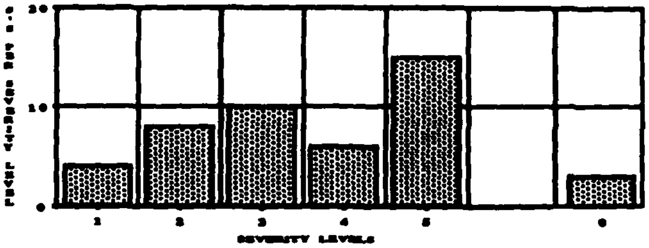


FIGURE A-8: ERROR SEVERITY FOR PROGRAM XX

Metric: KSLOC

Description:

KSLOC is the number of executable lines of code (excluding blanks and comments) of a program divided by 1000.

Feedback/analysis:

KSLOC can be applied to different programming languages and compared to outside organizations especially in comparing estimates for work that is to be contracted out. KSLOC can be used in conjunction with NUMMON and NUMPEO to estimate future program development resource allocation/staffing. It also can be used to calculate other metrics (e. g. DEFECT DENSITY). Knowing the relative sizes of programs is important when used in conjunction with other metrics to make an overall assessment of the software process/product.

Figure A-9 illustrates how KSLOC might be used to compare programs. These programs might be different software items or they might be the same program tracked over different years. The data can also be used to estimate the size of future efforts. If a new program is being developed that is roughly comparable to programs 2 and 3, a developer could estimate a KSLOC of 25 for the new program based on this data.

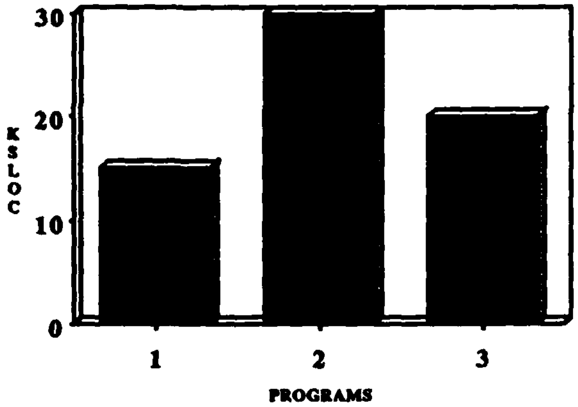


Figure A-9: KSLOC Sample Report

Metric: KTLOC

Description:

KTLOC is the total number of all lines of code of a program divided by 1000.

Feedback/analysis:

KTLOC may be used in connection with KSLOC to determine how well a program is documented. It may be used in determining future program development resource allocation and staffing.

The following figure is an example report that could be generated comparing the KTLOC from three programs. In addition this report could combine the report of KSLOC to indicate how many non-executable statements there are. From this comparison one could get an idea of how much in-line documentation and/or blank lines were being put into a program and how this number varied across programs.

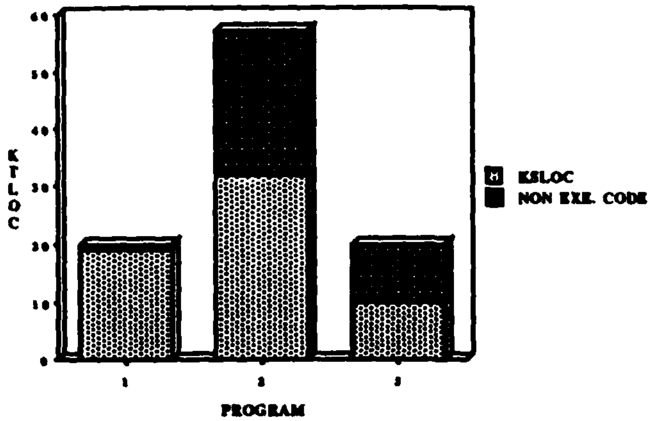


Figure A-10: EXAMPLE KTLOC AND KSLOC REPORT COMBINED

Notice how Program 1 has very little in-line comments and/or blank lines while Program 3 has about 50%. Program 1 may be difficult to maintain and/or test.

Metric: MODCHG

Description:

Percent of modules within a program that have changes (modified, added, or deleted) from one baseline/revision to another.

$$\text{MODCHG} = \frac{\text{\#of modules changed}}{\text{total \# of modules (NUMMOD)}} * 100$$

Feedback/analysis:

MODCHG will be used to evaluate program stability and to estimate resources needed by knowing how many modules have changed. It will be used in conjunction with requirements changes to estimate how many modules in a program will be changing in the future.

Figure A-11 illustrates a graphics report of MODCHG. From this report the V&V group can determine whether testing should be performed and, if so, the extent. From the Figure, the testing group might decide to put little or no effort into Program 1 while Program 2 is a good candidate for a complete V&V effort. Program 3 may need some testing depending upon the nature of the changes.

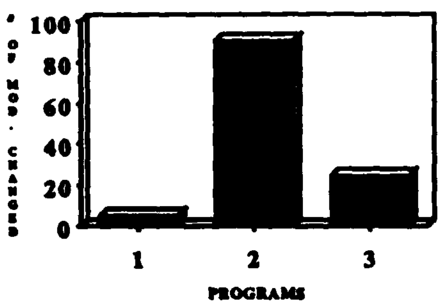


Figure A-11: Example Report for MODCHG

Metric: MTTF

Description:

Mean time to software failure occurrences of a software program.

Feedback/analysis:

MTTF can provide an indication of the quality of the software. It can thus be used to determine both the testing effort required and the release time from V&V testing. For example, if the MTTF is low, additional testing many be required in order to increase it. In addition, management can use this metrics to determine how much additional testing is required to achieve a desired level for this metric. This metric can be computed from the Statistical Modeling and Estimation of Reliability Functions for Software (SMERFS) developed by one of the authors.

This metric can be compared across program revisions to aid in determining maintainability issues. Figure A-12 illustrates a plot of this metrics for three software revisions. As can be seen MTTF is going down. This indicates the quality appears to be degrading, requiring additional testing effort and/or possible rewrite of the software.

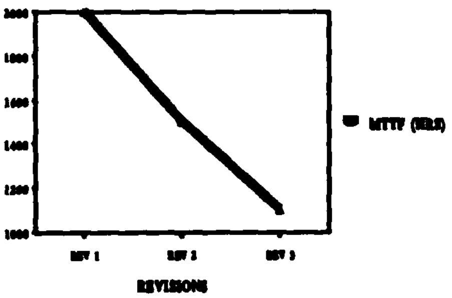


Figure A-12: MTTF

Metric: NUMMOD

Description:

The number of modules within a program. Do not include any external library routines in your count of modules.

Feedback/analysis:

NUMMOD provides program sizing information. It will be used in conjunction with KSLOC to estimate complexity for evaluation of design, development, or V&V resource allocation of a program. In Figure A-13 one can see that the number of modules steadily grew through three revisions and then dropped slightly for revision 4. For the V&V team the amount of testing could reflect this same behavior, i.e. testing increases through revision 3, then drops for revision 4.

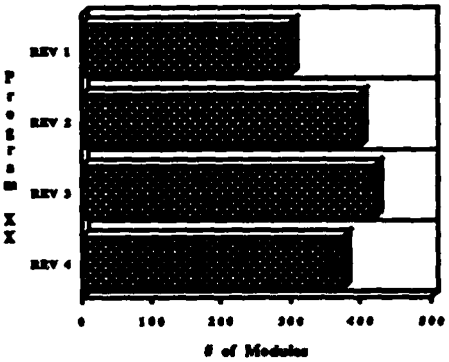


FIGURE A-13: NUMMOD FOR PROGRAM XX

Metric: NUMMON

Description:

The metric NUMMON is the number of elapsed time (in calendar months) it takes to complete a project. The metric will also be used with NUMPEO to determine AVSTAFF.

Feedback/analysis:

NUMMON would be used by the project leaders to determine how well actual scheduling meets the original projection. This metric can also be used as data points to create scheduling and staffing estimates for future endeavors. Note: calendar months with no effort are counted in NUMMON.

For this example given in Figure A-14, the program was in the requirements/design phase until April, and completed testing for implementation by December. The number of months and staffing during the design phase was relatively low but increased after April. The management might question why this occurred especially in the later phases of the lifecycle.

DATE	NUMMON			PHASE
	NUMPEO	AVSTAFF		
Jan	2	2	1.0	Req Definition and Analysis
Apr	3	3	1.0	Design Development
Oct	6	2	2.0	System Integration and Testing
Nov	6	2	3.0	Verification and Validation
Dec	6	3	2.0	Delivery and Production

Metric: **NUMPEO**

Description:

The metric NUMPEO is the total number of person-months used to complete a program. Be sure to include manager's time spent reviewing work on tasks as well as any contractor support given to tasks. The metric will also be used with NUMMON to determine AVSTAFF.

Feedback/analysis:

NUMPEO would be used by the project leaders to determine how well actual staffing meets the original projection. This metric can also be used as data points to create scheduling and staffing estimates for future endeavors.

For this example given in Figure A-15, the program was in the design phase until April, and completed implementation by December, during which time AVSTAFF continually increased until November. If this was not expected, the development manager would investigate what caused this spike in the figure (changing requirements, time management, planning was too crude, etc.) What should especially be questioned is the large value for NUMPEO beginning with coding. Poor scheduling may be indicated.

DATE	NUMMON			PHASE
	NUMPEO	AVSTAFF		
Jan	2	2	1.0	Req Definition and Analysis
Apr	1	2	1.0	Design Development
Jul	3	3	1.0	Code and Unit Testing
Oct	6	3	2.0	System Integration and Testing
Nov	6	2	3.0	Verification and Validation
Dec	6	3	2.0	Delivery and Production

Figure A-15: STAFFING INFORMATION FOR PROGRAM X

Metric: NUMPR

Description:

The number of problem reports (PRs) submitted against a CSCI.

Feedback/analysis:

Analysis of this metrics will highlight anomalies that require further investigation. e.g., an unusually high number of PRs in a given period of time could indicate requirements changes are still being made afte ersion baseline is established. Use this metrics to evaluate program stability- the key is that an unusally large number of PRs warrants a more detailed look at the software development process to see if corrective action is justified.

Metric: RELIABILITY

Description:

Probability the software will not fail in a specified time within a specified environment.

Feedback/analysis:

Reliability can provide an indication of the quality of software. It can thus be used to determine both the testing effort required and the release time from V&V testing. If the reliability is low additional testing may be required in order to increase it. In addition, management can use this metric to determine how much additional testing is required to achieve a desired level for this metric. This metric can be computed from the Statistical Modeling and Estimation of Reliability Functions for Software (SMERFS) developed by one of the authors.

This metric can be compared across program revisions to aid in determining maintainability issues. Figure A-16 illustrates a plot of this metric for three revisions. As can be seen the Reliability is going down. This indicates that the quality appears to be degrading, requiring additional testing effort and/or possible rewrite of the software.

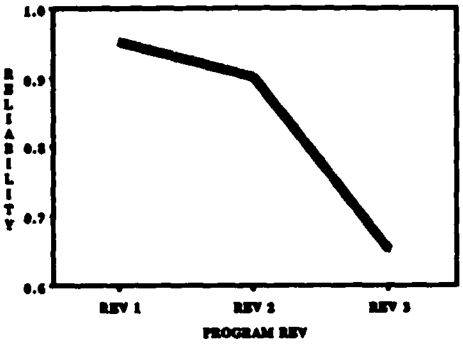


FIGURE A-16: RELIABILITY

Metric: REQCHG

Description:

Percent of requirements of a specific CSCI that has changed (modified, added, or deleted) from one baseline/revision another.

Feedback/analysis:

REQCHG will be used to evaluate software stability and to estimate resources needed by knowing how many of the requirements have changed. Figure A-17 illustrates how REQCHG might be used.

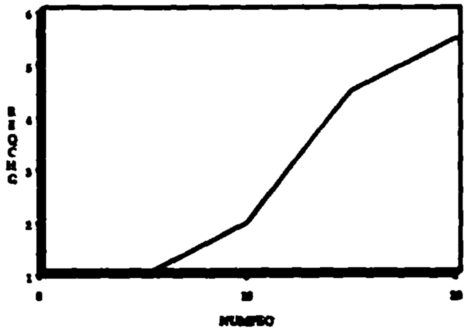


FIGURE A-17: REQCHG

The figure historical information about previous maintenance efforts for a project. Using this figure the developer could predict that NUMPEO is roughly equal to REQCHG.

Metric: REQSIZE

Description:

REQSIZE: Number of requirements of a CSCI.

Feedback/analysis:

REQSIZE is used to calculate REQCHG. Knowing the number of requirements of a CSCI is important when used in conjunction with other metrics to make an overall assessment of the software process/product. Figure A-18 illustrates how REQSIZE might be used with NUMPEO.

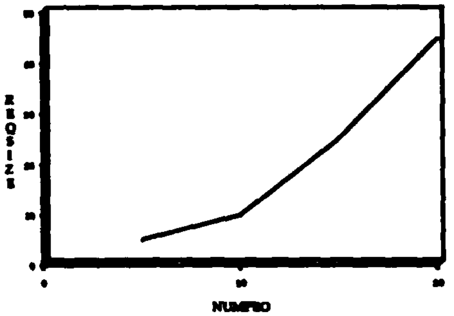


FIGURE A-18: REQSIZE

The figure displays historical information about previous baseline efforts for similar pieces of software. Using this figure the developer could predict an estimated NUMPEO based on REQSIZE.

Customized Software Evaluation Tools: Application of an Enabling Technology for Reengineering

Lawrence Markosian, Russell Brand and Gordon Kotik*

Abstract

This paper describes a new approach to developing tools for verifying source code compliance with coding standards. The approach is based on an enabling technology for software evaluation and reengineering. The key technical ideas underlying the technology are to represent source code in the form of abstract syntax trees in an object-oriented database, and to use a library of utilities to analyze software represented in this way. This enabling technology supports rapid implementation of project-specific coding standards. Coding standards verification tools implemented this way can be used for evaluating legacy systems that are being reengineered, as well as for performing quality assurance on the reengineered systems. A major focus of the paper is the enabling technology, which is applicable to other reverse engineering and reengineering tasks. The paper also discusses several examples of coding standards implemented using this technology. Finally, we summarize our experience using this approach.

1 Introduction

A key component of a software evaluation process is a procedure for checking source code for conformance with coding standards. Prior to reengineering a legacy system, this procedure can be used to help determine whether the system should be reengineered and, if so, what reengineering approach should be used. For example, the procedure can help identify modules that must be redesigned or rewritten by hand. Another goal is to ease maintenance of the reengineered system by making the code more understandable and by reducing the likelihood of defect introduction during maintenance.

The coding standards may be based on published standards such as the Software Productivity Consortium's *Ada Quality and Style: Guidelines for Professional Programmers* [1] for Ada, or they may be based on corporate guidelines. Often there are project-specific guidelines that deliverable code must comply with.

Coding standards typically cover a wide range of coding practices including commenting style, formatting, identifier naming, language standard, coding idioms, file organization, and use of libraries.

While the simplest of such coding standards may be checked by regular expression-based text analysis tools, others require a deeper model of the implementation language.

The great variety of possible coding standards, including project-specific standards, makes it difficult to find an adequate commercially-available, off-the-shelf (COTS) automation tool. Also, there is often little support for the old languages and dialects in which legacy applications were written.

This paper describes an approach to building software evaluation tools that makes it feasible to implement a wide range of coding standards, including those that require semantic analysis of the program being analyzed.

Section 2 describes Software Refinery^{TM} and the Refine Language Tools^{TM}, products from Reasoning Systems that support this approach.

Section 3 contains examples of specific coding standards implemented using this approach. We provide examples for C. We give an informal characterization of each coding standard and show its implementation

*Reasoning Systems, Inc., 3260 Hillview Avenue, Palo Alto, CA 94304

in Software Refinery.

Section 4 summarizes industrial experience with software evaluation tools implemented using this approach.

2 Enabling technology

We use a new, enabling technology for reengineering in building tools for coding standards verfication. The central technical ideas underlying the technology are:

- represent software in the form of abstract syntax trees (ASTs) in a persistent object-oriented database;
- use libraries to analyze and transform code represented in this form.

The coding standards verification tools are built using Software Refinery and the REFINE Language Tools, reengineering products from Reasoning Systems that incorporate this technology. These products run on SPARCstation^{TM}, IBM RS/6000^{TM} and HP 9000/700^{TM} workstations.

The remainder of this section describes the features of Software Refinery and REFINE/C^{TM}, the REFINE Language Tool for analyzing C programs. This section also discusses the data model that REFINE/C uses for representing C programs.

2.1 Software Refinery

Software Refinery is an environment for developing reengineering tools. We used Software Refinery to build the REFINE Language Tools, including REFINE/C. For building the C coding standards verification tools, we used Software Refinery to customize REFINE/C.

Software Refinery provides:

- a parser and printer generator,
- a persistent object-oriented database,
- a very-high-level language for analyzing and transforming software in this database,
- a library of utilities for operating on code in the database,
- an X Windows-based graphical user interface toolkit, and
- a programming environment that permits seamless integration with existing UNIX tools and networking utilities.

Figure 1 shows the major components of Software Refinery. A complete description of Software Refinery is provided in [3, 4, 5, 6].

DIALECT^{TM} is an LALR(1) parser generator. It provides a mechanism for handling non-LALR(1) languages such as COBOL.

Parsers developed using DIALECT retain surface syntax, which includes comments and formatting information. A coding standards verification tool can make use of this information—for example, to determine whether indentation conventions are followed. Also, during the code conversion process, code that is transferred into the reengineered system without modification can be printed exactly as it appears in the original source program.

Software Refinery's very high level language, REFINE^{TM}, is the implementation language for our coding standards verification tools. REFINE includes:

- symbolic mathematical operations including first-order logic and set operations,
- a library of tree operations such as copy, compare, traverse and substitute,
- a transformation operator for specifying modifications to source code, and
- pattern-matching against ASTs in the objectbase.

The REFINE compiler supports unit-level incremental compilation and dynamic linking, which speeds up the edit-compile-run loop.

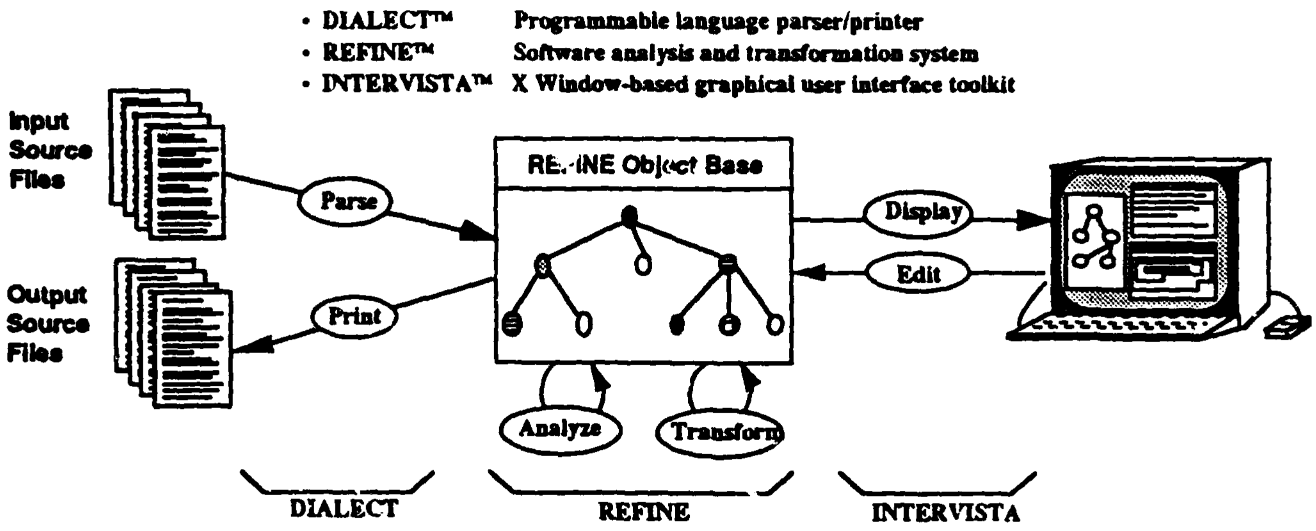

• DIALECT™ Programmable language parser/printer
• REFINE™ Software analysis and transformation system
• INTERVISTA™ X Window-based graphical user interface toolkit
Input Source Files
Output Source Files
Parse
Print
REFINE Object Base
Display
Edit
Analyze
Transform
DIALECT
REFINE
INTERVISTA

Figure 1: The major components of Software Refinery

2.2 REFINE/C

REFINE/C is a C reengineering workbench developed by Reasoning Systems using Software Refinery. REFINE/C supports:

- modelling C source code as ASTs in the REFINE object base;
- associating variable definitions with their references;
- generating graphical reports for each C program, including:
  - structure chart,
  - data-flow diagram,
  - function control-flow diagram and cyclomatic complexity report,
  - coding standards report,
  - functions table,
  - variables table, and
  - types table;
- navigating from any report directly into the corresponding source code;
- exporting to a forward-engineering CASE tool;
- PostScript^{TM} or ASCII printing of reports; and
- handling large C programs.

REFINE/C has a graphical user interface with pull-down menus, on-line help, multiple windows displaying text, tables and graphs, and hyperlinking among reports and source code.

The programmer's guide for REFINE/C documents its application programmer's interface (API)[2]. The API supports customization and extension of the tool's capabilities.

The next section gives an overview of the representation of C code in REFINE/C. This data representation is used by analysis tools in REFINE/C such as coding standards verification.

2.3 Representation of C code in REFINE/C

This section gives an overview of how C programs are represented in REFINE/C.

The AST of a C program is the basic data structure used during program analysis and transformation steps. REFINE/C provides an object-oriented model for C abstract syntax. Object classes correspond to non-terminal nodes in the AST. Attributes (slots) hold subtrees.

Figure 2 shows three views of the REFINE/C representation of the following C `if` statement:

```
if ( players[player].dollars == 0)
   { players[player].bet = 0 ;
     return ;
   }
```

The upper-left window contains the "Pretty Print" view of the statement–the surface syntax as it appears in a source code file. The lower-left shows a diagram of part of the AST of the same statement. The upper-right window shows the "Frame Print" view of the statement as an object in the database. This object is the root of the AST that REFINE/C creates when it parses the C `if` statement. The lower-right window is a history window that assists in navigation.

REFINE/C models C `if` statements with the `if-statement` object class. The first line in Frame Print window says that the particular statement being viewed is an instance of this object class. The `if-statement` object class has three attributes that hold subtrees: `if-condition`, `then-part` and `else-part`. They are shown in the Frame Print window. Since there are no `else` actions in the example `if-statement`, the value of the `else-part` attribute is `undefined`. The value of the `if-condition` is an instance of the `equality` object class that represents the condition

```
players [player] . dollars == 0
```

in the `if` statement.

Similarly, the value of the `then-part` is an instance of the `block` object class that represents the block

```
{ players [ player ] . bet = 0 ;
  return ;
}
```

in the `if` statement.

The parser creates these objects and sets the values of their attributes when it parses the original `if` statement. The top part of the AST for the `if` statement is shown as a diagram in the lower-left window (labelled "Abstract Syntax Tree" in Figure 2. The full AST for this `if` statement is shown in Figure 3.

The complete abstract syntax for C is provided in the REFINE/C API.

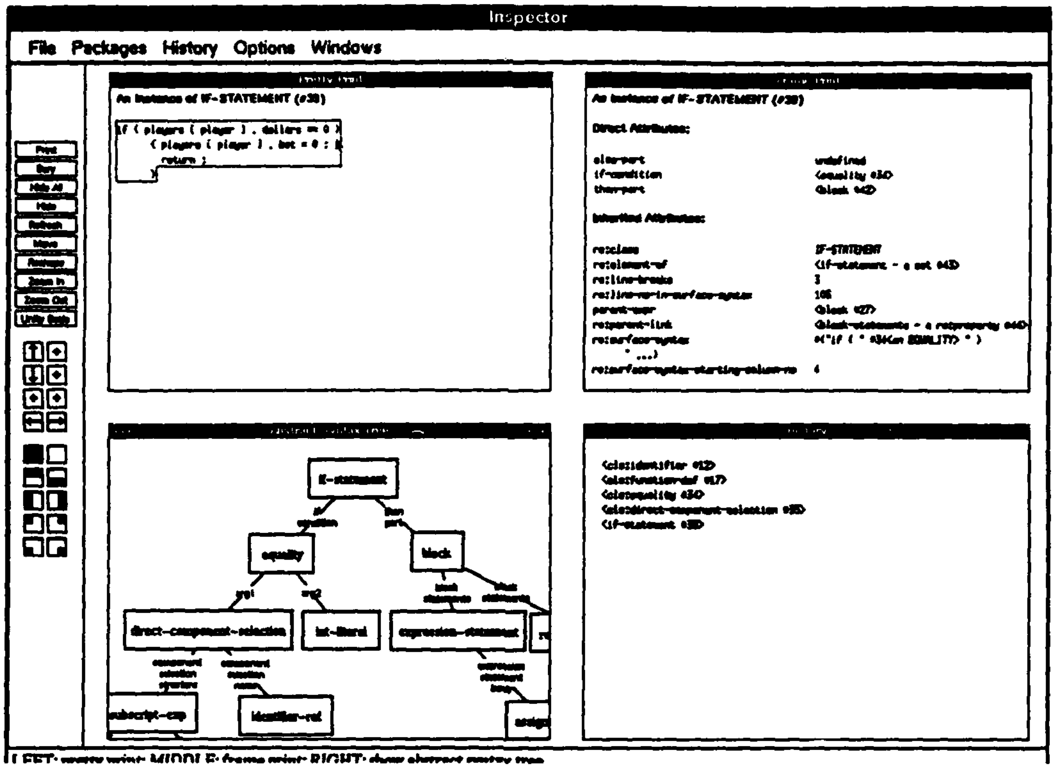
3 Implementation of coding standards

This section describes several related coding standards for C code and how they are implemented in REFINE/C.

The first coding standard we discuss is the following: "Do not use an LVAL unnecessarily on the left side of a comparison." An "LVAL" is a variable or other construct that can be assigned a value. In C, these other constructs include array elements (for example, `a[i]`), pointer dereferences (`*ptr`). and component selectors (`EmployRecord.name` and `EmployRecord->name`). The motivation for this coding standard is that during maintenance, the comparison operator could be incorrectly edited to become an assignment. This editing error would not be detected by the compiler, would not always generate a runtime error, and could "silently" cause incorrect answers to be generated by the program. For example, `x != 8` might be edited incorrectly to become `x = 8` when `x == 8` was intended. Usually this incorrect edit will not generate a compiler error. However, if the original expression had been `8 != x` and this had been changed to `8 = x`, a compiler error would be generated.

The condition in the `if` statement that we examined above violates this coding standard because it contains a component selector, which is an LVAL, on the left side of the equality test.

Figure 4 shows the REFINE/C representation of the



An Instance of IF-STATEMENT (#38)

```
if ( players [ player ] . dollars == 0 )
     { players [ player ] . bet = 0 ;
       return ;
     }
```

An Instance of IF-STATEMENT (#38)

Direct Attributes:

```
else-part                                  undefined
if-condition                               <equality #34>
then-part                                  <block #42>
```

Inherited Attributes:

```
re::class                                  IF-STATEMENT
re::element-of                             <if-statement - a set #43>
re::line-breaks                            3
re::line-nr-in-surface-syntax              105
parent-expr                                <block #27>
re::parent-link                            <block-statements - a re::property #44>
re::surface-syntax                         #("if ( " #34<an EQUALITY> " )
      " ...)
re::surface-syntax-starting-column-no      4
```

```
<cl::identifier #12>
<cl::function-def #17>
<cl::equality #34>
<cl::direct-component-selection #35>
<if-statement #38>
```

LEFT: pretty print; MIDDLE: frame print; RIGHT: show abstract syntax tree

Figure 2: Three views of the REFINE/C representation of an IF statement

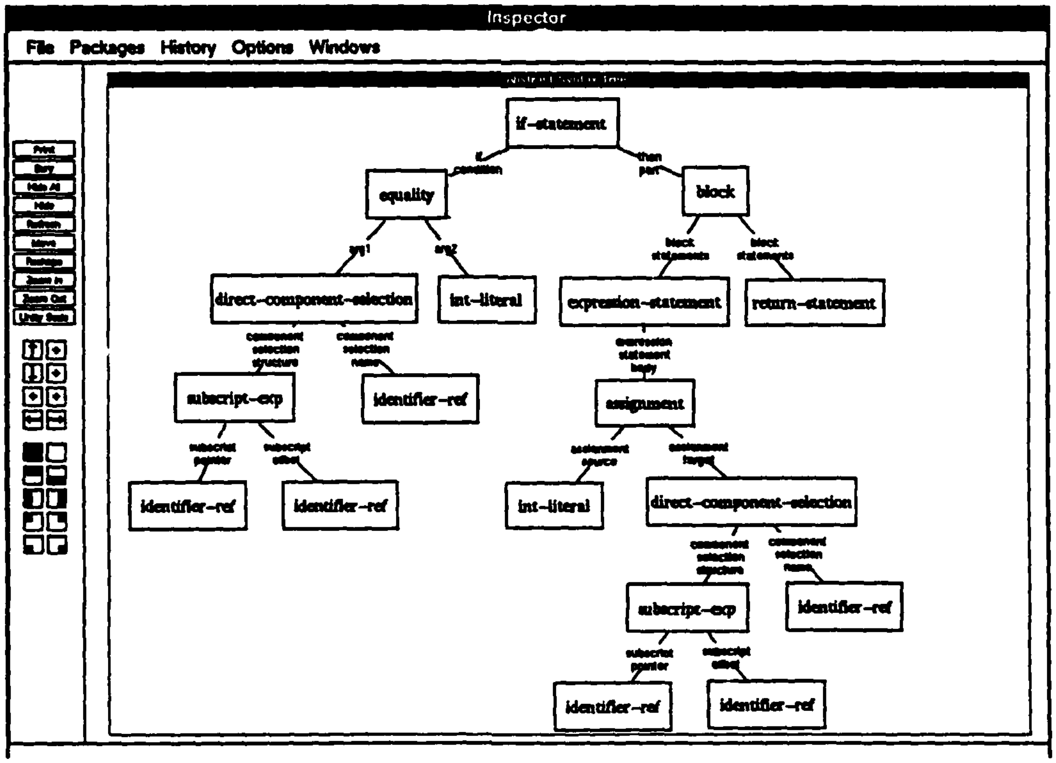
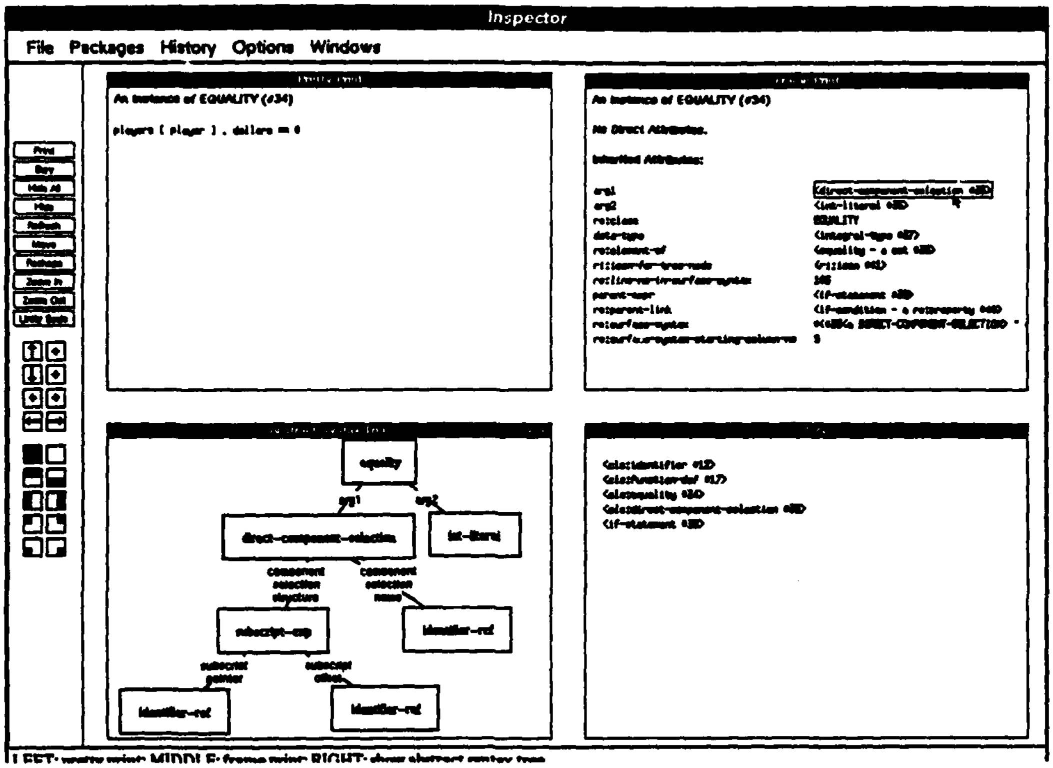


Figure 3: The AST for an IF statement



An instance of EQUALITY (#34)

```
players [ player ] . dollars == 0
```

An instance of EQUALITY (#34)

No Direct Attributes.

Inherited Attributes:

arg1	<direct-component-selection #[illegible]>
arg2	<int-literal #[illegible]>
re::class	EQUALITY
data-type	<integral-type #[illegible]>
re::element-of	<equality - a set #[illegible]>
ri::item-for-tree-node	<ri::item #[illegible]>
re::line-number-in-surface-syntax	[illegible]
parent-expr	<if-statement #[illegible]>
re::parent-link	<if-condition - a re::property #[illegible]>
re::surface-syntax	#<[illegible] DIRECT-COMPONENT-SELECTION>
re::surface-syntax-starting-column	3

```
<ele:identifier #12>
<ele:function-def #17>
<ele:equality #34>
<ele:direct-component-selection #[illegible]>
<if-statement #[illegible]>
```

LEFT: pretty print; MIDDLE: frame print; RIGHT: show abstract syntax tree

Figure 4: REFINE/C representation of an equality expression

`equality` expression in the `if-statement`. Note that the value of the `arg1` attribute of the `equality` is a `direct-component-selection`.

The following REFINE function is a predicate that returns `true` if its argument is an LVAL:

```
function is-an-lval? (node: c-object):
 boolean =
     IDENTIFIER-REF(node)
  or SUBSCRIPT-EXP(node)
  or DIRECT-COMPONENT-SELECTION(node)
  or DEREFERENCE(node)
```

`IDENTIFIER-REF`, `SUBSCRIPT-EXP`, `DIRECT-COMPONENT-SELECTION` and `DEREFERENCE` are names of REFINE object classes in the abstract syntax of C. They are used as predicates in the definition of `is-an-lval?`.

Since the value of the `arg1` attribute of the `equality` is a `direct-component-selection`, `is-an-lval?` returns `true` for this node.

Here is a rule that checks whether its argument, a node in a C AST, is an equality that violates the coding standard.

```
rule check-for-lhs-lval
 (node: c-object)
   <node, 'check-for-lhs-lval>
     ~in *already-checked*
 & EQUALITY (node)
 & is-an-lval? (ARG1 (node))
 & ~is-an-lval? (ARG2 (node))
  -->
   <node, 'check-for-lhs-lval>
     in *already-checked*
 & <node, 'check-for-lhs-lval>
     in *violations*
```

This rule has a single argument, a node in an AST. The rule contains a transform, as indicated by the transform arrow, -->. On the left hand side of the transform are the preconditions for the transform to fire. The preconditions include tests that the node is an instance of the `equality` object class, and that the `arg1` attribute of the node is an LVAL while the `arg2` attribute is not.

On the right hand side of the transform are postconditions that will hold in the state that results if the rule fires. One of the postconditions is that the offending node and the name of the rule that it violates are placed in a set of violations.

The example rule above can be generalized to detect other similarly error-prone uses of LVALs:

```
rule
 check-binary-expr(node: C-OBJECT)
    <node, 'check-binary-expr>
      ~in *already-checked*
  & BINARY-EXPRESSION (node)
  & ~LOGICAL-AND(node)
  & ~LOGICAL-OR(node)
  & ARG1(node) = the-arg1
  & ARG2(node) = the-arg2
  & (is-an-lval?(the-arg1)
     or ASSIGNMENT(the-arg1))
  & ~is-an-lval?(the-arg2)
   -->
    <node, 'check-binary-expr>
      in *already-checked*
  & <node, 'check-binary-expr>
      in *violations*
```

Code that violates a coding standard can, in many cases, be converted into equivalent code that conforms with the standard. For example, the following rule corrects the order of elements tested in equality comparisons:

```
rule flip-equality-comparisons
 (node: c-object)
   <node,
    'flip-equality-comparisons>
     ~in *already-checked*
 & EQUALITY (node)
 & rhs = ARG1 (node)
 & lhs = ARG2 (node)
 & is-an-lval? (rhs)
 & ~is-an-lval? (lhs)
  -->
   lhs = ARG1 (node)
 & rhs = ARG2 (node)
 & <node,
    flip-equality-comparisons>
      in *already-checked*
```

This rule can be generalized to work with all types of comparison operations.

4 Experience with the approach

Software Refinery users have implemented coding standards for a number of languages including Ada, C, FORTRAN, COBOL and the PL language family [7, 8].

As part of a project to automate a quality assurance process for C developers using Microsoft/Windows^{TM}, we recently implemented 12 coding standards in REFINE/C. The rules were based primarily on a client's corporate style guideline document. The rnles are being used in a batch processing environment to simplify the client's review and acceptance process for code developed by its vendors. Implementation of these rules took two days.

We used REFINE/COBOL^{TM} to implement coding standards for analyzing control flow in COBOL programs using PERFORM statements. Different dialects of COBOL treat nested PERFORM statements differently. This leads to a number of porting difficulties. Also, leaving and reentering PERFORM ranges by way of GOTOs leads to unpredictable results.

References

[1] SOFTWARE PRODUCTIVITY CONSORTIUM. *Ada Quality and Style: Guidelines for the Professional Programmer, SPC-91061-CMC, Version 02.01.01*, Software Productivity Consortium, Herndon, Virginia, 1992.

[2] REASONING SYSTEMS. REFINE/C *Programmer's Guide*. Reasoning Systems, Palo Alto, California, 1992.

[3] REASONING SYSTEMS. REFINE *User's Guide*. Palo Alto, CA, 1992

[4] REASONING SYSTEMS. DIALECT *User's Guide*. Palo Alto, CA, 1992

[5] REASONING SYSTEMS. INTERVISTA *User's Guide*. Palo Alto, CA, 1992

[6] REASONING SYSTEMS. WORKBENCH *User's Guide*. Palo Alto, CA, 1992

[7] BUSS, E., and HENSHAW, J. "Experiences in Program Understanding," Technical Report TR-74.105, IBM Canada, Toronto, Canada, 1992

[8] TROSTER, J., "Assessing Design-Quality Metrics on Legacy Software," Technical Report TR-74.103, IBM Canada, Toronto, Canada, 1992

Using Design Knowledge to Extract Real-Time Task Models

Lester Holtzblatt, Richard Piazza, Howard Reubenstein, Susan Roberts
The MITRE Corporation
Bedford, MA 01730
lester@mitre.org

Abstract

The utility of most commercially available reverse engineering tools is limited by their ability to extract only information concerning the sequential execution of a computer program. With the exception of tools that support Ada and its explicit tasking constructs, reverse engineering tools fail to capture information concerning the flow of information between tasks. One of the primary reasons for this situation is that reverse engineering tools only extract information that is explicitly represented in the syntax of the programming language. Since older programming languages do not explicitly represent tasking constructs, reverse engineering tools for these languages fail to capture information concerning how these tasks interact. In this paper, we describe an approach for extracting extra-linguistic information from the source code. This approach was used to support the recovery of task flow information from a command and control system written in CMS-2.

1.0 Introduction

Commercially available reverse engineering tools extract certain aspects of the design of a software system from source code. These tools can provide maintainers good insight into the structure of a program particularly when analysis reports are coupled with good source code navigational aids. For example, using one of the family of tools available from Reasoning Systems (Refine/C, Refine/Ada, Refine/Fortran, Refine/Cobol), a software maintainer can interactively navigate through code by selecting different portions of code to view from a structure chart. A maintainer may also begin to gain insight into the potential impact of changes he plans to introduce into a program by using these tools to identify areas of the program that may be affected by his change. In each of these cases, reverse engineering tools may improve the productivity of a software maintainer both by providing insight into the structure of a program and by making relevant portions of a program readily accessible.

In spite of the potential of these tools, the utility of these tools for many DOD software systems is limited by their ability to extract only information concerning the sequential execution of a computer program. Real-time military systems frequently consist of individual units of execution (tasks) that can operate concurrently on different processors or by interleaving their functioning on the same processor. These concurrent tasks typically exchange both control information as well as data through a variety of mechanisms. However, with the exception of tools that support Ada and its explicit tasking constructs, reverse engineering tools fail to capture information concerning the flow of information between tasks. As a result, these tools provide limited support for understanding the structure of real-time systems

One of the primary reasons for this situation is that understanding design constructs relevant to the execution of concurrent tasks requires more than an implementation level understanding [1] of the software. The syntax of programming languages, particularly older legacy languages such as CMS-2 or Fortran, do not make constructs such as inter-task communication and task synchronization explicit. Instead, the inter-task behavior of a system often depends on the design of the specific operating system and the way in which the application code interacts with the operating system. Since reverse engineering tools only extract information that is represented explicitly in the syntax of the programming language, tools for sequential programming languages can only extract information concerning the sequential execution of individual tasks. These tools will fail to capture information concerning how these tasks interact.

However, people can often extract knowledge about how concurrent tasks interact from the source code of older systems, even though such information is not explicitly available in the syntax of a programming language. Capturing this information requires knowledge about the type of processing model used by the system software and how this processing model has been implemented in a particular system.

In addition, it is also necessary to understand the idiosyncratic techniques used by a system to implement these constructs. For example, although tasks may not be explicitly represented through syntactic constructs in the code, specific recurring patterns of code may be used to represent a task in a particular application. As a result, it may still be possible to recognize those specific portions of code that implement a particular task. Similarly, the specific actions through which these tasks communicate with each other may be implemented through particular types of calls to the real-time operating system. Interpreting how specific tasks communicate with each other will depend on being able to interpret the meaning of these specific calls.

As can be seen, the ability of a person to manually extract extra-linguistic information from the source code of a program depends on his ability to use knowledge about how specific design constructs are implemented in the source code. Reverse engineering tools are not designed to make use of such meta-design knowledge. However, unless techniques are developed to make use of meta-design knowledge, reverse engineering tools will fail to extract more than the implementation level detail of a program. As long as tools can only provide limited visibility into the structure of a program, they will not be able to provide the insight required to understand the design of a real-time system.

2.0 Recovering Task Flows for the MCE System

Although one of the goals of this work has been to develop techniques to recover the inter-task behavior of real-time systems in general, our initial efforts have centered on recovering this information from one system in particular, the Modular Control Equipment (MCE) system. MCE is a command and control system written in the Navy source language CMS-2. It also contains a relatively small amount of embedded assembler language. The assembler code is less than ten percent of the system and is predominately located in the real-time operating system (RTOS). The MCE software runs in a distributed, multiple CPU hardware environment. The software consists of 14 functional subprograms that comprise 44 CMS-2 modules. The software modules are distributed across the different CPUs. RTOS enables the software on different CPUs to communicate, sharing both data and control (task invocation).

Tasks in MCE are executable units within a module and are comprised of many different procedures. Tasks spawn a variety of actions on themselves or other tasks through procedure calls to RTOS. These actions include scheduling a task, terminating a task, or removing a previously scheduled task. We have focused primarily on developing techniques for determining which tasks schedule other tasks although this approach can be extended to recover information regarding other types of operating system calls.

Determining the flow of tasks within MCE requires extracting information that is not directly available in the MCE source code. Extracting the task flows requires extracting the two primary pieces of information required to understand any task flow: who called a task and what task was called. Neither piece of information is explicitly represented in the source code. The following two sections will describe the overall strategy that was required to automatically extract this information from MCE.

2.1 Determining Tasks Called by RTOS

The task scheduled by an RTOS call is uniquely determined by a set of arguments passed to RTOS by the RTOS call. These arguments identify a module and the task contained in that module. A module/task pair uniquely identifies a task in the MCE system.

In order to determine the task spawned by an RTOS call, it is necessary to determine the state of the two variables that uniquely identifies this task at the particular point in the program when an RTOS call is made. In some cases determining the value of these variables is relatively straightforward since these values are set once and then remain constant throughout the execution of that module. Other variables, however, are set multiple times within a module. In these cases it is necessary to statically evaluate a portion of the program that determines the state of the variables.

When a variable is not preset, its state can be determined by identifying and evaluating the set of statements that may impact the value of that variable. Algorithms for identifying the minimal set of statements that may impact the state of a variable are known as program slicing [3]. We implemented a program slicing algorithm to use in identifying the minimal set of statements impacting the module and task variables within an RTOS call. For our purposes, this technique assumed that the state of the module and task variables was completely determined within the scope of a task since the program slicing algorithm does not trace data dependencies across task boundaries. This assumption was valid for all but one module in the MCE system.

For any particular RTOS call, module and task variables may assume different values in different contexts. Because a program slice contains the set of all statements that may influence the state of a variable, only a subset of these statements may actually be executed under a particular context. In order to evaluate each program slice under each possible context, each syntactically possible execution thread within a module that may reach a designated RTOS call is evaluated. Each of these evaluations derives a distinct value for the module and task variables for the particular RTOS call. These values identify the maximal set of tasks that may be called by a specific source code RTOS call.

2.2 Determining the Calling Task

RTOS calls are made within the context of a particular task. A task is said to spawn some action on another task when an RTOS call is made within the context of that task. One of the difficulties in determining task flows is in determining which task spawned a particular action. This is because there exists no syntactic structure, such as a procedure, that corresponds to a task in CMS-2. Therefore, one cannot simply read the source code to determine the task containing a particular RTOS call.

Although no syntactic structure exists in CMS-2 that corresponds to a task, it is possible to determine which task spawns another through a call to RTOS by identifying the calling context for that call. Because this calling context is associated with a task, once the calling context is identified, the task that spawned this call can be identified. To do so, we needed to define a set of recognition rules that could be used to identify occurrences of MCE tasks. These recognition rules were based on our understanding of how tasks were implemented in MCE.

Tasks are activated in MCE when an "entry-procedure" for a module is called by RTOS. This entry-procedure is implemented by a CMS-2 construct known as a p-switch, which will pass control to one of a set of procedures depending on the value of the argument passed by RTOS to the p-switch. Each procedure to which an entry procedure can pass control represents the root procedure of a different MCE task. A task continues executing in MCE until the root procedure terminates.

To recognize the occurrences of tasks, we needed first to identify objects in the code that represented "entry-procedures" for modules. Once these entry-procedures were recognized the root procedures for each task could be identified by tracing through the p-switch. In order to recognize these entry-procedures we used information extracted from external documentation. Since this documentation was available in a structured format, we wrote a parser to extract the relevant information from the documentation. We used this information to select the p-switch in a file that functioned as the entry-procedure for a module. Once the entry-procedure for a module was identified we could identify the root procedure for each task contained in that module.

The identification of the root-procedure for each task provided the knowledge necessary for identifying the context of an RTOS call. As noted in Section 2.1, a specific RTOS call may be made within different contexts, resulting in different values for the module and task variables and hence spawning different tasks. Each calling environment contains a root procedure that corresponds to the calling task. Therefore, determining the task that spawned a new task requires determining the calling environment for a particular RTOS call passed a specific set of module/task values. This was done as part of evaluating each execution thread through a program slice.

2.3 Recognizing Design Constructs

The overall strategy for determining task flows required the implementation of a set of recognition rules that identified a small set of design constructs (e.g., tasks and modules) in the MCE code. This information was then supplemented with techniques for evaluating the states of specific variables in the code that identified the tasks spawned by a particular RTOS call. These evaluations required determining the program slice for the module and task variable in each RTOS call. This program slice was evaluated within the calling environment of each task within the module containing the RTOS call. This evaluation returned a value for the module and task variables together with the calling environment in which these values were computed. These values identified the task spawned by a particular RTOS call and the calling environment identified the task spawning the new task. In this section, we will describe the approach we implemented for recognizing design constructs in MCE source code. In the following sections, we will describe how program slicing was implemented and how a program slice was evaluated to determine the task flow.

The purpose of design construct recognition is to identify instances of the design constructs implemented in a software system and their interrelationships. We created a domain model that identifies both a small set of design constructs in the MCE code (e.g., tasks and modules) and a small set of events through which tasks interact with each other (e.g., task spawning). The current implementation hard codes recognition rules for these design constructs. Each recognition rule creates an instance of an abstract design construct or determines the value of one of its attributes.

Because of the difficulty of recognizing these abstract design constructs from information contained solely in the source code, we implemented recognition rules that operated on both design documentation and the parsed representation of the source code. We were able to identify a portion of the on-line documentation for MCE which described each of the 44 modules of the system. For each module, the module's name and a list of tasks was listed. A list of files relevant to the module and the file that contained the entry procedure for the module were also identified.

Although this documentation was written in English, it was fairly structured. Thus, with a minimal amount of editing, we were able to automatically parse the documentation using a recursive-descent parser written in Refine. The parser automatically created module and task objects for each module and task identified in the documentation. The module and task names, and the list of relevant files for each module, were also set automatically during parsing.

After obtaining as much information about modules and tasks as possible from the documentation, we turned to the source code to complete the model. As noted in Section 2.1, each module is associated with an entry procedure. Because the documentation only identifies the name of the file containing a module entry procedure, we needed to find this procedure from the source code. This is done by generating the procedure calling hierarchy. The module entry procedure is equivalent to the root procedure in the procedure calling hierarchy. To avoid orphan procedures, the root of the largest disconnected subgraph is used. As stated in Section 2.2, the module entry procedure contains a CMS-2 construct called a p-switch. The p-switch passes control to the entry procedure for a particular task depending upon the value of the p-switch variable. Therefore, from the p-switch we were able to determine the names of the task entry procedures for each of the tasks in that module.

Once modules, tasks, and their entry procedures have been recognized, it is possible to determine the behavior of each task by identifying and interpreting RTOS system calls used by a task. Our domain model represents each of the events produced via an RTOS call and its associated attributes. We implemented event recognition algorithms that identify occurrences of these events.

The first step is to find all of the RTOS calls in the source code. This is easy to do by traversing the abstract syntax tree and testing for the name RTOS in each procedure call object encountered. The next step is to evaluate the value of the arguments used by RTOS to determine the task behavior the RTOS call represents. An RTOS call has two arguments, the type of the RTOS call and a table (a CMS-2 data structure) containing information for that type of call. The fields in the table vary depending on the type of RTOS call invoked. For example, if the RTOS call schedules a task, then the table includes two fields which contain the information necessary for the operating system to determine which task to schedule. For each RTOS call identified in the code, the first argument identifying the type of RTOS call is accessed and the appropriate event object is created to represent the event. Our algorithm then determines the task invoking this event and the values of designated fields in the table to determine the value of the event's attributes. This is done by computing and evaluating a program slice for the relevant fields in the table.

2.4 Implementing Program Slicing

A program slice on some variable *v*, or set of variables, at statement *n* consists of those statements that contribute to the value of *v* just before statement *n* is executed. In the current implementation, we compute a program slice from a data flow graph.

A data flow graph is constructed by identifying a set of "reaching definitions" for each variable used in a program. Statement *m* is a reaching definition for variable *v* used by statement *n* when statement *m* defines the value of *v* actually used at *n* through some execution path. Note that a variable *v* in statement *n* may have several reaching definitions under different execution paths. *n* "backward depends" on *m*, and *m* "forward depends" on *n*. A backward (forward) program slice is computed on statement *n* by taking the transitive closure of all backward-depends (forward-depends) relations on statement *n*.

2.4.1 Intra-procedural data dependence analysis: The first step needed to generate an intra-procedural data flow graph is to generate a control flow graph (CFG). A control flow graph for a procedure is a directed graph that contains an initial node which represents the entry point for a procedure and a final node which represents the procedure's exit point and a set of remaining nodes that each represent sequences of simple statements in the procedure represented by the CFG. Each edge in the graph represents a possible flow of control.

The next step in data dependence analysis is to identify the reaching definitions for each *location* used in a procedure. The term location is used instead of variable because it is necessary to keep track of arrays and data structures. Each node in a CFG is mapped to a set of locations defined and a set of locations used in the statement represented by a node. There exists a reaching definition between a definition and a use of a location if there is a path in the CFG between the node that contains the definition of the location and the node that contains the use of the location. Since a location may be defined multiple times within a procedure, there are many potential candidates for the definition that actually reaches a use of a location at a statement. It is possible for a location to have several reaching definitions because the definitions for that location are in the body of conditionals. However, one definition can also cancel another, eliminating the canceled definition as a reaching definition for all subsequent uses of that location.

2.4.2 Inter-procedural data dependence analysis: We extended the concept of reaching definitions to take into account reaching definitions between statements contained in different procedures. Our extensions only consider reaching definitions contained within the scope of a single task. Reaching definitions that occur between tasks are not considered by our algorithm. Inter-procedural data flow analysis considers both global variables and parameter passing between procedures.

In order to support inter-procedural data flow analysis, the process is done in several steps. First, the control flow graph is generated. Second, within each procedure the definitions and uses of a location are computed. Third, reaching definitions are computed for all locations used in a procedure. Finally, the relations forward-depends and backwards-depends are computed. During this process the reaching definitions for global variables are found. Each step is done for all procedures, via a post-order traversal of the procedure calling hierarchy.

The inter-procedural reaching definitions for a global variable use can be in found one of three places: within the procedure (an intra-procedural reaching definition), in a procedure called by the procedure, or in a procedure which calls the procedure. Each of these are considered in order. First, the reaching definitions within the procedure are considered. If the reaching definition is a regular assignment statement, that statement is returned. Second, a procedure call contains the reaching definition, if the global variable was defined within that called procedure. The called procedure must be investigated to find the assignment statement which is the actual reaching definition. The intra-procedural information for all global variables defined within a procedure is summarized in the unique exit node of the CFG, so it is easy to access. Finally, if no definitions are found within the procedure or a called procedure, then all procedure calls related to the procedure must be investigated. This process is a recursive one, traversing the calling hierarchy as needed.

If the variable is a formal parameter, and is not defined within the procedure then the reaching definition must be the one implicit in parameter passing. Therefore all of the calls to the procedure are the reaching definitions. This will work for call by value parameter passing. The issue of aliases (call by reference) or other parameter passing schemes has not been investigated.

An example of a program slice is shown below, with emphasis on the inter-procedural data flow.

```
int some_global_variable;

int p()
{
   int i = 0, z, x = 1, y = 2;
   z = x * y;
   if (i == 0)
   {
         i = 5;
   }
   else
   {
         i = 6;
   }
   t(0);
   z = i + some_global_variable;
   return z;
}

int t(x)
   int x;
{
   some_global_variable = 1;
}
```

The program slice on *z* at *return z* is:

```
p:  i = 5
p:  i = 6
p:  t(0)
p:  z = i + some_global_variable
p:  return z
t:  some_global_variable = 1
```

In this program the value of *z* at return *z* is computed using the previous statement, therefore the value of z depends upon *some_global_variable* and *i*. The value of *i* depends only on statements in procedure *p*. *i* is set conditionally, so both assignments appear in the slice. If conditionals were included in the slice, *i == 0* would also appear. Note that even though it is easy to determine that *i* does equal *0* and therefore *i = 5* is executed and not *i = 6*, both still appear in the slice because there is no way, in general, to determine statically what will happen when the program is executed. The value *of some_global_variable* is set in *procedure t* which is called by *p*. Therefore the call to *t* and the assignment are included in the slice by using inter-procedural data flow analysis.

2.5 Evaluating Execution Threads Through a Program Slice

Once a program slice is available it is possible to evaluate the slice to determine the possible values of the table fields used by an RTOS system call. Evaluation of a slice is made somewhat easier in CMS-2 because procedure invocation does not introduce a new scope. All variables in a CMS-2 program, including formal arguments of procedures, are global.

Inputs given to the evaluator are the variables of interest, a list of the variables for which values are requested, and the statement of interest (i.e., the statement in the slice for which the variable values should be evaluated). The execution of a slice is guided by a pre-order traversal of the procedure calling hierarchy. As it is traversed, each procedure that is encountered may contain some statements that are found in the slice. They are evaluated in the order in which they occur in the procedure (i.e., statements are sorted by line number) and their values are saved for use in other computations of the evaluator. When the statement of interest (the RTOS call in this case) is encountered, the values of the variables of interest are checkpointed. If the statement is encountered again, the values at that time are also checkpointed. Sets of checkpointed values together with the calling environment for each set are returned from the evaluator.

Given the values for the module/task pair and the calling environment, it is possible to compute the calling tasks and the called tasks of the RTOS call. A calling task is one whose entry procedure is contained in the calling environment. A called task is the one that corresponds to the module/task pair. If the module is the same as the one under investigation, the task number is an index into the module's entry procedure p-switch statement, which can be thought of as a list of all intra-module task entry procedures. If the module number corresponds to another module, then information from the documentation is used to determine the name of the task so that it can be displayed in the task flow graph or table.

2.6 Results

We have evaluated task flows for 29 of the 44 MCE modules (the others contain classified information). The results so far are very encouraging. The capability described above has produced the completely correct graph for 8 modules. For 14 modules we generated a graph that is slightly different (~90 % similarity) from the graphs produced by hand. Results are not currently available for the other 7 modules.

There are several factors to account for the discrepancies in these results:

1. Bugs in the program slicer or evaluator. The program slicer and evaluator are under development and could still contains bugs.

2. Limitations of the program slicer or evaluator. The program slicer and evaluator do not handle all conditions. For example, the program slicer does not include conditional statements in a slice. When the slice is evaluated without these conditionals, this could cause certain possible variable values to be eliminated, which could easily explain missing arcs in the graph. Another limitation is that the program slicer currently does not handle recursive procedures. This makes it impossible to generate graphs for two modules.

3. Limitations of the tasking graph generator. In at least one module there exists an RTOS call whose table argument is a variable, not a pointer to a table. To handle this discrepancy a second program slice must be generated, which currently is not done. Also, as stated before, this technique assumed that the state of a module and task variable were completely determined within the scope of a task. This is not the case in at least one module. Both of these limitations cause missing arcs in the module's graph.

4. Modules are too large. Currently, we do not have enough computer resources to generate the data flow information for four of the modules. For example, one of these modules has a parse tree on the order of 100 megabytes of memory, *before* data flow information is generated.

5. Incorrect hand generated tasking graphs. For at least two modules the graphs produced

automatically were correct and the hand generated ones were missing arcs.

We are investigating all of these avenues in order to improve our results. The program slicer is currently being enhanced to handle recursive procedures. With respect to assignments in conditionals, we are investigating a measure to recognize when this is occurring. Eventually both the program slicer and the evaluator must be enhanced to handle conditional code. Additionally, the tasking graph generator must be extended to handle table specifications in RTOS calls via variables. To handle the modules that are too large we are adding more memory for our machines. The main area of focus, however, will be to make the data flow analyzer more efficient in its use of resources. We also need to investigate more of the graphs to see if the hand generated graphs are indeed correct. Lastly, we continue to find and fix bugs.

There are several factors that can be used to measure the success of this effort. First, automating the process to generate task flow graphs reduces the level of effort required to generate these graphs. The process described above will take less than an hour to produce a graph once the data flow analysis has been done. The time required to produce the same graph by hand is on the order of several days to weeks. The reduced level of effort to generate task flow graphs will be particularly useful during maintenance of the MCE system. New versions of the source code are being released on a regular basis. When the new version of the source code becomes available it could be a matter of several days to automatically regenerate all of the graphs for each module. If these graphs needed to be generated by hand it could require significantly more time for a maintainer to notice how changes between versions impact the task flow.

Second, the discrepancies between automatically generated documentation and the manually generated documents point to possible errors in the manually generated reports. As a result, automatically generating documentation will ensure that the documentation that is available to the maintainer is more reliable than may otherwise be possible.

Finally, this process generates on-line documentation for maintainers which is integrated with the source code listing. As a result, it can be used for source code navigation and can be integrated into other reports via hyperlinks.

3.0 Conclusions

The real-time tasking tool implemented for MCE demonstrates that a significant level of design recovery can be obtained when a small amount of design knowledge regarding a system is encoded into a powerful set of tools and then applied in an analysis across the entire system. The current implementation *hard codes recognition rules* for a small set of MCE relevant objects (e.g., tasks and modules) and a small set of MCE relevant events (e.g., tasking spawning via RTOS calls). The notion of "objects" of interest in a program and "events" of interest that relate objects to each other [2] is a generic way to view the design of a software system. In the future, we intend to build a framework that supports the recognition of objects and events in an attempt to capture what can be termed the architecture of a software system. This recognition framework will support the specification of recognition rules for object and event types (versus hard coded system specific rules) and provide powerful visualization facilities for the set of events recognized in a program.

The techniques used in the current system are all static analysis techniques and thus are inherently limited by the degree to which static analysis can be used to evaluate run time behavior. We have implemented a program slicing technique that enables static evaluation of program values where feasible. This capability is currently limited in its ability to deal with name aliasing (an intractable problem) but we are continuing to increase its abilities to resolve aliases and statically evaluate resulting program slices.

By applying powerful program analysis capabilities in concert with recognition rules derived from some basic system design knowledge a significant level of system design recovery can be achieved. The information is derived directly from the source code and traceable back to that source code. As software baselines change, design recovery can be reapplied to produce current design information, yielding a form of "living" documentation that can reliably aid in program maintenance and understanding.

References

[1] T. Biggerstaff
Design Recovery for Maintenance and Reuse
IEEE Computer, July 1989

[2] M. Harandi and J. Ning
Knowledge-Based Program Analysis
IEEE Software, 7-1, 1990

[3] M. Weiser
Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352-357. July, 1984

Maintenance Process Reengineering: Toward a New Generation of CASE Technology

Judith Ahrens[1], Noah Prywes[2] and Evan Lock
Computer Command and Control Company
Philadelphia, PA

Business process reengineering is generating considerable interest in the business community because it can reduce costs and increase an organization's responsiveness to competitive challenges and opportunities. The software maintenance crisis – escalating costs and sluggish responsiveness to changing requirements – is not unlike the corporate problems business process re–engineering has been turning around. This article applies business process re–engineering principles and insights to the problem of software maintenance. Three of its principles are explored in depth: surfacing tacit assumptions and rules, defining processes, and using information technology as an enabler. In particular, three integrated, multi–tool CASE environments are proposed as the enabling technological infrastructure for maintenance process re–engineering. These environments include the Software Specification Environment, the Software Engineering Environment, and the Software Re–engineering Environment. Scenarios of operation illustrate how the integrated environments leverage human resources, not only for maintenance activities, but also for the phased forward development method described in DoD–STD–2167A, for the prototyping and evolutionary development methods proposed in the Software Design Document (DoD–STD–SDD), and for software development based on domain reuse libraries.

1. Introduction

Maintenance consumes up to 70–75 percent of a software system's cost over its lifetime, dwarfing the phases of requirements analysis, specification, design, implementation, testing, and production (Bloom, 1990; CSTB Report, 1990). This statistic is even more stunning when combined with the opportunity cost of diminished funds for replacing existing systems and lengthened payback periods for justifying new systems. Escalating costs and an inability to respond quickly to changing requirements also characterize many problems found in business organizations. Squeezed-by rising costs and global competition, many corporations have turned to business process re–engineering (BPR). Business process reengineering has achieved radical breakthroughs in solving seemingly intractable organizational problems (Hammer and Champy, 1993; Stewart, 1993; Hall, et al., 1993). This article describes the application of BPR thinking to the intractable problem of maintenance in the software life cycle.

BPR is defined as *the fundamental rethinking and radical redesign of business processes to achieve dramatic improvements in performance measured in such areas as cost, quality, service, and speed* (Hammer and Champy, 1993). A *process* is defined as *a collection of activities that takes one or more kinds of input and creates an output that is of value to the customer.* We begin the fundamental rethinking of maintenance in Section 2 by applying the BPR principle of *examining tacit assumptions and rules for error, obsolescence or irrelevance.* This examination revealed that prevailing assumptions about software maintenance regard it either as separate and distinct from software development activities, or as an activity that can be accomplished largely through redesigning software and generating new code with forward CASE tools. Consequently, CASE technology has been neglected that integrates domain and application engineering for software reuse and forward engineering, with the maintenance activities of re–engineering, i.e. reverse engineering, program understanding and restructuring.[3]

Fundamental rethinking is continued in Section 3 where we apply the BPR principle of defining processes. This resulted in a fundamental reconceptualization of mod–

[1] Also affiliated with Drexel University

[2] Also affiliated with University of Pennsylvania

[3] Reengineering includes reverse engineering, program restructuring, software understanding and translating legacy software into modern programming languages. Reverse engineering is the recovery of program design information from its code. Reengineering not only recovers design information from existing software, but may involve program restructuring. In program restructuring, the software engineering uses reverse engineered design information to alter or reconstitute the existing system in an effort to improve its overall quality. In most cases, reengineered software reimplements the function of the existing system. But at the same time, the software developer also adds new function and/or improves overall performance (Pressman, 1992). Program restructuring requires software understanding. To achieve software understanding, programs need to be understood from a number of viewpoints, e.g. scope of variables, concurrent structure, sequential and temporal execution behavior. This reduces the intellectual difficulty of conceptualizing large, complex software systems (Brown et al., 1992) Re–engineering also comprehends conversion of legacy software into modern programming languages, such as Ada and C++. (Pressman, 1992).

ern software practices and the capabilities required for their support. Re–engineering, i.e. maintenance processes, are seen to be present in domain and application engineering and in forward engineering's iterative phases. Although the reasons for initiating these activities differ, they employ identical technical processes. According to BPR, there is no justification for separating software development from maintenance with respect to software engineering jobs and organizational structures, management and measurement systems, and values and beliefs.

After fundamental rethinking comes radical redesign, where the third BPR principle, ***use information technology as an enabler***, is applied. Section 4 describes the capabilities of a new generation of CASE technology and the three advanced prototype environments that implement these capabilities: the Software Specification Environment, the Software Engineering Environment, and the Software Re-engineering Environment. Section 5 illustrates the re– engineered processes and Section 6 summarizes and concludes the article.

2. Examine Assumptions for Errors, Obsolescence, or Irrelevance

Assumption 1: Maintenance starts after software release

In the software engineering literature, several authors have criticized the prevailing assumption about software maintenance because it portrays software practice in an unrealistic manner (Paul and Simon, 1989). This assumption, which originated with the Waterfall model (Pressman, 1992), views maintenance as a separate software life cycle that begins after software enters production. That is, before the first release of the software, all software activities are assumed to be associated with forward engineering.

Assumption 2: Maintenance is defined by administrative concerns

In assumption 2, technical similarities between development and maintenance activities are recognized, but the separation is justified based on administrative concerns. Both new software development and perfective, adaptive, and preventive maintenance (described in Section 3) involve requirements analysis, specification, design, implementation, and testing. Nevertheless, the difference in how these activities are constrained and controlled justifies their separation.

New software development is constrained by customer requirements, as well as by their budgets, existing resources, and acquisition policies. Additionally, software maintenance is constrained by a preexisting software system (Ford and Gibbs, 1989). While new software development control is mandated by distinct developmental phases defined in military standards, (e.g. DoD–STD–2167A, 1988), software maintenance is problematic at best.

For example, software developers know that change occurs from the earliest design stages as initial expressions of customer requirements are refined. After implementation, controlling changing requirements involves controlling ongoing iterations that mix old code (typically with inadequate documentation of original specifications as well as modifications made over time), new programmers, and new technology. The control process is ad hoc and the problem grows over time: the larger the installed base of legacy code, the more formidable the problems.

Assumption 3: Forward CASE technology can perform almost all maintenance

Under this assumption, maintenance is understood to involve not only fixing bugs, but also responding to changes in customer requirements, to new technology, to changes in the external environment, e.g. regulatory changes. Thus, maintenance is viewed as a component of the forward development process, involving modifications to requirements specifications, designs, and source code, testing, and performing configuration management. Most maintenance, therefore, can be performed with the same techniques and products used during forward software development, but some more specific tasks must be approached with ad hoc techniques and tools such as reverse engineering (Fuggetta, 1993). Since technology now permits code to be generated directly from design specifications, most "maintenance" activities can be accomplished through forward redesign and code generation.

Discussion of maintenance assumptions

This section discusses the consequences of the three assumptions. Assumption 1 seems to have legitimized the high costs, poor technological support, and poor management of maintenance activities (CSTB Report, 1990). For example, the management of the same software system may be split between a development manager and a maintenance manager. Similarly, the development and maintenance tasks are frequently allocated to different developmental and maintenance software engineers. In general, maintenance work is perceived as less desirable than new software development, and few programmers aspire to a career in maintenance.

BPR views this condition as a symptom of the industrial organization structure introduced by Adam Smith. This structure is characterized by the specialization of work, which is broken down into simple tasks and assigned to separate organizational units. When work is reengineered into processes, several jobs that cut across organizational boundaries are combined into one, and generalists performing multidimensional work replace specialists (Hammer and Champy, 1993).

Rather than dividing software into specialized development and maintenance activities, we should instead conceptualize software as a process that originates with customer need and ends with software that meets that need. Co.:ceptualizing all software as a process will change the structure of software engineering jobs, the associated management and measurement systems, and people's values and beliefs about the significance of their work.

According to BPR, Assumption 2 errs by organizing processes around administrative concerns rather than around the work itself. This leads to a process that is severely fragmented in.o specialized jobs and administrative and control procedures. Each separate task requires a "handoff" to the next task. Each handoff in turn requires administrative coordination or control, adding to process overhead. The integrated tools described in Section 4 will eliminate the necessity of both administrative and technical handoffs. Additionally, assumption 2 is weak because once new software enters testing, a software system exists.

Assumption 3 suffers from a common failure of BPR efforts, that of defining a process too narrowly (Hall et al., 1993). For example, bugs that appear during testing or after the software is released may be at too low a level than that provided by program design specifications. Hence it is necessary sometimes to understand software at the level of code. Ideally, the software engineer could reverse engineer from the code to a low level abstraction, correct or restructure the abstraction, and then forward engineer new code. However, assumption 2 does not propose the integration of tools for forward and reverse engineering (Fuggetta, 1993).

These assumptions may have influenced developers of CASE technology to neglect tools that support maintenance processes in favor of tools that support forward software development processes (Chen and Norman, 1992). Consequently, the current generation of CASE technology does not support software engineering activities present in the entire life cycle.

To support the entire life cycle, integrated tools are needed having capabilities not only for software specification, design, code generation and testing, but also for reverse engineering, re–engineering legacy software, and evaluating components in a reuse library. Modern software practice thus requires a new generation of integrated CASE technology that supports the complete software life cycle process.

Accomplishing this objective requires coordinated managerial and technological support. Integrated tools that support both software development and maintenance activities could provide the technological foundation, but improving administrative controls can come only from standards that recognize the reality of change in the software life cycle.

SDD – A step in the right direction

The new DoD draft standard for software development, Software Design Document (SDD), is a step in this direction (DoD–STD–SDD, 1992). SDD, in addition to encouraging software reuse, permits the 2167A life cycle model to be supplemented or replaced with alternative models that explicitly recognize change as inherent in large systems. Thus, both mission critical and information systems development under SDD could follow the spiral (Boehm, 1988), the evolutionary (Gilb, 1988), or the prototyping model (Boar,1984). These alternative models are realistic because they recognize the necessity of iteration within and between life cycle phases.

The evolving SDD standard provides an excellent opportunity for the software engineering community to reconceptualize the role of maintenance ac.ivities in the software life cycle. Once it is recognized that maintenance activities include an intensive mix of forward and re–engineering processes, integrated tools and environments can be developed that reduce the time needed to transition among these activities. Administrative controls linked to technology use can then be introduced to monitor and control maintenance costs.

3. Define Processes

This section identifies the capabilities required to support maintenance processes in the software life cycle. Maintenance and new software development are shown to require many common capabilities, indicating the need for integrated toolsets and environments. Maintenance processes have been studied by the National Institute of Standards and Technology (NIST) (CSTB Report, 1990). A description of each maintenance process and the percentage of time devoted to each follows:

Perfective maintenance, or enhancements (50% or more) introduces major transformations in form, functions, anc objectives.

Adaptive maintenance, (25%) responds to changes in the external environment, including conversion of legacy code ir'o modern programming languages.

Corrective maintenance, (20%) includes diagnosis and correction of design, logic or coding errors.

Preventive maintenance, (5%) improves future maintainability and reliability.

Along the horizontal axis of Table 1 are listed the maintenance processes and the percentage of time devoted to each. Along the vertical axis are listed the capabilities required by a software engineer from CASE tools in order to perform these processes effectively and efficiently. Note that some capabilities, such as testing, understanding the existing system's software architecture and interfaces, and configuration management, apply to multiple processes.

Software engineers spend most of their time thinking about software. A perusal of these capability requirements suggests that software engineers could devote less time to maintenance processes if tools were available for facilitating the understanding of complex software relationships and the performance of *thought–intensive cognitive tasks*. For example, tools for software understanding could reduce the time needed for all maintenance processes. Tools for creating and updating requirements specifications could reduce the time devoted to perfective and adaptive maintenance. Tools for automating the translation of legacy software into modern programming languages could significantly impact adaptive maintenance.

4. Use Information Technology as an Enabler

BPR success depends upon information technology. Integrated databases, networks, computer–supported cooperative work environments, client–server architectures and expert consultative systems provide BPR infrastructure. Maintenance process reengineering requires a CASE technology infrastructure. This section proposes such an infrastructure based on the research and development work at Computer Command and Control Company (CCCC). CCCC has been engaged in developing software reengineering and specification technology as part of the research and development programs at the Navy Surface Warfare Center (NSWC) and the Joint Logistics Commanders

	Maintenance Process			
Required Capabilities	**Perfective (50%)**	**Adaptive (25%)**	**Corrective (20%)**	**Preventive (5%)**
Generate current software abstractions (specifications) from implementeo software.	X	X	X	X
Generate/revise/confirm software specifications from a specification reuse repository.	X			
Reengineer/update a domain from specification.	X	X	X	X
Analyze commonality/ variability in the domain to engineer the application.	X	X		
Query, retrieve, analyze, and understand reuse repository of code specifications.	X			
Generate code from specifications.	X	X	X	X
Perform software concurrency analysis.	X	X	X	X
Perform software simulation.	X	X	X	X
Testing.	X	X	X	X
Configuration management.	X	X	X	X
Convert from legacy language to modern language.		X		X
Understand converted legacy software.		X		X
Restructure system boundaries to support partial retirement of legacy systems.		X		X
Reorganize/restructure converted legacy code into new programming paradigms, e.g. object–oriented.		X		X
Extract components from legacy systems for reuse libraries and/or new systems.		X		X
Migrate to new development and production environments, e.g. open systems.		X		
Understand code visually, (i.e. graphically) from different perspectives.	X	X	X	X
Manipulate visual code representations.	X	X	X	X
Generate new code from visual representations.	X	X	X	X

Table 1: Maintenance Process/Required Capabilities Matrix.

(JLC). These types of tools are being integrated with the forward softwareengineering tools of the Domain Specific Software Architecture (DSSA) (Mettala and Graham, 1992) and with the ARPA/STARS Software Engineering Environment (SEE) (Foreman, 1992).

Figure 1 shows how these environments are integrated to support the complete software life cycle processes.

Software Specification Environment (SSE)

The SSE appears at the top left of Figure 1. SSE facilitates the creation and updating of software specifications conforming to DoD Standard 2167A (DoD–STD–2167A, 1988). Subsystems store, compose, and update Data Item Description (DID) documents, including the System/Segment Specification, the System/Segment Design Document, the System Requirement Specification, and the Interface Requirements Specification.

The inclusion of SSE reflects the importance of software specifications for an orderly software life cycle. SSE is an integrated set of information repositories and tools. SSE guides, instructs and informs staff in composing, updating and evaluating preliminary requirements and specifications. Typical users of SSE are Development Managers, Software Support Activities, or Contractors. SSE allows a user to manage, query and update its application system repository. Staff may ask complex technical questions about the software specifications and retrieve answers. Fragments of retrieved answers can be extracted for inclusion as updates to relevant new documents. SSE leads the inexperienced specifier in a "step–by–step" manner and provides traceability to the source documents used to update specifications. SSE has been used in a demonstration project for the Tactical Air Mission Planning System (TAMPS) program at the Naval Air Warfare Center, Warminster, PA.

SSE subsystems include:

- Document Manager: This is used by the data administrator to create and catalogue documents in the repository
- Assignment Manager: This is used by the manager to enter the work plan for subordinates who compose or update documents
- Step–by–step: This is used to guide specifiers in searching previous documents and composing or updating requirements and DIDs
- Evaluate: This is used to provide feedback on completeness of DID coverage

These subsystems are supported with commercial off–the–shelf software (COTS):

- Document loading and publishing – Interleaf
- Editor – MS WORD and Wordperfect
- Search – Zyindex
- CASE – depends on use by Program Office

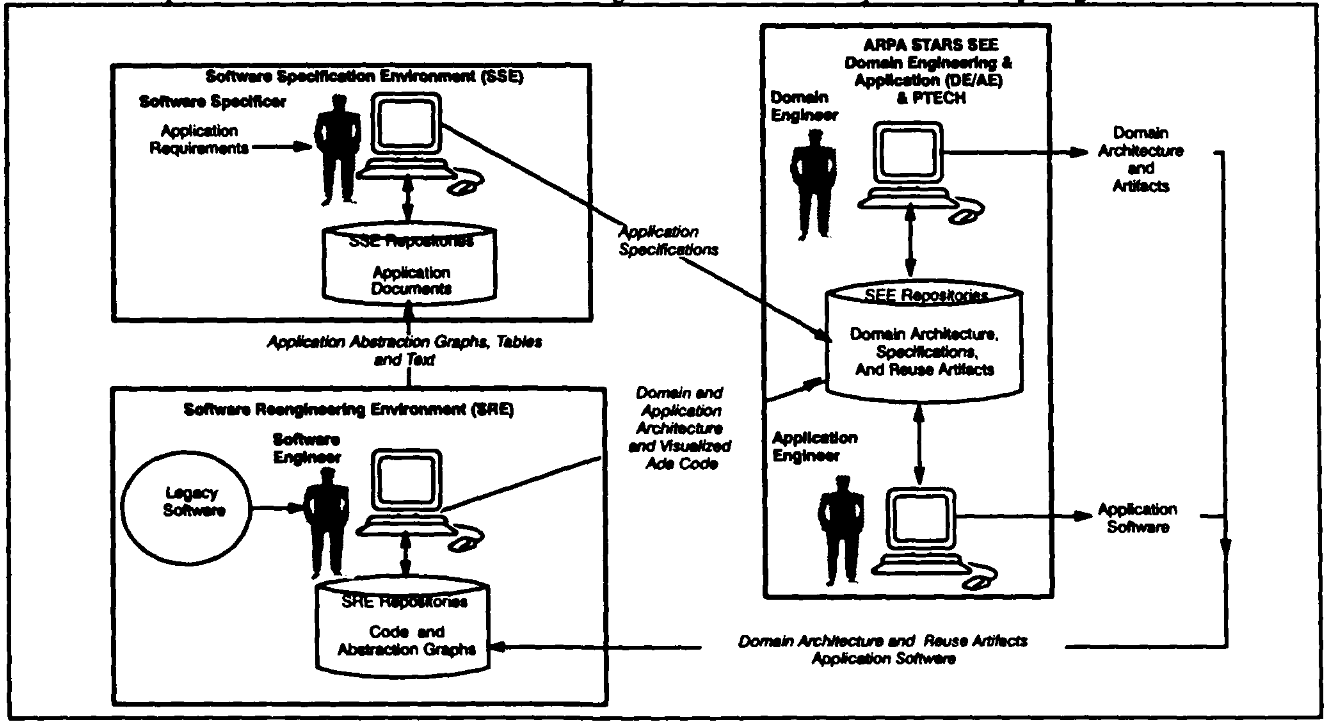

Software Specification Environment (SSE)
Software Specifier
Application Requirements
SSE Repositories
Application Documents
Application Specifications
Application Abstraction Graphs, Tables and Text
Software Reengineering Environment (SRE)
Software Engineer
Legacy Software
SRE Repositories
Code and Abstraction Graphs
Domain and Application Architecture and Visualized Ada Code
ARPA STARS SEE
Domain Engineering & Application (DE/AE) & PTECH
Domain Engineer
Domain Architecture and Artifacts
SEE Repositories
Domain Architecture, Specifications, And Reuse Artifacts
Application Engineer
Application Software
Domain Architecture and Reuse Artifacts
Application Software

Figure 1: Overview of Integrated Tools for the Software Life Cycle

Software Engineering Environment (SEE)

The SEE (Foreman, 1992) is shown at the top right of Figure 1. SEE incorporates new software development technology for software reuse and for automatic program generation, following ARPA's Domain Specific Software Architecture (DSSA) Program (Mettala and Graham, 1992). The ARPA STARS SEE for NAVAIR PMA 205 includes Domain Engineering (DE), which enables a domain engineer to define the process of producing software for a class of related applications in a domain, and Application Engineering (AE), which enables an application engineer to produce software for an application that belongs to the domain (SPC, 1992). The Domain Engineering and Application Engineering (DE/AE) facilities are language independent.

DE/AE facilities are employed as follows. A specific domain is comprised of software for a closely related family of applications. Once a domain software architecture (Agrawala, 1992) is developed, applications can be generated. Domain Engineering contains a repository of reuse software artifacts and associated tools. The creator of a domain architecture is called the Domain Engineer. The DE/AE facilitates selection of reuse software and generation of software to create a specific application system. The user of the repository and of the tools is called the Application Engineer (SPC, 1992).

The reuse software is part of the DE/AE repository (see Figure 1). The reuse software is organized as a hierarchy of software artifacts that follow the domain architecture. For example, DoD software follows a standard hierarchical tree structure of software units called System, Segment, Computer Software Configuration Item (CSCI) and Software Unit (DoD–STD–2167A). Each software unit in the hierarchy has a specification of its position in the architecture hierarchy, its capabilities, interfaces and dependencies on other software units. Six types of Software Abstraction Documents are created to document the architecture. These include:

- Hierarchical decomposition diagram: This graph shows the decomposition of the overall software into hierarchical units
- Flow diagram: This graph shows the flow of data and control within and between hierarchical units
- Interface table: These tables show the structure of inputs and outputs of each hierarchical unit
- Object/Use diagram: This graph shows, for each hierarchical unit, where types or generics are defined and where they are used
- Context diagram: This graph shows the library units and where they are used
- Comments text: This text records the software comments found in each hierarchical unit. They are assumed to contain information about the hierarchical unit's capabilitities

The capabilities of the hierarchical software units determine commonality and variability among them. It is possible to navigate through the domain hierarchy tree by referring to capabilities and selecting hierarchical software units based on commonality and variability of their respective capabilities. Hierarchical software units may be parameterized and a code generation tool may be used to select parameters of generic software. Alternatively, hierarchical units may be completely generated based on models of their functionality. A series of tools is also available in the DE/AE for application modelling, unit testing, and conversion to concurrent operations.

The SEE also contains PTECH (PTECH, 1992). PTECH permits a software engineer to design object–oriented software and to generate object–oriented programs in Ada and in C++ automatically.

Finally, the SEE incorporates meta tools for tool integration.

Software Reengineering Environment (SRE)

The SRE (CCCC, 1992) is shown at the bottom left of Figure 1. The SRE incorporates the technologies of software visualization and visual programming. *Software visualization* overcomes the essential invisibiiity (i.e. non–physical quality) of software by representing the program structure, control flow, and data graphically. An abstract, graphical representation can facilitate a software engineer's visual perception and cognitive understanding of complex software during debugging, monitoring, and especially, program restructuring (Roman and Cox, 1993; Meyers, 1990).

Visual programming is a methodology of programming as well as of program maintenance. In visual programming, a graph is composed and edited on the screen of a terminal, primarily through use of a pointing device. This is contrasted with conventional textual programming of keying–in textual statements.

To date, software visualization and visual programming technologies have been developed for forward software engineering, facilitiating software design to be followed by implementation. The design diagrams in these systems also produce software code, partially automatically and partially manually. In contrast, the SRE derives design diagrams automatically from software code. In this way, maintenance can be performed on the reverse engineered design which is consistent with the old code, and the entire new code can be produced automatically.

The SRE has three main capabilities:

(i) Software understanding:

Software Understanding consists of query and retrieval of graphic diagrams that illustrate the software from various perspectives. These diagrams are used to visualize specific aspects of the software. The diagrams are first divided into in– the–large and in–the–small diagrams. In–the–large diagrams help a software engineer to visualize declarations of objects. In–the–small diagrams help a software engineer to visualize execution statements within individual program units.

Ada program diagrams are stored in a graphic form in the repository of a customized CASE system. A graphic query language is provided for ad–hoc browsing of the Software Abstraction Documents in the graphic repository. These graphs show relations between high or low level hierarchical units. This facilitates the understanding of the software's architecture as well as its detailed code. Changes to the program for debugging or program restructuring can be made via the graphics used for visualization.

Examples of graphic query retrieval capabilities in the SRE for understanding and creating reusable software include:

- Display Base View
- Query Base View to create Subview
  - Select root node, e.g. "within Package x..."
  - Select node type, e.g. "show me all of the procedures..."
  - Select relations, e.g. "and their Input/Output"
  - Select depth, e.g. "any children...?"
- Query subviews as needed for progressive graphical browsing
- Save subviews as needed for documentation
- Generate Ada code from any view

(ii) Software abstraction and documentation

SRE partitions the software into multi–level hierarchical software units conforming to the 2167A standards for describing the software architecture. Software Abstraction Documents are then generated that describe the architecture of these units from different perspectives.

(iii) Software capture and its transformation to Ada

The SRE translates CMS II code, statement by statement, into a pseudo–Ada, called Elementary Statement Language (ESL) Ada. Next, the ESL–Ada programs are transformed into the Ada programming paradigm in a series of passes that achieve 100% translation to Ada. Each pass translates different aspects of the programming paradigm of the source language into the Ada programming paradigm (e.g., object declarations and execution statements). During the transformation process, seven different sets of relations, each defining a different view of the associations among programming objects, are generated. At the end of the transformation process, these relations are converted into graphic structures in the form of Entity–Relation–Attribute (ERA) diagrams. The ERA representation is the main vehicle for graphic program analysis and visualization. Visualization is used for query, retrieval, understanding, restructuring and generating the documentation of programs.

The integration of the SSE, SRE, DE/AE and PTECH is accomplished through two interfaces, also shown in Figure 1. Their descriptions follow:

Interface between SRE and SSE:

This interface is shown at the middle left of Figure 1 (Prywes et al., 1993). This interface provides a reverse process to produce information for the software requirements and specifications and other documentation from program code. SSE receives the documentation from SRE. The Software Abstraction Documents map into specific paragraphs of the 2167A/SDD specifications, as shown in Table 2. For example, Table 2 shows that the Hierarchy graph can be used for the System Architecture diagram required in paragraphs 3.1 and 3.2.3 of the System/Segment Specification of 2167A and SDD.

Interface between SRE and SEE

This interface is shown in the left–hand portion of Figure 1 (Prywes and Lee, 1993). The SRE provides DE/AE and PTECH with software documentation in the form of high–level graphic views of the architecture as well as detailed graphic views of algorithms. The SRE can process legacy code as well as reuse code from the DE/AE repository. The SRE generates key parts of the specifications of each hierarchical software unit (Table 2). The capabilities of each hierarchical software unit in the specifications are employed to establish commonality and variability among the domain architecture's hierarchical software units.

Tools for the SSE and SRE environments and their interfaces were developed by CCCC.

Abstraction Document	S/SS System /Segment	S/SDD System /Segment	SRS CSCI	IRS CSCI
Hierarchy Diagram	Par. 3.1, 3.2.3 System Architecture Diagram	Par. 4 System Architecture Diagram	Par. 3.1 CSCI External Interface Diagram	Par. 3.1. CSCI Internal Interface Diagram
Flow Diagram			Par. 3.3 CSCI Internal Interface Diagrams	
Interface Table			Par. 3.3 CSCI Internal Interface Diagrams	Par. 3.x.1 Data Element Table
Context Diagram	For Ada Compilation			
Object/Use Diagram	For Object Orientation			
Comments Text	For Capabilities			

Table 2: Mapping Software Abstraction Documents into Software Specifications.

The required maintenance capabilities listed in Table 1 are now mapped into their respective environments in Table 3.

5. Illustrations of Re–Engineered Life Cycle Processes

Several scenarios illustrating iterative use of the tools shown in Figure 1 are possible. The scenarios depend on the history of software development, previous life cycles and maintenance upgrades. Typically, the tools will be used iteratively until a desired new or maintained application system is obtained. The following two scenarios illustrate the need for integrated forward, reverse, and reengineering capabilities throughout the software life cycle. The first scenario illustrates how the integrated capabilities facilitate software development with DE/AE reuse repositories. The second illustrates how the integrated capabilities facilitate the re–creation of software specifications.

Facilitating software development with domain and application engineering reuse repositories

Assume that totally new application software is desired. The preliminary requirements have been generated by the application's Program Manager. The platform to be used and its dynamics may be derived through modelling and simulation. The Specifier, with the aid of SSE, uses the requirements to compose hierarchically structured specifications. The capabilities are then communicated to the Application Engineer who uses the specifications to select architecture units from the domain and to generate new code to create application software. If unable to do so, the Domain Engineer may be called to expand the scope of the domain. Expanding the scope of a domain requires understanding the impact of commonality and variability in capabilities of each architecture unit and its code. In either case, the SRE tool will be used to document and display the new domain and/or application software. Software Abstractions will be reverse engineered from the code, giving the architecture of the application software. The Software Abstractions are next used by the Specifier, who employs SSE to update the specifications. The Domain Engineer will use the Software Abstractions to verify and update the domain software. The Application Engineer will use the reverse engineered software abstractions to document the application software. This cycle may be repeated a number of times until satisfactory application software is realized.

The SRE can process software from the various sources (domain, application, legacy) and augment the domain architecture to satisfy new capabilities. Visualization graphs and Software Abstractions portray the architecture of the system as well as Ada code artifacts. The Software Abstractions are communicated to the SSE so that the Specifiers can incorporate them in updated specifications. The abstractions and code are communicated to the Domain Engineer who can use them to update the domain. The abstractions and program visualizations are communicated to the Application Engineer who can use them to create documentation of the implemented application software.

Facilitating re–creation of software specifications

Reverse and re- engineering capabilities can also play a significant role in the interim before domain and application engineering become a reality. For example, the reverse engineering capability can be used to confirm software specifications developed by outside vendors. In the context of new software development, DoD Standard 2167A (1988) prescribes that software specifications for mission critical software be partitioned into Computer Software Configuration Items (CSCIs). CSCI development is contracted to outside vendors. Mandated periodic reviews determine whether the code under development meets the original specifications. Program Offices receiving delivery of new code, as part of a review, need to confirm conformance of code to specifications.

	Software Specification Environment				Software Engineering Environment			Software Reengineering Environment		
Required Capabilities	Document Manager	Assignment Manager	Step-by-Step	Evaluate	Domain Engineering	Applic. Engineering	Define Domain Specific Reuse Oriented Processes	Capture and Transformation	Understanding	Abstraction and Documentation
Generate current software abstractions (specifications) from implemented software.					X	X				Software Abstraction Documents
Generate/revise/confirm software specifications from a specification reuse repository.					X					
Reengineer/update a domain from specification.						X				
Analyze commonality/variability in the domain to engineer the application.			X			X				
Query, retrieve, analyze, and understand reuse repository ofcode specifications.			X						Graphic query & retrieval	X
Generate code from specifications.						X				
Perform software concurrency analysis.						X				
Perform software simulation.						X				
Testing.							X			
Configuration management.	X	X		X	X	X	X	X		X
Convert from legacy language to modern language.								X		
Understand converted legacy software.			X						Structured Repository	X
Restructure system boundaries to support partial retirement of legacy systems.					X	X		X	Graphic query & retrieval	Visualization from different perspectives
Reorganize/restructure converted legacy code into new programming paradigms, e.g. object-oriented.					X	X	X	X	X	X
Extract components from legacy systems for reuse libraries and/or new systems.	Specification Repository	Process Mgt.			X	X	X	X	X	X
Migrate to new development and production environments, e.g. open systems.					X	X		X		
Understand code visually, (i.e. graphically) from different perspectives.			X						X	X
Manipulate visual code representations.									X	X
Generate new code from visual representations.						X				

Table 3: Required Capabilities/CASE Technology Matrix.

This confirmation can be accomplished by reverse engineering the delivered software and comparing the software abstractions to the original specifications. The earlier deviations are found, the greater the future savings in development time and cost. The reverse specification capability would also enable more objective formal reviews during software development because the Software Abstraction Documents can be compared to the original specifications.

Newer software life cycle methods, such as those that include prototyping, joint application development, and evolutionary development, will increase the frequency with which the reverse specification capability will be needed because these methods do not assume the existence of a complete set of requirements before design and program development begins. Additionally, changes typically are made quickly at a user's request at the level of code, not specifications. Therefore, Software Abstractions generated automatically from evolutionary or prototype software can keep the specifications current at very little cost.

The reverse engineering capability is also needed for the software maintenance activity of updating obsolete specifications or creating specifications for undocumented software. Software managers are continually confronted by problems associated with outdated or unavailable software specifications. Updates to specifications typically lag updates to software. Pressured by time and budget constraints, it is convenient to make modifications to the code and neglect the corresponding changes to the respective software specifications. This is a critical problem since the specification plays a central role in the contracting, scheduling, planning, design, implementation and post–deployment support. The reverse specification capability will make it easier for software program managers to update obsolete specifications and create specifications where none existed.

6. Summary and Conclusions

This article described the application of three business process reengineering principles to the problems of software maintenance within the software life cycle. The first principle, *examine tacit assumptions and rules for error, obsolescence or irrelevance*, revealed that prevailing obsolete assumptions about software maintenance regard it either as separate and distinct from software development activities, or as an activity that can be accomplished largely through redesigning software and generating new code with forward CASE tools. The case was made for technology that integrates DE/AE for software reuse and forward engineering, with the maintenance activities of re–engineering, i.e. reverse engineering, program understanding and restructuring.

The second principle, *define processes*, resulted in a fundamental reconceptualization of modern software practices. Re–engineering, i.e. maintenance processes, were shown to be present in software development, i.e., domain and application engineering for software reuse and forward engineering's iterative phases. Although the reasons for initiating these activities differ, they employ identical technical processes. According to BPR principles, there is no justification for separating software development from maintenance with respect to process definition, software engineering jobs and organizational structures, management and measurement systems, and values and beliefs.

The third principle, *use information technology as an enabler*, was employed to describe the design of CASE technology having capabilities supporting all software life cycle processes. These capabilities include:

- generating, revising, and confirming software specifications,
- generating new software,
- performing domain and application engineering to select, analyze and generate software from reuse libraries,
- reengineering legacy software into modern programming languages,
- visualizing, understanding and restructuring software for maintenance and quality improvements,
- reverse engineering software to recapture its design.

Three advanced prototype environments that implement these capabilities were de<scribed: the Software Specification Environment, the Software Engineering Environment, and the Software Reengineering Environment.

Illustrations of the reengineered processes described how the integrated environments leverage human resources not only for maintenance activities, but also for the phased forward development method described in DoD–STD–2167A, for the prototyping and evolutionary development methods proposed in the Software Design Document (DoD–STD–SDD), and for software development based on domain reuse libraries.

In conclusion, these integrated environments have the potential to enable the DoD to realize the maintenance performance improvements in cost, quality, service and speed promised by BPR. This article demonstrated that the maintenance process can be re–engineered. However, *implementing* a re–engineered process requires concomitant changes in the job descriptions, organizational structures, management and measurement systems, and values and beliefs of the organizational participants. Certainly, the Capabilities Maturity Model deserves close examination for its contribution to the management and measurements aspects (Humphrey, 1989).

Although BPR can be a formidable implementation task, the new generation of CASE technology described above can provide the infrastructure needed to begin reversing the maintenance crisis.

References

Agrawala, A., et al, "Domain–Specific Software Architectures for Intelligent Guidance, Navigation & Control," Proceedings of the DARPA Software Technology Conference 1992, Los Angeles, CA, April 1992.

Bloom, P., "CASE Market Analysis," Volpe, Welty and Co., San Francisco: 1990.

Boar, B., *Application Prototyping*, Wiley-Interscience, 1984.

Boehm, B., "A Spiral Model for Software Development and Enhancement," *IEEE Computer*, Vol. 21, No. 5, May 1988, pp. 61–72.

Brown, A, Earl, A, and McDermid, J., *Software Engineering Environments: Automated Support for Software Engineering*, McGraw-Hill, London, 1992.

Computer Command and Control Company, Technical Report, "Software Intensive Systems Reverse Engineering", prepared under Naval Surface Warfare Center Contract N60921-90-C-0298, April 1992.

Chen, M., and Norman, R., "A Framework for Integrated CASE," IEEE Software, March, 1992, pp.18–22.

CSTB Report, "A Research Agenda for Software Engineering," CACM Vol. 33, No. 3, March 1990, pp.281–293.

DoD-STD-2167A, 1988, "Defense System Software Development," September, 1988.

DoD-STD-SDD, 1992, "Software Design Document," Draft December, 1992.

Ford. G., and Gibbs, N. "A Master of Software Engineering Curriculum," IEEE Computer, September, 1989, pp.59–70.

Foreman, J. "STARS: State of the Program," STARS '92 Conference, 1992, pp. 20–41.

Fuggetta, A., "A Classification of CASE Technology," IEEE Computer, December, 1993, pp. 25- 38.

Gilb, T. *Principles of Software Engineering Management*, Addison-Wesley, 1988.

Goldman, S.L. and Roger N. Nagel, "Management, Technology, and Agility: the Emergence of a New Era in Manufacturing," International Journal Technology Management, Vol. 8, No. 1/2, pp. 18–38, 1993.

Hall, G., Rosenthal, J., and Wade, J. "How to make reengineering really work", Harvard Business Review, November-December, 1993, pp. 119–133.

Hammer, M. & Champy, J. *Reengineering the Corporation, A Manifesto For Business Revolution*, HarperCollins, 1993.

Humphrey, W. *Managing the Software Process*, Addison Wesley, Reading, MA: 1989.

Mettala, E., and Graham, M. "Domain Specific Software Architecture Program," Proceedings of the DARPA Software Technology Conference 1992, Los Angeles, CA, April, 1992.

Meyers, B., "Taxonomies of Visual Programming and Program Visualization." J. Visual Languages and Computing, Vol. 1 No. 1, 1990, pp. 97–123.

Nielsen, J., "Non-command User Interfaces," CACM (36), No. 4, April 1993, pp. 83–99.

Paul, J. and Simon, G. "Bugs in the system: Problems in federal government computer software development and regulation," U.S. Government Printing Office, Washington, D.C., September, 1989.

PTECH Design and PTECH Code, Release 3.5, Tool User's Guide," Associative Design Technology, March, 1992.

Pressman, R. *Software Engineering: A Practitioner's Approach*, 3rd ed,McGraw Hill, NY: 1992, pp 680–683.

Prywes, N., Ingargiola, G., and Ahrens, J., "Automatic Reverse Engineering of Software to Confirm/Update Requirements Specification," Computer Command and Control Company, Philadelphia, PA, 19103, June, 1993a.

Prywes N., Lee, I. "Integration of Software Specification, Reuse and Reengineering," Computer Command and Control Company, Philadelphia, PA, 19103, June 1993b.

Roman, G. and Cox, K. "A Taxonomy of Program Visualization Systems," IEEE Computer, December, 1993, pp. 11–24.

SPC "Domain Engineering Guidebook," Technical Report SPC-92019-CMC, Software Productivity Consortium, December 1992.

Stewart, T., "Reengineering: The Hot New Managing Tool," Fortune, pp 41–48, August 23, 1993.

Texas Instruments and Price Waterhouse, "Reengineering for Information Engineering White Paper," October 5, 1992.

A Syntax-Directed Tool for Program Understanding and Transformation*

William G. Griswold Darren C. Atkinson

Department of Computer Science & Engineering, 0114
University of California, San Diego
San Diego, CA 92093-0114 USA

Abstract

Software maintenance is often too expensive. Part of the problem is that the repeated modifications of a software system degrade its structure, making it difficult to understand and modify. Semantically-rich techniques can help restore the structure of a system, but they may require concurrency analysis, timing analysis, or dependence analysis on pointers, which are difficult to implement efficiently. We propose a fast, programmable tool that can perform syntactically-oriented text processing tasks for use in program understanding and transformation. Because our tool is syntactically-oriented, the tool user must find ways to discover the required semantic information. However, we believe that syntactic information complemented by design and domain knowledge is often sufficient to obtain the needed semantic information. However, to do so may entail iteratively refining a query to find the right information, requiring a fast tool. We take a "traditional" compilers approach to the problem to provide a tool with the flexibility and speed of UNIX tools like awk. Early performance measurements suggest that this approach can produce results substantially faster than previous approaches.

1 Introduction

Software is perceived to be too expensive relative to its quality. Since maintenance is the dominant phase in the program life-cycle [16], substantially reducing the cost of software requires lowering the cost of maintenance. A significant part of this cost is due to the fact that as useful software ages, modifications to meet the needs and demands of users are layered upon the original implementation. Modifications unanticipated in the design sometimes are not easily integrated into the existing implementation, requiring changes to multiple modules in the system to complete a single change [19]. As repeated modifications are made, the design and implementation become increasingly less understandable until maintenance becomes unacceptably expensive, and the only solutions are to reimplement the system or restructure it [3][2, p. 113].

An example of a system experiencing this problem is the Comprehensive Health Care System (CHCS), an on-line system for keeping track of hospital patient records, billing, communication, and prescription ordering. After evolving from an initial prototype, it is now eight years old and consists of approximately 600,000 lines of code. Currently, to make a single enhancement to the system costs a minimum of $100,000.[1]

Semantics-based techniques are one method of improving the maintainability of such systems. For example, a technique called ***tool-assisted program restructuring*** can help improve a system's degraded structure [10]. A restructuring tool user, based on the needs of the desired enhancements of the system, uses the tool to transform the system (while preserving its meaning) to a more appropriate structure—in particular, one in which the enhancements can be made as local changes to the system, rather than as changes throughout the system. There are two benefits of this approach. First, the structure of the system is improved, allowing changes to be made more easily. Second, the tool performs all the global changes required to make a structural change, and ensures that the input-output behavior of the program is unchanged, guaranteeing no errors are introduced by the restructuring and freeing the programmer to concentrate on choosing the appropriate structure. If the tool cannot

*This work supported in part by SAIC and NSF Grant CCR-9211002.

[1] Emmett Burke, SAIC, Personal Communication.

perform a transformation so that it preserves meaning, it prohibits the structural transformation and reports the source of the problem.

Unfortunately, the cost of implementing and running such tools can be prohibitive. The restructuring techniques use global data flow analysis, which is at best difficult to implement efficiently and precisely in the presence of pointers. Furthermore, in a system such as CHCS, which is implemented in MUMPS [15], an **execute** command is frequently used to translate a text string into code that is then executed. CHCS also uses concurrency. Such features make cheap, precise data flow analysis untenable. For example, **execute** commands might invoke any instruction on any data structure. Unless the cost cf implementing such analyses can be amortized over several systems, the cost may not be warranted.

However, our investigations of how programmers and designers work have led us to hypothesize that there may be alternative or complementary approaches that are easier to implement and also support a variety of important tasks in software maintenance. First, we believe that much of the semantic information that is expensive to compute, such as data flow information, can be inferred straightforwardly from the program source if some additional information can be obtained from the design of the system or details of the application domain. For example, if it is known that the execution of an **execute** command in CHCS never causes side-effects, then it can be inferred that they have no impact on pointer dependences. Second, we believe programmers and designers frequently want to perform tasks related to restructuring—that is, tasks that are global but semantically constrained—but for which no specialized tool exists. However, if the information required to make the change can be found inexpensively, the programmer knows how to systematically make the changes, perhaps with the help of a tool.

Our approach, then, is to design a syntax-directed program understanding and transformation tool that depends on the tool user to employ design and domain information to make up for the lack of detailed data flow information. Such a tool reads in a script that specifies a program understanding or transformation task to be performed, reads in the system to be processed, and then performs the task on the system. Because the task can be specified on the spot, it can use characteristics of the system known to the programmers (but not readily inferred from the program text) to cheaply acquire needed semantic information.

Such a tool must be exceptionally fast, easy to use, and flexible. If the tool is not fast, programmers sometimes will not use the tool for reasons of expediency, increasing the chance of incorrect changes. A tool that is fast will encourage programmers and designers to pose "what-if" queries to freely explore possible enhancements and design alternatives. Also, there is a complex relationship between syntactic information and semantic information, so several syntactic queries may have to be performed and successively refined to obtain the desired semantic information. If it is not easy to specify program understanding and transformation tasks, programmers will again be tempted not to use the tool. If the tool is not flexible, then it is likely to be outgrown, so more time and effort must be invested in other tools.

Existing tools, such as TXL [8] and REFINE [7], provide much of the needed functionality, but employ computational paradigms that compromise performance, ease of use, or flexibility, limiting their ability to perform quick queries or extensive analyses on large systems. Our hypothesis is that both flexibility and good performance can be achieved by understanding what programmers and designers need, and by using existing compiler technology. As Johnson argued, the theory and practice of compilers have yielded tools, insights, and heuristics that significantly simplify the construction of a compiler [13]. We employ standard, highly-optimized parser tools such as lex [14] and yacc [12], and build Abstract Syntax Trees (ASTs) in a low-level language (C) to achieve good performance and compact representation. Because lex and yacc generate much of the C code, and the basic technology for describing, building and searching ASTs is well-understood, it is not a burden to use a low-level, efficient language like C for the implementation of such a tool. Only the scripting language provided to the tool user needs to be high-level, and this can be implemented in C directly on top of the abstract interface to the AST. The language's computational model is inherently imperative so that it is easy to use by traditional programmers.

Our goal, then, is to provide low-cost program understanding and transformation with a flexible, efficient tool in the style of the UNIX tool awk [1], except with syntactic, rather than lexical processing. The matching and transformation features of our tool's scripting language make straightforward tasks easy to specify, but the general-purpose features of the language allow performing more complicated tasks such as graphics. Because the task language of the tool is generic and the parser and AST are adaptable, it can easily accommodate a variety of programming languages.

Preliminary results indicate that our tool can perform a simple interrogation of the 1 million MUMPS commands of CHCS in under 10 minutes, including parsing, AST construction, and simple matching. Other techniques require orders of magnitude more time or space to perform the same task.

In the following we examine two typical scenarios in which a program understanding and transformation tool can be expected to be used. We then discuss the designs of two existing tools, describe the design and implementation of our tool, and its customization to a specific programming language.

2 Tool use and design

We have identified two key scenarios in which a program understanding and transformation tool might be used. These have helped identify key requirements and have driven our design choices. We use this analysis to examine the design decisions of two existing tools.

2.1 Designer's scenario

A designer may wish to discover how various parts of the system are related in order to assess the difficulty of proposed enhancements and to plan evolution of the system's structure in order to accommodate those enhancements. For example, the designer might wish to know the set-use properties of the system, that is, which functions use which global variables. Potential modules can be inferred by clustering functions according to how they share information via variables [21]. Using a syntax-directed tool, the designer needs to specify a one-pass traversal over the AST to find the variable references in each component, recording in a table all references to global variables. When the entire program has been processed, the tool would print out a two-dimensional array, with global variables on the rows and functions on the columns. Each cell of the array contains the number of sets and uses of a variable by a particular function. An enhancement of this task might include a hypertext view of the array—built from a standard graphics library—that allows perusing the program source by clicking on various cells, components, or variables.

Unfortunately, determining aliasing (i.e., with pointers) of local variables to global structures is a potentially complicated calculation, as is inferring the effects of `execute` commands and concurrency. The designer may have to use domain and design information to customize the search to inexpensively resolve ambiguities. In fact, the designer might have to perform several related queries to derive all the necessary information, examining the last query and formulating a new one to augment the prior results. At the extreme, the designer can use the transformation component of the tool to modify the existing system so that it records dynamic control flow and the contents of pointers to verify hypotheses about the system. In particular, extra statements can be added to perform dynamic tracing of what addresses are being dereferenced and what statements are being executed. Running the modified system on the existing test suites can then provide intuition about how the system is put together.

2.2 Programmer's scenario

The programmer's scenario is more bottom-up. When a programmer is making changes, there are at least three concerns. First, the written code must perform the right calculation. Second, any non-local effects of this calculation must be assessed. For example, are any global variables being affected by the change, and is this intended? Third, any effects on this calculation by other calculations must be assessed. For example, are the actual inputs to the calculation the ones that the programmer desires? These last two concerns require non-local reasoning about the program's behavior; they can be thought of as backward- and forward-slices [23][18] of the new code, respectively. Using a syntax-directed tool, the programmer would specify a backward (or forward) search from the proposed new component, looking for references to variables that the new component reads (or writes). For most languages, the structure of the program provides fairly accurate control flow information, and the symbol table provides precise scope information, ensuring a reasonable dependence analysis. But, as with the designer's scenario, alias information has to be handled carefully. Note that on large projects such as CHCS, there are dozens of programmers who need to perform such calculations.

In either of these scenarios, the next step might be to use the tool to perform non-local program transformations to make structural changes to help make enhancements.

Analyzing these scenarios reveals that the frequency of use or the number of iterations for each use demand that a program understanding and transformation tool have good performance. Otherwise, the tool will not be used, likely leading to a more ad hoc software maintenance process. Furthermore, a program understanding and transformation tool must be able to perform quick "what-if" queries of the program and have an extendable scripting language that allows the tool user to easily write and extend such queries and transformations. Because traditional programmers are expected to use these tools, the queries must be expressible in a familiar paradigm, such as an imperative programming language. Existing tools meet some, but not all, of these requirements.

2.3 Existing tool designs

TXL is a program transformation tool based on a tree rewriting computational paradigm [8]. Basically, the TXL rewriting engine applies a rewrite rule to the parse tree of a program until there are no more constructs in the parse tree that are applicable to the rule. This paradigm, however, complicates searches, one-pass transformations, or other tasks outside the rewrite paradigm. Although there are hooks to the underlying language for more general tasks, the awkward interface discourages frequent use. TXL's matching language uses concrete syntax for specifying pattern matching. Although appealing, this approach complicates access to a part of a program that is in a deeply nested construct, because all of the surrounding text to the desired part must be specified. It can be difficult to create an abstract interface to the program source to overcome this problem. Finally, TXL's custom-built parser-generator accepts ambiguous grammars, resulting in slow processing of the program source.

REFINE is a more general matching and transformation tool based on a strongly-typed first-order set-theoretic logic [7]. Although this paradigm is elegant and powerful, it requires users to learn a new computational paradigm, and complicates performing higher-order operations. According to estimates provided by REFINE's implementors, entering CHCS into REFINE would require about 8 hours.[2] Although a subsystem has to be reentered only when it is changed, some subsystems in CHCS would require 9 minutes to reenter. In order to ensure accurate global analyses, several files might have to be reentered several times a day as they evolve. An AST may is about 20-40 times larger than the program text, depending on what attributes are specified for the AST nodes.

To provide its many flexible features, REFINE is implemented in the REFINE language, which itself is implemented in Common Lisp [22]. Although the resulting tool is compiled, Common Lisp is still slower and uses more space for data compared to lower-level compiled languages such as C. Like TXL, REFINE's scripting language uses the concrete syntax of the language for specifying pattern matching, Although REFINE allows using abstract syntax, some of the special features associated with concrete syntax matching are then not available. Like TXL, REFINE has its own parser-generator based on Extended Backus-Naur Form (EBNF), although it handles a more standard subset of the context-free languages.

REFINE allows storing of the ASTs and other computed information as objects in persistent storage for later retrieval. This can be useful for managing very large structures when the swap space of the machine is inadequate to store them. It also allows saving data that is costly to recompute (such as data flow information) between uses of the tool. Retrievals from the database are performed on a per-file basis, and retrieving files is estimated to be 5 times faster than entering files.[3] Space may be conserved by a specification of what subset of AST attributes to preserve. Tuning the storage size, however, can be a complicated task.

Many of REFINE's problems are accidental in nature, and a few essential [6]. For example, it implements the core of its paradigm, sets, with linked-lists, although with some additional effort they could be implemented more efficiently as hash tables. On the other hand, the use of the very high-level REFINE language for the REFINE tool and its instantiation to operate on a particular language poses inherent risks because of the complexity of compiling a high-level language to an efficient form. However, REFINE's implementors are now working on retargeting the REFINE language to generate code for a low-level language rather than Common Lisp. They are also beginning to use techniques for improving the selection of representations for high-level data types [4].

2.4 Design choices for a new tool

Achieving both flexibility and good performance, regardless of the chosen software architecture, is difficult. Flexibility typically demands the use of elaborate constructs that can be inefficient. Our primary challenge, then, was determining exactly where flexibility or performance could be sacrificed in favor of the other, in order to achieve both in typical uses of the tool. This goal can be achieved by understanding what the scenarios above require, and by using exist-

[2] Lawrence Markosian, Personal Communication. This estimate is based on REFINE applications for FORTRAN and COBOL.

[3] Philip Newcomb, Personal Communication.

ing compiler technology. The theory and practice of compilers have yielded tools, insights, and heuristics that can significantly simplify the design and implementation of a program understanding and transformation tool, because such a tool is quite similar to a compiler. The following describes the impact of using a compilers paradigm for the design of our tool.

Parsing: We found that lexing and parsing do not need to be especially flexible, but they have to be efficient since processing every character in a program is I/O and compute intensive. Nearly all programming languages in use can be lexed and parsed by existing, fast parsing tools such as lex and yacc, and any language constructs that are difficult to process can be handled by incorporating extra semantic actions triggered by these tools. The declarative input to these tools and programmers' familiarity with them make them easier to use than other parsing techniques. Also, many lexer and parser descriptions are publicly available for commonly used languages such as C and Ada.

ASTs: Similarly, we found that flexibility is not a key concern in implementing ASTs. ASTs have a regular tree structure that is analogous to the language grammar. This relationship guides the design of the AST, and the actions of the grammar's parsing rules can straightforwardly construct an AST for a program. To make the programming of AST construction actions even easier, we have implemented a preprocessor, that extends the syntax of yacc using additional parsing directives. The new directives allow for creating new subtrees directly, or indirectly by invoking a user-defined action. The preprocessor can also generate a header file of function prototypes and templates for the user-defined actions and for routines that can print the AST.

To avoid the need for special techniques for storing ASTs off-line, the space consumed by an AST needs to be minimized. An AST representation may be 10 times larger than a textual representation, even when care is exercised in its design. (Significant space can be saved by optimizing the representation of the leaves of the AST, but this can complicate accesses and updates to it.) Consequently, we were led to design ASTs in C, which ensures compact representation of structures. Also, using C for constructing ASTs integrates well with lex and yacc. (We had considered using an object-oriented language, such as C++, but rejected it because of persisting problems with the technology, such as space and execution overhead. These problems can be overcome with sufficient effort, but this negates the benefits of object-orientation.) To minimize the storage required for an AST node, non-intrinsic information is typically not stored in the node. Instead, the scripting language provides a relation data type for mapping from AST nodes to non-intrinsic information.

To further decrease memory requirements and improve paging behavior, for one-pass scripts we incorporated the ability to process ASTs as they are constructed, rather than waiting for a program's entire AST to be constructed. This interleaving allows discarding an AST fragment as soon as it is processed. Consequently, memory requirements are greatly reduced, and the accompanying paging time, which can be substantial, is eliminated. In some cases the actual AST computation time is also reduced. Currently, this feature requires the script programmer to specify the granularity of processing. In practice this has not been an onerous responsibility.

Matching and transformation language: The main challenge in providing both flexibility and performance is designing the language constructs provided to the tool user for manipulating the AST. The tool user does not want to be distracted by memory management, search strategies, typing, or complex syntax. Typically, eliminating such distractions via a high-level computational paradigm comes at the expense of performance or generality of the language. Such a paradigm also may be difficult for a traditional programmer to use, indicating that an imperative programming paradigm might be appropriate. In fact, we have found that many scripting tasks, such as coding standards checking (i.e., syntactic and static semantic checking) and flow analysis, are well-known compiler problems and have been shown to be relatively easy to code efficiently in a traditional imperative language. Also, a high-level paradigm compromises the ability to use existing C code for graphics and other complex tasks. Although calling from a high-level to a low-level language is not always difficult, storing high-level types in the low-level language presents problems, as does performing callbacks to the high-level language. Callbacks are common technique in systems like the X window system [17].

We believe that the advantages of scripting languages are usually limited to small programs, since they emphasize simplicity, making large programs difficult to write. For large programs, typing, modular structure, and good performance become more important, but supporting such needs complicates writing smaller programs. Thus, simple programs are more easily written in the scripting language, but more complex programs are more easily written in C or C++ and called from a script. This approach also allows using the growing number of existing C code libraries.

Our approach, then, which is still under development, is to create a simple imperative scripting language that can be translated directly to C. Only a few high-level facilities are provided, such as a simplified syntax, goal-directed searching, tables, sets, bit-sets (e.g., for flow analysis), AST printing, and memory management. By keeping the number of high-level features small, a simple translator can produce efficient code. Types of variables need not be declared by the programmer, but the translator infers type declarations for a variable from the type of the initializing value of the first of its definitions. The types of built-in functions and linked-in C functions, which are known, also contribute to the inference process. Memory management is provided by a conservative collector [5], although the AST is treated specially. To provide an interactive environment for using our tool, the shared library features of many UNIX platforms can be used to implement dynamic linking and execution of scripts written on the fly.

2.5 Tool architecture

Because our tool consists of many components of various levels of functionality, we chose a layered design for our tool. Layering is useful, since successively higher layers can incrementally provide more expressive functionality. Since a higher layer cannot be more efficient than its underlying layers, the lower layers must be especially efficient. The challenge is to provide successively more expressive layers without compromising performance.

The following are the layers of an instance of our tool for a specific language, listed from the lowest to the highest layer. Note that the lower layers are not always strictly layers in that each also has the full power of the implementation language available to it. They are layers in the sense of the new service provided by that layer.

1. **AST construction:** The lowest layer of the tool is the AST construction layer. It consists of definitions for the basic AST node types, constructor functions, and functions to link subtrees. Symbol table construction is also performed in this layer.

2. **Lexer and parser:** The lexer and parser read in the source files and call the AST construction routines to assemble an AST and fill the symbol table. Note that the layers above determine how the parser is invoked. For example, the parser may be invoked to assemble an AST one file, routine or statement at a time. The lexer and parser are different for each programming language, and are generated from declarative lexical and syntactic descriptions using lex and yacc (See Section 2.6).

3. **Abstract syntax interface:** The abstract interface to the AST consists of basic access, update, and searching functions for the AST. It also allows a request to build a new AST, add one to the existing tree, or controlling the granularity of the AST on one pass-algorithms (Section 2.4).

4. **Matching and transformation language:** This layer provides high-level linguistic mechanisms for examining and transforming the AST. These mechanisms include the abstract interface to the AST, memory management, dynamic typing, and goal-directed pattern matching, in addition to more standard language constructs. Straightforward translation to C supports accessing existing code such as graphics libraries.

5. ***User script:*** *The highest* layer of the tool is the tool user's script, written in the matching and transformation language. A script implements a specific program understanding or transformation task such as determining set-use information or performing dead-code elimination. A sophisticated user can write his or her own script, but a novice user may perform only simple actions or borrow scripts from others.

Note that the AST *module* spans two non-adjacent layers (1 and 3). This division permits controlling the granularity of AST construction without introducing circularities between the layers or forcing the AST into two different modules [11][20].

2.6 Generic infrastructure

In order to amortize the costs of building an instance of this tool for a specific language, it is advantageous to develop a generic tool infrastructure—like that provided by TXL and REFINE—that can be easily adapted to a specific language. Thus there are really two tools, the generic framework for creating a tool for a specific language, and the tool for a specific language.

The generic tool includes the scripting language, the UNIX tools lex and yacc, the yacc-preprocessor that provides concise parsing directives for AST construction, and the symbol table and AST datatypes. The tool retargeter is responsible for providing the standard lexical and syntactic descriptions to lex and yacc, along with the appropriate AST node and symbol table entry definitions, AST linking directives and symbol table actions for building an AST for a program in the target language. Finally, the retargeter must design an appropriate abstract interface to the AST, which becomes the set of built-in procedures of the matching and transformation language. This interface needs to be carefully chosen to faithfully convey the language syntax and to allow easy access to nested constructs.

3 Discussion

3.1 Design history

Our early efforts in designing this tool focused on discovering the tradeoffs between performance and flexibility. Our first attempt at implementing a parser and ASTs used Icon [9]. We chose Icon because of its support for goal-directed pattern matching, flexible aggregate data structures, dynamic typing, garbage collection, rapid edit-compile-execute cycle, and optimizing compiler. Additionally, we believed it could be more efficient and expressive than Common Lisp for the tasks we wanted to perform. However, we found that yacc-like bottom-up parsing in Icon was too slow and the resulting code was difficult to compile. Likewise, although the ASTs were reasonably compact, they were still twice as large as ASTs implemented in C. Overall, Icon was about 5–10 times slower than what we finally achieved in C.

After this failed attempt, we realized that performance outweighed most of our concerns about flexibility. Much of the flexibility we desired is available by using lex and yacc, whose declarative syntax eases making small changes during development. Likewise, we found that the analogous structures of the AST allowed implementing ASTs in a relatively declarative fashion.

Once we implemented the basic tool, we sought additional techniques to speed processing. For example, some queries are simple enough that they can be performed during parsing without an AST. Hence, we wanted to provide a mechanism for specifying that a query should occur as part of a parsing action. However, we found the approach impractical. The rules that would have to be added to the grammar to accommodate the on-the-fly processing either complicated the grammar unacceptably or introduced parsing conflicts that could not be resolved satisfactorily by yacc's LALR(1) parsing strategy. Consequently, we settled on the the approach of interleaving processing with AST construction (Section 2.4).

The problem of inserting processing actions into the grammar is an example of a broader class of issues created by using standard, highly optimized tools such as lex and yacc. For one, these tools were not created with tasks such as AST printing and AST interface design in mind. For example, both lexical and syntactic information is needed to create an AST printer. However, because lex and yacc are separate tools, and some of their processing is handled by C code rather than declarations, insufficient information is available to successfully automate the generation of an AST printer. Also, their underlying algorithms and input syntax were designed primarily for performance at the expense of other needs. For instance, a yacc grammar is best written in a left-recursive fashion to minimize stack depth. Unfortunately, such an encoding of a grammar does not necessarily yield a grammar that is suitable for automatic derivation of abstract interfaces.

Our experience so far indicates that the time required to program components such as AST printing and the AST interface is acceptable, especially since they are programmed just once for each retarget of the tool to a new language. On the other hand, once the tool is retargeted, it will be used repeatedly in contexts where fast performance is paramount.

3.2 Performance results

On a Sun Sparc 10 model 41, with 96MB of RAM and 500MB of swap space, the resulting prototype parses MUMPS code at 250,000 lines per minute. It can parse code and produce an AST at 67,000 lines per minute. If the AST is discarded as it is traversed (See Section 2.4), our tool can proceed at 110,000 lines per minute due to reduced paging overhead. An AST node requires 24 bytes (for application data, pointers to parent, children, and siblings). Our tool can construct an AST for CHCS—consisting of 600,000 lines of code—in about 9 minutes using 250MB of storage. REFINE, with its current technology, parses code and builds ASTs at about 1200 lines per minute. With estimates based on a Sparc 10 with 500MB of RAM, it could construct a full AST for CHCS in about 8 hours using about 600MB of storage.

A simple set-use computation on CHCS, which can be performed at the granularity of a MUMPS command, takes 10 minutes (about 6 minutes for parsing and 4 minutes for the set-use computation). At this granularity, the AST never exceeds 4KB of storage. In a scenario in which there are many programmers on a project, this permits several programmers on the same machine to run one-pass scripts simultaneously. Although the time to parse and reconstruct the ASTs dominates for simple operations, it is small enough to permit reconstructing the AST for each script. When several script runs are anticipated, it is possible that building the whole AST and then running scripts interactively inside our tool might provide better performance. For a scenario with several programmers, a client-server approach to allow sharing the AST might be best. However, it is difficult to assess how effective these techniques will be in the typical case because of the tradeoff between reconstructing the AST and swapping due to the increased memory requirements. The amount of RAM available and the anticipated size of the data structures are key.

The performance of our tool is adequate for performing "what-if" queries and repeatedly refining queries to converge on a desired solution. For instance, the set-use computation was refined from a naive script into a usable form over several iterations. If the script were tested on the whole system (which is necessary in some cases to reveal certain omissions and performance problems), the script could be tested 6 times per hour (not including coding time).

4 Conclusion

Automating semantically-precise techniques for manipulating programs can help maintain large systems, but they can be too costly to implement or use. An alternative is to use a simpler, faster, syntax-directed tool that captures much of the needed information. However, missing information must be provided by the tool user, perhaps by repeatedly running the tool with a variety of related queries. Designing such a tool with enough performance and flexibility to offset its semantic shortcomings is difficult. However, by taking advantage of the theory, experience, and tools of traditional compiler and language design, this goal can be met.

We have implemented the lower layers of our tool but are still designing the programming layer. Once we have completed our tool, then we can verify our hypothesis about how programmers and designers can use domain and design information to permit using syntactic analysis for cheaply acquiring and using semantic information. Use of the tool on CHCS will be used in the evaluation. However, our early results suggest that using traditional compiler techniques in an established technology base can provide substantial performance benefits and ease of use.

Acknowledgments

We are grateful to Philip Newcomb of Boeing for providing detailed information on his experience with REFINE, and to Lawrence Markosian of Reasoning Systems regarding key details of REFINE's implementation. Thanks to James Cordy for detailed information on TXL. Thanks to Jennifer Schopf for her comments on a draft of this paper.

References

[1] A. V. Aho, B. W. Kernighan, and P. J. Weinberger. Awk - a pattern scanning and processing language. *Software—Practice and Experience*, 9(4):267–280, April 1979.

[2] L. A. Belady and M. M. Lehman. Programming system dynamics or the metadynamics of systems in maintenance and growth. Research Report RC3546, IBM, 1971. Page citations from reprint in M. M. Lehman, L. A. Belady, editors, *Program Evolution: Processes of Software Change*, Ch. 5, APIC Studies in Data Processing No. 27. Academic Press, London, 1985.

[3] L. A. Belady and M. M. Lehman. A model of large program development. *IBM Systems Journal*, 15(3):225–252, 1976. Reprinted in M. M. Lehman, L. A. Belady, editors, *Program Evolution: Processes of Software Change*, Ch. 8, APIC Studies in Data Processing No. 27. Academic Press, London, 1985.

[4] L. Blaine and A. Goldberg. *DTRE—A Semi-automatic Transformation System*, pages 165–204. North-Holland, Amsterdam, The Netherlands, 1991.

[5] H. J. Boehm and M. Weiser. Garbage collection in an uncooperative environment. *Software—Practice and Experience*, 18(9):807–820, September 1988.

[6] Frederick P. Brooks. No silver bullet: Accidents and essence of software engineering. *IEEE Computer*, pages 10–19, April 1987.

[7] S. Burson, G. B. Kotik, and L. Z. Markosian. A program transformation approach to automating software re-engineering. In *Proceedings of the Fourteenth Annual International Computer Software and Applications Conference*, pages 312–322, 1990.

[8] J. R. Cordy, C. D. Halpern-Hamu, and E. Promislow. Txl: A rapid prototyping system for programming language dialects. *Computer Languages*, 16(1):97–107, 1991.

[9] Ralph E. Griswold and Madge T. Griswold. *The Icon Programming Language*. Prentice-Hall, second edition, 1990.

[10] W.G. Griswold and D. Notkin. Automated assistance for program restructuring. *ACM Transactions on Software Engineering and Methodology*, 2(3), July 1993.

[11] A. N. Habermann, L. Flon, and L. Cooprider. Modularization and hierarchy in a family of operating systems. *Communications of the ACM*, 19(5):266–272, May 1976.

[12] S. C. Johnson. Yacc—yet another compiler compiler. Computing Science Technical Report 32, AT&T Bell Laboratories, 1975.

[13] S. C. Johnson. A portable compiler: Theory and practice. In *Proceedings of the 5th Symposium on Principles of Programming Languages*, pages 97–104, January 1978.

[14] M. E. Lesk. Lex—a lexical analyzer generator. Computing Science Technical Report 39, AT&T Bell Laboratories, 1975.

[15] J. M. Lewkowicz. *The Complete MUMPS: An Introduction and Reference Manual for the MUMPS Programming Language*. Prentice Hall, Englewood Cliffs, N.J., 1989.

[16] B. Lientz and E. Swanson. *Software Maintenance Management: A Study of the Maintenance of Computer Application Software in 487 Data Processing Organizations*. Addison-Wesley, Reading, MA, 1980.

[17] J. McCormack, P. Asente, and R. R. Swick. X toolkit intrinsics—c language x interface. Technical report, Massachusetts Institute of Technology, 1988.

[18] M. Moriconi and T. C. Winkler. Approximate reasoning about the semantic effects of program changes. *IEEE Transactions on Software Engineering*, 16(9):980–992, September 1990.

[19] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–58, December 1972.

[20] D. L. Parnas. Designing software for ease of extension and contraction. *IEEE Transactions on Software Engineering*, SE-5(2):128–138, March 1979.

[21] R. W. Schwanke and M. A. Platoff. Cross references are features. In *2nd International Workshop on Software Configuration Management*, pages 86–95, Princeton NJ, 1989.

[22] Guy L. Steele. *COMMON LISP, the Language*. Digital Press, Burlington, MA, 2nd edition, 1991.

[23] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, July 1984.

SOFTWARE REENGINEERING IN THE SF FRAMEWORK

Alfs T. Berztiss

Department of Computer Science, University of Pittsburgh, Pittsburgh, PA 15260, USA
(e-mail: alpha@cs.pitt.edu; fax: +412-624-8854)
and
SYSLAB, University of Stockholm, Sweden

ABSTRACT: The SF (Set-Function) specification and prototyping language has numerous properties that make it an attractive intermediate language for the reengineering of embedded control systems. An existing control system is mapped to SF, changes are made to the SF prototype, and the new prototype is then transformed into an implementation in the language of the user's choice. We show how a fairly major redesign task to an elevator controller was very rapidly introduced into the SF prototype.

1. Introduction

In general, software controls a process, provides access to information, or changes the form of data, and we speak of a control system, an information system, or a data transformer. Examples of the three kinds: an elevator controller, an inventory information system, and a program that transforms cartesian coordinates into polar coordinates. Most systems are composite software-hardware hybrids, which may be supervised by a human operator. For example, a milling machine transforms a piece of metal, but is controlled by software selected by an operator, the software in turn refers to tables in an information base, and this information has to be put into an appropriate form by a data transformer. For simplicity, here we shall restrict ourselves to systems that consist of a software controller embedded in a hardware device to be controlled. We shall refer to the two components of the system as controller and host.

Even when discussion is limited to controller-host systems, numerous types of change that necessitate reengineering have to be considered. We list three broad categories of such changes.

- Shift of the controller-host boundary. The boundary is shifted to alter the balance between system reliability and system performance. If more functions of the combined system are taken over by the host, functional reliability can be improved because duplication of hardware components allows hardware reliability to be increased to an arbitrarily high level. On the other hand, if performance needs to be improved, the boundary has to be shifted in the other direction.

- Controller enhancement. An existing system is enhanced when more of the tasks that were originally performed by human operators or were not performed at all are taken over by the controller.

- Host modification. An existing host is modified, necessitating a change in the controller. This is a common problem facing developers of controllers under concurrent engineering, but often modifications continue to take place after release of the system. Host replacement by a new model is an extreme case of modification.

The reengineering process can be applied directly to existing code, e.g., a controller written in a mix of Fortran and assembler language may be adapted to new conditions using these same languages, but we cannot recommend such an approach. There are several reasons why it is preferable to derive a specification from the existing code, make the required changes in this specification, and then translate the specification into executable code. First, the existing software has to be thoroughly understood before a change is made. Program understanding has been estimated to account for 40% of maintenance costs [1]. Although there exist tools to aid in program understanding [2], and these tools could in principle be used to help a software reengineering team to make changes in the existing software directly, it has to 'be established that the altered code corresponds to sponsor expectations. This is easier done in specifications than in code. Second, when the change also entails a language switch, e.g., from Jovial to Ada, the need for thorough validation by the sponsor assumes even greater importance. Third, once a specification has been derived, future modifications will be much easier to carry out and validate in the specification than in code. Hence the writing of specifications can be regarded as preventive maintenance. This presumes, though, that the specification language is appropriate for the task.

2. Properties of specification languages

In what follows we shall be concerned with *formal* specifications of software systems, i.e., specifications written in a language with a well-defined syntax and semantics. A formal specification is a bridge between system sponsors and the people who develop, maintain, and reengineer the system. A formal specification of a controller must be an unambiguous statement of the expectations of the sponsors, but expressed in such a way that they can make sure that their expectations are in fact being met. It follows that the mode of presentation of a formal specification has to satisfy four criteria: requirements must be expressed unambiguously, they must be verifiable, they must be understandable by sponsor representatives with little effort, and their form must allow a sponsor representative to indicate clearly any changes that are to be made to the specifications. Simply put, specifications must be at the same time precise and readable, and this is difficult to achieve. We shall nevertheless attempt to achieve both goals with SF, to be described further down. This language will be found to satisfy the following ten properties of specification languages. The properties are consistent with the principles of specification and prototyping languages listed in [3-6].

1. The language is to be totally unambiguous so that system developers and maintainers have a precise understanding of the system. This means that the language must have sound theoretical foundations.

2. The language is to allow a reading knowledge to be acquired with little training. There is to be an economy of concepts.

3. The language is to allow the specification of a conceptual model of a controller without having to define a detailed design or implementation model. However, it has been found that some intertwining of specification, design, and implementation of software is inevitable [7-9].

4. The language is to encourage development of a specification in the form of modules. Of the ten most serious sources of risk in the software development process, seven can be managed by incremental modular software development [10]. Modularization based on data types is a natural choice.

5. The language is to require that the specification of a module consists of distinct parts that deal with the structural aspects of its information base, its behavioral aspects, and the actual control process. This contributes to the clarity of a specification and allows the specification of a module to be developed by several members of a reengineering team working in parallel.

6. The language is to enforce the principles of data abstraction – a data type is to consist of a set and of functions defined on this set.

7. The language is to allow easy representation of integrity constraints on the information base of a module.

8. The language is to provide mechanisms for linking individual processing tasks into control processes, and is to allow easy definition of interactions of a controller with its host, including real-time interactions.

9. The language is to be supportive of an evolutionary development of specifications, i.e., it should allow easy modification and augmentation of an initially incomplete specification. Under reengineering this means that the specification can be derived from existing controller software in incremental steps.

10. The language is to support reuse. For example, it should not require much effort to convert the specification of an existing controller into a controller for a similar application.

3. An introduction to SF

The specification and prototyping language SF (short for Set-Function) has been developed over the past eight years. The language was introduction in 1986 [5]; a later version (somewhat simplified since then) is described in [11]; examples of SF specifications can be found in [12]. A specification in the SF (Set-Function) language is composed of segments, which correspond to data types. A segment has three parts: a schema that describes an information base, events that change the information base, and a control component, which consists of transactions that allow events to be joined together into a process. The schema definition identifies the set of interest for the segment and defines functions that map from this set. A control process is built up by means of *signals* that are either "on" or "off", and that link events and transactions within and across segments. Events switch signals on; transactions switch them off. A transaction may be implemented in software or hardware, thus providing a high degree of flexibility. A signal may be a carrier of additional information that is to be sent from one segment to another, or merely from one event to another. Hence switching on a signal is in effect the sending of a message.

We shall use a library system to provide us with an example of a schema. We do so because schemas for realistic controller-host systems are much too large for an introductory exposition (a schema for an elevator controller can be found in [12]). Moreover, we shall at times deviate from the rules of SF to avoid getting distracted by technical detail of little relevance in such an exposition. There can be several copies of the same title in the library, so there is a need for a copies and a titles segment. There is also a borrowers segment. Only the schema for copies will be considered here.

```
SEGMENT Copies;
  IMPORTEDSIGNAL TitleFixed(Copy, Title);
  SIGNALS
       CatalogCopy(Copy,'Title);
       AddTitle(Copy, Title);
       AdjustCount(Borrower, Integer);
  TYPE Copy;
    SET C (SUBSETS: AV, CO, BR, L, R);
    FUNCTIONS
       BookId:   C ⟶  Title;
       History:  C ⟶  (Borrower × Date)-set ;
  ENDTYPE ;
```

The set for data type *Copy* is *C*, and it is partitioned into subsets *AV*, *CO*, *BR*, *L*, and *R*. A copy is in the corresponding subset if it is available for borrowing, checked out, being repaired, lost, or removed, respectively. For each copy, function *BookId* tells to what title this copy belongs, and *History* gives its complete borrowing history. Three signals originate in this segment, and one is imported into it (from segment *Titles*). Signal *AddTitle* is raised when the first copy of a title arrives in the library, and there no information about this title in the catalog. The *Titles* segment picks up the signal, updates the catalog, and sends back signal *TitleFixed*, which initiates the introduction of the copy information into the catalog. Signal *CatalogCopy* has the same effect as *TitleFixed*, but is raised if this is not a first copy, i.e., it is raised in the *Copies* segment itself if the title information is already in the catalog. The remaining signal causes adjustment of the count of copies held by a borrower.

The specification of events consists of preconditions and postconditions. Preconditions determine under what circumstances an event may take place. They serve as a check on the feasibility of input values, and embody integrity criteria for the information base of the segment. Postconditions are divided into dataconditions and sigconditions. Dataconditions indicate the changes that sets and maps undergo in consequence of an event taking place. Sigconditions send signals to transactions. An event is regarded as atomic, i.e., the objects with which the event deals are not accessible to other events until the event is done with them. We give two events of an elevator controller as examples. Event *MoveIdle* is initiated by a dispatcher segment. Its purpose is to move an idle elevator *e* to a holding floor *f* selected by the dispatcher. The elevator has an agenda of floors to visit. For an idle elevator the agenda has been empty, and it now becomes the set { *f* }. The elevator goes from the idle state into a halt state, and signal *S¯SetInMotion* will cause it to be set in motion toward floor *f*. Note that there are two halt states: from one the next state transition will be into an upward motion state; from the other it will be into a downward motion state.

When a set or a function is changed as a result of an event, the prime symbol denotes the changed object. Equation $S' = S \; op \; X$ tells that S', the set after the event, is *S*, the set before the event, modified by *X* according to the operation *op*. With equation $fun'(x) = t$ there can be two cases. If function *fun* already contains some pair $<x, s>$, then $<x, t>$ replaces this pair; if not, then pair $<x, t>$ is simply added to the function.

```
EVENT MoveIdle(e: Elevator, f: Floor);
  DATACONDITIONS
    Agenda'(e) = { f } ;
    f > FloorNow(e)  ⟶  State'(e) = "uphalt";
    f < FloorNow(e)  ⟶  State'(e) = "dhalt";
  SIGCONDITIONS
    (S¯SetInMotion(e))ON;
ENDEVENT;

EVENT OpenDoor(e: Elevator);
  PRECONDITIONS
    State(e) ∈ {"idle", "uphalt", "dhalt"};
  DATACONDITIONS
    Clock'(e) = Time.Now;
  SIGCONDITIONS
    (S¯ProcessHalt(e))ON;
    (DoorOpen(e))ON;
ENDEVENT;
```

Event *OpenDoor* is essential so that people can get out who somehow find themselves in an idle elevator; raising the flag *S¯ProcessHalt* ensures that the opened elevator door will ultimately close again. An event takes place only if all its preconditions are satisfied. Here we have just one precondition: the elevator is to be idle or in a halt state, i.e., pressing the "Open Door" button has no effect if the elevator is in motion. The purpose of *Clock* is to measure the time for which the elevator door is to remain open; *Time. Now* is the identifier of a function *Now*, which belongs to type *Time* and returns the current clock reading.

Let us now look at what happens to signals. A signal is raised by an event, and is then picked up by a transaction in the same or another segment. A signal remains alive until it is explicitly turned off. A signal may be raised in just one segment, which we call its home segment, but it can be turned off in more than one segment, by any applicable transaction that picks up the signal first. The need for signals to have a home segment was the reason for having the two signals *CatalogCopy* and *TitleFixed* in the library example. Suppose segment A wants segment B to perform some action. It makes the request by raising a signal. This signal is *exported* by segment A and *imported* by segment B, where it triggers a transaction. Our example of a transaction is triggered by a signal that originates in the dispatcher segment, and it initiates an event in the elevator segment.

```
TRANSACTION;
  @( ): ON(S¯MoveIdle(e, floor))OFF:
    MoveIdle(e, floor);
ENDTRANSACTION;
```

Consider the components of this transaction. All transactions have times associated with them. Thus @(17:15) indicates that every day at 5:15 pm a check is made whether the transaction so marked is enabled. The time marker @( ) requires the check to be made continuously. Transactions are enabled if the signals in their definitions are on, or if no signal is shown. The signals are switched off, and actions are performed. In our example the action is the automatic initiation of an event. Actually, @(*a*) is an abbreviation for @(*a*, *a*), and @(*a*, *b*) indicates that the transaction must take place within the time interval (*a*, *b*); @( ) is an abbreviation for @(*Time. Now*, *Time. Now*).

Sometimes a situation arises in which a transaction realizes that an event is to take place, but cannot initiate the event because the necessary arguments have to be supplied by a human operator. It then *prompts* the operator to initiate the event. A third alternative is a *reminder* – a transaction can remind a human operator of anything at all.

The separation of the action of sending out a message (i.e., switching on a signal as part of an event) from the definition of the ultimate effect of this message (in a transaction, which may reside in a differen' segment) has several attractive features. First, events can be defined independently of transactions, and all decisions regarding the precise effect of a message can be postponed. Second, because messages are not addressed, more than one transaction can pick up a message, which adds to the flexibility of the entire system. For example, the availability of a resource can be made known to several processes that may wish to use this resource, without naming the processes. Third, all indications of the time dependence of the effect of a message belong to the definition of transactions, i.e., the specification of time-related aspects of the system is confined to transactions. Fourth, all iterative actions are confined to transactions, which means that events are given a particularly simple structure.

Also in the interests of simplicity we have eliminated a number of features that were to be found in earlier versions of SF, in particular sensors and mechanisms. A sensor is a device that supplies information, e.g., a thermocouple supplies a temperature reading. As regards an elevator, an indication of the floor at which it finds itself could be provided by a hardware sensor or by a software function that is adjusted each time the elevator passes a floor. Therefore, at a sufficiently high level of abstraction, there is no essential difference between sensors and functions. A mechanism is a device that interfaces with the external world, e.g., causes an elevator door to be opened. Again, there is no essential difference between a signal that causes a software change and a mechanism that causes hardware change. We therefore eliminated the latter, and *DoorOpen* became a signal.

Prompting transactions point out tasks that are candidates for automation. A prompting transaction knows that an event is to take place, but cannot supply all its arguments. If the supplying of the arguments is taken over by an expert system, the transaction can initiate the event on its own. If all the activities of an organization are specified in SF, then prompting transactions indicate all opportunities for the introduction of expert systems, in each instance a cost-benefit analysis can be made, the opportunities ranked according to the results of the analysis, and expert systems developed incrementally in order of the ranking. If full automation is not called for, a decision support system can be made to assist the human decision maker arrive at the appropriate inputs to the event to be initiated. When full or partial automation is introduced by means of an expert system or a deci: ɔn support system, the new process will probably be more complex than the approach it replaces. Reengineering does not always mean simplification – usually it means increased effectiveness, but this may require a more complex process structure.

The only way to remove ambiguity from specifications is to use a language based in mathematics. Necessarily, this is also the case with SF, but we have tried to make the mathematics not too forbidding. Thus, although all preconditions and dataconditions are assertions in logic, the notation is not obtrusive. The assertions use operations on sets, finite functions, and numerical and boolean data types. All such operations have well understood mathematical definitions. Signals relate events and transactions: only an event can switch on a signal; only a transaction can switch it off. This follows the Petri net formalism in which places are linked with transitions, but no two places or two transitions may be directly linked. The nets are actually variants of *time Petri nets* [13, 14] because of the use of time markers of the form @(*a*, *b*). Manipulation of signals corresponds to the movement of tokens. The theory of Petri nets thus provides mathematical foundations for SF processes.

4. Categories of change and SF

In Section 1 we introduced three categories of change: shift of controller-host boundary, controller enhancement, and host modification. We shall now relate the features of SF to these categories. Under the first category the total system, as seen by an outside observer, has unchanged functional capabilities, but the capabilities of its software component have either increased or decreased.

Let us consider an example. The determination of the current position of an elevator can be established by means of a hardware floor indicator or by counting floors as the elevator moves up and down. The SF schema contains the function *FloorNow*, mapping from elevators to integers. If the values of this function are supplied by hardware floor indicators (sensors), then the making of changes to this function is no concern of the software reengineering team. But suppose now that in the future the values of *FloorNow* are to be adjusted by software. We stipulate that there exist sensors between floors in every elevator shaft, and that a signal *NextFloorSensor*(*e*) is raised whenever elevator *e* passes one of the sensors. In the specification of the elevator controller this signal initiates the event *PassingSensor*. Under the present design the agenda of the elevator is consulted, and, if the next floor in the direction of motion of the elevator is on its agenda, the elevator will be halted at the floor. Under the new design these actions still need to be performed, but now, in addition, the value of *FloorNow* is changed: if elevator *e* is moving up, the value of this function for *e* is incremented; if it is moving down the value is decremented. These changes are defined by the assertions

$State(e) = \text{"up"} \longrightarrow$
$\quad FloorNow'(e) = FloorNow(e) + 1$;
$State(e) = \text{"down"} \longrightarrow$
$\quad FloorNow'(e) = FloorNow(e) - 1$;

As a design rule, we recommend that initially as much as possible of the system be specified in the form of software capabilities. Then the event *PassingSensor* would contain the assertions that define the changes in value of *FloorNow* from the very beginning, and, in implementing the design under which the value of *FloorNow* is supplied by a hardware indicator, these assertions would be changed into comments. This has two advantages. First, these assertions-comments express the semantics of *FloorNow*. Second, if a change from a hardware to a software implementation is to be made, as we were considering it above, this is accomplished by changing the comments back into assertions. It seems that whenever a function can be implemented by hardware or software, ultimately there is a need for sensors. Thus, the software implementation of *FloorNow* still depends on next-floor-sensors. Even if the value of *FloorNow* were to be worked out by reference to the time an elevator is in actual motion, this would require consulting a clock, which is again a kind of sensor.

Controller enhancements do not affect the hardware-software interface, i.e., every change is either a modification of existing software or an addition to existing software. This means that there are not to be any changes to functions that are implemented as sensors, signals that activate mechanisms, and sensors that raise signals. Let us consider the system components that may change. The structure of the information base is very likely to change with the addition of entire new segments, changes in subset partitions of existing sets, or addition of new functions in existing segments. The most radical change is the addition of new segments, but, because of communication by signals, this does not affect existing segments to any great extent. It may happen, though, that the new segment needs information that has not been gathered in the past, and that the logical place for holding some of this information is in the segments already there. Wherever the new functions may be located, they are likely to undergo changes, and the changes have to be made explicit in the definition of events. Moreover, new processes may have to be defined, which is done by stringing together events.

Although usually changes are enhancements, there can be exceptions. Suppose that our library is to be "downsized", meaning that in the future books will no longer be lent out. The changes brought about by this decision are easily implemented: segment *Borrowers* disappears; segment *Titles* is not affected; in segment *Copies* signal *AdjustCount* and function *History* are eliminated, as are the events that deal with the borrowing and return of copies. This list indicates also the full extent of the changes that would have to be made in the reverse direction to effect an enhancement that changes a non-lending library to a lending library.

The most complicated changes take place when the host system is modified. Our recommendation is that such a modification be considered in two parts. First, the host change brings about changes in the interface between host and controller. Second, the interface changes necessitate changes in the software system. The difficult part is to deduce what changes of the second kind are to arise from changes of the first kind. Once this has been determined, the modification of the controller follows the same pattern as controller enhancement.

5. Case study: an elevator controller

Consider a system that consists of a bank of *k* elevators. There are two modules, the elevator module and a dispatcher module. The set of the elevator module consists of the *k* elevators. One of the functions is *agenda*, which is set-valued, and consists of all the floors the elevator is to visit. When a person inside the elevator presses a floor button, this floor is entered into the agenda. When a person arrives at the bank of elevators and presses an up or down button, this is registered by the dispatcher. The dispatcher selects the elevator that is to visit this floor, and adds the floor to the agenda of the elevator by means of a signal. The dispatcher has two further functions: when an elevator becomes idle, the dispatcher decides on a holding floor to which this elevator is to be moved, and it reactivates an idle elevator when a need for its services arises. A full SF specification of the elevator module can be found in [12].

Now a change was proposed. Because there had been communication breaks between dispatcher and elevators, people had been waiting for elevators that never came. The new strategy: when an up or a down button is pressed, the floor is added directly to the agenda of every elevator; when an elevator visits this floor, the floor is taken off the agendas of the other elevators. Even with a dispatcher-elevator communication break we have a fail-safe situation – elevators will stop at some floors unnecessarily, but nobody will be forgotten.

The required changes to the specification of the elevator module were made in less than an hour. The following changes were made:

- Two "pickup" agendas for up and down requests were defined as extensions of the existing agenda. Let us call them up-agenda and down-agenda, respectively. Events *AddToUp* and *AddToDown* were provided for making additions to these agendas. However, in contrast to the existing agenda, of which there is one for every elevator, there is just one up-agenda and one down-agenda.
- The signal from the dispatcher that added a floor to the agenda of an elevator is now redundant – hence the transaction that picked up the signal, and the event that made the actual change to the agenda were deleted.
- The pressing of an up (down) button was interpreted as the activation of event *AddToUp* (*AddToDown*).
- In event *PassingSensor* one of the "pickup" agendas is to be consulted in addition to the existing agenda.
- Events *TakeFromUp* and *TakeFromDown* were defined. When an elevator stops at floor *k* in upward (downward) movement, floor *k* is taken off the up-agenda (down-agenda) by event *TakeFromUp* (*TakeFromDown*).

Since the up- and down-agendas are common to all elevators, the dispatcher does not have to be concerned with taking floors off agendas, so that the change has simplified the interaction between the dispatcher and the elevator modules. The simplification can be taken even further. Instead of an idle elevator being moved to a holding floor and taken out of the idle state, an empty elevator just remains at the floor at which it becomes empty, and any addition to the up- or down-agenda reactivates the elevator. There is then no need for the dispatcher module at all. However, the simpler system introduces one minor inconvenience. With the elevator selection under the control of the dispatcher, precisely one elevator would have stopped at a particular floor to pick up riders. Now more than one elevator may stop there before this floor is taken off the appropriate agenda. Further, to minimize the effect of possible communication breaks, each elevator should have its own processor, and its own copy of the elevator module – this raises consistency problems, but the maintenance of consistency of this distributed system is not our concern here.

6. SF and other specification languages

Past efforts at the specification of software systems have rarely managed to combine properties 1 and 2 of the list of Section 2, and where they have done so, their domains of applicability have been rather narrow. They certainly have not satisfied all ten requirements. In fact, there seems to be a polarization, with a multitude of diagramming conventions at one pole, and sophisticated mathematical notations at the other. Let us look now at a few specification methodologies that have found some measure of acceptance in industry, particularly in Great Britain, and that go beyond inadequately interpreted diagramming techniques.

Jackson System Development [15]. A JSD specification makes use of diagrams, but the diagrams are not intended to do much more than help arrive at an intuitive understanding of a system. The actual specification resembles a program. The specification defines a distributed network of processes that communicate by message passing. However, the interpretation of the meaning of JSD constructs is left largely to the user's intuition, which makes it difficult to tell the precise nature of the message-passing mechanism. A serious flaw of JSD is its extreme process orientation, which leaves the data aspect underemphasized. First, this hinders modularization. Second, when a JSD system refers to a data base, the data base is regarded as being outside the system. Data base constraints and operations are therefore the concern of a data base administrator rather than the developer of a controller. In the 1970s this was considered an advantage, but today we realize that data base constraints are rarely absolutes, i.e., that they are for the most part determined by the processes that make use of data.

The Statechart visual formalism [16, 17]. Statecharts are process diagrams with very elaborate diagramming conventions that allow a controller and its interaction with its host to be described in rigorous terms. The emphasis is on states, and on transitions between states. Transitions carry labels. On the diagram the labels are represented in a shorthand notation, but a dictionary contains them in a fully expanded form, and the expansions are written in a formal language. Some process designers may feel happy working with diagrams supported by text, and for them Statecharts are probably the most appropriate notation. Others may feel happier working with text supported by diagrams. They may find the SF formalism more to their liking.

The specification language Z [18]. A Z (pronounced "zed") specification consists of a declaration of the components of a data base, an expression in logic that defines valid states of the data base, and definitions of operations. Operations change data base states. Some background in discrete mathematics allows a reading knowledge of Z to be acquired fairly easily, but this does not mean that Z specifications are always easy to understand. The problem is that the state into which an operation brings the data base need not be a valid state, so that additional *implicit* operations may be needed to restore validity. Consequently Z specifications are mathematically elegant, but are not easy to validate by potential system users who often lack the required mathematical sophistication. Moreover, Z lacks facilities for dealing with real-time embedded systems. The emphasis that Z puts on states rather than types implies that there is no natural base for the modularization of large Z specifications.

Vienna Development Method (VDM) [19, 20]. Originally developed for precise definition of programming language constructs, VDM has become a widely used specification language. In VDM data are structured into records, and these records can serve as a base for modularization. However, just as with Z, there are no real-time facilities. We regard SF as a modernization and extension of VDM.

7. An agenda for the future

We have accumulated considerable experience regarding the effect of major design changes on SF specifications. The modular structure of an SF specification, and communication by message passing have ensured that in all cases the changes could be carried out cleanly and rapidly. There are two major tasks to be undertaken. First, a methodology has to be developed for transforming an existing software system into an SF specification. This is a very difficult task, and the transformations will require investment of significant amounts of time by system personnel, but the investment will pay off in the long run. As we noted in Section 2, it is to be regarded as preventive maintenance.

There are also reliability considerations. The reliability of a software system is the probability that it will not fail during a given time interval of execution, where by failure is meant a deviation from requirements. Unfortunately this definition presents difficulties. First, failure of a controller need not be due to deviation from requirements – the situation that leads to the failure may not have been anticipated. Second, a controller is often a reactive system, and execution time is then not the appropriate metric for the reliability computation. We have begun to address these difficulties.

However, the main problem is how to preserve reliability when controllers of very high reliability are being adapted to a changing host and changing operating conditions. It has been demonstrated that the reliability of software of ultra-high reliability cannot be determined by testing [21]. However, for software that has been in the field at multiple sites over a period of several years, the reliability can be determined – for an operating system that supports a monitoring system coupled to a nuclear reactor, a history of failures gathered at 5000 sites allows prediction of a future reliability of 0.9 per 1000 years of operation under two different reliability models [22, p.205]. Such software should be adapted in a way that preserves its known reliability.

The second major task is translation of SF specifications into programming languages, e.g., Ada or C. Although a pilot project has shown that SF can be translated into C [23], it is intended for prototyping alone. Extensive real time features allow easy expression of performance requirements in SF, but an SF implementation is unlikely to be sufficiently efficient to allow these requirements to be satisfied. We need, therefore, to develop a methodology for making an initial implementation (in Ada, say) more efficient by means of source-to-source transformations.

References

[1] Sneed, T.H., The myth of "top-down" software development and its consequences. *Proc. IEEE Conf. Software Maintenance, 1989*, pp.22-29.

[2] Van Zuylen, H.J., *Understanding in Reverse Engineering: the REDO Handbook*. Wiley, 1992.

[3] Balzer, R., and Goldman, N., Principles of good software specification and their implications for specification languages. *Proc. IEEE Conf. Specifications Reliable Software*, 1979, pp.58-67.

[4] Roman, G.-C., A taxonomy of current issues in requirements engineering. *Computer* **18**, 4 (April 1985), 14-23.

[5] Berztiss, A., The set-function approach to conceptual modeling. In *Information System Design Methodologies: Improving the Practice*, T.W.Olle, H.G.Sol, and A.A.Verrijn-Stuart (Eds.), North-Holland, 1986, pp.107-144.

[6] Luqi, The role of prototyping languages in CASE. *Int. J. Software Eng. Knowledge Eng.* **1** (1991), 131-149.

[7] Guttag, J.V., and Horning, J.J., Formal specification as a design tool. *Proc. 7th Symp. POPL, 1980*, pp.251-259.

[8] Swartout, W., and Balzer, R., On the inevitable intertwining of specification and implementation. *Comm. ACM* **25** (1982), 438-440.

[9] Berztiss, A., *Programming with Generators*. Elli Horwood, 1990.

[10] Boehm, B.W., Software risk management: principles and practices. *IEEE Software* **8**, 1 (Jan.1991), 32-41.

[11] Berztiss, A.T., The specification and prototyping language SF. SYSLAB Report 78, Department of Computer and Systems Sciences, The Royal Institute of Technology and Stockholm University, Electrum 230, S-16440 Kista, Sweden, 1990.

[12] Berztiss, A., Formal specification methods and visualization In *Principles of Visual Programming Systems*. S.-K. Chang (Ed.), Prentice-Hall, Englewood Cliffs, NJ, 1989, pp.231-290.

[13] Merlin, P., and Farber, D.J., Recoverability of communication protocols – implications of a theoretical study. *IEEE Trans. Commun.* **COM-24** (1976), 1036-1043.

[14] Berthomieu, B., and Diaz, M., Modeling and verification of time dependent systems using time Petri nets. *IEEE Trans. Software Eng.* **17** (1991), 259-273.

[15] Cameron, J.R., *JSP and JSD: The Jackson Approach to Software Development*, 2nd Ed. IEEE Computer Society Press, 1989.

[16] Harel, D., On visual formalisms. *Comm. ACM* **31** (1988), 514-530.

[17] Harel, D., Lachover, H., Naamad, A., Pnueli, A., Politi, M., Sherman, R., and Shtull-Trauring, A., Statemate: a working environment for the development of complex reactive systems. *IEEE Trans. Software Eng.* **16** (1990), 403-414.

[18] Potter, B., Sinclair, J., and Till, D., *An Introduction to Formal Specification and Z*. Prentice-Hall, 1991.

[19] Jones, C.B., *Systematic Software Development Using VDM*. Prentice-Hall, 1986.

[20] Cohen, B.W., Harwood, W.T., and Jackson, M.I., *The Specification of Complex Systems*. Addison-Wesley, 1986.

[21] Butler, R.W., and Finelli, G.B., The infeasibility of experimental quantification of life-critical software reliability. *Proc. ACM SIGSOFT '91 Conf. Software for Critical Systems* (*Software Engineering Notes* 16, 5 (Dec. 1991)), 66-76. [Reprinted *IEEE Trans. Software Eng.* 19 (1993), 3-12.]

[22] Musa, J.D., Iannino, A., and Okumoto, K., *Software Reliability – Measurement, Prediction, Application*. McGraw-Hill, 1987.

[23] Berztiss, A.T., and Liu, C.-T., The prototyping language SF and its implementation. *Proc. 2nd Internat. Conf. Software Eng. and Knowledge Eng.*, June 1990, pp.51-57.

Efficient Methods for Validating Timing Constraints in Multiprocessor and Distributed Systems *

Jane W. S. Liu and Rhan Ha
Department of Computer Science
University of Illinois
1304 West Springfield Avenue
Urbana, Illinois 61801

Abstract

This paper discusses the difficulties in validating timing constraints of dynamic multiprocessor and distributed systems. Some worst-case bounds and efficient algorithms now exist for the special case where jobs are independent. These results are summarized.

1 Introduction

In a *real-time system*, many jobs are time-critical; their execution must meet certain timing constraints. The term *job* refers to a unit of work to be scheduled and executed. A job may be the computation of a control law, the transmission of an operator command, the retrieval of a file, etc. To execute, it requires a computer, a data link, a disk, respectively; we refer to them all as *processors*. The length of time a job requires to complete if it were to execute alone is called its *execution time*. In the simplest form, the timing constraint of a job are specified in terms of its release time and deadline: the job cannot begin to execute until its *release time* and must complete its execution by its *deadline*. The failure of a job to complete by its deadline is considered to be a time fault, and a real-time system functions correctly only in the absence of time faults. To validate a real-time system, its builder must be able to demonstrate convincingly not only that the system meets all of its functional requirements but also that every time-critical job in it always completes by the job's deadline.

"Non-functional, or quality, aspects of large systems are often treated in an ad hoc manner, even when they are critical to the system's ultimate success." This statement by Salasin and Waugh [1] is especially true for real-time systems. Traditionally, when building or reengineering a real-time system, one first focuses on its functional requirements. Whether the system can meet its real-time requirements is checked only after most of the design decisions have been made and, often, after parts of the system have been implemented. Timing constraints are validated by exhaustive simulation or testing. This approach is time consuming and costly. To ensure that the system can be reliably tested, one is forced to restrict the choices of scheduling strategy, operating system, and underlying system architecture. For this reason, modern scheduling paradigms that lead to easy-to-modify/maintain systems are not used. Almost all real-time systems that support critical applicat:ons use clock-driven or cyclic scheduling strategies. Such a system is brittle, difficult to maintain and extend. Because a small change in the application software or the underlying hardware and system software can produce unpredictable timing effects, the system must be tuned and tested exhaustively after every change.

This situation has improved in recent years. There are now reliable and tractable validation methods for static multiprocessor and distributed systems [2-7]. By *static system*, we mean a system in which jobs are statically assigned and bound to processors and are migrated among processors on a relatively infrequently basis. Jobs on each processor are scheduled according to a uniprocessor scheduling algorithm. A new generation of analysis and validation tools built on these recent theoretical advances are now beginning to emerge. (An example is PERTS [8].)

*This work has been partially supported by ONR Contract Nos. N00014-89-J-1181 and N000-92-J-1815 and NASA Grant No. NAG 1-1613.

In contrast, efficient methods for validating dynamic multiprocessor and distributed systems are not yet available. In a *dynamic system*, jobs ready for execution are placed in a common queue and are dispatched and scheduled on available processors in an event-driven manner. Although numerous dynamic scheduling algorithms are available, the lack of efficient, reliable and provably correct ways to validate that all deadlines are met in dynamic systems prevents the practical adoptions of these algorithms.

This paper first gives an overview of existing analytical and efficient methods for validating static systems built on well-known scheduling algorithms. It then describes several new worst-case bounds and efficient algorithms for validating dynamic systems that cannot be validated using existing methods. The special cases of the validation problem considered here are concerned with independent jobs that have arbitrary release times, arbitrary deadlines, and variable execution times. Jobs are scheduled according to a priority-driven algorithm. A scheduling algorithm is *priority-driven* if it does not leave any resource idle intentionally. Such an algorithm can be implemented by assigning priorities to jobs and placing all jobs ready for execution in a queue ordered by their priorities. The available processor(s) is (are) allocated to the job(s) at the head of this queue. Priority-driven algorithms differ from each other in the rules they use to assign priorities to jobs. Almost all commonly used event-driven scheduling algorithms, such as FIFO, LIFO, shortest-processing-time-first, earliest-deadline-first, rate-monotonic, and deadline-monotonic algorithms are priority-driven.

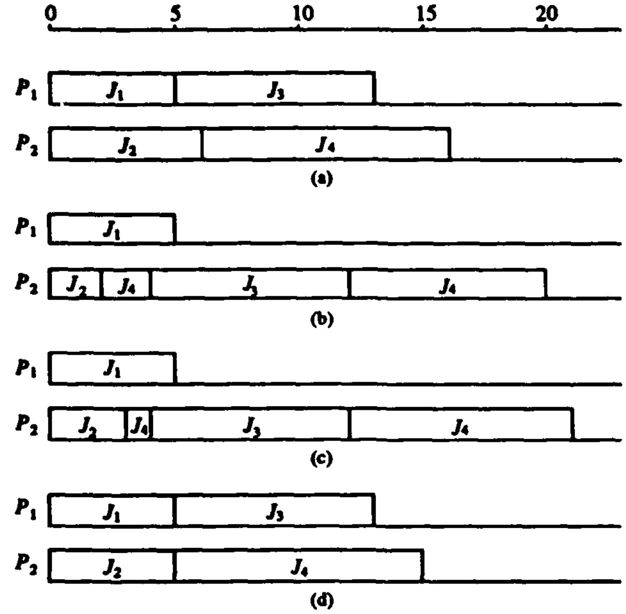
Following this introduction, Section 2 discusses the difficulties in validating timing constraints in dynamic systems and gives a formal definition of the validation problem considered here. Section 3 gives an overview of methods for validating static systems. Section 4 summaries the worst-case bounds and efficient algorithms for validating timing constraints of independent jobs in dynamic systems. Section 5 discusses the work that remains to be done in order to build a comprehensive strategy for validating dynamic systems.

2 Validation Problem

It is well-known that a system in which jobs are scheduled in a priority-driven manner may exhibit scheduling anomalies. Graham [9] has shown that the completion time of a set of jobs can be later when more processors are used to execute them and when jobs have shorter execution times and fewer dependencies. When jobs have arbitrary release times and share nonpreemptable resources, scheduling anomalies can occur even when there is only one processor and the jobs are preemptable. These anomalies make ensuring full coverage in simulation and testing difficult whenever there are variations in job execution time and resource requirements and jitters in job release times. Unfortunately, these variations are often unavoidable. Given an arbitrary scheduling algorithm, there is no efficient way to find the worst-case completion time of each job. This is why exhaustive simulation and testing are impractical and unreliable when used to determine whether all jobs always complete in time in large and dynamic systems.

Figure 1 shows an illustrative example. The simple system in this figure contains 4 independent jobs and 2 identical processors. The release time, deadline and execution time of job J_i are denoted by r_i, d_i and e_i, respectively. These job parameters are listed in the table. In this example, the execution times of all the jobs are known except for J_2. Its execution time can be any value in the range [2,6]. The scheduling algorithm in this example is preemptive and priority-driven; the priority order is J_1, J_2, J_3 and J_4 with J_1 having the highest priority. A constraint is that jobs are *not migratable.* In other words, once a job begins execution on a processor, it is constrained to execute on that processor until completion. We want to validate that all deadlines can be met, assuming that the scheduler works correctly, that is, it never schedules any job before its release time. A naive way is to simulate the system twice: when the execution time of J_2 has the maximum value 6 and when it has the minimum value 2. The results are the schedules shown in parts (a) and (b) of Figure 1. By examining these schedules, we would conclude that all jobs can complete by their deadlines. This conclusion is incorrect because the simulation test does not give us full coverage. This fact is illustrated by the schedules in parts (c) and (d). The worst-case schedule is shown in (c); the completion time of J_4 is 21 when the execution time of J_2 is 3. The best-case schedule is shown in (d); J_4 completes at time 15 when the execution time of J_2 is 5. To find the schedules in (c) and (d) by simulating the system, we need to exhaustively try all possible execution times of J_2.

job	r_i	d_i	e_i
J_1	0	10	5
J_2	0	10	[2,6]
J_3	4	15	8
J_4	0	20	10



(a)

(b)

(c)

(d)

Figure 1: An example illustrating scheduling anomalies

Our objective is to find analytical expressions and efficient algorithms with which we can determine reliably whether every job can meet its deadline. In other words, given a set of jobs, the set of resources available to the jobs, and the scheduling (and resource access-control) algorithm to allocate processors and resources to jobs, we want to find in polynomial time an upper bound of the completion time of every job. Because jobs have different properties and there are different rules governing resource usage, this problem has many variants. The periodic-task schedulability analysis problem whose solutions are summarized in Section 3 is a variant. This section describes other variants of this problem that remain to be solved, as well as the variant solved by the results presented in Section 4. It also introduces the notations that will be used later.

Variants of the Validation Problem

We characterize the workload to be scheduled and, hence, analyzed as a set $J = \{J_1, J_2, \cdots, J_n\}$ of jobs. Each job J_i is defined by its release time r_i, deadline d_i and execution time e_i. When there is jitter in its release time, r_i can have any value in the range $[r_i^-, r_i^+]$ where r_i^- and r_i^+ are the *earliest release time* and the *latest release time* of J_i, respectively. Without loss of generality, we assume that $r_i^- \geq 0$ for all i, that is, no job is released before $t = 0$. We say that the jobs have fixed release times, or there are no jitters, when $r_i^- = r_i = r_i^+$ and that they have identical, or zero, release times when $r_i^- = r_i^+ = 0$ for all i. The actual execution time e_i is in the range $[e_i^-, e_i^+]$ and therefore can be as small as its *minimum execution time* e_i^- and as large as its *maximum execution time* e_i^+. $[r_i^-, r_i^+]$ and $[e_i^-, e_i^+]$ are given parameters of J_i. J_i's actual execution time e_i may depend on its input data, as well as the underlying hardware configuration and run-time environment, and may be unknown until the job's execution completes. Similarly, the actual release time r_i of J_i becomes known when J_i is released.

The jobs in J may be dependent; data and control dependencies between them impose *precedence constraints* in the order of their execution. A job J_i is a *predecessor* of another job J_j (and J_j is a *successor* of J_i) if J_j cannot begin execution until the execution of J_i completes. Two jobs J_i and J_j are *independent* if they can be executed in any order.

We confine our attention here to off-line scheduling. In other words, the scheduler knows the parameters $[r_i^-, r_i^+]$, $[e_i^-, e_i^+]$ and d_i of every job J_i before any job begins execution. The scheduling algorithm is priority-driven. It assigns fixed priorities to jobs. It may assign priorities to jobs based on the known job parameters. Some algorithms, such as the FIFO algorithm, assign priorities to jobs according to their actual release times. However, none of the algorithms considered here assign priorities to jobs based on their actual execution times.

Therefore, the given scheduling algorithm is completely defined by the list of priorities it assigns to the jobs. Without loss of generality, we assume that the priorities of jobs are distinct. We will use the list $(J_1, J_2, \cdots, J_n)$ in decreasing priority order except where it is stated to be otherwise. In other words, we always index the jobs so that J_i has a higher priority than J_j if $i < j$. $J_i = \{J_1, J_2, \cdots, J_i\}$ denotes the subset of jobs with priorities equal to or higher than the priority of J_i.

In a dynamic system containing m identical processors, the scheduler maintains a common priority queue and places all jobs ready for execution in the queue. There are the following three cases:

(1) preemptable and migratable: In this case, a job can be scheduled on any processor. It may be preempted when a higher priority job becomes ready. Its execution may resume on any processor.

(2) preemptable and nonmigratable: As in case (1), each job can begin its execution on any processor and is preemptable. However, it is constrained to execute to completion on the same processor. Figure 1 gives an example of this case.

(3) nonpreemptable: Each job can be scheduled on any processor. Some or all of the jobs are nonpreemptable.

In addition to processors, the system may also have a set of serially reusable resources. A job may require some of these resources, as well as a processor, in order to execute. When some of the resources required by two or more jobs are the same, the jobs are said to be in *resource conflict*. We assume that a resource access-control protocol is used to resolve resource conflicts among jobs, and this protocol controls priority inversion and prevents deadlock. Therefore the length of time any job J_i may be blocked from execution due to resource conflict is bounded from above. This bound is called the ***worst-case blocking time*** of J_i and is denoted by b_i. For a given resource access-control protocol, b_i is given for every job J_i. We need not be concerned with the specific details about the protocol.

The validation problem has four dimensions. Two dimensions are whether jobs are dependent and whether they share any resource. Section 4 is concerned only with independent jobs that do not share any resources. The other two dimensions of the problem are the release-time characteristics of jobs and rules in scheduling. Each special case based on these two dimensions is referred to by three capital letters separated by "/". The first letter denotes preemptability. It can be either "P", for preemptable, or "N", for nonpreemptable. The second letter defines the migratability of jobs. It can be either "M", for migratable, or "N", for nonmigratable. The third letter describes the release time characteristics. It can be either "Z", for zero release times, or "F", for fixed arbitrary release times, or "J", for jittered release times. For example, by jobs being P/M/F (or P/M/F jobs), we mean jobs that are preemptable and migratable and have fixed arbitrary release times. N/N/Z jobs are nonpreemptable (and therefore not migratable) and have zero, or identical, release times.

Definitions and Notations

Let J_i^+ denote the set $\{J_1^+, J_2^+, \cdots, J_i^+\}$ of jobs in which every job has its maximum execution time. Similarly, J_i^- denotes the set $\{J_1^-, J_2^-, \cdots, J_i^-\}$ in [illegible] every job has its minimum execution time. We [illegible] to the schedule of J_i produced by the given algorithm [illegible] as the ***actual schedule*** A_i and the schedule of J_i^+ (or J_i^-) produced by the same algorithm as the ***maximal*** (or the ***minimal***) ***schedule*** A_i^+ (or A_i^-) of J_i.

Let $S(J_i)$ be the instant of time at which the execution of J_i begins according to the actual schedule A_n. $S(J_i)$ is the (actual) ***start time*** of J_i. Let $S^+(J_i)$ and $S^-(J_i)$ be the ***observable start times*** of J_i in the schedules A_n^+ and A_n^-, respectively. Clearly, for jobs with fixed release times, $S^+(J_i)$ and $S^-(J_i)$ can easily be found by constructing the maximal and minimal schedules and observing when J_i starts according to these schedules. In contrast, $S(J_i)$ is unknown until J_i actually starts. Moreover, because of varied execution times, J_i may start at different times when we repeatedly simulate the system. We say that the start time of J_i is ***predictable*** if $S^+(J_i) \geq S(J_i) \geq S^-(J_i)$.

Similarly, let $F(J_i)$ be the instant at which J_i completes execution according to the actual schedule A_n. $F(J_i)$ is the ***completion time*** of J_i. The ***response time*** of a job is the length of time between its release time and its completion time. Let $F^+(J_i)$ and $F^-(J_i)$ be the ***observable completion times*** of J_i according to the schedules A_n^+ and A_n^-, respectively. The completion times of J_i is said to be predictable if $F^+(J_i) \geq F(J_i) \geq F^-(J_i)$.

We say that ***the execution of J_i is predictable*** if both its start time and completion time are predictable. When the execution of J_i is predictable, the completion time $F^+(J_i)$ in the schedule A_n^+ minus the minimum release time r_i^- of J_i gives J_i's worst-case response time. J_i meets its deadline if $F^+(J_i) \leq d_i$.

Let $w_i(t,\ t')$, for time instants $t < t'$, denote the sum of execution times of all the jobs in the set J_i whose release times are in the interval $[t,\ t']$. The job J_i and all jobs with higher priorities than it require at most $w_i(t,\ t')$ additional units of processor time in the interval $[t,\ t']$. We call $w_i(t,\ t')$ the ***incremental (processor) time demand*** of J_i in the interval $[t,\ t']$. $w_i(t) = w_i(0,\ t)$ is, therefore, the ***total (processor) time demand*** of J_i before t. It is equal to the amount of processor time required by all jobs that are in J_i and have release times at or earlier than t. Similarly, let $w_i^+(t,\ t')$ (or $w_i^-(t,\ t')$) be the sum of the maximum (or minimum) execution times of all jobs that are in J_i and have release times in $[t,\ t']$. Let $w_i^+(t) = w_i^+(0,\ t)$ (or $w_i^-(t) = w_i^-(0,\ t)$). $w_i^+(t)$ (or $w_i^-(t)$) is the ***maximum*** (or ***minimum***) ***time demand*** of J_i before t. Clearly, $w_i^-(t) \leq w_i(t) \leq w_i^+(t)$ for all i and t.

3 Methods for Validating Static Systems

Again, almost all existing analytical and efficient methods for bounding the worst-case completion times are for static systems. In this case, a general strategy is to first determine how late each job J_i can be delayed from start and completion by jobs that are assigned on the same processor with it and, then, take into account of additional delays due to synchronization with jobs on all processors.

The best known and the most comprehensive set of bounds and algorithms are those based on the periodic-task model [2-7]. In this model, the set of jobs assigned and executed on each processor is partitioned into n subsets, each called a task. Some tasks are periodic; each periodic task T_i is a sequence of jobs whose release times are spaced nominally at regular intervals. There may be jitters, but the lengths of these intervals are never less than p_i, called the period of the task. The release time f_i of the first job in a task T_i is called its phase. The length of time δ_i between the release time of every job in T_i and its deadline is called the relative deadline of T_i. δ_i is usually equal to or less than p_i. With a slight abuse of the notation, we use e_i^+ to denote the maximum execution time of each job in T_i. $u_i = e_i^+/p_i$ is the maximum fraction of time the jobs in T_i use the processor and is called the utilization of the task T_i.

Some tasks are (periodic) servers [4]. A periodic server is created to handle the execution of a stream of jobs whose release times and execution times are random variables. Jobs handled by each server are placed in a priority queue. Whenever the server is scheduled and allocated the processor, the job at the head of this queue executes. Each server T_s is characterized by its period p_s, execution time (budget) e_s^+, and relative deadline δ_s. The scheduler treats each server as a periodic task with these parameters. Therefore when we try to bound the completion times of jobs in periodic tasks, there is no need to treat the servers differently.

The precedence constraints between jobs in the same task, if any, are naturally taken care of by making the release time of every predecessor job equal to or earlier than its successor jobs and by executing jobs in the task in the FIFO order. Similarly, data and control dependencies between jobs in different tasks can be taken care of by adjusting the phases of the tasks so that the deadline of every predecessor job is earlier than the release times of its successor jobs. In this way, we can ignore precedence constraints and treat all jobs as if they are independent.

In our notation, most of the jobs are P/N/J jobs; they are scheduled preemptively (and are not migrated). Their resource accesses are controlled by a protocol (such as the ones in [6, 7]) that ensures the blocking time of every job in T_i due to resource conflicts with all jobs in the system is never more than b_i. For such systems, there are several sufficient conditions, which, when satisfied by a task, allow us to conclude that all jobs in it always complete by their deadlines. An example is the inequality

$$\sum_{k=1}^{i} u_k + b_i/p_i \leq i(2^{1/i} - 1) \qquad (1)$$

assuming that we index the tasks so that $p_1 < p_2 < \cdots < p_n$. When there is only one processor and tasks are scheduled on the rate-monotonic basis (that is, the shorter the period, the higher the priority) and synchronized according to the priority-ceiling protocol [6], all jobs in T_i with $\delta_i = p_i$ always complete by their deadlines as long as (1) is satisfied. Similar conditions are known for many other fixed-priority algorithms (which assign the same priority to all jobs in each task and schedule the jobs in the same task in the FIFO order) and for arbitrary values of δ_i less than or equal to p_i. Bounds also exist for the earliest-deadline-first algorithm, when there is only one processor and a protocol such as the stack-based protocol [7] is used. It is straightforward to generalize the conditions to account for the effec's of nonpreemption if some jobs are not preemptable. It is also straightforward to use these conditions to bound the worst-case completion times of jobs in periodic job-shops and flow-shops where each job consists of subjobs which execute in turn on two or more processors and have end-to-end deadlines [10].

The known sufficient conditions, such as (1), are particularly robust. Specifically, the values of the periods and worst-case execution times of jobs in tasks $T_1, T_2, \cdots, T_i$ do not appear in the left-handed side of (1), only their utilizations. Moreover, (1) assumes that the job being analyzed is released at an instant between which and the deadline of the job the total processor time demand of all higher-priority jobs is the largest. Therefore if the job were actually released at this instant, it would have the largest response time. Because of this assumption, the conclusion that a job can complete before its deadline based on such a sufficient condition remains true no matter what its actual release time is.

However, a test based on a sufficient condition like (1) is sometimes pessimistic. If (1) fails to hold for T_i, for example, its jobs may nevertheless always complete in time. An algorithm that makes use of the known parameters p_i and e_i^+ can give a more accurate prediction of the worst-case response times. Such algorithms are used in PERTS [8]. They are based on a more exact characterization [5] of the rate-monotonically scheduled periodic tasks. To determine whether any job in a task T_i can meet its deadline, the algorithm takes the release time of the job as the time origin 0 and computes the maximum time demand $w_i^+(t)$ between 0 and its deadline at δ_i. To compute $w_i^+(t)$, the algorithm uses as phases of the other tasks the values that maximize $w_i^+(t)$ for all t. (Usually, the choices are $f_k = 0$ for all k.) The job completes for sure at the earliest time instant t when $w_i^+(t) + b_i \leq t$. In other words, the worst-case completion time of the job is $w_i^+(t) + b_i$ after its release time.

We note that this approach of checking when the supply of time meets the demand for time is general enough that it can be generalized and applied to validate most static systems. Many systems do not fit the periodic-task model; jobs have arbitrary release times and precedence constraints. Because jobs on each processor are scheduled by themselves, a way to ensure that the scheduler can always enforce the given precedence constraints is to work with the effective deadlines and release times of jobs rather than their given deadlines and release times. A job with successors must be completed before them. Hence, the effective deadline of the job is the earliest deadline among its deadline and the deadlines of its successors. Similarly, its effective release time is the latest time among its release time and the release times of its successors. Working with effective release times and deadlines allows the scheduler to temporarily ignore the precedence constraints between jobs and make scheduling decisions as if the jobs are independent. To validate whether every job completes before its deadline, we can use the algorithms for validating independent jobs. Unfortunately, this method does not work when the system is dynamic. For this reason, the results on independent jobs presented in the next section cannot be readily extended to deal with dependent jobs in dynamic systems.

When the release times of jobs are arbitrary, rather than periodic, it is slightly more complicated to determine the values of release times that maximize the potential demand for time in the interval between the release time and deadline of the job being analyzed. This can be done efficiently, however, and Algorithm $\mathcal{IPMJ}$ presented in Section 4 for transforming a set of jobs with jittered release times to fixed release times can be used for this purpose.

4 Methods for Validating Dynamic Systems

We now focus our attention on dynamic multiprocessor systems. Specifically, the performance bounds and algorithms presented are for jobs that are independent and do not share any resources. Proofs of theorems in this section and examples to illustrate them can be found in [11].

Conditions for Predictable Execution

It is easy to find the worst-case and best-case completion times of independent P/M/F jobs. In particular, the following theorem and corollary allow us to conclude that the execution of independent P/M/F jobs is predictable. To find the worst-case (or best-case) response time of a job J_i in a set $\mathbf{J}_n$ of independent P/M/F jobs with arbitrary and fixed release times, we apply the given scheduling algorithm on the set J_n^+ (or J_n^-) where all jobs have their maximum (or minimum) execution times. The response times of J_i according to the resultant schedule A_n^+ (or A_n^-) is its largest (or smallest) possible response time. We are sure that J_i always meets its deadline if it meets its deadline in the maximal schedule A_n^+.

Theorem 1 *The start time of every job in a set of independent P/M/F jobs is predictable, that is,* $S^+(J_i) \geq S(J_i) \geq S^-(J_i)$.

Corollary 1 *The completion time of every job in a set of independent P/M/F jobs is predictable, that is,* $F^+(J_i) \geq F(J_i) \geq F^-(J_i)$.

When there are jitters in release times, whether J_i is schedulable depends not only on its own release time and execution time but also on the release times and execution times of all the higher-priority jobs. In trying to find the worst-case completion time of a job, we cannot simply choose the earliest or the latest release times of higher-priority jobs. Algorithm $\mathcal{IPMJ}$ is based on this observation. This algorithm tries to find bounds of start times and completion times of independent P/M/J jobs, by considering one job at a time, from the job with the highest priority to the job with the lowest priority. It transforms J_i and the jobs that have priorities higher than J_i into jobs with fixed release times as follows. Let K_k denote the job transformed from J_k and $\boldsymbol{K}_i$ denote the set of transformed jobs $\{K_1, K_2, \cdots, K_i\}$. Algorithm $\mathcal{IPMJ}$ has the following three steps:

(1) Step 1 computes the parameters of the transformed job K_i from the parameters of J_i. Specifically, K_i's execution time is equal to J_i's maximum execution time, e_i^+, plus the length of its jitter interval, $(r_i^+ - r_i^-)$. K_i's release time is J_i's earliest release time, r_i^-.

(2) Step 2 computes the parameters of K_k for each of $k = 1, \cdots, i-1$. K_k's execution time is e_k^+, that is, the maximum execution time of J_k. K_k's release time is chosen among r_i^-, r_k^+ and r_k^- so that the overlap between the feasible intervals of K_i and K_k is as large as possible. Specifically, if $r_k^- < r_i^- < r_k^+$, K_k's release time is r_i^-; if $r_k^+ \leq r_i^-$, K_k's release time is r_k^+ (as late as possible); and if $r_k^- \geq r_i^-$, K_k's release time is r_k^- (as early as possible).

(3) Step 3 schedules $\boldsymbol{K}_i$ according to the given preemptable, migratable, priority-driven algorithm. An upper bound of the completion time $F(J_i)$ of J_i is equal to the completion time of K_i in the resultant schedule $\boldsymbol{A}_i'$.

Because $\boldsymbol{K}_{i-1}$ is not a subset of $\boldsymbol{K}_i$, $\boldsymbol{K}_i$ needs to be constructed for every job J_i. Consequently, the complexity of Algorithm $\mathcal{IPMJ}$ is $O(n^2)$. The following theorem allows us to conclude that if K_i can complete by the deadline d_i of J_i in the schedule $\boldsymbol{A}_i'$ generated by the $\mathcal{IPMJ}$ algorithm, then J_i is schedulable for all possible combinations of release times and execution times.

Theorem 2 *The completion time $F(J_i)$ of J_i is no later than the completion time of the transformed job K_i in the observable schedule $\boldsymbol{A}_i'$ of $\boldsymbol{K}_i$ generated by Algorithm $\mathcal{IPMJ}$.*

In the special case when independent jobs have zero, or identical, release times and jobs have fixed priorities, preemption and migration can never occur. Therefore, it does not matter whether preemption and migration are allowed or not. The following theorem follows straightforwardly from this observation.

Theorem 3 *The execution of the independent N(P)/N/Z jobs is predictable.*

When jobs have arbitrary release times and are not migratable, their execution behavior is no longer predictable. This fact is illustrated by the example in Figure 1. While the execution of independent P/N/F jobs is not predictable for arbitrary priority assignments, it is predictable when the jobs are scheduled in the order in which they are released. This fact is stated formally in the following theorem.

Theorem 4 *When the priorities of independent P/N/F jobs are assigned on the FIFO basis (that is, the earlier the release time, the higher the priority), the execution of the jobs is predictable.*

Completion Times of P/N/F Jobs

When independent P/N/F jobs are not scheduled on the FIFO basis and their execution is no longer predictable, we can bound their start times and completion times according to the following theorem. We note that when we want to determine whether J_i is schedulable, there is no need to consider

$J_{i+1}, J_{i+2}, \cdots, J_n$, since jobs are preemptable and independent. Therefore, we can confine our attention to $\mathbf{J}_i$. The theorem is stated in terms of the set $\mathbf{D}_i$; $\mathbf{D}_i$ is a subset of $\mathbf{J}_i$ in which each job J_k is released after some job in $\mathbf{J}_i$ with a priority lower than itself (that is, J_k) and is not scheduled to start and complete before J_i on the same processor as J_i in the maximal schedule. In the example in Figure 1, $\mathbf{D}_1$, $\mathbf{D}_2$ and $\mathbf{D}_3$ are null. $\mathbf{D}_4$ is $\{J_3\}$.

Theorem 5 $S(J_i) \leq S^+(J_i) + \sum_{J_k \in D_i} e_k^+$, *and* $F(J_i) \leq F^+(J_i) + \sum_{J_k \in D_i} e_k^+ - (e_i^+ - e_i)$.

The worst-case bound of completion time given by this theorem is sometimes too pessimistic. For example, this theorem tells us that the completion times of the jobs in Figure 1 are no greater than 5, 6, 13 and 24, respectively. The bound for $F(J_4)$ is pessimistic. As another example, we consider a simple system containing m independent jobs and m identical processors. The release times of the jobs $J_1, J_2, \cdots, J_m$ are such that $r_m < r_{m-1} < \cdots < r_1$. The priority order is $J_1, J_2, \cdots, J_m$ with J_1 having the highest priority. Obviously, every job J_i can be scheduled immediately after its release time and can always complete at or before its observable worst-case completion time $F^+(J_i)$. The bound of the worst-case completion time of J_m computed from Theorem 5 can be as large as m times the actual completion time of J_m, and is therefore not useful.

Sometimes, we can use the information provided by the two observable schedules to derive more accurate predictions of job completion times. For example, we consider tests that begin by examining the sequences in which the jobs start execution according to the minimal and maximal schedules. Let $\wp_i^+(t)$ (or $\wp_i^-(t)$) be the sequence of jobs whose observable start times are at or before t according to the maximal schedule $\mathbf{A}_i^+$ (or minimal schedule $\mathbf{A}_i^-$) of $\mathbf{J}_i$; the jobs in the sequence appear in order of increasing start times. For example, in Figure 1, $\wp_4^+(S^+(J_4))$ is (J_1, J_2, J_3, J_4) and $\wp_4^-(S^-(J_4))$ is (J_1, J_2, J_4). Similarly, let $\wp_i(t)$ be the corresponding sequence of jobs in increasing order of their actual start times according to the actual schedule $\mathbf{A}_i$, including all jobs whose actual start times are at or before t. We call $\wp_i(t)$ the *actual starting sequence*, and $\wp_i^+(t)$ and $\wp_i^-(t)$ the *maximal* and *minimal observable starting sequences* according to $\mathbf{A}_i^+$ and $\mathbf{A}_i^-$, respectively. We say that a sequence X is a *subsequence* of a sequence Y if Y contains X and the elements in both X and Y appear in X and Y in the same order. In Figure 1, $\wp_4^-(S^-(J_4)) = (J_1, J_2, J_4)$ is a subsequence of $\wp_4^+(S^+(J_4)) = (J_1, J_2, J_3, J_4)$. Similarly, $\wp_4^+(S^+(J_3)) = (J_1, J_2, J_3)$ is a subsequence of $\wp_4^-(S^-(J_3)) = (J_1, J_2, J_4, J_3)$.

When we want to determine whether J_i is schedulable, we first examine whether there is preemption in the maximal schedule. In the simpler case; no job in $\mathbf{J}_i$ is preempted in the maximal schedule $\mathbf{A}_i^+$. Then, we examine whether the two observable starting sequences $\wp_i^+(S^+(J_i))$ and $\wp_i^-(S^-(J_i))$ are identical. If the two sequences are identical, we can conclude that no job in $\mathbf{J}_i$ is preempted and the orders in which jobs start execution are the same, according to all the schedules of $\mathbf{J}_i$ for all combinations of execution times of jobs in $\mathbf{J}_i$. Therefore, the latest completion time of J_i is $F^+(J_i)$. This fact is stated in Theorem 6.

Theorem 6 *If no job in $\mathbf{J}_i$ is preempted according to $\mathbf{A}_i^+$ and the two observable starting sequences $\wp_i^+(S^+(J_i))$ and $\wp_i^-(S^-(J_i))$ are identical, then $F^-(J_i) \leq F(J_i) \leq F^+(J_i)$.*

Obviously, the upper bound of $F(J_i)$ given by this theorem is tight. For example, for the system in Figure 1, no job in $\mathbf{J}_3$ is preempted according to the maximal schedule $\mathbf{A}_3^+$, and $\wp_3^+(S^+(J_3))$ is identical to $\wp_3^-(S^-(J_3))$. Consequently, we can conclude that the completion time of J_3 is never later than $F^+(J_3)$, which is 13 according to Figure 1 (a). Similarly, for the system consisting of m jobs that is mentioned earlier, no job in $\mathbf{J}_m$ is preempted according to the maximal schedule $\mathbf{A}_m^+$, and $\wp_m^+(S^+(J_m))$ is identical to $\wp_m^-(S^-(J_m))$. Hence the completion time of each job J_i is at most equal to $F^+(J_i)$.

A natural question to ask at this point is whether a tight bound can be derived in a similar manner for the case when there is no preemption in the maximal schedule and $\wp_i^-(S^-(J_i))$ is a subsequence of $\wp_i^+(S^+(J_i))$. Figure 2 illustrates the impossibility. Parts (a) and (b) show the maximal and minimal schedules, respectively. Part (c) shows a possible actual schedule. J_1, J_2, J_3 and J_4 have the same parameters as the ones in Figure 1 except that the execution time of J_2 is in the range [1,6] and the release times of J_4 is 2. The parameters of J_5, J_6 and J_7

are listed in the table in Figure 2. $\wp_7^-(S^-(J_7)) = (J_1, J_2, J_7)$ is a subsequence of $\wp_7^+(S^+(J_7)) = (J_1, J_2, J_3, J_4, J_6, J_7)$, but the actual starting sequence $\wp_7(S(J_7)) = (J_1, J_2, J_4, J_3, J_6, J_5, J_7)$, is not. Furthermore, the set of jobs in the actual starting sequence $\wp_7(S(J_7))$ is not a subset of the set of jobs in the maximal starting sequence $\wp_7^+(S^+(J_7))$. (J_5 is not in $\wp_7^+(S^+(J_7)$ but is in $\wp_7(S(J_7))$.) The actual start time of J_7 is larger than the maximal start time $S^+(J_7)$, illustrating that the start time is not predictable in this case. Moreover, according to both A_7^+ and A_7^-, no job is preempted before the start time $S(J_7)$ of J_7, but J_4 and J_6 are preempted before $S(J_7)$ according to the actual schedule in Figure 2 (c).

job	r_i	d_i	e_i
J_1	0	10	5
J_2	0	10	[1,6]
J_3	4	15	8
J_4	2	20	10
J_5	18	25	3
J_6	5	200	100
J_7	0	22	2

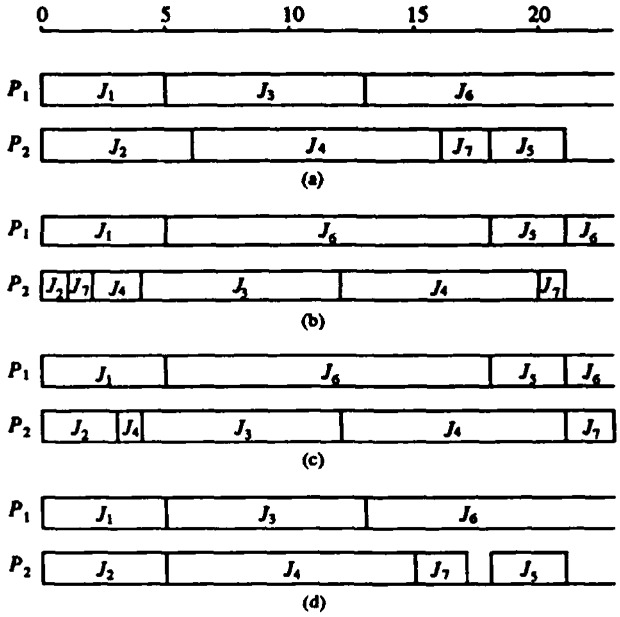


Figure 2: An example illustrating unpredictable start times and completion times

Similarly, when some job(s) in J_i is preempted before the completion time of J_i in the maximal schedule, the actual starting sequence may be different from the observable starting sequences, even though the two observable starting sequences are same. When some job(s) in J_i is preempted before the completion time of J_i in the maximal schedule, the start time and completion time of J_i may be unpredictable, even though all the starting sequences are same. In the actual schedule, a job may be preempted by a different job from the one in the maximal schedule, even though all the starting sequences are same. Examples illustrating these facts can be found in [12]. These examples lead us to believe that it is unlikely for us to find tighter bounds than the one given by Theorem 5 when there is preemption in the maximal schedule. In many examples, in fact, the completion times of the jobs are accurately predicted by the upper bounds given by Theorem 5.

Completion Times of N/N/F Jobs

We now consider the case when all jobs are nonpreemptable and the release times of all jobs are arbitrary but fixed. A lower-priority nonpreemptable job whose release time is earlier than J_i may be executed to completion after J_i in the observable schedules but before J_i in the actual schedule. Consequently, we cannot ignore such lower-priority jobs when trying to find the start time and the completion time of J_i. Let $\boldsymbol{N}$ denote the set of nonpreemptable jobs and $\boldsymbol{N}_i$ denote the subset of nonpreemptable jobs that have release times earlier than J_i and priorities lower than J_i. Let $\boldsymbol{P}_n^-$ be a schedule of $\boldsymbol{J}_n^-$ constructed according to the given scheduling algorithm, but assuming that all the jobs are preemptable and migratable. Let $\boldsymbol{B}_i$ denote the set of jobs in $\boldsymbol{N}_i$ which start before J_i in $\boldsymbol{P}_n^-$.

It has been shown [11] that any job J_l not in $\boldsymbol{B}_i$ cannot start before J_i in the actual schedule $\boldsymbol{A}_n$. Algorithm $\mathcal{INNF}$ makes use of this fact to eliminate some of the lower-priority jobs in $\boldsymbol{N}_i$ from consideration when trying to bound the worst-case completion times. It considers one job at a time, from the job with the highest priority to the job with the lowest priority. In order to find the worst-case completion time of J_i, Algorithm $\mathcal{INNF}$ transforms J_i and jobs in $\boldsymbol{J}_{i-1}$. In this transformation, every job is transformed into two jobs. Let G_k and H_k denote the two jobs transformed from J_k. Algorithm $\mathcal{INNF}$ has three steps:

(1) In Step 1 the parameters of the transformed jobs G_i and H_i are computed from those of J_i. G_i's

execution time is equal to the largest of the maximum execution times of jobs in $\boldsymbol{B}_i$, if $\boldsymbol{B}_i$ is nonempty and is equal to zero if $\boldsymbol{B}_i$ is empty. G_i's release time is release time r_i of J_i. H_i's execution time is equal to J_i's maximum execution time e_i^+ and H_i's release time h_i is r_i plus G_i's execution time. Let 0 be a priority that is higher than the priority of J_1. The priority of G_i is 0 and the priority of H_i is equal to that of J_i. G_i simulates the job that may block J_i, and H_i simulates J_i blocked by G_i. Step 1 also computes the parameters of G_k and H_k for each J_k for $k = 1, \cdots, i-1$. G_k has release time r_k. Its execution time is equal to the largest of the maximum execution times of jobs in $\boldsymbol{B}_k$ if $\boldsymbol{B}_k$ is nonempty and is equal to zero if $\boldsymbol{B}_k$ is empty. G_k's priority is equal to 0. H_k is a job with jittered release time $[h_k^-, h_k^+]$ where h_k^- is equal to r_k and h_k^+ is equal to r_k plus the execution time of G_k. H_k's execution time is equal to e_k^+ and its priority is equal to that of J_k.

(2) Step 2 uses Algorithm $\mathcal{IPMJ}$ presented earlier and transforms each job H_k into a job L_k with a fixed release time, for $k = 1, \cdots, i-1$. Let $\boldsymbol{G}_i$ and $\boldsymbol{L}_{i-1}$ denote the sets of jobs $\{G_1, \cdots, G_i\}$ and $\{L_1, \cdots, L_{i-1}\}$, respectively.

(3) In Step 3, $\boldsymbol{G}_i$, $\boldsymbol{L}_{i-1}$ and H_i are scheduled according to the given nonpreemptable, priority-driven algorithm. An upper bound of the completion time $F(J_i)$ of J_i is equal to the completion time of H_i in the resultant schedule.

$\boldsymbol{L}_{i-1}$ is not a subset of $\boldsymbol{L}_i$; $\boldsymbol{L}_i$ needs to be constructed for every job J_i. Consequently, the complexity of Algorithm $\mathcal{INNF}$ is $O(n^2)$. The following theorem allows us to conclude that if H_i can complete by the deadline d_i of J_i in the schedule generated by $\mathcal{INNF}$ algorithm, then J_i always completes by d_i.

Theorem 7 *The completion time $F(J_i)$ of J_i is no later than the completion time of the transformed job H_i in the schedule of $\boldsymbol{G}_i$, $\boldsymbol{L}_{i-1}$ and H_i generated by Algorithm $\mathcal{INNF}$.*

Because lower-priority preemptable jobs do not block any job, we can use Algorithm $\mathcal{INNF}$ with very little change to find the completion times of all jobs when some jobs are preemptable and migratable. Specifically, Step 3 of Algorithm $\mathcal{INNF}$ treats (1) all the jobs in $\boldsymbol{G}_i$ as nonpreemptable, (2) L_k in $\boldsymbol{L}_{i-1}$ as preemptable (nonpreemptable) if J_k is preemptable (nonpreemptable), and (3) H_i as preemptable (nonpreemptable) if J_i is preemptable (nonpreemptable).

Figure 3 illustrates how Algorithm $\mathcal{INNF}$ predicts the worst-case completion time of the job J_5. The schedules in parts (a), (b) and (c) are the maximal schedule, the actual schedule, and the schedule generated by Algorithm $\mathcal{INNF}$, respectively. In the actual schedule, J_4 blocks J_3, and J_3 delays J_5 since it has a higher priority than J_5. In other words, J_5's start time is delayed because J_4 blocks J_3. In the schedule generated by Algorithm $\mathcal{INNF}$, the blocking of J_3 by J_4 is accounted for by its blocking job G_3 which has release time r_3 and execution time e_4^+.

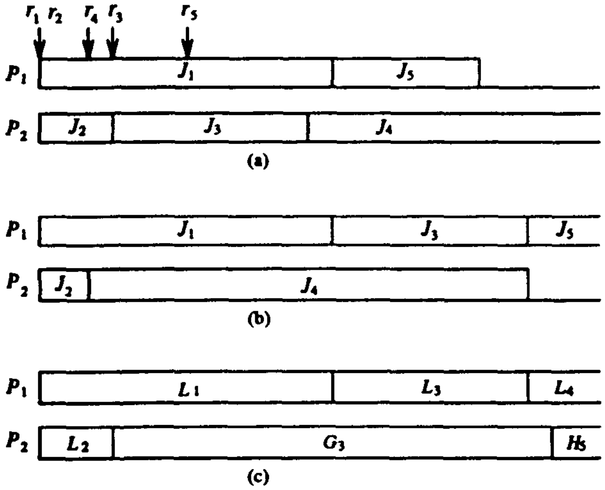


Figure 3: An example illustrating Algorithm $\mathcal{INNF}$

In the case where some jobs are nonpreemptable and some jobs are preemptable but not migratable, the actual start time and completion time of J_i may be postponed beyond $S^+(J_i)$ and $F^+(J_i)$ by two kinds of jobs. A nonpreemptable lower-priority job may block some jobs in the actual schedule but not in the maximal schedule. Some higher-priority jobs may preempt some jobs in the actual schedule but not in the maximal schedule.

When preemptable jobs are not migratable, a lower-priority job in $\boldsymbol{N}_i$ but not in $\boldsymbol{B}_i$ can start before J_i in the actual schedule. We must consider all the jobs in $\boldsymbol{N}_i$ when trying to bound the completion time of J_i. Also, the schedule of J_n^- constructed by assuming all the jobs are preemptable and nonmigratable gives us no information on which lower-priority jobs can actually start before J_i. Algorithm $\mathcal{INNF} - \mathcal{N}$ can be used to bound the completion times in this case. It consists of two steps. Step 1 considers the delays in the start time of each job J_i by nonpreemptable lower-priority jobs. It uses Algorithm $\mathcal{INNF}$ to construct a schedule for each J_i, using the set $\boldsymbol{N}_i$ instead of $\boldsymbol{B}_i$ in Step 1 of Algorithm $\mathcal{INNF}$. Let $F_*^+(J_i)$ denote the completion time of H_i in this schedule. Then it computes the delays in the completion time of J_i due to higher-priority jobs which may preempt J_i, or some job starting before J_i, in the actual schedule. Step 2 makes use of Theorem 8, stated below. This theorem is stated in terms of the set $\boldsymbol{E}_i$; $\boldsymbol{E}_i$ is a subset of $\boldsymbol{J}_i$ in which each job J_k is released after some preemptable job in $\boldsymbol{J}_i$ with a priority lower than itself (that is, J_k), and L_k (the job created in Step 2 of Algorithm $\mathcal{INNF}$) *is not scheduled on the same processor as* H_i to complete before H_i in the schedule constructed by Algorithm $\mathcal{INNF}$. The complexity of Algorithm $\mathcal{INNF} - \mathcal{N}$ is $O(n^2)$.

Theorem 8 $F(J_i) \leq F_*^+(J_i) + \sum_{J_k, L_k \in \boldsymbol{E}_i} e_k^+ - (e_i^+ - e_i)$.

5 Summary and Future Work

In this paper, we have discussed some of the challenges in validating timing constraints of dynamic multiprocessor and distributed systems. In recent years, many load-balancing and scheduling algorithm that dynamically dispatch and schedule jobs on available processors have been developed. These algorithms often achieve better response times and fuller resource utilization than the traditional algorithms that statically assign and bind jobs to processors. Unfortunately, analytical methods and efficient algorithms for validating that all jobs always complete by their deadlines in dynamic systems do not exist, and exhaustive simulation and testing are unreliable and expensive. Until reliable and efficient validation methods become available, the modern scheduling paradigms cannot be used in hard real-time systems.

We have presented here several worst-case upper bounds and efficient algorithms. They can predict reliably the worst-case completion times of independent jobs in homogeneous dynamic distributed systems. One of the algorithms allows us to take into account release times jitters. The others assume fixed arbitrary release times but take into account the effects of nonpreemptability and nonmigratability. These results constitute a small part of the theoretical basis needed for a comprehensive validation strategy that is capable of dealing with dynamic distributed real-time systems. Much of the work on this problem remains to be done. For example, we must be able to deal with dependencies between jobs. Ways to reliably predict the worst-case completion times of jobs that have precedence constraints and/or share resources are yet not available. This, as well as the work on predicting the completion times of jobs in heterogeneous systems is a part of our future work.

Acknowledgement

The authors wish to thank Drs. C. L. Liu and W. K. Shih for their comments and suggestions.

References

[1] J. Salasin and D. Waugh. An approach to analyzing non-functional aspects during system definition. Proceedings of July 1993, DSSA Meeting.

[2] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the Association for Computing Machinery*, 20(1):46–61, January 1973.

[3] J. Leung and J. Whitehead. On the complexity of fixed-priority scheduling of periodic, real-time tasks. *Performance Evaluation*, 2:237–250, 1982.

[4] B. Sprunt, L. Sha, and J. P. Lehoczky. Aperiodic task scheduling for hard-real-time systems. *The Journal of Real-Time Systems*, 1:27–60, 1989.

[5] J. P. Lehoczky, L. Sha, and Y. Ding. The rate monotone scheduling algorithm: Exact characterization and average case behavior. In *Proceedings of IEEE 10th Real-Time Systems Symposium*, pages 166–171, December 1989.

[6] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, September 1990.

[7] T. P. Baker. A stack-based allocation policy for real-time processes. In *Proceedings of IEEE 11th Real-Time Systems Symposium*, pages 191–200, December 1990.

[8] J. W. S. Liu, J. Redondo, Z. Deng, T. Tia, R. Bettati, A. Silberman, M. Storch, R. Ha, and W. Shih. PERTS: A prototyping environment for real-time systems. Technical Report UIUCDCS-R-93-1802, University of Illinois at Urbana-Champaign, 1993.

[9] R. L. Graham. Bounds on multiprocessing timing anomalies. *SIAM J. Appl. Math.*, 17(2):416–429, March 1969.

[10] R. Bettati. *End-to-End Scheduling to Meet Deadlines*. PhD thesis, University of Illinois at Urbana-Champaign, 1994.

[11] J. W. S. Liu and R. Ha. Theoretical foundations of efficient methods for validating real-time constraints. to appear in *Principles of Real-Time Systems*, edited by S. Son, Prentice Hall.

[12] R. Ha and J. W. S. Liu. Validating timing constraints in multiprocessor and distributed real-time systems. Technical Report UIUCDCS-R-93-1833, University of Illinois at Urbana-Champaign, 1993.

Massively Parallel Systems Design for Real-Time Embedded Applications

Thomas C. Choinski and Chin-Hwa Lee

Naval Undersea Warfare Center
Newport Division
New London, CT 06320

Naval Postgraduate School
Monterey, CA 93943

Abstract

This paper describes a generic approach to mitigate risk when reengineering for high throughput massively parallel systems. The approach entails baselining the existing system, capturing the functional requirements, estimating initial processing requirements through a high level analysis, benchmarking a subset of the functionality on a low throughput computer, and modeling the high throughput application to determine the detailed processing requirements for scaling.

1: Introduction

"Once the architecture begins to take shape, the sooner contextual constraints and sanity checks are :nade on assumptions and requirements, the better."

Eberhardt Rechtin, Systems Architecting: Creating & Building Complex Systems [1]

Commercial massively parallel processing (MPP) architectures offer a solution to TERAFLOP (one trillion operations per second) computing applications in the Navy. Computing density (TERAFLOP/cubic foot) and cost (dollars/TERAFLOP) have decreased in recent years; however, the challenge of real-time embedded processing requirements poses a high risk for complex systems. The high risk relates to: inefficient match of applications to architectures, low availability of high throughput architectures, the accuracy of forecasted downward spiraling price projections, and immature software development tools (e.g., parallelizing compilers).

This paper proposes a generic approach to mitigate the risk when investing in a specific MPP architecture. The approach proposes a series of intermediate steps to assess the compatibility of the architecture and the requirements. Each step refines the assessment and leads to the final tradeoff study. The approach embodies reengineering considerations when pursuing new implementations for cost or technology upgrades. Specifically, the approach entails:

1. defining the system requirements,
2. sizing an architecture using static benchmarks,
3. allocating resources using systems engineering tools,
4. developing a full scale model,
5. validating the full scale model with dynamic benchmarks,
6. assessing the compatibility of the architecture with the real-time embedded applications, and
7. selecting the appropriate design approach based on a trade-off analysis.

The tradeoff analysis incorporates an assortment of design tools to expedite and facilitate the decision making process. Examples of tools used to date include: VHSIC Hardware Description Language (VHDL), RDD-100 and OMTool. The approach will integrate the systems engineering tools developed under the direction of the Naval Surface Warfare Center (NSWC) within the Office of Naval Research's (ONR) Engineering of Complex Systems (ECS) Block Program as they become available.

The generic approach suits a myriad of applications ranging from radar to sonar systems. Accordingly, air, surface and subsurface platforms can benefit from the approach outlined in this paper. This paper uses the case study method to showcase the approach.

The case study method applies the generic approach to a practical application. This paper discusses one case study to demonstrate the utility of the approach. The research will explore additional case studies as interest arises.

This paper describes the case study, outlines the generic systems engineering design approach, presents high level architectural sizing techniques, and discusses detailed modeling and requirements allocation issues. The

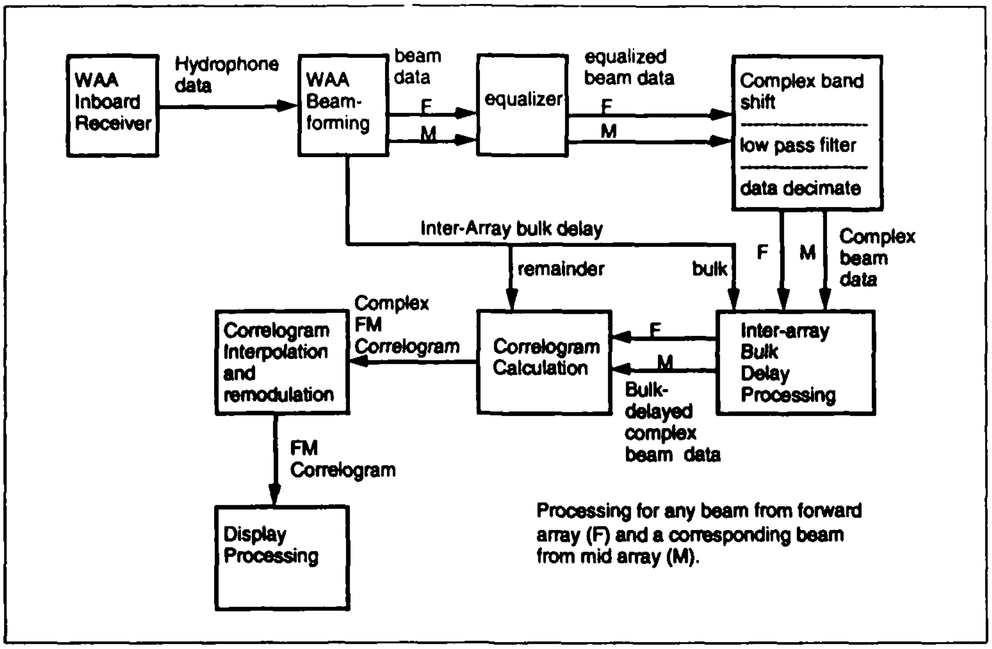


Figure 1. WAA Processing

Wide Aperture Array (WAA) beamforming problem serves as the initial case study for the application of the generic approach. The approach will evolve based on the comments, suggestions and progress of research performed in the ECS Program under the direction of the Naval Surface Warfare Center (NSWC).

2: Overview of WAA case study

NUWC chose the WAA in-board electronics application for two reasons. First, reengineering the WAA system contributes to the incremental insertion of commercial off-the-shelf equipment (COTS) into submarine warfare systems [2]. Second, a new implementation for the WAA inboard electronics suite would fulfill the Navy's need to reduce cost, as well as align with the Director of Defense Research and Development's (DDR&E) thrust areas. Two of DDRE's seven thrust areas emphasize *Affordability,* and *Sea Control and Undersea Superiority* [3]. Therefore, the reengineering of the existing WAA system responds to changing cost and commercial technology requirements.

This paper provides a brief description of the Wide Aperture Array full detection system. As interest arises, the approach will incorporate diversified case studies based on other existing Naval systems.

The cost effective implementation of the in-board electronics for a Wide Aperture Array full detection system serves as the primary objective for this case study. The WAA system can perform the detection function for a submarine. Sea test data indicates the Wide Aperture Array system can provide an attractive capability. In addition, current fiscal requirements within the Navy (i.e., the New Attack Submarine Program) have created the need to significantly reduce the cost of the existing WAA system.

The WAA full detection concept conceived by DePrimo and Choinski, and published in "An Efficient Approach to Systems Evolution (EASE)" [4], offers a cost effective way to implement the inboard electronics with additional full detection capability. This implementation serves as the case study for this paper.

The implementation uses commercial massively parallel processing technology developed within the High Performance Computing and Communications (HPCC) Program and made available under the Director of Defense Research and Development's (DDR&E) Defense Modernization Plan.

Figure 1 illustrates the WAA signal processing specified for the reengineering process. The signal processing includes additions to the existing WAA system.

This paper proposes a generic approach to mitigate the risk when investing in a massively parallel architecture. The approach concentrates on the importance of constraints and sanity checks throughout the design process.

3: System engineering approach

Figure 2 embodies the intent of Rechtin's heuristic, as quoted at the beginning of the paper, by setting up a process to assess the compatibility of the architecture and

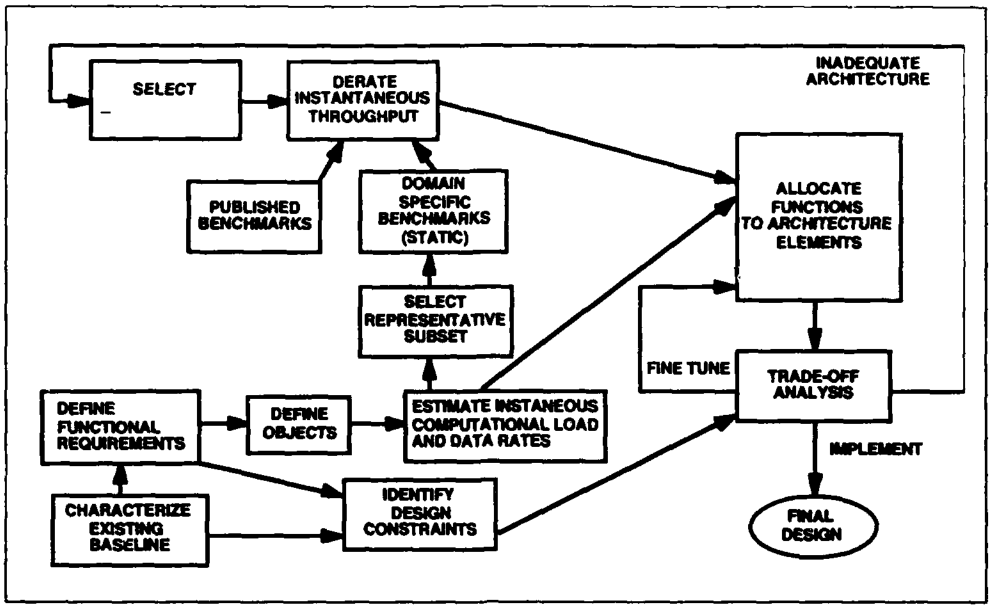

SELECT
DERATE INSTANTANEOUS THROUGHPUT
INADEQUATE ARCHITECTURE
PUBLISHED BENCHMARKS
DOMAIN SPECIFIC BENCHMARKS (STATIC)
ALLOCATE FUNCTIONS TO ARCHITECTURE ELEMENTS
SELECT REPRESENTATIVE SUBSET
FINE TUNE
TRADE-OFF ANALYSIS
DEFINE FUNCTIONAL REQUIREMENTS
DEFINE OBJECTS
ESTIMATE INSTANEOUS COMPUTATIONAL LOAD AND DATA RATES
IMPLEMENT
IDENTIFY DESIGN CONSTRAINTS
CHARACTERIZE EXISTING BASELINE
FINAL DESIGN

Figure 2. Systems Engineering Design Approach

the requirements throughout the design process. Each step progressively refines the assessment. The approach includes reengineering considerations when systems engineers pursue a new implementation for cost or technology upgrade reasons.

The process starts with the definition of functional requirements and the selection of a candidate architecture. For reengineering problems like the Wide Aperture Array, the process includes a step to characterize the existing system. The existing system characterization provides the baseline for the tradeoff analysis.

An object oriented software design follows the functional specification. The inclusion of the object oriented design step translates functional requirements to objects suitable for software design. This step will determine if object oriented design facilitates software portability and reuse. In practice, a systems designer could bypass this step in favor of functionally based software design.

The sustained throughput and data rate estimates follow the object oriented design. The throughput and data rate estimates enable a preliminary architectural sizing using the performance data from existing libraries or static benchmarks. Static benchmarks provide single processor performance data for metrics like efficiency. Therefore, the architecture sizing obtained at this point allows for an initial assessment of the instantaneous throughput levels quoted by manufacturers.

Given the preliminary architectural sizing, the systems designer can perform a detailed analysis of the architectural requirements for the given application. The detailed analysis consists of a combination of modeling, simulation and dynamic benchmarking.

Dynamic benchmarking entails the implementation of a processing subset on a scaled down MPP architecture. In this manner, dynamic benchmarking reduces risk. The move from using a single processor to multiple processors differentiates dynamic benchmarking from static benchmarking. Dynamic benchmarking also introduces partitioning, input/output (I/0) issues, and event driven processing attributes.

In addition, the dynamic benchmarks validate the detailed architecture models simulated in this step. The concept of using modeling, simulation and benchmarking for architecture validation was first introduced by Muñoz of the Naval Undersea Warfare Center [5]. Figure 3 elaborates on the allocation process identified in figure 2.

After allocating the functions, the final tradeoff analysis uses a set of previously defined metrics to compare the performance of the proposed implementation to the existing baseline system. The results of the tradeoff analysis determine whether to accept, modify or eliminate the candidate architecture.

3.1: Metrics

The basis of the tradeoff analysis rests with the extraction and comparison of metrics. Modeling and simulation permit measurement of the metrics for the proposed system. The measured data can be compared to the existing system baseline data.

Numerous metrics have been identified for consideration in the tradeoff analysis of MPP

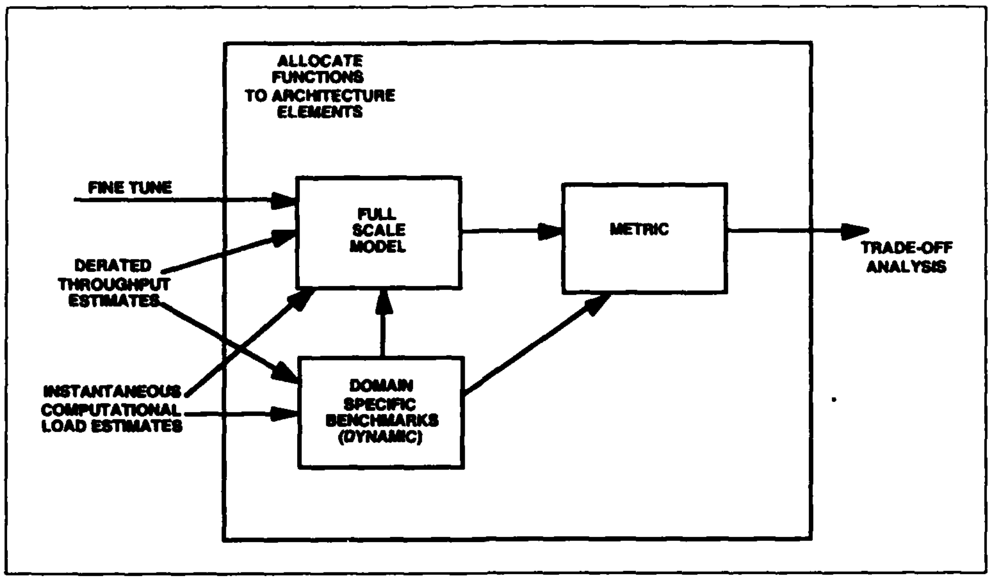

ALLOCATE FUNCTIONS TO ARCHITECTURE ELEMENTS
FINE TUNE
DERATED THROUGHPUT ESTIMATES
INSTANTANEOUS COMPUTATIONAL LOAD ESTIMATES
FULL SCALE MODEL
DOMAIN SPECIFIC BENCHMARKS (DYNAMIC)
METRIC
TRADE-OFF ANALYSIS

Figure 3. Functional Requirements Allocation

architectures. Table I presents the MPP metrics and their definitions. These metrics have been discussed in detail by Lee [6] and the team of Sweetman and Muñoz [7]. The design capture view metrics outlined by the ECS research block can also be added to this general list.

4: High level architectural sizing

The high level architectural sizing provides the preliminary estimate for the size and configuration of a compatible architecture. The high level architectural sizing consists of four parts:

1. capturing functional requirements,
2. baselining the existing system,
3. generating an object oriented design,
4. establishing a preliminary architectural sizing from static benchmarks.

Once completed, these four steps lead to a detailed architectural design and development. Unlike the detailed design, the high level architectural sizing does not address software issues or partitioning of the functions.

4.1: Functional requirements definition

The functional requirements definition phase of the generic system engineering design approach results in system level specifications for the application. For the case study highlighted in the paper, a systems engineer first documented the WAA functional requirements previously depicted in figure 1.

A systems engineering design tool like RDD-100 can capture the functional requirements and facilitate traceability throughout the design process. Initially, a word processor was used to capture the WAA requirements; however, NUWC will also use tools like RDD-100.

RDD-100 brings several capabilities to the design process including: requirements capture, functional behavior modeling, full scale architecture modeling, resource allocation, dynamic analysis and documentation of results. Other tools are also available to provide this capability.

4.2: Baseline system characterization

Ideally, the systems design engineer should baseline the existing system using metrics necessary to complete the tradeoff analysis. Under these conditions, the designer completes the tradeoff analysis by comparing the new and existing systems on equal footing.

Unfortunately, even the best documentation from a military system will fall short of supplying all the previously defined metrics for the tradeoff analysis. Design engineers document their work for development and not reengineering purposes. Therefore, the tradeoff analysis will embody comparisons between similar but not equivalent metrics.

The reengineering process was initiated by using the design capture views established by the ECS Block to capture the implementation of the existing WAA

Table I. Metrics

1.	**Computation Bandwidth**	A description of the frequency of operations per unit time measured in MFLOPS/second.
2.	**Communication Bandwidth**	A description of the I/O rate measured in MBytes/second.
3.	**Memory Bandwidth**	A measure of the memory access requirements per unit time represented by Bytes/second.
4.	**FLOPS-I/O Ratio**	A ratio which compares the computation load(MFLOPS) to the I/O (Bytes/sec) load.
5.	**Latency- FLOPS Product**	A characterization of the ability to support communications requirements versus the computational bandwidth requirements of a module or architectural element.
6.	**Power/Weight/Volume**	Values used to characterize the physical attributes of a system. Power is characterized by Watts, weight by pounds (lb) and volume by cubic feet (ft^3).
7.	**dB/Watts**	A measure which combines process gain (dB), algorithm efficiency, dB/gate-Hz, technology cost, gate - Hz/watts, architecture efficiency, and percent duty cycle. An alternative is to use noise recognition differential (NRD) instead of process gain for a measure of sonar system performance.
8.	**Architecture Diameter**	An integer which represents the maximum number of communication paths that a message or data may be required to travel from processor to processor.
9.	**Architecture Latency**	The maximum time, in seconds, a message takes to propagate across the path that determines architecture diameter.
10.	**Processor Memory Ratio**	A ratio that captures the memory available to an individual processor. For local memory systems the ratio would be the local memory per processor. For shared memory systems the ratio would be computed by dividing the total system memory by the number of processors and adding the amount of local cache memory per processor.
11.	**Average Message Size per Processor**	A value computed by dividing the total number of message bytes sent during the time it takes to execute an algorithm, divided by the number of processors.
12.	**Response Time**	The time in seconds that is required to execute an algorithm. The time begins when the first processor starts executing and ends when the last processor stops executing.
13.	**Processor Utilization**	A percentage computed by dividing the sum of the individual times that the processors are executing by the total time it takes to execute the algorithm, times the number of processors in the system. $= \frac{t1 + t2 + t3 + ... tn}{NT}$
14.	**Program Size**	The size in bytes of the program.
15.	**Speed Up**	A value computed by dividing the response time for an algorithm executing on a single processor by the response time for an algorithm executing on several nodes in a system.

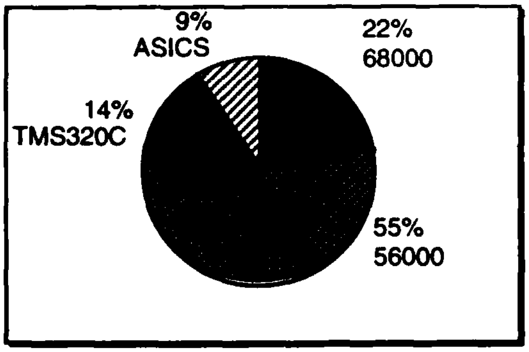


Figure 4. Processors

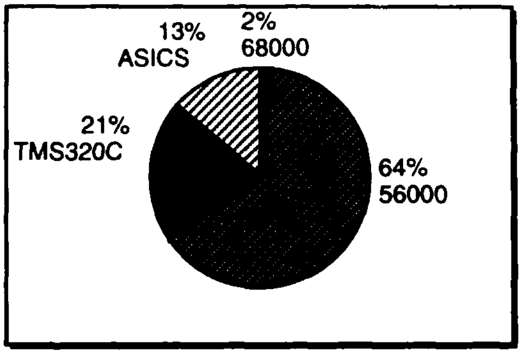


Figure 5. Throughput

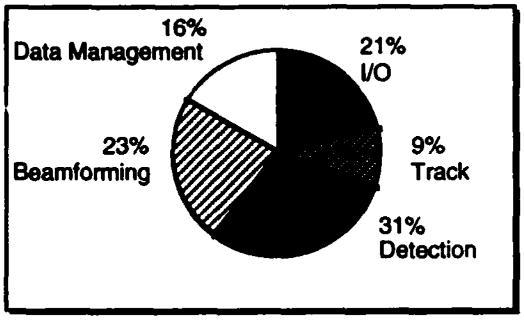


Figure 6. Processor/Function

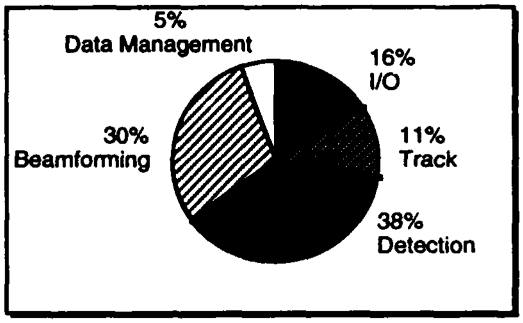


Figure 7. Throughput/Function

system. The information presented in figures 4-7 illustrates the types of data documented for the existing system. These figures represent a sample of the data used for the baseline characterization of the Wide Aperture Array System case study.

Although the existing baseline uses a distributed processing architecture, some of the experiences can be carried over to the massively parallel processor architecture. For example, since trackers do not require large amounts of throughput, the MPP implementation for trackers probably would not change significantly. Figures 4-7 present four different views of the Wide Aperture Array System. Partitioning functions to resources has become the focal point for the case study because of its significance in massively parallel array architectures.

4.3: Object oriented design

The object oriented software design follows the functional specification, and the existing system baseline. The inclusion of the object oriented step translates functional requirements to objects suitable for software design. Object oriented design should facilitate the reuse and portability of software.

Once the functional requirements have been designed, a software engineer determines the set of software objects necessary to achieve the desired functionality. An analysis takes place with the assistance of an object oriented design tool like OMTool. Future versions of products like OMTool will perform the functional to object oriented translation automatically; however, OMTool cannot perform the translation at this time.

OMTool provides functional, object and data flow views for a given application. In addition, the tool produces C++ code. This paper neither endorses nor denounces the use of OMTool. Engineers working on the WAA case study use OMTool because of the features available for the given price range.

Figure 8 illustrates the object oriented array system design. Figure 9 expands the object oriented beamformer design. These diagrams represent a synopsis of the object oriented design which will be used to reengineer the WAA system. Note that although the object oriented design started with the WAA application in mind, the high level software suits any array processing problem using a 2 stage time delay beamformer.

In the future a systems engineer specifying the functional level requirements could expedite the object oriented design if a link was developed between tools like OMTool and RDD-100. The link could further automate the design process. In addition, the link would also ensure the consistency of requirements between object oriented and system design tools.

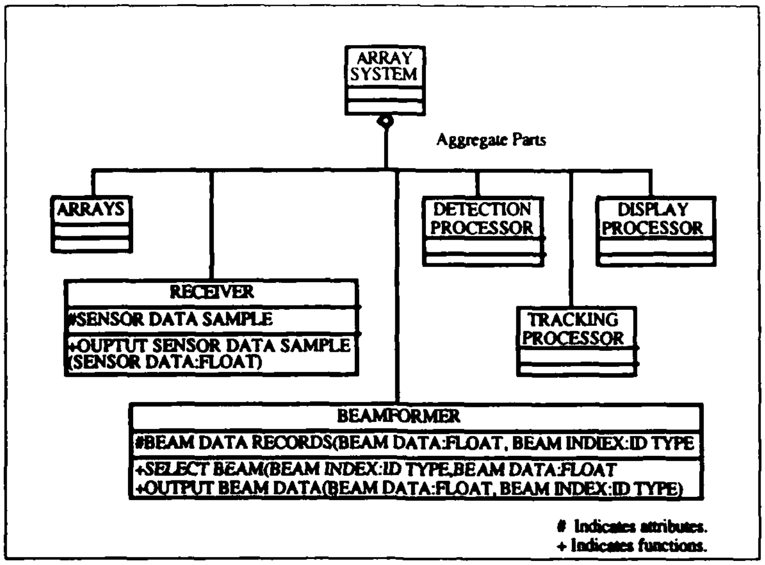

ARRAY SYSTEM
Aggregate Parts
ARRAYS
DETECTION PROCESSOR
DISPLAY PROCESSOR
RECEIVER
#SENSOR DATA SAMPLE
+OUPTUT SENSOR DATA SAMPLE (SENSOR DATA:FLOAT)
TRACKING PROCESSOR
BEAMFORMER
#BEAM DATA RECORDS(BEAM DATA:FLOAT, BEAM INDIEX:ID TYPE
+SELECT BEAM(BEAM INDEX:ID TYPE,BEAM DATA:FLOAT
+OUTPUT BEAM DATA(BEAM DATA:FLOAT, BEAM INDEX:ID TYPE)
# Indicates attributes.
+ Indicates functions.

Figure 8. Object Oriented Array System Design

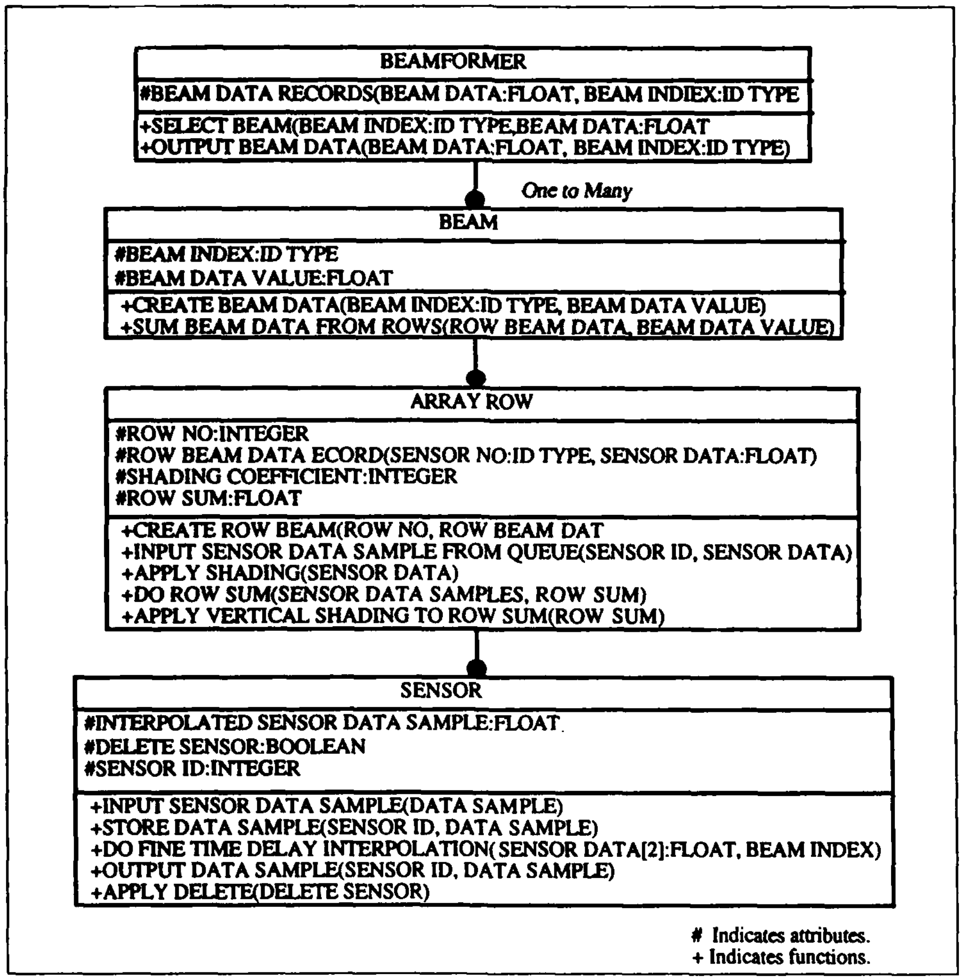

BEAMFORMER
#BEAM DATA RECORDS(BEAM DATA:FLOAT, BEAM INDIEX:ID TYPE
+SELECT BEAM(BEAM INDEX:ID TYPE,BEAM DATA:FLOAT
+OUTPUT BEAM DATA(BEAM DATA:FLOAT, BEAM INDEX:ID TYPE)
One to Many
BEAM
#BEAM INDEX:ID TYPE
#BEAM DATA VALUE:FLOAT
+CREATE BEAM DATA(BEAM INDEX:ID TYPE, BEAM DATA VALUE)
+SUM BEAM DATA FROM ROWS(ROW BEAM DATA, BEAM DATA VALUE)
ARRAY ROW
#ROW NO:INTEGER
#ROW BEAM DATA ECORD(SENSOR NO:ID TYPE, SENSOR DATA:FLOAT)
#SHADING COEFFICIENT:INTEGER
#ROW SUM:FLOAT
+CREATE ROW BEAM(ROW NO, ROW BEAM DAT
+INPUT SENSOR DATA SAMPLE FROM QUEUE(SENSOR ID, SENSOR DATA)
+APPLY SHADING(SENSOR DATA)
+DO ROW SUM(SENSOR DATA SAMPLES, ROW SUM)
+APPLY VERTICAL SHADING TO ROW SUM(ROW SUM)
SENSOR
#INTERPOLATED SENSOR DATA SAMPLE:FLOAT.
#DELETE SENSOR:BOOLEAN
#SENSOR ID:INTEGER
+INPUT SENSOR DATA SAMPLE(DATA SAMPLE)
+STORE DATA SAMPLE(SENSOR ID, DATA SAMPLE)
+DO FINE TIME DELAY INTERPOLATION(SENSOR DATA[2]:FLOAT, BEAM INDEX)
+OUTPUT DATA SAMPLE(SENSOR ID, DATA SAMPLE)
+APPLY DELETE(DELETE SENSOR)
# Indicates attributes.
+ Indicates functions.

Figure 9. Object Oriented Beamformer Design

4.4: Instantaneous Load Estimation and Static Benchmarking

A preliminary sizing for the WAA case study demonstrates the application of instantaneous load estimation and static benchmarking. The number of floating point multiply and addition operations were calculated for the functions identified in figure 1. The sustained throughput estimates in Table II reflect these multiply and addition estimates coupled with input data rates.

Intel Corporation provided the efficiency and peak numbers in Table II based on Paragon single processor implementations written in Fortran. The peak numbers do not reflect scaling effects due to I/O and partitioning. As a result, efficiencies for a final massively parallel version would probably be lower. Therefore, Table II presents the results of a static benchmarking effort and represents a preliminary sizing for the WAA processing problem. Basically, initial estimates indicate the WAA processing requires a massively parallel architecture capable of providing 36 GFLOPS of peak throughput.

Table II. Static Benchmarking Load Estimation [8]

Operation	Sustained	Efficiency	Peak
Beamforming	7.50	32%	23.44
FIR Filters	0.54	25%	2.16
Complex FFTs	2.02	56%	3.61
Cross PSD	0.23	14%	1.64
Auto PSD's	0.16	14%	1.14
Integrate Auto Spectra	0.01	14%	0.07
Inverse Complex FFT	1.01	56%	1.80
Normalized XCOR/ Up Sample and Interpolate FIRs	0.25	14%	1.79
Total GFLOPS	11.72		35.65

5: Detailed modeling and requirements allocation

The detailed level modeling and requirements allocation method provides a specific design for a MPP architecture. The method presented in this paper addresses four issues:

1. technology independence,
2. software partitioning,
3. full scale modeling, and
4. dynamic benchmarking.

Technology independence means that it is possible to retarget the software. Partitioning involves dividing the processing into pieces which can run on individual processing elements.

Massively parallel architectures can have an assorted collection of heterogeneous analog or digital processors. The program that runs the real-time embedded system typically can have hundreds of thousands of lines of source code. The system is generally very complex, difficult to design, and hard to maintain.

Large combat systems historically use a number of heterogeneous processors connected in a distributed network structure. Continuing this trend would lead to expansive custom MPP architectures.

Custom MPP architectures have thousands of processors connected as nodes in some kind of network structure. Commercial processors like the Intel i860, Sun Sparc, or DEC Alpha chip perform the processing functions in the nodes. Hypercube, mesh, hierarchical ring, or tree topologies form the basis of the networks.

Companies like Intel, Thinking Machines and Kendall Square Research have developed the MPP architectures into commercially available systems. These systems may have homogeneous or heterogeneous processing elements. The difference between commercial and custom MPP architectures lies in the user base. System engineers optimize custom architectures for one specific application for a limited market. Companies build commercial architectures as products for a more generic user community. The commercial architectures may not fit a particular application as well as a custom architecture; however, the commercial architecture will fit a broader range of applications. Table III characterizes custom and commercial architectures.

Table III. Custom Versus Commercial MPP Architectures

MPP Architecture	H/W Cost	S/W Cost	Portability
Custom	high	high	none
Commercial	medium	medium-high	partial

If MPP markets develop successfully, the hardware cost of commercial MPP architectures will shrink faster than custom MPP architectures. The software development costs of the custom architectures are

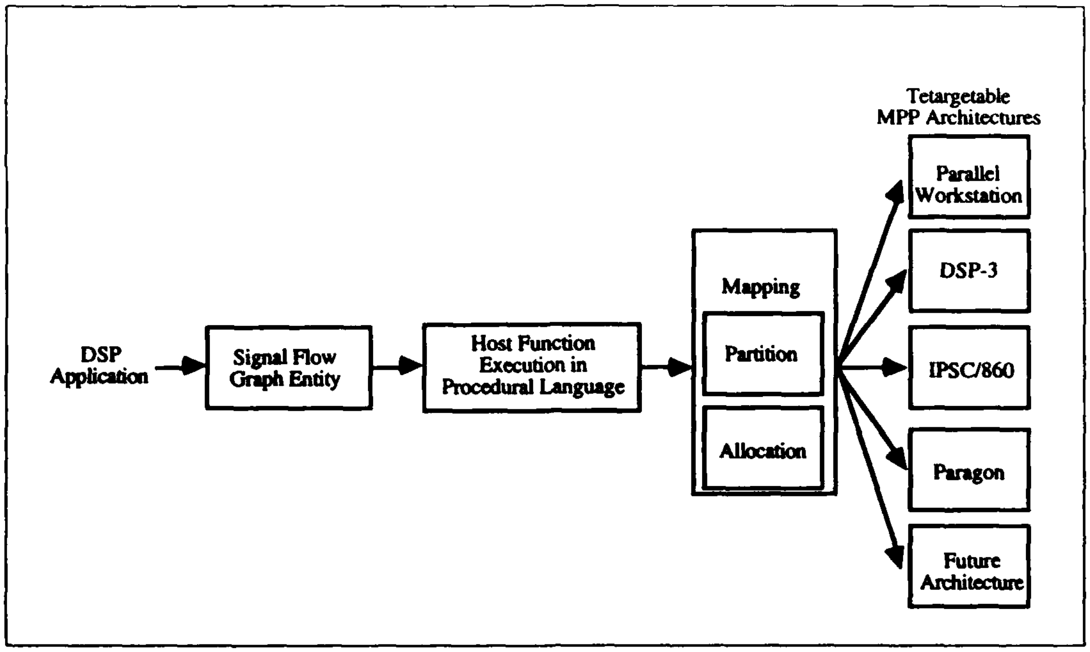


Figure 10. Technology Independent Application

prohibitively high. Life cycle costs for custom architectures are also high because of lack of portability. With appropriate research and development in the engineering of complex systems, the software for commercial MPP architectures can achieve lower cost through partial portability. One objective of the Massively Parallel System Design task is to address detailed level MPP software mapping and portability.

Despite continuing research efforts in parallel processing, two challenges exist for MPP architectures:

1. The MPP scalability problem presents a major obstacle. Efficiencies from benchmarks with large (thousands of processors) MPP architectures measure less than 10%. For vector processors like the Cray supercomputer, the efficiencies measure higher than 10%. These inefficiencies create a high incentive to increase the speed-up of MPP systems.

2. The MPP programming problem necessitates a significant up front development effort for partitioning. Software engineers cannot program MPP architectures easily. One dominating issue relates to the mapping process. The mapping process determines partitions and allocates functions on MPP architectures. The absence of automated mapping tools requires software engineers to manually complete the mapping process.

Scalability and partitioning are correlated. Good software partitioning methods generally lead to good scaling. Generally, the efficiency and scalability increase with effective software development techniques. Note however, that this relationship is not linear and is algorithm dependent.

5.1: Technology independence

Technology independence presents a significant hurdle to real-time embedded MPP architectures. Figure 10 shows one approach for attaining technology independence. In general, the objective and procedures are similar to other previous works. The uniqueness lies in the details of the methodology. The method concentrates on using commercially available tools whenever possible. Many of these tools have graphical user interfaces.

Graphical interfaces facilitate the use of signal flow graphs for representing real-time embedded applications. Node labels represent computation loads in the signal flow graph. Directed edges symbolize data dependency in the graph. Edge labels characterizes the communication delay of signals from node to node. Simple FIFOs between the nodes can represent communication delays for some target architectures.

This kind of programming method uses block diagrams, large grained data flow graphs, and synchronous data flow graphs. The graphics facilitates the entry of a digital signal processing (DSP) application. The task software may be written in any procedural language so that simulation of the function can be done on host processor before it is mapped into a target MPP system.

This method meets the scalability and portable software challenges. Software engineers can use one of three different techniques to program MPP architectures. The first one takes a regular sequential program and compiles it for a MPP system. This technique is referred to as the parallelizing compiler approach. The second recodes the program in a parallel language such as LINDA, FORTRAN 90, or functional (applicative) language. This technique is called parallel languages. The first technique does not require a large effort when rewriting software. A parallelizing compiler capable of dealing with thousands of lines of code simply does not exist, and the ones available for small programs suffer from performance problems. The second approach requires a new culture for programmers. However, using parallel language still falls short of acceptable performance.

The third approach follows the message passing methodology which involves explicit parallel environment control. Hence, the third technique is called message passing. The programming takes place in an environment like PVM or EXPRESS with utilities to handle parallel message passing. The last technique requires some user awareness of the topology of the MPP architecture, but it can achieve the highest scalability and efficiency. Table IV describes the software techniques.

Table IV. MPP Software Approaches

Technique	Efficiency	Mapping
Parallelizing Compiler	Not Proven	Automatic
Parallel Language	≤ 0.01%	Automatic
Message Passing	1% ~ 10%	Manual

Unfortunately, automatic mapping technology for partitioning and allocation does not exist. Good performance in programming MPP architectures relies on tedious manual mapping methods.

Single Instruction Multiple Data (SIMD) MPP and the Multiple Instruction Multiple Data (MIMD) MPP architectures further complicate the portability challenge. SIMD MPPs encompass the connection machine and the MASPAR architectures. MIMD MPPs consist of the iPSC 860, CM-5, DSP-3, and Paragon shown in Figure 10. This paper concentrates on MIMD architectures.

The four salient features for the portable massively parallel systems design (MPSD) method discussed in this paper include:

1. task level function module parallelization (coarse grain),
2. high level procedural language and messaging passing,
3. portable software for different MIMD MPPs, and
4. calibrated performance metrics for mapping.

After graphic entry and host function simulation a mapping procedure is required before the program can run on a MPP system. Figure 10 shows a set of MPP architectures. MIMD with distributed memory and message passing are our target systems. The Mapping Procedure consists of partition and allocation.

For MIMD with distributed memory and message passing, scheduling may be done at the compile time. Unlike real-time scheduling, compile time scheduling is the most straightforward way to handle the real-time requirement which dominates military applications.

5.2: Partitioning

The Calibrated mapping performance prediction paradigm (CMPP) leads to detailed modeling allocation. The CMPP paradigm is discussed in this section of the paper.

Because of a large and multidimensional solution space, heuristic methods provide the first pass solution. Therefore, automation and design aids would expedite a broader search of a good solution for the total system design.

Technology independence depends also on the MPP mapping procedure. Mapping involves partitioning and allocating function modules on the MPP architecture. The absence of parallelizing compilers and languages leaves only design aids to ease the mapping process. The user has to couple the procedural modules with message passing operations. The process is slow when the user has to do a manual mapping for all the pieces (thousands), as well as run the MPP execution to decide whether the mapping works. Figure 11 shows this mapping procedure in detail. This cycle will be repeated to optimize each performance metric.

This paper proposes a calibrated mapping performance prediction paradigm. Figure 12 illustrates the paradigm. The key idea concentrates on performance model simulation. Rather than do a functional execution on full scale MPP to collect dynamic performance metric data, the benchmark is collected from model simulation. The full scale model then provides estimates of the architectural performance in terms of the previously defined metrics.

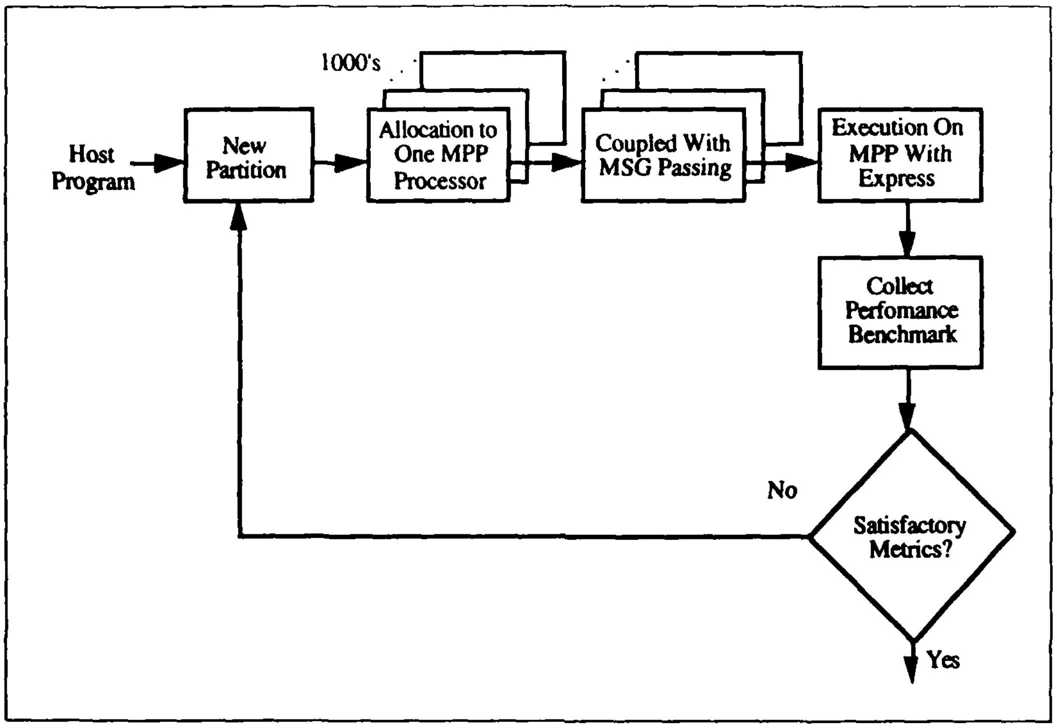

Host Program
New Partition
1000's
Allocation to One MPP Processor
Coupled With MSG Passing
Execution On MPP With Express
Collect Perfomance Benchmark
Satisfactory Metrics?
No
Yes

Figure 11. Detailed Mapping Procedure

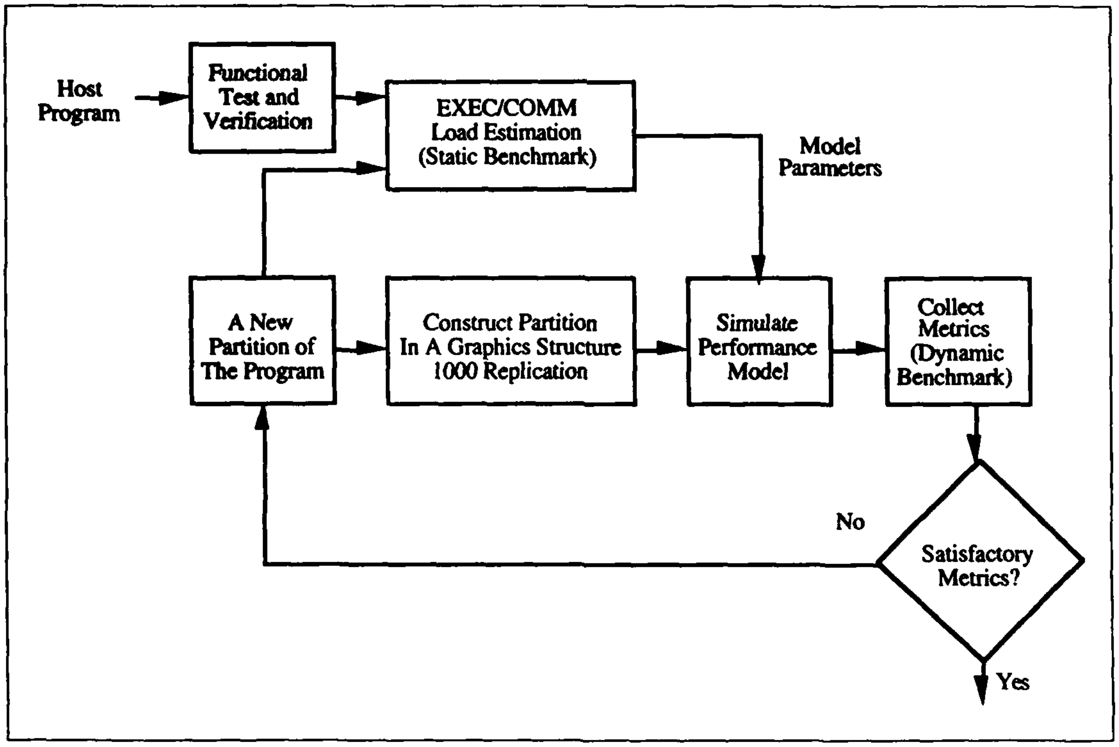

Host Program
Functional Test and Verification
EXEC/COMM Load Estimation (Static Benchmark)
Model Parameters
A New Partition of The Program
Construct Partition In A Graphics Structure 1000 Replication
Simulate Performance Model
Collect Metrics (Dynamic Benchmark)
Satisfactory Metrics?
No
Yes

Figure 12. Calibrated Mapping Performance Prediction Paradigm

5.3: Full scale modeling

The performance model requires two kinds of modules: the execution module (EXEC), and the communication module (COMM). The execution time metric as the initial focus, since execution time is directly related to the speed up in MPP architectures. EXEC load, EXEC bandwidth, COMM load, and COMM bandwidth characterize these modules. The host program estimates the EXEC load and COMM loads for all the partitioned pieces of a specific mapping. The collected data become model parameters to annotate the performance model before simulation. Each new partition requires repetition of the load estimation and extraction process. Any automation that can be added would be desirable.

EXEC modules and COMM modules are used to build the performance model with token networks. The token network handles multiple transmitters like real network situations. Presently the model can only handle Ethernet simulation. Construction of the performance model is done in the graphics mode. The VHDL feature simplifies the replication of thousands of identical modules. The Calibrated Mapping Performance Prediction (CMPP) paradigm hides many of the details of message passing so that the designer can concentrate on the partition and allocation problems. The right environment enables replication the modules many times. This environment reduces the problem of scaling to thousands of processors.

The CMPP paradigm discussed in this paper used the VHDL environment. Note that VHDL is not used here for hardware design; instead VHDL allows the designer to construct the structure, simulate the performance, and collect metric data. Both PC's and workstations support VHDL environments at low cost. VHDL will be available for hardware and system design for a long time. In addition, constructs of the VHDL language can replicate modules as shown in Figure 12 in a straight forward manner. VHDL *generic* constructs also help annotate model parameters before simulation. The manual EXEC/COMM load estimation and extraction is a disadvantage of the CMPP paradigm. An automatic procedure would strengthen the CMPP paradigm.

The CMPP paradigm allows the partition and allocation results to be portable to different types of MPPs. Remapping is necessary due to different network bandwidths, topologies and throughput rates in different MPPs. However the CMPP minimizes the portability effort as much as possible. The CMPP paradigm reduces the effort needed to run a real-time embedded application on different MPP architectures.

The FLOPS-I/O ratio characterizes the proportion of computation done versus communication (I/O) required in the partition. The ratio can characterize the architecture element once maximum throughput requirements are fulfilled. If the peak FLOPS-I/O ratio of an architecture element is less than that of the application module, it is possible to fit the application module into the element. If the peak FLOPS-I/O ratio of an element is greater than the application module, the partition will encounter problems.

Essentially, the FLOPS-I/O ratio characterizes computational activities relative to communication activities. With this metric, it will be easier to analyze the results of different mapping processes by examining granularity. One definition of fine grain tasks refers to small FLOPS-I/O ratios. Fine grain application modules can only be efficiently accommodated in fine grain architecture elements.

The FLOPS-I/O ratio metric makes it possible to find a common partition of an application for a set of MPPs. The common partition usually can not achieve the best speedup and efficiency in a specific MPP, but the partition can be accommodated in a number of MPPs. Further development of the CMPP paradigm will demonstrate this situation in the future.

A collection of Sparc workstations on an Ethernet was used to demonstrate the CMPP approach during 1993, since the researchers did not have access to a commercial MPP architecture. The researchers also used a message passing development environment called EXPRESS. EXPRESS addresses the portability challenge for the CMPP paradigm.

The development consists of three parts. First, the EXEC module characterizes a piece of the execution that occurred in the architecture element of the MPP. EXEC modules represent a source that generates a load token, a feed-through that accepts input tokens and produces output tokens, or a sink that consumes a load token. The following VHDL parameters characterize EXEC modules:

INST => unique module name
Unit => 1 Kbytes
Size_info => statistic size in units
Throughput_info => (#/sec) statistic throughput rate
Latency_info => statistic delay (usec.)

- * Duty_cycle_info=>(#/sec) statistic duty cycles
- * Only relevant for source EXEC modules.

Figure 13 shows a VHDL structure for the modules. The left most block depicts a source EXEC module, and the right most one a sink EXEC module. The INST generic describes a unique name for the module in the model. The size *unit characterizes the load of execution.

The term "unit" represents the basic data size such as in bytes or Kbytes. The throughput characterizes the speed of this EXEC module. The latency feature permits a more accurate delay account. Duty cycle is relevant if the EXEC module is a source that generates periodic loads.

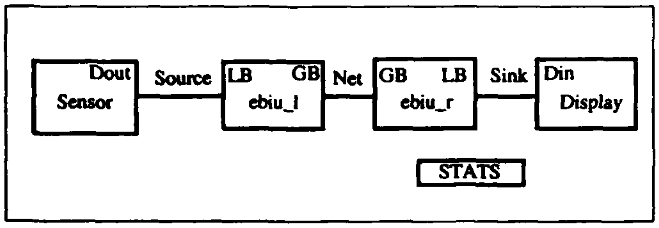


Figure 13. A Structure of EXEC Modules, COMM Modules, and Ethernet

Figure 13 shows two COMM modules called ebiu. The COMM module can receive or transmit to or from a local port. The data transfer on the global port is also bi-directional. The following VHDL generics characterize the COMM module:

INST => unique module name
bw_unit_per_sec => unit size (byte)
bandwidth_info => statistic bandwidth (byte/sec)
Tx_latency_info => statistic transmit latency delay (usec)
Rx_latency_info => statistic receive latency delay (usec)
Bus_timeout_info => statistic time-out (usec)
Ack_time_info => statistic acknowledgment time (usec)

Bandwidth and bw_unit_per_sec characterize the channel limitation of the bus. For the case of Ethernet, part of the Ethernet features reside in the COMM module, and the other features like arbitration reside in the token signal resolution function.

The VHDL resolution function is a special facility available in the VHDL language that handles multiple signal drivers. The signals in this model are all data token types. A special VHDL resolution function is implemented to model the Ethernet.

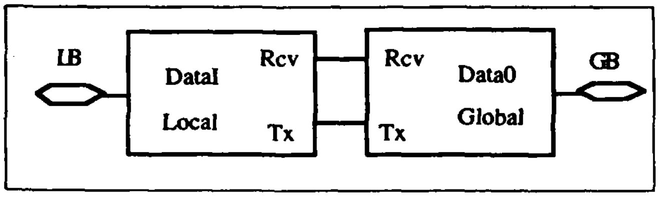


Figure 14. Ebiu Sub Module Structure

The ebiu is in turn built from two sub modules: Local and Globalnet. The sub module structure is shown in Figure 14. The VHDL environment can build these entities, module structures, and sub module structures hierarchically. Graphics windows permit editing, checking, and simulation. The bottom level behavior of the EXEC or COMM modules are written in VHDL.

5.4: Dynamic benchmarking

Dynamic benchmarking helps to validate the full scale models. The data from the dynamic benchmarks help refine the simulation models to reduce the risk associated with the scaling process. Due to the lack of availability of a commercial MPP architecture during 1993, the Naval Postgraduate School researchers explored the CMPP paradigm using Sun Workstations connected by Ethernet.

One important feature in the CMPP paradigm involves the calibration process for EXEC/COMM model parameters. The calibration process requires dynamic benchmarking for fine tuning. The EXEC/COMM parameters are extracted from a functional program. The results enable calibration of the model. The calibration process refers to the adjustment of parameters by comparing a benchmark from the CMPP prediction to the parameters from the actual execution. The calibration process ensures the validity of the model.

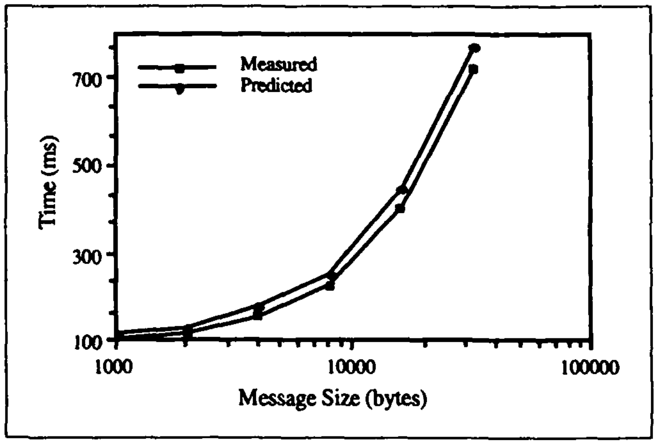


Figure 15. Ethernet Delay for Versus Message Size

The crucial step for the experiments developed during 1993 was to model and characterize the Ethernet correctly. The aforementioned calibration process tuned the COMM modules (ebiu). Figure 15 presents the actual message delays and the model predictions. The message size varied from 1 Kbyte to 32 Kbytes. The predicted and measured data matched very well. The model parameters that yielded this prediction consist of:

```
bw_unit_per_sec => byte
bandwith_info => 48,000
Tx_latency_info => 41.280 ms
Rx_latency_info => 10.000 ms
Ack_time_info => 41.280 ms
bus_timeout_info => 10 sec
```

In addition to the Ethernet modeling, two beamformers were coded and tested. One beamformer used a frequency domain algorithm, and the other a time domain algorithm. The time domain algorithm reflects the type used in the Wide Aperture array system.

The frequency domain beamformer demonstrated the advantage of using MPP systems. The hypothetical beamformer assumed 96 sensors in the system. Beam response covered 0 to 180 degrees with 1 degree resolution. A host program in FORTRAN 77 was written and checked with the test data to assure correctness. The mapping procedures outlined in this paper were used to partition and execute the application under the parallel EXPRESS environment. The metric plotted in figure 16 is the execution time. This mapping procedure was repeated for 1, 2, 4, 6, and 8 architecture elements on the network of workstations. The results show that increasing processing elements decreases the execution time.

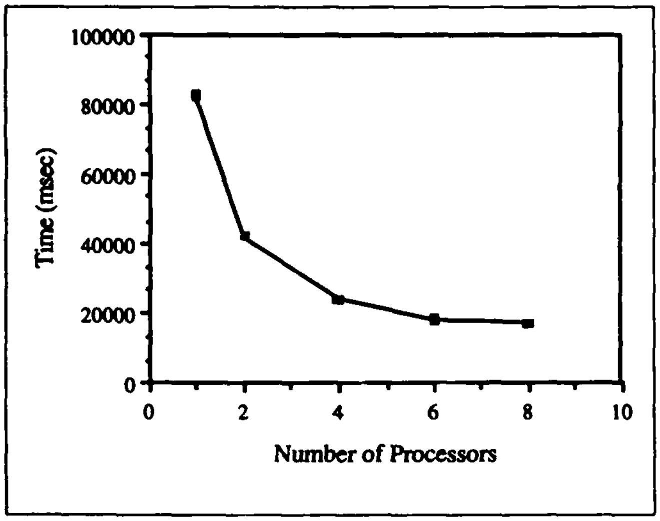


Figure 16. Execution Time Versus Number of Processors

Figure 17 exhibits the computational and communication loads for the frequency domain beamformer. The loads were estimated and extracted as described in the CMPP paradigm in Figure 12. These estimates represent the loads for each processor.

The two main execution modules are: the FFT module and the Vector-Matrix product module. The other modules are executed in the host. The diagram shows that the largest execution load occurred in the Vector-Matrix module. The heaviest traffic on the Ethernet was the message shuffle between the FFT and Vector-Matrix modules. The information in figure 17 was accumulated using the Sun operating tcov command. COMM loads were estimated using EXPRESS profilers.

The parallel EXPRESS environment can also provide an event profile which shows the communication activities, and the execution activities of the processors.

After the analysis, the next step is to construct a partition structure in the VHDL environment that simulates the performance. A structure for the 8-node partition was developed. The objective is to be able to predict performance such as execution time shown in figure 16. Progress is ongoing and encouraging.

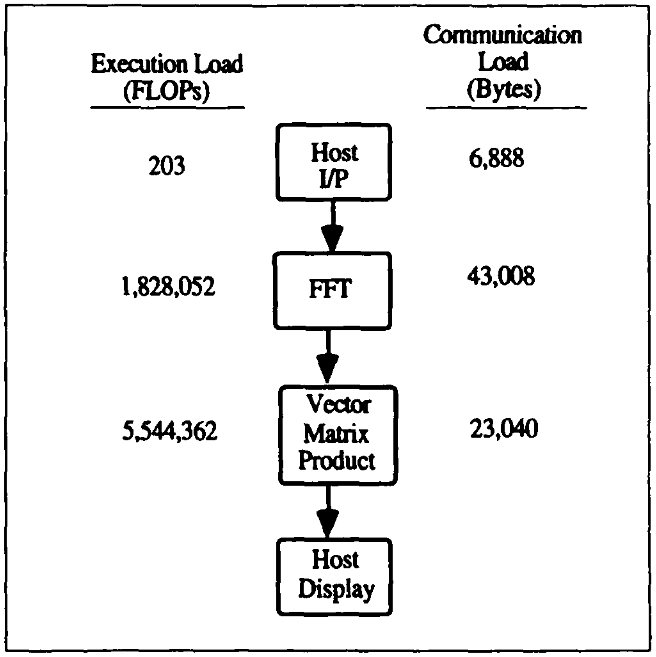


Figure 17. EXEC and COMM Loads for an 8-Processor Partition

A single panel of the WAA beamformer was also programmed during 1993. The WAA program includes test data generation, time delay memory, 1:3 interpolation, full beam vertical shading, and beam summation. The program has been tested and verified.

EXPRESS will be used to map the application to the Sun Workstation environment. Table V reveals preliminary execution time data for the WAA beamformer program on three high speed computers: the Sparc 630MP (2 processors), the Navy TAC-3 (HP 900/730), and the Cray YMP/EL. The Cray yielded the best execution time, but the TAC-3 yielded the smallest execution code size.

The TAC-3 is about 10 times slower than the Cray YMP/EL, but requires 25 times less code.

TABLE V. Time Domain Beamformer Benchmark

Architecture	Execution Time (sec)	Execution Code (Bytes)
Sparc 630MP (2 processors)	833.8	237,568
TAC-3 (HP9000/730)	339.6	32,768
Cray YMP/EL (4 processors)	35.4	802,320

6: Summary

This paper proposed a generic approach to mitigate the risk when investing in a specific MPP architecture. The approach proposes steps to assess the architecture and requirements compatibility which include: defining the system requirements, sizing an architecture using static benchmarks, allocating resources using systems engineering tools, developing a full scale model, validating the full scale model with dynamic benchmarks, assessing the compatibility of the architecture with the real-time embedded application, and selecting the appropriate design approach based on a trade-off analysis. The approach embodies reengineering considerations for cost or technology upgrades.

The Wide Aperture Array case study demonstrated the approach's usefulness. This paper presented a functional specification, existing system baseline, object oriented design and a series of benchmarks for the WAA application.

In addition to the WAA case study, the paper also included detailed modeling and simulation data for a frequency domain beamformer implemented on several Sun Workstations networked with Ethernet. This test demonstrated the utility of the Calibrated Mapping Performance Prediction Paradigm. The information presented included an explanation of the detailed VHDL models that a researcher fine tuned to reflect the actual operation of the network.

Several issues surfaced during the course of the research described in this paper. These issues offer possibilities for extended research in the future. The issues include:

1. compatibility between engineering tools (e.g., RD-100, OMTool, etc.),
2. portability of object oriented design software,
3. technology independent software for massively parallel architectures,
4. scalability of massively parallel architectures,
5. availability of commercial massively parallel architectures, and
6. suitability of the massively parallel systems design approach to other case studies.

Research will continue to pursue the massively parallel system design framework discussed. Plans encompass implementation, modeling, benchmarking and simulation of the Wide Aperture Array functions. A continued focus will be placed on using tools like VHDL, RD-100 and OMTool, in addition to integrating tools emerging from the Engineering of Complex Systems Block Program.

Acknowledgment

The authors acknowledge several individuals for their significant contributions to this paper: Daniel Organ for the characterization of the existing WAA system, Warren Axtell for the object oriented design, and Adam Siconolfi for the WAA functional requirements specification. In addition, the authors would like to thank Dr. José Muñoz for his helpful comments and suggestions, and John DePrimo for providing the technical review of the paper.

References

[1] Eberhardt Rechtin, Systems Architecting: Creating and Building Complex Systems, Prentice-Hall, 1991, p.48.

[2] Thomas C. Choinski, "Economical Development of Complex Computer Systems," Proceedings of the Complex Systems Engineering Synthesis and Assessment Technology Workshop, July 1992.

[3] Director of Defense Research and Engineering, Defense Science and Technology Strategy, July 1992.

[4] Thomas C. Choinski and John J. DePrimo, "An Efficient Approach to Systems Evolution," Proceedings of the Complex Systems Engineering Synthesis and Assessment Technology Workshop, July 1993.

[5] José L. Muñoz, et al., "An Architecture Assessment Environment for Massively Parallel Computations," Proceedings from the IEEE International Conference on Systems, Man, and Cybernetics, Volume II of III, November 14-17, 1989.

[6] Chin-Hwa Lee, "Massively Interconnected Models for a Beam Former," Proceedings of the 1992 Complex Systems Engineering Synthesis and Assessment Technology Workshop, July 1992.

[7] Denman E. Sweetman and José Muñoz, "Measures of Effectiveness (MOEs) for Parallel Architectures," Proceedings from the IEEE International Conference on Systems, Man, and Cybernetics, Volume II of III, November 14-17, 1989.

[8] Thomas C. Choinski and John J. DePrimo, "An Efficient Approach to Systems Evolution," Proceedings of the Complex Systems Engineering Synthesis and Assessment Technology Workshop, July 1993.

Knowledge-Based, Metalanguage-Based Object Abstraction for Automatic Program Transformation

Romel Rivera
Member, IEEE

Xinotech Research, Inc.
1313 Fifth Street Southeast, Suite 213
Minneapolis, MN 55414
romel@xinotech.com

Abstract

This paper describes Xinotech Research's knowledge based, metalanguage-based programming environment to support automatic program transformation and object abstraction for forward and reverse engineering. In this environment, both knowledge extraction and knowledge abstraction are metalanguage-based and thus language independent. The transformation engine is accessible through the interactive syntax-directed tools for program construction or for massive reengineering. This transformation infrastructure is operational for Ada, and can be applied to transform existing programs to support object-oriented methodologies, to port existing software to new libraries and platforms, to translate automatically between languages, to change the meaning of programs, or to enforce the semantics of applications or programming standards. It also supports specification and prototyping languages, and can be retargeted to other programming languages.

I. Introduction

"By now it is hard to imagine that any computer professional has not become aware of the bottleneck in software development. For both commercial and government applications, the annual bill for software is rising at a rapid pace. For example, the U.S. Department of Defense (DoD) spent over $3 billion on software in 1980 and their expenses are projected to grow to $30 billion per year by 1990 (DoD Annual report FY '81). Moreover, these costs are only the tip of the iceberg, as the impact of faulty software, delayed software, and continuing maintenance costs drive the real costs even higher" [12].

Ten years later, K.A. Banniuck confirms the above prediction. His study estimates that software expenditures in 1990 were over $185 billion worldwide with approximately $90 billion being spent in the U.S., and of that, $27 billion spent by the DoD. [2]

"We might well ask, why this phenomenal growth in the cost of software? There are several major reasons. One is the fact that the requirements for new software systems are more complex than ever before. A second reason for the rising cost of software is the increased demand for qualified software professionals. A third reason, is the fact that our software development tools and methodologies have not continued to dramatically improve our ability to develop software.

"...It has for a long time been recognized that one fundamental weakness of software creation is the fact that an entirely new software system is usually constructed 'from scratch'. This is clearly an unfortunate situation, as studies have shown that much of the code of one system is virtually identical to previously written code. For example, a study done at the Missile Systems Division of Raytheon Company observed that 40-60 percent of actual program code was repeated in more than one application [11]. Therefore the idea of reusability would seem to hold one answer to increasing software productivity. And yet the simple notion of reusability (i.e., code reusability) has been considered by computer professionals over the years but has never been entirely successful." [12]

Methodologies for reusability must be seamlessly integrated into the design, coding, testing and maintenance phases of the software cycle, which according to E. Horowitz [12], account for 87% of the total life-cycle effort.

Any methodology that can be proposed, could fail to be implemented if it is not supported by tools that synthesize software understanding and automate the transformation of programs so that reusability and modernization techniques can be applied automatically and on a large scale. The task of manual application of evolutionary transformations on large software systems would be found overwhelming and quickly discarded. The problem is that in order to create these tools, they need to be supported by an environment infrastructure with the following properties:

1. Structural and semantic knowledge of the program-

ming language (e.g. Ada).

2. Reusable language knowledge that:
   i. Supports quick fabrication of a multitude of language tools outside the realm of language translation, and
   ii. Allows customization of the created tools.
3. Formal Specifications. The production of language tools requires that the realization of structural and semantic language rules be available to the tool designer so that they can be applied to implement particular transformations, measurements and analyses, infer or complement other rules, etc. For example, a successful environment must provide the ability to specify new applications such as transformations to modularize source code, thus complementing the typical, hard-coded, predefined functions of object code generation. Language knowledge reusability is best supported by a system where languages can be defined with formal specifications, independent and separate from the application tools.
4. If forward and reverse engineering tasks are to be unified, it is essential that certain tools be interactive. This means that the formal mechanisms to manipulate language structures must be incremental. Traditionally, incrementality has been supported through domain-dependent, algorithmic approaches. For the sake of generality, it is desirable that incrementality be derived from the semantics of the metalanguage.

From these requirements, it is clear why language-based tools to automate the otherwise impossible task of manual transformation of source code have not proliferated and matured.

II. The Xinotech Environment

The environment is designed to be language-independent. Knowledge extraction (e.g. parsing, creating abstract syntax trees, and deriving semantic attributes) is expressed using a formal notation called XML, the Xinotech Meta-Language. Knowledge abstraction (the process of recognizing program patterns and transforming them into higher-level structures) is expressed through an XML component called XPAL, the Xinotech Pattern Abstraction Language. The system can thus be retargeted for other languages and applications at a fraction of the original cost.

XPAL is designed to express complex program patterns and to specify transformations of these patterns into more cohesive higher-level concepts. The alternative approach of using an intermediate "universal" language to which programs are first transformed, causes the unnecessary loss of the original model and still does not provide the means to tailor transformations. Only with a pattern language does the task of specifying a vast evolving library of patterns and their transformations become feasible, allowing pattern specification to become an application-oriented task.

XPAL makes use of a complete semantic notation and a comprehensive semantic library. Because XPAL is a component of XML, the extraction meta-language, XPAL has access to XSSL (XML's semantic notation) as well as all of the semantic equations written to properly define a particular programming language. For example, writing patterns that require the use of language scoping rules can be done by simply referring to the corresponding semantic equations.

XPAL transformations can also be used as the vehicle to formalize and document the implicit relations needed to abstract object oriented (OO) models from non-OO programs.

The environment is designed to support interactive software development, including syntax-directed construction, graphical abstraction, and standards and guidelines detection and enforcement. All these tools are built on top of the metalanguage engine. Pattern transformations are available interactively through these tools' user interfaces. Transformation libraries have been developed to support object orientation, conversion to Ada 9X from Ada 83, and translation to Ada from CMS-2 and Jovial.

III. XML, the Xinotech Meta-Language

The language-based, language-independent infrastructure of the Xinotech environment is provided by the implementation of XML. XML is a highly-readable language for specifying the abstract grammar, external syntax (views) and semantics of languages. XML is an *environment meta-language*, because it supports the design, implementation, embedding, revision and evolution of the various languages used in a software development environment, such as specification, documentation, design, programming, testing, and configuration languages. XML provides support for quick language prototyping, reusable language descriptions through module decomposition and inheritance, inter- and intra-language transformations, and separation of embedded and annotation languages. It provides an open architecture for integration to other traditional semantic analysis tools such as STARS ASIS for Ada.

XML supports *modules* for the hierarchical decomposition of languages. Modules are collections of related symbols. Modularization allows the language designer to logically divide the specification to enhance its readability. A language specification can import modules from other XML specifications. This encourages reusability when prototyping new languages.

A *construct* is defined in terms of its intrinsic language properties, such as abstract grammar, views (unparsing syntax) and semantics. Other clauses describe details for the environment, such as menus, placeholders, etc.

XSSL, the Xinotech Semantic Specification Language, is a component of XML. XSSL is a general notation: it supports, e.g., the expression of Ada scoping rules, type checking, data flow relations, and language translation. XSSL supports structured types and generalized lists, and it incorporates efficient abbreviation schemes to reduce the complexity of expressions due to explicit semantic flow. It uses object-oriented encapsulation to achieve the reuse of semantic structures throughout multiple constructs. XSSL supports incremental evaluation as well as the semantics of inter-compilation-unit relations.

IV. XPAL and Pattern Abstraction

Pattern abstraction is the transformation process of automatically condensing or abstracting low-level source code patterns found in existing software into high-level program concepts. XPAL, the Xinotech Pattern Abstraction Language, a declarative, constraint language, is the vehicle to express these program patterns and their transformations. Since XPAL is a component of XML, the Xinotech Meta-Language, these transformations can be written for any language specified with XML. Therefore, the entire mechanism is language independent.

Pattern abstraction is valuable because it recognizes implied or concealed relationships in low-level source code and, by representing them with existing higher-level structures, makes the relationships explicit and conceptual, and the code more cohesive and less fragmented. This reduces the complexity of the representation while increasing the expressive power of the resulting programs, thus enhancing its maintainability, understandability and reusability. This process is the inverse of top-down synthesis, such as program compilation.

In XPAL, patterns can be specified in terms of other patterns. Because XSSL, a component of XML, is a general notation for expressing the semantics of languages, patterns can use or complement these semantic equations. The approach traditionally taken in designing reengineering environments is that of providing some semantic capabilities through a limited set of hard-coded functions. In the XML family, graph operations, such as transitive closures for data and control flow, can be specified on the relationships characterized by XSSL equations. A language this comprehensive makes pattern abstraction very powerful.

Advantages of having XPAL as a component of XML. Because the XPAL notation is embedded within XML, it has the advantages of full access to the abstract grammar and semantics of the programming language, access to a general semantic notation, the use of XML extraction mechanisms, such as parsing views, to express tree patterns textually, and the use of multiple views which allow syntactical transformations to be expressed in the syntax of the programming language.

V. The Xinotech Program Composer

The Composer is the central application tool built on top of the XML language infrastructure. It is a syntax-driven, interactive semantic tool for tne design and construction of programs. Programs are managed as abstract syntax trees (AST), with multiple textual representations or views. An incremental parser and an incremental unparser provide the mappings between the textual and the AST representations. One of the main areas of concern during the design of the Composer was the functionality and behavior of its incremental bottom-up parser. This parser was designed to support a smooth left-to-right insertion while providing full interactive language support such as automatic template generation, placeholders, menus, and formatting while typing. The user can select levels of template generation during insertion. Templates are non-intrusive, since the user can type over to skip optional clauses. Text files not created with the Composer are automatically imported the first time they are opened.

Views can be used to create multiple formatting schemes, or to combine or isolate programs with embedded documentation and/or PDL structures. The Composer supports browsing through libraries, and provides program outlines from any point in the program.

VI. The Graph Abstractor

The Graph Abstractor is an analysis and maintenance tool designed to display XSSL-generated semantic relations. These relations can be displayed graphically or structurally. The Graph Abstractor is designed to minimize the size complexity of graphs and isolate the relations of current relevance.

VII. The Guideliner

The Xinotech Guideliner is an interactive program analyzer. It verifies adherence to programming guidelines, standards and metrics, and transforms programs automatically to comply with these guidelines. These guidelines are written using XPAL. The design goals of the Guideliner were as follows:

1. To provide an integrated, incremental capability to prevent and/or detect and flag user-defined guideline deviations during interactive program construction with the Composer.
2. To provide batch processing to obtain detailed and statistical reports regarding non-compliance with user-defined guidelines and standards. This can be useful during the quality assurance phase of code acceptance from contractors.
3. To provide the automatic translation of source code to comply with user-defined guidelines and standards. This process can be applied to any source code, regardless of whether it was created with the Composer.
4. To provide a wide range of metrics measurements that can be requested by the user as part of the guidelines and standards to be analyzed.

VIII. Reengineering Applications

XPAL is a general language for program recognition and transformation. It can be used to:

1. Translate programs from one language to another, such as CMS-2 or Jovial to Ada.
2. Detect and correct violations of user-defined guidelines and standards.
3. Transform existing non-OO programs into object-oriented programs.
4. Port existing programs from one supporting library to another. This helps automate migration to newer standard libraries, or to different operating systems and hardware platforms. As new libraries are created, existing applications can be searched for potential matches, so that the application can be modernized and expressed in terms of the new reusable components.
5. Modify the meaning of programs. Transformations can be written to modify existing programs so that they perform new functions, thus helping create new applications from existing ones.
6. Apply isolated transformations interactively. XPAL libraries can be created to generate bodies out of package specifications, to split packages or procedures, improve module decomposition, etc.

8.1 Language translation

Typically, language conversion is an abstraction process, very much the opposite of top-down synthesis or compilation. This is the case whenever the target language is a higher-level language, as in the case of translating CMS-2, Jovial or FORTRAN to Ada. Compilation technologies do not lend themselves well to this process, and pattern abstraction is highly desirable so that low-level, implicit, global relationships can be identified and abstracted into explicit higher-level constructs. XPAL was designed to support such abstraction. These are some examples of XPAL applications when converting CMS-2 to Ada:

1. Patterns can be defined to map different operating-system dependent multi-tasking models in CMS-2 to the construct-based tasking model in Ada. These transformations can be done very effectively since they are a classic example of implied relationships made explicit by the abstractor. Patterns can be written for the following:
   - i. Building the task structures out of CMS-2 modules and entry point tables.
   - ii. Building the "Message_Center" task out of the specification of the message broadcasting table for the linker.
   - iii. Abstracting critical regions by localizing and encapsulating the shared data into tasks, from the fragmented test-and-set protected access found in CMS-2. Such abstraction supports code migration towards an object-oriented methodology.
   - iv. Customizing patterns to support the direct translation of CMS-2 library procedures for some of these functions (e.g. critical regions), if they exist.
2. Abstracting block structure such as *for*, *while* and exit-based closed loops from goto-based control flow.
3. Creating procedures to modularize code or to eliminate unstructured loops, and creating enumeration types from sets of constants and related variables.

These are some of the advantages of XPAL-based translation:

1. *Fully Customizable*. This is a requirement for the case of CMS-2 or Jovial to Ada, since the translation will depend on the dialect, the executive in use, and library and other environment dependencies, as well as on the customization of the translated code to Ada guidelines such as the STARS Ada Reusability Guidelines.
2. *Fully reusable during subsequent system evolution*. Components developed for translation, since they are language-independent, can be used interactively during continuing Ada development (as Ada-to-Ada reengineering tools).
3. *Powerful dual translation and development environments*. Part of the success of the reverse-engineering process (i.e. translation) depends on how well it is integrated with the forward-engineering process (i.e. development). Such integration dictates the success of the translation system for interactive use.
4. *High-quality of the resulting code*. By devising sophisticated schemes for code abstraction, the transla-

tor designer can make more comprehensive use of the features of the target language (e.g. Ada). This results in more condensed and readable code. By not discarding the original implementation through a very-high-level intermediate language, this approach is able to maintain comparable efficiency levels.

5. *Predictability.* The Xinotech approach, using external specifications for the translation, allows the user to verify and approve *in anticipation*, the ways in which source language structures have been chosen to be translated. In a system where the implementation was discarded, the efficiency of the resulting code would be completely unpredictable.
6. *Life Cycle Orientation.* The XPAL-based approach takes into account the fact that the translated system will continue to evolve, so tailored patterns can prepare it for further growth, by supporting 2167A documentation generation and traceability with the PDL of choice, extraction of high-level graphs, and compliance with user-defined standards.
7. *Formally Specified Translation.* Another advantage of using formal specifications is that they provide a highly modular and functional decomposition of the translation system, resulting in an accessible mechanism for verifying the translator's reliability.
8. *Low-risk development path.* This is the result of two factors: predictability, and the fact that this technology is implemented progressively, with practical appreciable benefits available from day one. These benefits continue to grow in proportion to the resources invested in the project. Its success can be measured and monitored throughout the development effort.

8.2 Support for Ada 9X compliance and Ada 9X philosophy

The Xinotech transformation environment includes a set of Ada 9X transformation libraries to support Ada 9X compliance as well as Ada 9X philosophy. In turn, these libraries are managed by the Guideliner's user interface.

Support for Ada 9X Compliance. The environment provides a library of transformations to automatically translate the 9X violations in existing Ada 83 sources to the Ada 9X standard. These transformations can be applied interactively or in batch mode: the result is compilable Ada 9X code. This library is used to translate to 9X for compliance, even though the resulting code may not be object oriented (OO) or otherwise embody Ada 9X philosophy in any way.

Support for Ada 9X Philosophy. An additional library transforms 9X-compliant programs into a model supporting OO and 9X philosophy. The OO Ada 9X programs resulting from these transformations take advantage of 9X-specific features for modularization, object-orientation, parallelism and synchronization. Examples:

1. Transforming a package into a hierarchy with children packages. This supports improved modularization by allowing the direct sharing of declarations among a closely-related family of packages.
2. Transforming Ada structures to support explicit Ada 9X vectorization. A few of these cases can be detected automatically. Conversely, the user is able to invoke these transformations interactively.
3. Transforming a synchronization model into one with explicit protected records. In some cases, the old synchronization model can be derived from the usage of a particular library.
4. Transforming record types with variants to tagged types with extensions. This transformation is requested by the user for a particular record type with variants. The particular record type is analyzed to determine if the transformation is possible, and if so, the transformation is performed. This transformations takes advantage of multiple dispatching to enhance the readability, object-orientation, and rcusability of the code. The simplest such case involves a record with a single variant whose discriminant is a value of an enumeration type.

8.3 Real-time prototyping environments

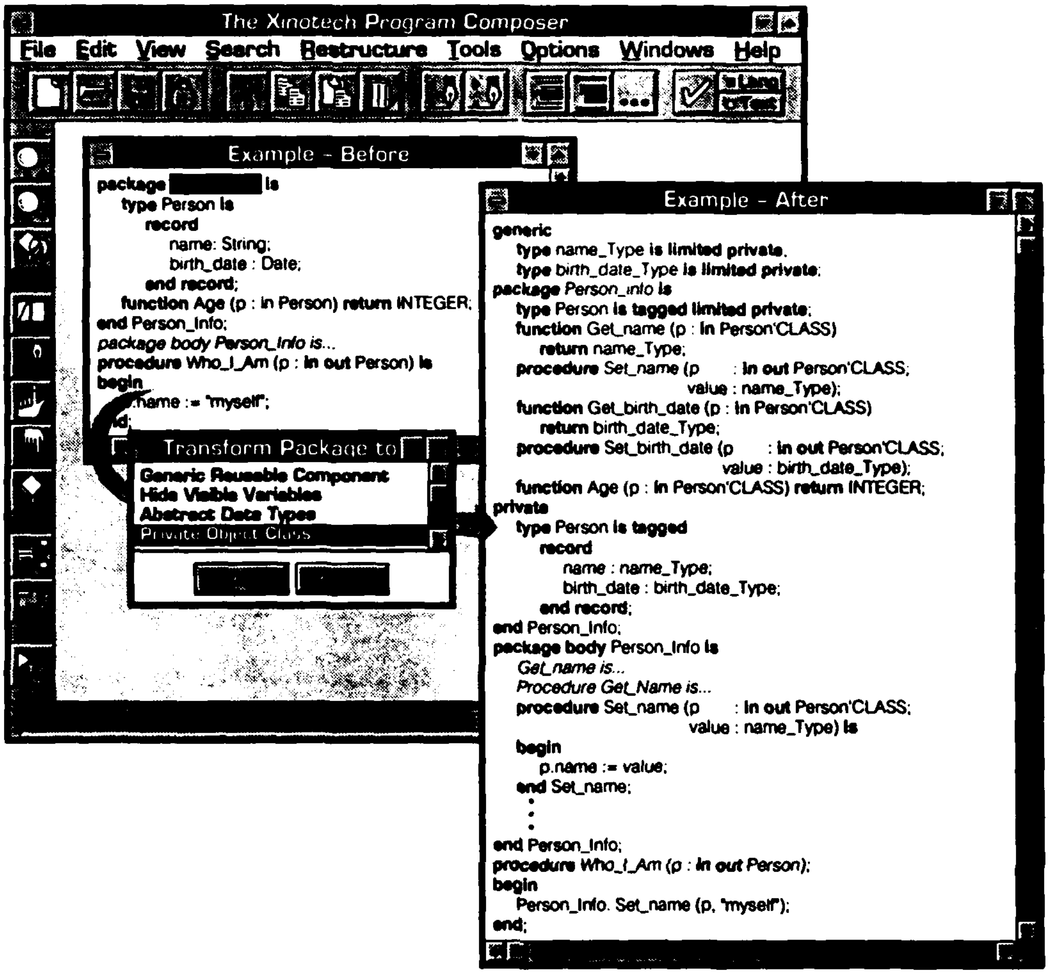
XPAL can be applied to support specification or prototyping languages such as Luqi's PSDL. [19], [22] Besides providing an integrated, interactive front-end for PSDL, XPAL can be used to verify adherence to design methodologies, to synchronize graphical with structured editing, and to map between specification and implementation languages.

8.4 Object abstraction

Object abstraction is the process of recognizing relationships in existing, non object-oriented (OO) Ada programs, and transforming these programs into a higher-level, object-oriented architecture with reusable components.

OO design methodologies have been in use for some time, and are very useful in helping to understand the behavior of systems and relationships between components (objects). It seems natural that obtaining an object-oriented design view of existing non-OO source programs through reverse-engineering will:

1. Help us understand the intended behavior of a system and its relationships.
2. Allow us to capture this OO design in an OO design language that can be manipulated textually or graphically by design tools, thus making it possible to use forward engineering (FE) technology to analyze, modify and browse through the design.
3. Allow us to restructure or redesign the existing code so


The Xinotech Program Composer
File Edit View Search Restructure Tools Options Windows Help
Example - Before
Transform Package to
Generic Reusable Component
Hide Visible Variables
Abstract Data Types
Private Object Class
Example - After

```
package [redacted] is
   type Person is
      record
         name: String;
         birth_date : Date;
      end record;
   function Age (p : in Person) return INTEGER;
end Person_Info;
package body Person_Info is...
procedure Who_I_Am (p : in out Person) is
begin
   .name := "myself";
 d;
```

```
generic
   type name_Type is limited private.
   type birth_date_Type is limited private;
package Person_info is
   type Person is tagged limited private;
   function Get_name (p : in Person'CLASS)
      return name_Type;
   procedure Set_name (p      : in out Person'CLASS;
                       value : name_Type);
   function Get_birth_date (p : in Person'CLASS)
      return birth_date_Type;
   procedure Set_birth_date (p      : in out Person'CLASS;
                             value : birth_date_Type);
   function Age (p : in Person'CLASS) return INTEGER;
private
   type Person is tagged
      record
         name : name_Type;
         birth_date : birth_date_Type;
      end record;
end Person_Info;
package body Person_Info is
   Get_name is...
   Procedure Get_Name is...
   procedure Set_name (p      : in out Person'CLASS;
                       value : name_Type) is
   begin
      p.name := value;
   end Set_name;
    •
    •
    •
end Person_Info;
procedure Who_I_Am (p : in out Person);
begin
   Person_Info. Set_name (p, "myself");
end;
```

Fig. 1. The package in the left window is automatically transformed into a private object class. Transformation rules (not shown) are specified in XPAL, the Xinotech Pattern Abstraction Language.

that it conforms to the OO recaptured design.

Examples of XPAL for object abstraction:

1. Transforming exported data objects into abstract data types. Data objects will be hidden, and made available only through access methods (procedures). This includes the automatic creation of initialization and finalization methods for the data types.
2. Transforming program units into reusable blueprints (e.g. generic units in Ada).
3. Transforming sets of variables into object classes by hiding them in structured types with access methods.
4. Transforming variant record types into a base class with subclasses (e.g. Ada 9X tagged types with extensions). These transformations will take advantage of multiple dispatching to enhance readability, object-orientation and reusability.

IX. Benefits

9.1 Benefits for Ada 9X

This environment represents a rather extensive solution for Ada reengineering, because it automates the evolutionary migration, from the legacy systems written in the proprietary languages of the sixties, towards the full, object-based, design philosophy of Ada 9X. For example, it can be used to:

1. Translate CMS-2 or Jovial programs into Ada.
2. Translate Ada 83 programs into Ada 9X.
3. Support the object-orientation of existing Ada code, according to the philosophy of the new Ada 9X features, thus enhancing reusability.
4. Automate the porting of existing Ada applications to new Ada 9X standard libraries, thus enhancing the interchangeability of the application components.

5. Automate transformations to change the meaning of existing programs, thus supporting the adaptation of existing programs to new applications.

9.2 General benefits

Support for All Languages in the Life Cycle. Pattern abstraction can be applied to all the languages in the software life cycle, from specification languages, to OO design languages, to annotation languages, programming languages, etc. XPAL may be used to automate top-down translation during program development, or to abstract design and specifications during reverse engineering.

Interactive Transformation Environment. Transformations can also be applied interactively during program construction. Forward and reverse engineering are thus integrated in a single homogeneous environment.

Support for Multiple Programming Languages. Through XML, the same homogeneous language-based environment is available for many programming languages. This is particularly attractive for translation between dialects. The Xinotech environment can also be instantiated (very cost effectively), for specialized languages, such as VHDL and database languages.

Open Architectures. The existing Xinotech environment supports the client-server model of an open heterogeneous architecture with a graphical user interface.

An Integrated Environment. Xinotech's approach was to create an integrated semantic environment for syntax-directed program construction, as well as for analysis and transformation. Forward and reverse engineering are indistinguishable. Vast transformation libraries can be expressed and customized with a metalanguage for pattern abstraction.

X. Bibliography

[1] Ada 9X Project. Ada 9X Requirements. Office of the Under Secretary of Defense for Acquisition, Washington, D.C., December 1990.

[2] K.A. Bannick. Breakdown of Software Expenditures in the Department of Defense, United States and in the World. Master's Thesis, Naval Postgraduate School, Monterey, CA, Sept. 1991.

[3] B. Barding, C. Thompson. Composable Ada Software Components and the Re-Export Paradigm —Parts 1 and 2. ACM SIGAda Letters VIII (1); pp. 58-79, 1988.

[4] Boyle, J.M., Muralidaran, M.N. Program Reusability Through Program Transformation. IEEE Transactions on Software Engineering, vol. SE-10, no. 5, September 1984

[5] C.L. Braun, J.B. Goodenough, R.S. Eanes. Ada Reusability Guidelines. Technical Report 3285-2-208/2, SofTech, Inc., Waltham, Massachusetts, Revised 1991.

[6] P.T. Breuer and K. Lano. Creating Specifications from Code: Reverse-engineering Techniques. Journal of Software Maintenance: Research and Practice, John Wiley and Sons, 1991. Reprinted in Software Reengineering, by Robert S. Arnold, IEEE 1993.

[7] Gianluigi Caldiera, Victor Basili. Identifying and Qualifying Reusable Software Components. IEEE Computer, Feb 1991. Reprinted in Software Reengineering, by Robert S. Arnold, IEEE 1993.

[8] G. Canfora, A. Cimitile, and U. de Carlini. A Logic-Based Approach to Reverse Engineering Tools Production. IEEE Trans. on Software Eng., Vol. 18, No. 12, December 1992.

[9] A. Cimitile and U. de Carlini. Reverse engineering: Algorithms for Program Graph Production. Software Practice and Experience, Vol 21, pp 519-537, 1991.

[10] W. Cunningham, K. Beck. Constructing Abstractions for Object-Oriented Applications, Journal of Object-Oriented Programming, 2,2,17-19, August 1989.

[11] W.L. Frank. What limits to software gains ? Computerworld, pp 65-70, May 4, 1981.

[12] E. Horowitz, J.B.Munson. An Expansive View of Reusable Software. Software Reusability, Vol. I, Edited by T.J. Biggerstaff and A.J. Perlis. ACM Press, 1989.

[13] S. Horwitz and T. Teitelbaum. Generating Editing Environments Based on Relations and Attributes. ACM Trans. on Programming Languages and Systems, Vol 8, No 4, Oct 1986.

[14] Ivar Jacobson, Fredrik Lindstrom. Re-engineering of old systems to an object-oriented architecture. Proc. OOPSLA, 1991. Also reprinted in Software Reengineering, by Robert S. Arnold, IEEE 1993.

[15] Gail E. Kaiser, Simon Kaplan. Parallel and Distributed Incremental Attribute Algorithms for Multiuser Software Development Environments. ACM Transactions on Software Engineering Methodology, January 1993, Volume 2, Number 1.

[16] K. Koskimies, O. Nurmi, J. Paaki. The Design of a Language Processor Generator. Software -Practice and Experience, Vol. 18 (2), Feb. 1988.

[17] Richard D. Linger. Software Maintenance as an Engineering Discipline. Proc. Conf. on Software Maintenance, pp 292-297. Reprinted in Software Reengineering, by Robert Arnold, IEEE 1993.

[18] S.S. Liu, and K.R. Johmann. A Tool Specification Language for Software Maintenance: Part I —Language Design, Part II —Usage. SERC Technical Report 36F, CSci Dept., University of Florida at Gainsville, November 1989.

[19] Luqi, V. Berzins, R. Yeh. A Prototyping Language for Real-Time Software. IEEE Trans. Soft. Eng., vol. 14, October 1988.

[20] D. Maier, and D.S. Warren. "Computing with logic", The Benjamin/Cummings Publishing Co. Menlo Park, CA, 1988.

[21] B. Meyer. Software Reusability: The Case for Object-Oriented Design. IEEE Software 4(2), 50-64, 1987.

[22] F. Naveda. Specifying a Prototyping Language in the Cornell Synthesizer and the Xinotech Program Composer for an Integrated Programming Environment. Proceedings 2nd IEEE International Conference on Systems Integration, IEEE, June 15-18, 1992.

[23] D. Parnas, P. Clements, D. Weiss. Enhancing reusability with information hiding. In Proc. Workshop Reusability in Programming, Sept. 1983, pp 240-247.

[24] William W. Pugh Jr. Incremental Computation and the Incremental Evaluation of Functional Programs. Ph.D. Dissertation, Cornell University, 1988.

[25] T. W. Reps, T. Teitelbaum, A. Demers. Incremental Context-Dependent Analysis for Language-Based Editors. ACM Transactions on Programming Languages and Systems, Vol. 5, No 3, July 1983.

[26] D.S. Rosenblum. A Methodology for the Design of Ada Transformation Tools in a DIANA Environment. IEEE Software 2(2):24-33, March, 1985. Also as Stanford CSL Technical Report 85-269, February, 1985.

[27] Robert W. Schwanke. An intelligent tool for re-engineering software modularity. Proc. Int'l Conf. on Software Engineering, IEEE 1991. Also reprinted in Software Reengineering, by Robert S. Arnold, IEEE 1993.

[28] I. Silva-Lepe. Abstracting graphed-based specifications of object-oriented programs. Tech. Report NU-CCS-92-4, College of Computer Science, Northeastern University, March 1992.

[29] A.I. Wasserman, P.A. Pircher, R.J. Muller. The Object-Oriented Structured Design Notation for Software Design Representation, IEEE Computer, March 1990.

[30] Waters, R.C. Program Translation via Abstraction and Reimplementation. IEEE Trans. on Software Eng., August 1988

Issues in Re-Engineering from Procedural to Object-Oriented Code

Ricky E. Sward
Department of Computer Science
USAF Academy, CO 80840
rsward@cs.usafa.af.mil

Dr. Robert A. Steigerwald
Department of Computer Science
USAF Academy, CO 80840
steiger@cs.usafa.af.mil

This paper presents issues in re-engineering including *familiarity*, *completeness*, *existing designs*, and *when to re-engineer*. The issues are discussed and a reverse engineering method based on natural language descriptions of procedural code is described in detail. These descriptions provide an appropriate level of abstraction of the procedural code and can be used as the basis of object-oriented forward engineering using any Object-Oriented Analysis and Design methodology.

Introduction

The appeal of object-oriented programming with it's extensibility, maintainability, and reusability has enticed many considering development projects. Organizations with millions of lines of aging procedural code are also looking into object-oriented programming. As systems become more complex and fragile, the strengths of the object paradigm become more appealing.

The organizations with millions of lines of code also have millions of dollars invested in that code. If they are to switch to the object paradigm, will they just abandon the investment they have made in the procedural code? Should they just start over and re-design the system from scratch? If at all possible, the organizations prefer to achieve the benefits of the object paradigm without buying a replacement system or building one from scratch.

This paper presents issues in re-engineering procedural code to the object paradigm. Re-engineering is the process of examining an existing system and implementing that system in some form, typically a new programming language. [Chikof90] This is in contrast to normal forward engineering which proceeds from requirements to a design. A specific re-engineering process may involve reverse engineering of the existing system. Reverse engineering is the process of examining an existing system to 1) find the system's components and their inter-relationships and 2) create representations of the system at some higher level of abstraction. [Chikof90]

There are certain issues to consider before re-engineering a system. These issues of *familiarity*, *completeness*, *existing designs*, and *when to re-engineer* are covered fully below. Organizations have many options for re-engineering, and discussing these issues will direct an organization toward some re-engineering method. Our analysis shows most organizations should choose a method that reverse engineers to some intermediate abstraction, and then forward engineers to object-oriented code.

This paper proposes a method for the reverse engineering portion of this process. Our method provides a way to reverse engineer procedural code into an abstract form that can be forward engineered into object-oriented code. It provides an abstraction that is appropriate for doing any form of object-oriented forward engineering, using any of the current object-oriented methodologies. The form of our abstraction is natural language descriptions of the services provided by the existing system. These natural language descriptions are derived from the procedural code in the existing system as described below.

Before describing our method, we look at the previous research in this area and the rationale for using natural language descriptions. After the method is presented in detail, we discuss conclusions and future research.

Previous Research

Much work has already been done in the field of Object-Oriented Analysis (OOA) and Object-Oriented Design (OOD). Booch [Booch91], Coad and Yourdon [Coad91], Jacobson [Jacob92], and Wirfs-Brock [Wirfs90] (to name a few) present methods for building OO designs from requirements specifications. These methods rely solely on the OO designer's ability to extract objects from the requirements specifications. These methods work for systems that are built from scratch, implementing specific requirements. They do not work for systems that are being re-engineered from

procedural code, i.e., using these methods, one cannot extract objects directly from procedural code.
Other methods [Alabiso88, Bailin89] have combined some sort of structured analysis and OOD. They provide mappings from data flow diagrams (DFDs) and data dictionaries to objects and classes. The methods rely on the OO designer's ability to extract the objects from the structured analysis of the requirements as shown in the form of DFDs and data dictionaries. These methods have some application in the re-engineering environment. If the DFDs and data dictionaries can be extracted from existing code using a reverse engineering tool [STSC92], then the DFDs and data dictionaries can be used to develop an OOD. A discussion of this technique follows in the 'issues' section below.

There has been some work in automatically converting code from one procedural language to another [Olsem93]. The Re-engineering Technology Report [Olsem93] provides a long list of source-code translators from different vendors. For example, some systems convert from Fortran to C or C++, and others convert from Pascal to Ada. Even if these translators convert to an object-oriented language, they do not recognize objects during the conversion, so they are not building object-oriented code. They are simply converting to *procedural style* code written in a hybrid object-oriented language such as C++.

Ong and Tsai [Ong93] have developed a method for extracting classes and objects directly from procedural code (Fortran). Some objects can be found with hints from the user, and others are pulled directly from the *common* data block of the Fortran code. Their method relies on data flow analysis of the Fortran code to extract methods from Fortran code. These methods are placed in the objects extracted earlier. They do not build an intermediate representation of the Fortran code, but go directly from Fortran to C++ code. Their prototype system generates C++ code for the classes and methods extracted.

Recent work by Jacobson [Jacob91] provides an incremental method to re-engineer procedural code to object-oriented code. Using this method, a part of the existing system is selected for re-engineering, and an interface to this new object-oriented part of the system is developed. During the conversion, the designer reverse engineers to a formal design diagram and extracts the objects from this formal design diagram. Jacobson's method develops an intermediate representation of the procedural code before converting that representation to object-oriented code. The designer is still left with the task of finding the objects in the formal design diagram created from the reverse engineering.

Re-Engineering Issues

There are certain issues that organizations must consider before re-engineering a system. These issues include *familiarity*, *completeness*, *existing designs*, and *when to re-engineer.*

Familiarity. A pervasive issue in re-engineering a system is how familiar you should become with the procedural system. If you have been using the system, you will already be familiar with the system, its overall purpose, functionality, and general features. But, how much familiarity with the low-level functions of the system is required? To what level of detail should you analyze the system?

Jacobson [Jacob91] proposes a procedural system must be reverse engineered all the way back to a formal design before the conversion. This could be a lengthy process for a large system, but it will help you become familiar with the system. Using this method (even with the incremental nature of the method) the reverse engineering process alone could take years for large systems. Is it really necessary to know the system in this much detail? Is it necessary to have an exact formal design diagram for the system? How do you know that you are familiar enough with the system to begin the conversion?

If you already have a formal design of the system, then you may want to use that design. If there is not one available, it is our opinion that an exact formal design diagram is not needed. The person doing the conversion (the converter) must be somewhat familiar and comfortable with the functions of the system, but the level of this familiarity can be quite fuzzy. Some of the familiarity will come from the iterative nature of the conversion process. The more the converter works with the procedural code, the more familiar he/she will become.

As the converter becomes more familiar with the system, they should be looking for things that may later become objects in the object-oriented code. Can these objects be found only in requirements specifications? Many OOA methods [Booch91, Coad91] start only from requirements specifications. If the objects can be found only in requirements specifications, then the converter must look for requirements of the system when becoming familiar with the system.

If objects can be found in other places than the requirements specifications, then the converter should look for other things while becoming familiar with the system. Ong and Tsai [Ong93] use the data structures found in Fortran code to extract objects. Objects are often found in other things than just the requirements, so the converter should be looking for other things than just requirements. The converter should be looking for the *services* being performed by the modules of the procedural system. These services will be used to identify the behavior of objects in the new system.

So, when you are becoming familiar with a procedural system, should you be looking for requirements or just services of the system? It is our opinion that you should look for services of the system. These services will tell you the current functionality of the system, and will help you determine the objects required in the system. The services will also help document the progress of the conversion as described below.

Completeness. An important question for the converter is "When is the conversion done?" The converter must be able to measure the progress of the conversion and determine when the job is complete. One measure of this is whether all the services or requirements of the original system are in the converted system.

If the converter has access to the original requirements document, they can reference this document to measure progress of the conversion. Of course, if the converter has the original requirements document, they may want to do OOA on the original requirements. These requirements may not accurately reflect the functionality of the current system, so the converter may need to rely on the services identified while becoming familiar with the system.

If no requirements are available, the converter must rely on the services found in the system to measure the progress and completeness of the conversion. A list of the services should be compiled by looking at the existing code. Our method, presented below, includes this step.

Existing Design. What if there is already an existing procedural design for the system? To what extent should the converter use this design? Alabiso's method [Alabiso88] converts a DFD to an object design. But, using the functionally decomposed design can cause the designer to miss some of the objects that should be built into the object design. [Sward93] If at all possible, the designer should not use an existing design to build an object design because the functional decomposition that was done to build the existing design will taint the object design.

The existing design *can* be used to become familiar with the system. The converter can look at an existing design to find out how procedures interact. The designer may even be able to find high level services that the system is providing by looking at the design.

When to re-engineer. Sittenauer and Olsem [Sitten92] present a model to estimate whether or not an organization should re-engineer or not. The question is a good one. How does an organization know when to re-engineer? Are there measurable indications of the appropriate time to re-engineer? The Sittenauer and Olsem model examines the complexity, importance, and lifetime of a candidate system. These indicators are measured and graphed to help the organization determine if the system should be re-engineered.

All of these issues must be considered before re-engineering. The next section presents our rationale for using natural language descriptions when becoming familiar with the procedural system.

Rationale for Natural Language

After an organization has determined it is time to re-engineer, the first step is to become familiar with the system and determine the services provided by the system. We are proposing that these services are best described using natural language for the following reasons: 1) natural language will capture the terminology of the problem statement that has been instilled into the procedural code, 2) natural language provides an appropriate intermediate level of abstraction, and 3) natural language descriptions will provide an input into object-oriented forward engineering.

Terminology. As the converter of a system becomes familiar with that system, they need to learn the terminology of the system. The terminology of the system holds clues to the objects that will be built in the object-oriented system. One way to learn this terminology is to write natural language descriptions of the services provided by the system. Representing services in natural language forces the converter to discover the terminology of the procedural system. More importantly, it helps the converter discover the terminology of the original problem statement. As the converter describes the services of the system in natural language, they may begin to get a feel for some of the objects that will appear in the object system.

Level of Abstraction. In the process of reverse engineering we develop an intermediate representation of the existing system. This representation must be at the right level of abstraction. If it is not abstract, there may still be remnants of the procedural design in the representation and these will surface in the object design. If the representation is too abstract, it may be too time consuming to build without providing much added value.

Alabiso's method [Alabiso88] of converting DFD's and data dictionaries to objects does not provide an appropriate level of abstraction. If the DFD's and data dictionaries are extracted from the procedural code, they will still be tainted with the functional decomposition done to build the designs. Ong and Tsai's method [Ong93] is also tainted by the functional decomposition when it looks for objects in the data structures of Fortran. The objects in the object design are not necessarily tied to the data structures in Fortran. The data structures can give us clues to the objects in the object design, but there is no one-to-one correlation.

Jacobson's method [Jacob91] on the other hand, requires we build a formal design describing the *existing* system. To develop a design at this level of abstraction takes a considerable amount of work. This may be too time consuming for large systems without providing much more capability in the design. We need a representation that is somewhere in between these two levels of abstraction.

Our method provides this intermediate level of abstraction by using ***natural language descriptions*** of the services of the existing system. Natural language descriptions do not have any procedural flavor to them, and they are not time consuming to construct. They provide an acceptable level of abstraction for representing procedural code and can be used for object-oriented forward engineering. The descriptions are built using the data structures and procedures in the existing system. Objects do not come directly from these data structures or procedures, but come from OOA on the descriptions of the services provided by them.

Forward engineering. The natural language descriptions that are built will be used as input to object-oriented forward engineering. Most OOA methodologies use requirements statements written in natural language. The natural language descriptions built with our method take a similar form to requirements statements, so OOA on the descriptions is a similar task.

Another benefit of the natural language descriptions is that designers already familiar with OOA methodologies do not need to learn a new methodology to design from the descriptions. The expertise gained already in finding objects can be used to find objects in the descriptions. Thus, organizations will not need to spend more money training their designers.

We have analyzed the issues in re-engineering presented earlier and for the above reasons have chosen natural language descriptions as the abstract representation produced from our reverse engineering method. Our method is described fully below.

The Method

This section presents a method for becoming familiar with a procedural system so that it can be converted to the object paradigm. The method helps the user develop natural language descriptions of the services of the procedural system. These natural language descriptions provide the raw material for OOA.

I. Describe the data structures
II. Describe the procedures

Figure 1 - Phases of the Method

The method is split into two phases as shown in figure 1. The first phase involves describing the data structures and listing where the data structures are defined. The second phase looks at the procedures and describes the services being performed by them.

Phase I. This phase of the method lets the converter become more familiar with the data structures and the information being stored. Figure 2 shows the three steps of Phase I.

1. List the data structures
2. List where the data structures are defined
3. Write a natural language description of the data structures

Figure 2 - Steps in Phase I

The first step is to list all the data structures in the procedural system. These may be defined in several different units of the system or in a separate unit just for definitions. The next step is to list where these data structures are located. This list is used for future reference. The third step is the most important one of this phase of the method. In this step, the converter writes a natural language description of each data structure. As described above, this type of description helps the converter become familiar with the terminology of the procedural system.

Since the natural language description is important, we provide an example description of the following Pascal data structure:

```
Module_List = RECORD
      mlist  : ARRAY[1..Max_Modules] OF
                       Module_Ptr;
      mcnt  : INTEGER;
END;
```

The natural language description of this data structure might be:

Module list is a record with two fields:

1) a list of pointers to modules stored in an array from 1 to the maximum number of modules and
2) an integer that points to the last module pointer in the array.

Once the natural language descriptions are built for all the data structures, the converter proceeds to Phase II.

Phase II. This phase of the method deals with the *procedures* of the existing system. In this phase, the converter tries to extract the *services* being performed by the procedures. The steps of this part of the method are shown in figure 3.

In this phase of the method, the converter should get a sense of the entire procedural system. As the converter builds the natural language descriptions of the procedural processing, he should better understand the services that the procedural system accomplishes.

4. List all the procedures in the system
5. List the parameters for each of the procedures
6. Focus on one data structure
7. List the parts of the data structure each procedure uses
8. Write a natural language description of the processing for each procedure
9. Describe the impact on the selected data structure using natural language
10. List any services found
11. Repeat steps 6-10 until all the data structures have been considered
12. Write natural language descriptions for any remaining procedures

Figure 3 - Steps in Phase II

In step 4, the converter lists the unit and procedure names. This is simply a reference list for the converter to make sure all the functions in the system have a natural language description. The next step is to add to the list of procedures all the parameters for the procedures. This is another list that helps the converter become familiar with the processing and terminology of the system. An example list is shown below.

Procedure	Parameters
edit_name	design_rec
edit_author	design_rec

In step 6, the converter picks one data structure and focuses attention on it. We suggest starting with data structures that have multiple attributes such as a Pascal *record* structure or a C *struct* structure. Which data structure to focus on is not crucial to the conversion process, but we recommend starting with large data structures because they tend to be central to more system processes.

An example of a Pascal *record* structure is shown below.

```
design_rec = RECORD
      dname    : STRING[30];
      fdir     : STRING[8];
      modules  : module_list_type
      author   : STRING[30];
      ...
```

In step 7, the converter lists the parts of the data structure each procedure in the system uses or modifies.

The converter should prepare a list with each procedure name followed by the parts of the data structure used or modified in the procedure.

It may be the case that the procedure passes the entire data structure into the procedure, but only uses part of the structure. It is at this point the converter should make every effort to partition out this *tramp* data. Tramp data is data extraneous to the processing of the procedure. The converter should list only those parts of the data structure that are used or modified in the procedure.

The list of procedures and parameters built in step 5 is helpful here. The converter can look on the list to see if the selected data structure is included in the parameter list for a procedure. If it is, the converter examines the extent the data structure is used or modified in the procedure. The parts of the data structure that are used or modified should be included in the list built for this step.

An example is shown below for the *edit_name* procedure. The entire *design_rec* data structure is passed as a parameter to *edit_name*. However, the only part of the *design_rec* data structure that is modified in *edit_name* is *dname*.

edit_name	dname

At this point, the converter will be getting a better idea about what parts of the selected data structure are used in the different procedures. It is a logical next step to describe the processing of these parts of the data structure in natural language. In step 8, the converter takes the lists of procedures and parts of the data structure as defined in step 7 and writes natural language descriptions for each of the procedures.

The converter should use only natural language to describe the processing of the procedure and avoid short-cuts such as using the procedure name or the data structure name. If there are any procedures that are called from inside this procedure, the converter should develop natural language descriptions of these procedures before moving on. This allows the converter to follow and understand the processing going on in the procedure. An example of the natural language description of edit_name is shown below.

> The procedure that edits the name of a design is passed the entire design record. It changes the value of the name of the design by prompting the user for a new value and then setting the design name to this new value. The new value is checked against all the old design names to assure no duplicates are permitted.

The next step, step 9, is to expand the descriptions built in step 8. The descriptions are expanded by including the impact on the data structure by each procedure. This is where the converter can show exactly to what extent the data stucture is changed by the procedure. Any tramp data can be delineated clearly in this step. By now, the converter should have a good feel for the terminology of the procedural system and the services being performed on the selected data structure. An example of the natural language generated for edit_name during step 9 is shown below.

> The procedure that edits the name of a design is passed the entire design record. It changes the value of the name of the design by prompting the user for a new value and then setting the design name to this new value. The new value is checked against all the old design names to ensure there are no duplicates.
>
> This procedure does not change the entire design record. Only the name field of the design record is modified. The entire processing involves the name of the design and the list of all the other design names.

Step 11 in this phase of the method is iteration. In this step, the converter repeats steps 6 through 10 choosing a new data structure to focus on (from step 6 This iteration continues until all the data structures have been considered. It could be the case that some procedures do not have parameters and will not be considered in steps 6-11. Step 12 provides for this by requiring the converter to make sure all procedures have natural language descriptions built for them.

After step 12, the converter has natural language descriptions of all the data structures and procedures in the procedural system. The converter can now proceed to forward engineer the system using whatever OOA and OOD method they prefer.

Conclusion

The reverse engineering method presented in this paper provides an intermediate representation of procedural code. This representation is at an appropriate level of abstraction in that it is not too close to the existing procedural code or so abstract that it takes great amounts of work to create. The natural language descriptions are built during the time a converter is becoming familiar with the system. The descriptions help the converter become more familiar with the services provided by the system. These services define the completeness and functionality of the new object-oriented system. The descriptions provide a medium to forward engineer using an object-oriented methodology with which the converter is familiar.

Future Research

In the future, we hope to partially automate our reverse engineering method. The generation of the lists in steps 1, 2, 4, and 5 can easily be automated, whereas the generation the natural language descriptions is a somewhat more difficult artificial intelligence problem.

References

[Alabiso88] Alabiso, B., *Transformation of Data Flow Analysis Models to Object-Oriented Design*, Conference Proceedings, OOPSLA '88, pgs 335-353

{Bailin89] Bailin, S., *An Object-Oriented Requirements Specification Method*, Communications of the ACM, May 1989, Vol 32, Number 5, pgs 608-623

[Booch91] Booch, Grady, Object-Oriented Design with Applications, Benjamin Cummings Pub Co, Redwood City, CA, c1991

[Chikof90] Chikofsky, Elliot J. and Cross, James H. II, *Reverse Engineering and Design Recovery: A Taxonomy*, IEEE Software, Jan 1990, pgs 13-17

[Coad91] Coad, Peter and Yourdon, Edward, Object-Oriented Analysis, Yourdon Press, 1991.

[Jacob91] Jacobson, Ivar and Lindstrom, Fredrik, *Re-Engineering of Old Systems to an Object-Oriented Architecture*, Conference Proceedings, OOPSLA '91, pgs 340-350

[Jacob92] Jacobson, Ivar, et al., Object-Oriented Software Engineering, A Use Case Driven Approach, Addison-Wesley Pub Co, Wokingham, England, c1992

[Olsem93] Olsem, Mike and Sittenauer, Chris, Re-Engineering Technology Report, Vol 1, Aug 1993, STSC Hill AFB UT

[Ong93] Ong, C. L. and Tsai, W. T., *Class and Object Extraction from Imperative Code*, JOOP Mar-Apr 1993, pgs 58-68

[Sitten92] Sittenauer, Chris and Olsem, Mike, *Time to Re-Engineer?*, Crosstalk, March 1992.

[STSC92] STSC, Sittenauer, Chris, Olsem, Mike, and Murdock, Daren, Re-Engineering Tools Report, April 1992, Rev-A, 6 Apr 1992, STSC Hill AFB, UT

[Sward93] Sward, Ricky, *Pitfalls in Re-Engineering from Structured Code to the Object Paradigm*, Conference Proceedings, Software Technology Conference '93

[Wirfs90] Wirfs-Brock, Rebecca, Wilkerson, Brian, and Wiener, Lauren, Designing Object-Oriented Software, Prentice Hall Pub Co, Englewood Cliffs, NJ, c1990

An Object-Based Framework for Reengineering Avionics Software

Noble N. Nkwocha and John J. Zenor

Naval Air Warfare Center Weapons Division
Embedded Computing Technology Office
Code C21C
China Lake, California 93555-6001

Abstract

We present results from a case study on the use of object oriented techniques to reengineer the Ballistic Trajectory Algorithm (BTA) software. The BTA software is used in the Operational Flight Programs (OFPs) of several Navy attack and fighter aircraft to determine the release point for ballistic weapons. Though the algorithm itself is simple, the nature of its implementation in space and time-limited OFPs has resulted in an extremely complex implementation—typical of OFP code. The structure of the basic algorithm is *entirely* hidden by the embedding of the integration method and scheduling considerations into the algorithm and by the large number of special cases introduced to handle specific weapons. Existing documentation of the basic algorithm is *totally inadequate* for understanding the actual program. This paper provides a demonstration of how object-oriented properties such as inheritance can be used to control the complexity of the implementation, yielding a much more understandable, maintainable, and reusable program, and describes the methodology used and lessons learned in the reengineering effort.

1 Introduction

Since the first airborne digital computers were introduced in operational Navy attack aircraft in the 1960s, significant amounts of resources have been invested in the design, development, and maintenance of avionics software systems, most of which are written in assembly language and all of which run on obsolete computers by today's standards. The existing engineering data, which include the assembly codes, structured flowcharts (called math flows), detailed test procedures for safety of flight and Fleet certification, and tactical manuals, are insufficient to support redevelopment using modern software engineering technologies. Indeed, for a recent upgrade to the hardware used by an OFP, redevelopment of the operational weapon systems software for a new computer was deemed too risky and costly. Consequently, new hardware was developed to emulate the obsolete mission computer in order that the current software could be run with limited changes in the new hardware. Possible concerns are

1. Long change cycles on the order of 18-24 months to fix errors and enhance functionality

2. Expensive, labor-intensive, error-prone procedures to analyze, design, code, and test changes

3. Difficulty in restructuring the software to accommodate significant modifications

4. Chronic shortages of sufficient processor throughput and memory, since the fragility and machine-specific nature of the software inhibit migration to new hardware

In the case of the Ballistic Algorithm, the program is currently stable, and "*maintenance*" in the sense of fixing bugs is not really a problem. The real problem is that several characteristics of the OFP (e.g., scheduling period, integration algorithm, and specific weapon types) are built into the code, such that the code is so complex that it is not feasible to consider improving the methods used in the ballistic algorithm. This results in time-critical code that cannot be improved, even though more up-to-date numerical methods that could improve performance are available. Considerable man hours are spent integrating new weapon types into the program, due primarily to a largely manual process of curve fitting for lift, drag, and other data, where changes to the method used in the basic algorithm might eliminate the need for this process entirely. The complexity of the code has increased with time as new weapons are added and as it is nearly impossible to remove anything from the code because of the possible side effects of removing obsolete code.

Because of the smaller defense budget, the recent need to cut operational costs within the Navy and the Department of Defense in general has provided additional incentive to develop techniques for reengineering deployed avionics software to reduce life-cycle costs. This paper examines the contribution that object-oriented programming (OOP) technology can make to reengineering legacy systems. Our reengineering expectations for an end-product are

1. Reengineering techniques that form the foundation necessary as a first step towards reengineering avionics software in real-time
2. A simpler, less complex code structure that is easier to understand
3. Reduced effort required to add new and/or modify existing avionics software functionalities, weapon types, or release conditions
4. Retention of the same or higher degrees of accuracy as currently provided in the existing avionics software
5. Reduced life-cycle costs of current avionics software
6. An existence proof that it is feasible to contemplate reengineering deployed avionics software using modern software engineering technologies to reduce life-cycle costs.

As representative of the computationally intensive operational software in Navy aircraft, the BTA software was selected as the target for reengineering. Our paper presents results from a case study to (1) investigate and evaluate techniques for reengineering the BTA software using detailed requirements and design information extracted from the existing programs and (2) demonstrate these techniques on selected components of the BTA software. We first describe the BTA software, and then discuss its analysis and redesign in the reengineering process. Our experiences and results in the reengineering process are presented.

2 Ballistic trajectory algorithm

Most unguided, free-fall, air-to-ground weapons launched from aircraft to attack targets on the ground or surface, e.g., bullets, drogued or retarded bombs, cluster munitions, streamlined bombs, and unguided rockets, can be described as ballistic weapons [1]. Forces considered on these weapons after release from the aircraft are lift, gravity, and aerodynamic drag. A computational requirement imposed on airborne digital computers aboard tactical Navy attack aircraft is the prediction of range and time of fall for these ballistic weapons, given the following:

1. Position of the target relative to the aircraft
2. Velocity of the aircraft relative to the air mass
3. Direction and magnitude of gravity
4. Velocity of the air mass relative to the ground
5. Velocity of the target relative to the ground

These sensor-supplied quantities makeup the weapon-release conditions as supplied by aircraft sensors (e.g., inertial platforms, air data sensors, radar or laser trackers, target tracking devices).

Given the above weapon-release conditions, the successful release of a ballistic weapon so that it impacts at a desired point on the ground requires an *a priori* prediction of the weapon's trajectory. This problem is illustrated in Figure 1.

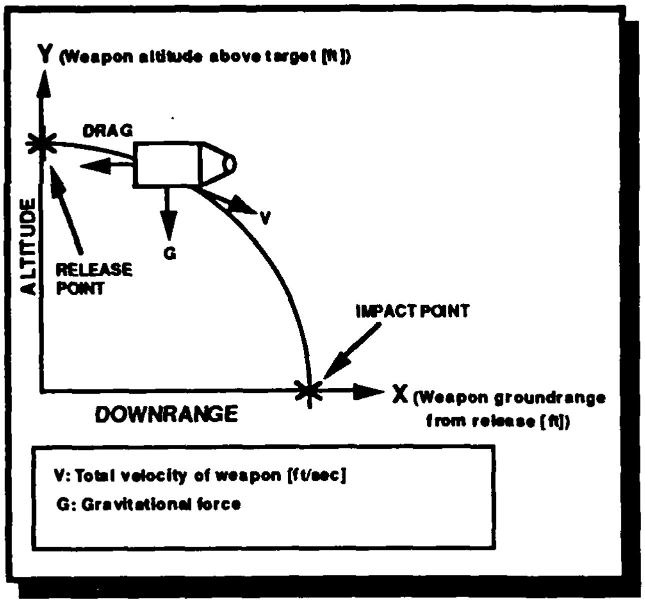

Y (Weapon altitude above target (ft))
DRAG
V
G
ALTITUDE
RELEASE POINT
IMPACT POINT
X (Weapon groundrange from release (ft))
DOWNRANGE
V: Total velocity of weapon [ft/sec]
G: Gravitational force

Figure 1. Ballistic Trajectory in the X-Y Plane

As depicted in Figure 1, everything necessary is assumed known about the release point. The impact range and time of flight are to be computed. The BTA software provides this important functionality. Initially, the BTA software was designed for single-stage, free-fall bombs, but has been modified over the years to accommodate newer, technologically advanced weapons, such as rockets and free-fall bombs with lifting surfaces.

Currently, two versions of the BTA software exist. The first is the actual real-time version written in assembly language for the airborne digital computers aboard the tactical attack aircraft. The second version consists of 76 FORTRAN routines. The FORTRAN version is used mainly to fine-tune weapon coefficients for the real-time version and is also used as a test program for safety of flight and Fleet certification.

While the BTA software has been used successfully since its development in the late 1960s, several of the computing constraints (e.g., speed, storage capacity, numerical algorithms) that confronted its design at the time no longer exist today. As currently designed and implemented, the BTA software does not provide sufficient flexibility to allow easy modification of weapon types or release conditions. One consequence has been that, over its 25-year life span, only small modifications have been attempted. The result is code tangled from years of patchwork.

Given the investigative nature of our task and the time and other resource constraints that are attributable in part to limited funds, we selected the ALGO subroutine component of the BTA software as the target for reengineering. ALGO is the '*main*' subroutine that forms the heart of the FORTRAN version of the BTA software. Though of moderate size, the high degree of complexity (illustrated in Figure 2), inherent in the ALGO subroutine is representative of both the BTA software and the existing avionics software systems in general.

3 Analysis

Our main task in the reengineering process was the analysis of the BTA software. The motivation behind the analysis is twofold:

1. To recover and capture the essential design and requirement information necessary for redesign and redevelopment

2. To identify the presence of software engineering anomalies (e.g., non-modularity, code/data dependency, and lack of clarity) in the existing code

A common barrier to software reengineering is the difficulty of understanding the design assumptions and constraints embedded in a piece of code [2]. Our own experience confirms this; the design and requirement information about what the BTA software does is inextricably embedded in the code. The code itself, tangled from years of patchwork, describes not what was done, but rather how it was done.

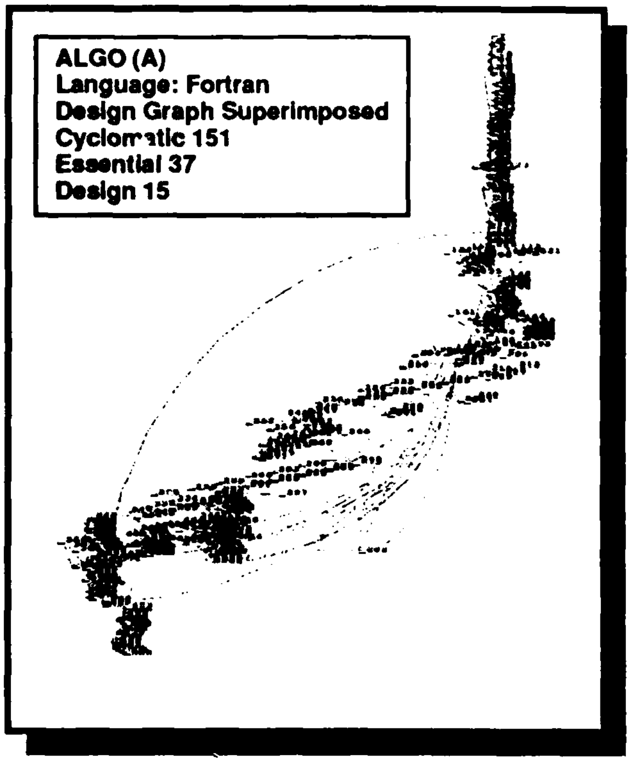


Figure 2. Complexity Analysis of ALGO subroutine component of the BTA software

In analyzing the ALGO subroutine component, we examined the 751 lines of FORTRAN code line by line to recover the design and requirements information. This process was both time- and labor-intensive. In one instance for example, recovering the implementation of a second-order Runge Kutta numerical integration method took in excess of 40 man-hours. This intrinsic complexity, depicted in Figure 2, stems from the original design.

The cyclomatic complexity value of 151 (Figure 2), represents a measure of the ALGO subroutine's decision structure. The essential and design values of 37 and 15 represent the degree to which the module contains unstructured constructs (i.e., branching in and out of loops and decision nodes) and the complexity of the module's calling patterns respectively. According to McCabe, a module whose flow graph has e edges and n nodes has a cyclomatic complexity of $e - n + 2$ [3]. A cyclomatic complexity value between 12 and 20 is considered to be within the norms of traditional software engineering.

The analysis effort on the ALGO subroutine was a five-step process.

In the first step, all loops and code blocks within the executable part of the subroutine were identified and assigned names and/or block labels.

In the second step, we identified all input/output (I/O) processing and separated this from the non-I/O processing. The set of actual I/O data, including data structures, was also identified. The data structures identified in this step included (1) five two-dimensional REAL arrays, (2) a one-dimensional REAL array, and (3) two one-dimensional INTEGER arrays.

In the third step, all hard-coded literals and constants in the executable part of the code were identified. The intent was to decouple physical data from the actual code. Named-constants were then declared, and the literals and constants within the executable part that could be replaced were then replaced with named-constants.

In the fourth step, all variable references throughout the code were reconciled. The objective was to identify exactly where and how, in the source code, each variable was referenced. This was particularly helpful in identifying the variables that were used only for specific weapon types and also facilitated the design of our object-class structure in the redesign phase.

Finally, in the fifth step, all commonly duplicated blocks of code, as exemplified in Figure 3, were identified for elimination by the establishment of reusable methods in the design phase.

Once analysis of the code was complete, the redesign and subsequent redevelopmen. in SmallTalk-80* was initiated. The intent was not to introduce yet another language such as SmallTalk into the long list of languages used for Navy software, but rather to use SmallTalk as a prototyping language for the design and initial implementation portion of our reengineering task and then complete the actual implementations in C++ and Ada to see how many of the beneficial Object Oriented concepts of SmallTalk-80 could be retained in the final implementations.

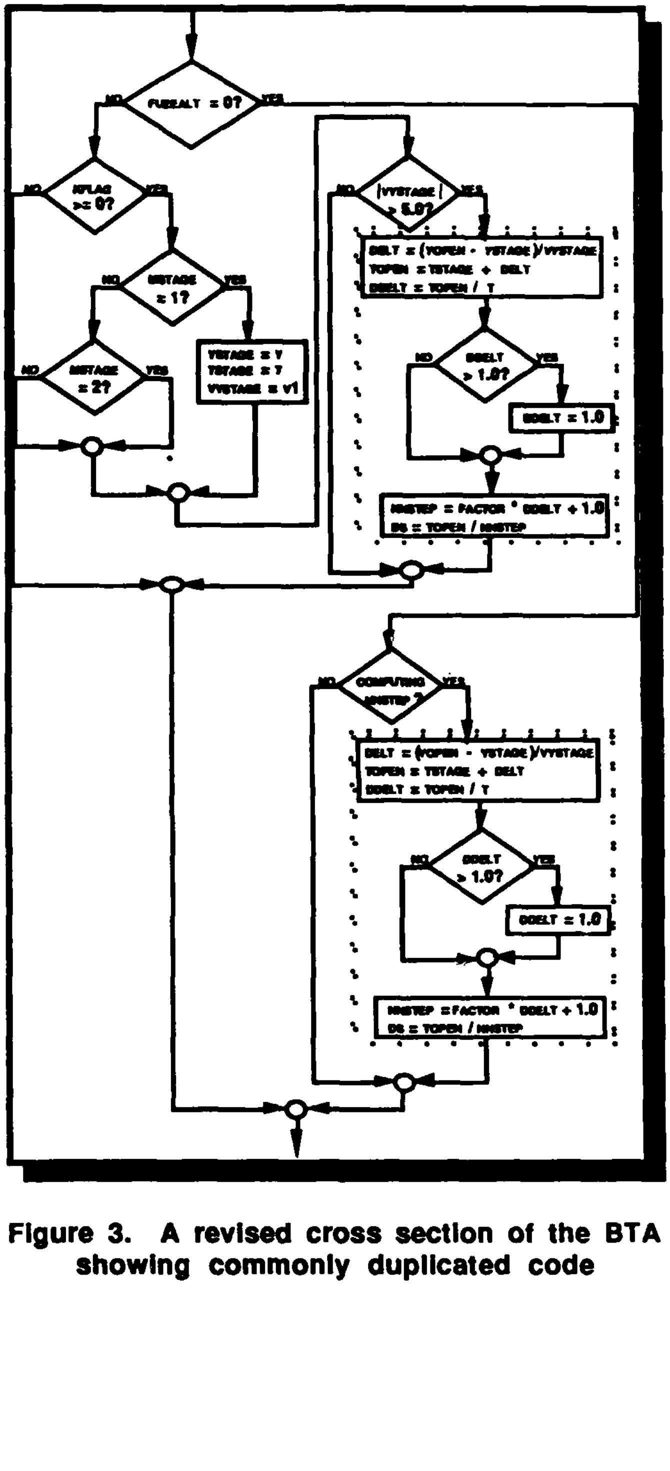


Figure 3. A revised cross section of the BTA showing commonly duplicated code

* SmallTalk-80 is a registered trademark of ParcPlace Systems Inc., Sunnyvale, CA 94086

4 Redesign

Our choice to redevelop the BTA software and to demonstrate the reengineering efforts in SmallTalk-80 is threefold:

1. Of all the OOP languages and systems to appear over the past decade, SmallTalk-80 remains one of the few '*pure*' OOP languages that possess all of the object-oriented concepts [4].
2. The class, object, and method browsers provided in SmallTalk-80 fulfill our need for a tool that assists in navigation through the resulting object-class library.
3. The SmallTalk-80 system provides an excellent environment for incremental development and testing of classes and methods as they are developed without the need for development of complex test drivers.

Although the initial redesign task is complete, the subsequent programming task is still ongoing and is therefore not discussed at great length here.

Some disadvantages to the use of SmallTalk-80 for initial implementation of this type of software include the following:

1. Methods were not available to read in the large data files of integer and floating-point data necessary for testing that were produced by FORTRAN formatted I/O. However, these methods were rapidly constructed from methods available in the class libraries provided.
2. Methods for handling common scientific data types such as two-dimensional arrays were not directly available and had to be constructed from more primitive methods.
3. A very steep learning curve is required to use SmallTalk-80 and its tool suite.

Our main challenge in the redesign process was the derivation of an object-class structure that supports a high degree of modularity, encapsulation, and ease of maintenance for the ALGO component of the BTA software. In deriving the object-class structure, we first identified the physical objects being modeled by the BTA, and then we determined the fundamental distinctions between these physical objects, as well as the commonalties among them. This information was then used in the derivation of the class hierarchy shown in Figure 4.

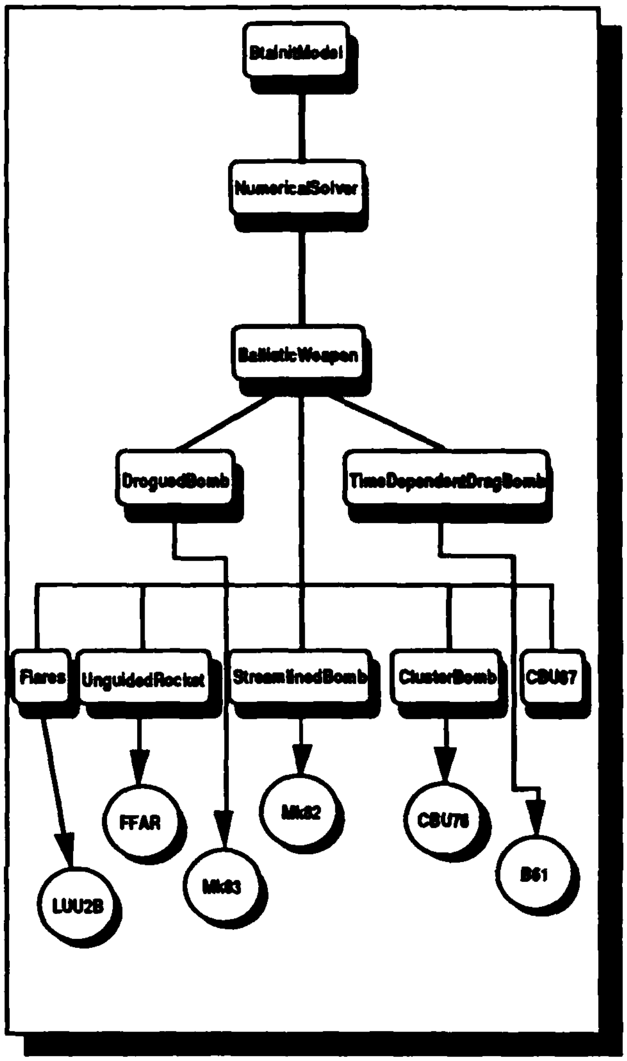


Figure 4. Inheritance structure of the redesigned ALGO classes

4.1 BtaInitModel class

The **BtaInitModel** class is an abstract superclass for all ballistic weapon objects modeled in ALGO. Its relationship with the other classes is based on the premise that for every ballistic weapon trajectory, it is necessary to initialize the working variables to their initial values based on the weapon's release conditions. The **BtaInitModel** class contains class-instance variables used for every ballistic weapon and implements methods for the initializations necessary at the beginning of each trajectory. These initializations include

1. Initializations required to account for the effects of altitude fuzing

2. Initializations required to account for the effects of time fuzing

3. Initializations required to account for the dynamic calculation of the number of integration steps needed in the first phase of the weapon's trajectory

4. Method used for expressing the weapon's drag as a function of its release velocity

4.2 NumericalSolver class—subclass of BtaInitModel

The **NumericalSolver** class implements numerical methods used for each ballistic weapon in the BTA software. As currently implemented, the BTA software uses a second-order Runge-Kutta numerical integration method for solving ballistic trajectory equations. This functionality is provided by the methods implemented in the **NumericalSolver** class. The separation of the functionality provided by this class from that provided by the **BallisticWeapon** class is intended to provide flexibility for future enhancements, especially for easy incorporation of new integration and other numerical analysis methods.

4.3 BallisticWeapon class—subclass of NumericalSolver

The **BallisticWeapon** class implements methods for

1. Determining integration step sizes
2. Computing accelerations attributable to gravity, thrust, and the weapon's drag
3. Making final corrections in the computed trajectory, using either the *straightLineMethod* or the *trajectoryExtrapolationMethod*
4. Sequencing the various functionalities

The six different methods, each of which may be used for determining the step sizes within each phase, implemented in the **BallisticWeapon** class are

1. *SpecifiedMethod*: computes step sizes as predefined arithmetic progression.

2. *VelocityMethod*: computes step size based on the weapon's velocity at the beginning of the step.

3. *PressureMethod*: computes non-constant step size based on the reciprocal of the dynamic pressure being experienced by the weapon.

4. *ComputedMethod*: computes the step size as a constant based on the number of integration steps left in the trajectory and the total estimated time of fall.

5. *TimedMethod*: computes step size as a constant based on the number of integration steps in the phase and the total time of that phase.

6. *RootingMethod*: computes step size as a variable based on the number of integration steps left in the trajectory and the weapon's altitude.

Each ballistic weapon's trajectory is divided into a number of phases depending on the ballistic weapon type. Associated with each phase are methods implemented to compute

1. The integration step sizes
2. The maximum number of integration steps for that phase

The number of phases, values of all phase parameters, and the type of each phase for a particular weapon are determined by analysis outside of the ballistic trajectory algorithm.

4.4 Base classes

The **DrougedBomb**, **StreamLinedBomb**, **ClusterBomb**, **Flares**, **Cbu87**, **UnguidedRocket**, and **TimeDependentDragBomb** classes form the base classes for the various weapon types in the BTA software. Each of these classes implements methods that depend strictly on that particular weapon type, and which also are necessary to adapt the algorithm to handle that particular weapon type. One of these methods is the *resetParameter* method, which resets the parameters (e.g., drag functions, step-size parameters, and number and size of phases) that change when choice of weapon changes.

The objects shown in circles (Figure 4), are actual instances of the various weapon classes. For example, Mk 83 (bomb with a BSU85B ballute retarding device), forward-fixing aircraft rocket (FFAR) (2.75-inch), and B61 a bomb with a 17-ft parachute are each instances of the **DrougedBomb**, **UnguidedRocket**, and **TimeDependentDragBomb** classes, respectively.

5 Summary and conclusions

Our findings relate to the reengineered BTA, the SmallTalk-80 system and its tool suite, and the reengineering process itself.

Regarding the reengineered BTA, we found that the present difficulty in restructuring the existing BTA software to accommodate significant modifications stems from the tight coupling of code, data, and methodology as evidenced in the ALGO subroutine component. Much of the complexity inherent in the BTA code had been added to the relatively simple BTA as a result of the constant handling and testing of special cases for individual weapon types. The tight coupling of code, data, and methodology contributes to the relatively high (3 man years per year) maintenance costs.

As we began analyzing the ALGO subroutine code, one consideration began to emerge: "Would our object-based approach lead to a solution that is better than that currently provided by the existing BTA software and at the same time fulfill the reengineering expectations?"

Our results thus far suggest that the object-based approach offers potential for success. Overall, our object-based framework resulted in a redesigned ALGO with far less complexity, and commonly duplicated blocks of code eliminated by the establishment of reusable methods, thus, resulting in fewer lines of code to maintain.

The use of inheritance in our object-class structure promises to provide considerably greater physical localization of all special code pertaining to any particular weapon type. The inheritance feature allowed special code for specific weapon types to be physically located within the weapon class rather than being included as a special case in the basic algorithm code. The result is that, methods in the weapon class are automatically used in preference to the general method for the algorithm, thus eliminating the need for continual testing of special cases in the algorithm code. The use of inheritance allows the physical grouping of all code for a specific weapon, making future modifications or even removal of the code for any weapon simple and possible without the fear of unforeseen side effects. Our object-class structure for the redesigned ALGO component promotes the derivation of new classes from working, tested, existing classes. The derived classes inherit existing methods while extending and enhancing the base functionalities, all without modifying their base classes. This extensibility reduces the effort required to add new and/or modify the existing functionalities and, indeed, forms the core of our object-based framework.

Regarding SmallTalk-80 and its tool suite, our experience indicates that the benefits of using SmallTalk-80 to redesign an object-oriented model of the ALGO subroutine component of the BTA far outweigh the steep learning curve experienced in the reengineering process. A necessary condition for the effective maintenance of a system is a level of understanding of that system sufficiently mature to support its maintenance. It is true that our object-based approach may lead to a reengineered BTA software that exhibits a high degree of modularity and elasticity. However, it presents new challenges to the maintainers. The maintainers need to understand the reengineered system to maintain it effectively. Inheritance derived from our object classes and the dispersal of the redesigned system's functionalities in the form of methods necessitate the need for tools to assist in navigation through the object-class structure. We feel that the class, object, and method browsers provided in SmallTalk-80 fulfill this need.

The rigid, object-oriented discipline enforced by the SmallTalk-80 system resulted in a reverse engineered ALGO component that is much simpler and promises to be better, at the least, in terms of design. The availability of a large reusable class library of existing methods coupled with the mature environment for browsing and the incremental development and testing of methods as they are developed was invaluable.

Regarding the reengineering process, one concern with the development of any new approach to reengineering legacy code such as that for the BTA is how to get the current users to accept the ideas and incorporate them into their work. Our experience suggests that getting the current users involved in the reengineering effort is the best approach, although historically this has not proven sufficient to overcome the inertia and pressures of the moment encountered by the current users. Given our perception that the risks of a reengineered software blur the intent of the reengineering process, we suggest that demonstrating the reengineering paradigm on (several, whenever possible) sub-components of existing systems allows current users to become comfortable with the process while considering the end-product. In particular, we contend that demonstrating the software reengineering process on sub-components of existing OFPs will serve to alleviate the risks, both real and perceived.

Acknowledgments

We are grateful to Brian McMahon, Randy Christensen, and Phil Niebuhr, custodians of the BTA software; and to Bob Westbrook, Lee Lucas, and Dick Nuckles—staff scientists at the Embedded Computing Institute and Embedded Computing Technology Office at the Naval Air Warfare Center Weapons Division (NAWCWPNS) for their insights and contributions to the reengineering process.

References

1. Duke, A. A., Brown, T. H., Burke, K. W. and Seeley, R. B. *A Ballistic Trajectory Algorithm for Digital Airborne Fire Control*, Tech. Pub. (TP) 5416. Naval Weapons Center, China Lake, Calif., December 1972.

2. Arango, G., Schoen, E. and Pettengil, R. "A Process for Consolidating and Reusing Design Knowledge," in Proceedings of the Irvine Research Unit in Software Research Symposium, University of California, Irvine, Fall 1993.

3. McCabe, T. J. and Assoc. "McCabe Tools Demo Manual OPEN LOOK®," Tech. Pub. McCabe & Associates, Inc., Twin Knolls Professional Park, Columbia, Md., 1993.

4. Lea, R. C., Jacquemot, C., Pillevesse, E. "COOL: System Support for Distributed Object-Oriented Programming," in Communications of the ACM, Special Issue on Concurrent Object Oriented F,ogramming, Vol. 36, issue 9 (September 1993), CS-TR-93-68. 1993.

An Object-Oriented Paradigm for Reengineering Complex Real-Time Systems

Kwei-Jay Lin*

Department of Electrical and Computer Engineering
University of California, Irvine
Irvine, California 92717

Abstract

Many have studied the issues on building systems using the object-oriented paradigm. Others have worked on how to design and implement *hard* real-time systems; i.e. systems which are used to control and respond to real-time events. Relatively few have tried to use the object-oriented paradigm for designing and building complex real-time systems. In this paper, we investigate the issues of reengineering hard real-time systems using the object-oriented paradigm. We propose an object-oriented real-time model which provides predictable scheduling framework and flexible performance real-time objects. With the model we may guarantee the real-time capability in reengineered systems and achieve desirable performances.

1 Introduction

It has been well recognized that real-time system problems cannot be completely handled by fast processors alone. Although high speed is desirable in most real-time systems, high speed by itself cannot solve all the issues in real-time systems. Real-time systems in general must utilize their resources in a more intelligent way, taking into account of both system capacities and job deadlines. In addition, real-time systems must be flexible so that they can provide timely responses to dynamic real-world events. Resiliency in face of abnormal events is especially important, since some real-time systems are the sole controllers of safety-critical applications and failures to provide timely responses could cause unthinkable disasters. Therefore, real-time systems must be *fast*, *predictable* and *flexible*.

In this paper, we propose an object-oriented paradigm for reengineering complex real-time systems. Using the object-oriented approach allows us to apply the principles of hierarchial structuring and component abstraction, which are essential in building any complex system. In addition, the object-oriented approach promotes component reusability which makes systems easier to maintain and to modify. To meet the strong requirement of graceful system degradation in safety-critical real-time systems, we propose to enhance the adaptability and flexibility of real-time objects. Finally, our proposal has a sound theoretical foundation on the scheduling model so that real-time systems are guaranteed to have a predictable performance.

Our contribution in this work is that we extend the conventional object-oriented paradigm to include the considerations for distributed and real-time applications. Our methodology has special measures targeted toward system reengineering. This is in contrast to most object-oriented methodologies that are more suitable for building systems from scratch. When reengineering a complex system, we should reuse object structures and implementations as much as possible. For this reason, most conventional object-oriented analysis and design methodologies fail to provide a satisfactory reengineering solution. Our methodology, on the other hand, utilizes theoretically-sound real-time scheduling algorithms as the system design backbone, so that the system schedulability can be constantly monitored during the reengineering process. We also apply the concepts of imprecision [4] and polymorphism [3] to the performance capability of real-time objects, so that a real-time object can provide different performances under different circumstances. We thus believe that our object-oriented methodology may provide a better reengineering solution.

*supported in part by the Office of Naval Research under grant N00014-94-1-0034 and by the National Science Foundation under grant CCR-89-11773.

2 Reengineering Complex Real-Time Systems

To help systems make intelligent and flexible decisions, real-time software must be able to *express*, to *maintain* and even to *adjust* timing constraints. Conventional system design languages do not specify timing constraints, as time usually is not a factor in deciding the system's correctness. In real-time systems, the notion of timing constraint is needed to trigger and to schedule real-time computations. Expressing timing constraints is thus an important feature of real-time software. To maintain timing constraints, scheduling algorithms are used to make decisions on resource management. By carefully selecting the right job (e.g. job with the earliest deadline) to execute first, a system may satisfy more real-time constraints. Finally, many external disturbances or interruptions may cause a system to overload. Whenever a timing constraint cannot be satisfied, the system must utilize its built-in flexibility to contain the failure and to adjust the actions and possibly the timing constraint itself. In many applications which deal with control problems, timing constraints can be modified by taking necessary actions. For example, the deadline for stopping a car from hitting an obstacle may be extended by slowing down the car or by switching to a lower gear.

When reengineering real-time systems, another important requirement is the predictability of system performance, whenever the target environment is changed or the system functionality is modified. Conventional approaches on building real-time systems attempt to set up systems so that they may operat acceptably even in the worst-case scenario. To achieve that, they are often hard-coded or hard-wired for those special situations. Such real-time systems are very fragile and also very difficult to reconfigure both statically and dynamically, since even a minor change may disrupt a previously well-tuned execution schedule and cause some deadlines to be missed.

A better approach to enhance the flexibility of real-time software is to change the computation structure so that the amount of work performed is based on the amount of time and resources available. In other words, instead of defining a fixed amount of work to be performed, we can define a set of workloads which are candidates for execution. During run-time, a subset of the workloads is executed using only the amount of time available. Some of the workloads may have dependencies between them when one cannot be executed until another is finished. This defines the precedence constraints. Workloads may also be structured as and/or trees. Each branch of the tree requires a different amount of time and resources, and also gives a different degree of rewards. The system design and scheduling issue is then to select the optimal subset of the workloads which gives the best reward using only the available time and resources.

In real-time systems, the approach can be implemented in several ways. A computation may actively evaluate its timing constraints to select the execution path with the most desirable response time. The run-time system, given global scheduling knowledge, may bind a real-time request dynamically to a server with appropriate time and resources available. Finally, a computation may resort to producing imprecise results if no feasible alternative exists and some response must still be generated. In all these different implementations, the execution time of a computation is modeled as a first-class object so that it can be evaluated or modified if necessary. By unifying the models of time, resource and normal objects, the timing property of a real-time computation can be better controlled.

3 Object-Oriented Paradigm

In an object-oriented system, all entities in the system are defined as objects. An object may invoke methods defined in itself or other objects for the services needed, which in turn may invoke methods in other objects. Each object is defined by a specific class. Classes may form a hierarchy where some classes inherit certain methods from their parent class. Cardelli and Wegner [2] have defined the three basic required elements of object orientation as:

object oriented =
data abstraction + object types +
type inheritance

The definition of object orientation has been extended in [6] for distributed systems:

object oriented =
encapsulation + abstraction +
polymorphism

In the above definition, *encapsulation* means data hiding and access control, *abstraction* means object grouping according to certain properties, and *polymorphism* means the overlapping and intersections of object functions. Objects form a natural model for distributed systems since each object has its natural boundary due to the physical distribution, plus a well-defined interface and also the message passing facility.

In engineeing or reengineering complex systems, two issues must be addressed: *interconnectivity* and *interoperability*. Distributed object structure provides a nice framework in finding the solutions for both issues. Moreover, the object structure naturally accommodates the heterogeneity and automony in most distributed systems.

For complex real-time systems, the above definition of object orientation can be extended further to cover the requirement of predictable performance:

Real-Time object oriented =
encapsulation + polymorphism +
predictable performance

There are two components in *performance*: the response time in method execution and the system schedule for meeting deadlines. Both must be predictable in an object-oriented real-time system. We have discussed the predictable response time issue in the previous section. For the rest of this paper, we discuss the predictable scheduling issue.

When designing real-time systems, system scheduling should not be left as the last step in the system development process. We believe that scheduling considerations should be integrated into the structures of real-time objects as well as the whole complex system. Moreover, it must be considered early in the system design when objects are being grouped and defined. For example, when structuring transportation vehicles, one can group passenger cars and buses in the same class since both have more than two wheels. However, from performance's point of view, it may be more appropriate to group motorcycles and passenger cars together since both may provide a similar kind of performance in terms of speed and convenience. For real-time systems, the class hierarchy defined in terms of performance capability may be more appropriate.

Object-oriented programming facilitates modular management of resources. Each object provides its users with certain resources in the form of methods that constitute the interface of the class. Some methods may be provided by other objects; these methods are bound based on the class hierarchy, and sometimes by the parameters of the invocation. For example, when a '+' method is requested, the actual operation executed may depend on the types of the objects presented as operands and as the result. The binding process and the service provided may differ for integer numbers and for real numbers. In traditional object-oriented systems, the actual realization chosen depends on the class of the object and parameters, and the class hierarchy. Inheritance allows classes to be specified and realized incrementally. Each subclass represents additional knowledge about the objects that are instances of the subclass. The binding of resources to requests can sometimes be performed statically (by using static typing and complete type specification), although many object-oriented systems also allow for dynamic binding.

4 Scheduling Issues in Performance Polymorphism

In real-time systems, the polymorphism concept can be generalized further to include the execution performance as one of the binding parameters. The flexibility is required for a number of reasons when long-lived systems are being considered. First, systems may be reconfigured, and the loads on the systems may change. New versions of a system also may be developed to enhance system capability. Moreover, new environmental constraints, e.g. modifications to a performance specification, may arise from time to time.

The concept of polymorphism has been used for different purposes in the object-oriented paradigm. This substitution based on architectural or performance criteria is a form of polymorphism that has not been considered in conventional systems. Instead of having multiple procedures that perform the same action on objects of different type, we now have multiple procedures that perform the same function based on different environmental constraints. This model of *performance polymorphism* provides a powerful system primitive in structuring flexible real-time software.

Formally, given a function F that must be performed. For this function, we have several implementations $F_1, F_2, \ldots, F_f$ that may be chosen. Choosing one of these implementations makes a resource commitment. In the packing model for system scheduling, it chooses a block of particular dimensions, in resources and time. For the purposes of studying the dimensions of the block, we need consider only the vector $(r_1, r_2, \ldots, r_n, t)$ describing its resource and time requirements. Note that at this point, the time requirement need not be distinguished from any other resource requirement; we have reduced the problem to an arbitrary set of coordinates, one of which happens to be time. Without loss of generality, we therefore describe the *resource vector* henceforth as $R = (r_1, r_2, \ldots, r_n)$.

Each of the choices F_i, therefore, represents a single point R_i in this configuration space. The availability of resources is modeled by a set of constraint inequalities, which will generally be of the form, $0 \leq r_j \leq M$; such an inequality corresponds to assertion, "At most M units of resource r_j may be used." (Obviously, more complicated constraints are possible; they can fit easily into the scheme.) If the point R_i satisfies all the constraints, then F_i may be bound to the invocation F. If not, then F_i is unacceptable, in that it consumes an excessive amount of some resource, and we must consider a different F.

It may happen that several of the F_i's satisfy all of the resource constraints. To choose among them, we assign to each choice a scalar q_i that represents the *figure of merit* associated with making that choice. When jobs are performance polymorphic computations, jobs do not have a fixed amount of execution time, but can be executed for a variable amount of time. Each version defines a reward function which specifies how much reward can be received for a given execution time. In some systems, we often want to evenly allocate resources (especially CPU time) to jobs such that all jobs have about the same reward. In other words, we often want to maximize the minimum reward for any job in the system. This is known as the *knapsack sharing* problem. Brown [1] has proposed an efficient algorithm that requires $O(n^3)$ operations for a problem with n continuous tradeoff functions, and further extended it to handle the cases where the tradeoff functions are piecewise functions.

Another possible objective is concerned with the allocation of resources to maximize the total value among all reward functions. It is known as the *knapsack* problem which is NP-complete in general. Heuristic algorithms have been used to solve the knapsack problem.

5 Conclusions

In this paper, we present an object-oriented paradigm targeted for real-time systems reengineering. The paradigm includes flexible performance and predictable scheduling so that real-time objects can be reused easily. We are currently working on the system design and implementation tools for the methodology.

One of the issues in adopting the object-oriented paradigm for system reengineering is that most existing systems were built by other models. Therefore, many believe that it is impossible to provide a smooth transition into a new paradigm. However, using our performance polymorphism model, a new system could coexist with an old one by treating the old components as another candidate computation to provide the required functionality but with an alternate performance. In this way, we believe our model can easily facilitate system transition, reuse and integration.

References

[1] J. R. Brown. The sharing problem. *Operations Research*, 27(2):324-340, March-April 1979.

[2] Luca Cardelli and Peter Wegner. Understanding types, data abstractions, and polymorphism. *ACM Computing Surveys*, 17(4):471-522, December 1985.

[3] Kevin B. Kenny and K. J. Lin. Structuring real-time systems with performance polymorphism. In *Proceedinngs of Real-Time Systems Symposium*, pages 238-246, Orlando, Florida, December 1990. IEEE.

[4] K. J. Lin, S. Natarajan, and J. W.-S. Liu. Imprecise results: Utilizing partial computations in real-time systems. In *Proceedings of the Eighth Real-Time Systems Symposium*, pages 210-217, San Jose, Calif., December 1987.

[5] C. L. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 10(1):46-61, 1973.

[6] J.R. Nicol, C. T. Wilkes, and F. A. Manola. Object orientation in heterogeneous distributed computing systems. *IEEE Computer*, 26(6):57-67, June 1993.

[7] L. Sha and S.S. Sathaye. A systematic approach to designing distributed real-time systems. *IEEE Computer*, 26(9):68-78, September 1993.

Next Generation Computer Resources Program: Strategic Direction

Rex A Buddenberg
Naval Postgraduate School
Monterey, Ca 93943
budden@nps.navy.mil

The Navy's Next Generation Computer Resources Program (NGCR) is now formally five years old and about seven or eight in conception. The author was asked to review the program for strategic direction; this paper is adapted from that review.

The opinions and recommendations are those of the author's.

The entire paper (less illustrations) is available for anonymous ftp on budden.nps.navy.mil in the /pub/Necessary directory.

NGCR program

The program was organized in SPAWAR as a vehicle to adopt, adapt and otherwise reuse commercial interface standards for mission critical applications including C^3I and combat direction systems. NGCR differs from other standardization efforts within DoD in these respects:

- focus is on commercial standards (FDDI, GOSIP, POSIX, Futurebus+, SQL, PCMT) — different than the traditional Mil-Std approach.
- standards working groups include industry participation, some of which is program-funded. Most Navy labs are involved. Participation from other services has been welcome from the beginning.
- strategy is to target standards as they are formed within industry rather than adoption after they are finished.

Assessment Methodology

A pyramid model was used to judge the NGCR program. This model was arrived at by viewing the evolution of tactical decision support systems as they are evolving from purpose-built stovepipe constructions to a more general purpose, sustainable form. The Joint Maritime Command Information System (JMCIS) evolution is a good example.

Unification thread

The unifying thread that runs through this taxonomy is how each layer treats an atom of data:

- how the communications system treats a packet or message,
- and passes this chunk of data to the operating system's file system,
- how the database management system organizes the data into tables and relations,
- how a correlator fuses different atoms of data into molecules of information, and
- how this information is displayed and operated on, and
- how the necessary security attributes remain attached to the data at each stage in the process.

Understanding of this thread allows us to cut to the core of the both interoperability and multi-level security problems and identify those truly central issues. It also allows us to relegate several standards issues to 'important, but secondary' status.

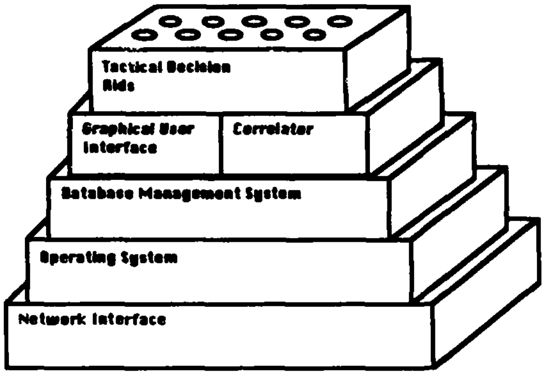


Figure 1. Anatomy of Decision Support Systems

Figure 1 illustrates our building-block anatomy[1].

- Layer 1. All decision support systems have a means of communications in order to receive incoming data and to transmit the decision maker's orders.
- Layer 2. Computer systems require an Operating System which schedules tasks for the central processing unit, organizes and maintains a file system, and controls communications input/output.
- Layer 3. The central function of a decision support system is to file data in a database and then extract appropriate views of the database for the user. In our anatomy illustration, the layers under the database management system are concerned with feeding data into the database. Those layers over the DataBase Management System (DBMS) are tasked with preparing the data and presenting a picture to the decision maker.
- Layer 4. The Graphical User Interface (GUI) is the means for making bits into graphics. When creating a tactical picture in the mind of a decision maker, graphics are far more effective at human communications than the written word so a powerful GUI is a pretty essential requirement in nearly all C3I decision support systems.

The Correlator is the process that fuses, or links, chunks of data in the database to each other. This is the essential process of transforming data into information.

- Layer 5. Finally, the Tactical Decision Aids are those processes that help the decision maker operate on the information and produce decisions. These aids allow operators to visualize 'what if' or they extrapolate existing trends and events into the future. Tactical decision aids are those which suggest resources to dispatch to a certain incident based on knowledge of resources, their capabilities and positions, probabilities of incidents and other factors.

If NGCR is doing its standards job correctly, then there is something to fill in each block and the set of standards hang together as a whole.

What's Right

The NGCR Operating System Standards Working Group and Database Standards Working Group appear to be squarely targeting the respective layers.

Additionally, the Program Support Environment Working Group should provide useful help for the Decision Aids area.

In each case, the assessment is on the relevancy of the NGCR work, not on its quality as that seems to be uniformly good throughout.

The Backplane Working Group suffers for irrelevancy reasons which become clear only when we proceed to the second part of the paper on models.

The SAFENET working group did a fine job in the local area networking standards but suffers greatly from lack of scope.

What's Missing

Three areas are of particular concern in the assessment of NGCR's standards adoption, adaptation and influence effort. Two deal with the scope of existing NGCR work:

- inadequate scope in the networking standards work,
- scant attention to the critical problem of software recycling in the correlator and decision aids areas where COTS software products will only be of limited value,

The third problem is the observation that even if the NGCR program places ideal standards into the Navy inventory, these tools will be of limited use without some serious attention to the rest of the information systems architecture problem. This modularization half of the architecture problem is the subject of the second half of this paper.

Correlator and Decision Aid shortcomings

The correlator and tactical decision aids modules cannot be defined by use of commercial standards and implementations are likely to have high proportions of government-owned software indefinitely. Because the Navy will carry the software maintenance burden, it is vitally important that the Navy modularize this software properly to gain economies through software recycling and sharing of modules amongst programs.

Far more important than the simple importation of COTS products, this part of the

[1] Resemblances between this description and the Common Operating Environment, DODIIS Technical Reference Model and the DISA Technical Reference Model are somewhat more than coincidental.

modularization problem is vital to controlling software costs and maximizing a sustainable combat capability that our sailors can actually use.

The correlator is the module that links different chunks of data in the database together, thereby transforming the data into information. The correlator is *dependent on the sensors* and the nature of the data to be fused. So we need different correlators for different sets of sensors.

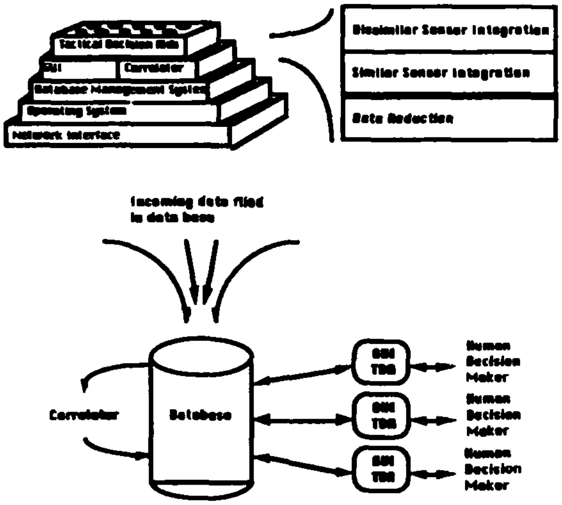


Figure 2. Correlator

The tactical decision aids are highly *dependent on the specific C3I requirements.* Some of those that a destroyer needs may not be appropriate for a CinC's command center ashore. And the decision aids needed by a destroyer engaged in support of a littoral peacekeeping operation are different than those needed by a destroyer protecting a convoy. But the library of tactical decision aids has room enough for all, and there still will be a lot of overlap, so we don't necessary need wholly different pieces of software. Indeed a multi-platform, multi-mission decision support software product is entirely practical.

The JMCIS Darwinian consolidation tactic, with a common library of decision aids, may indeed prove to be a sound approach.

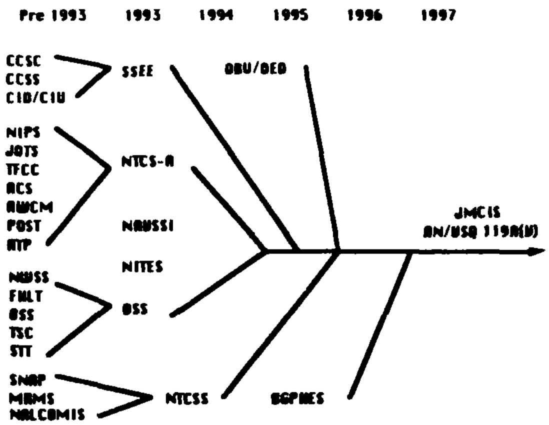


Figure 3. JMCIS program development path

The commercial information systems industry does seem to be producing useful standards for this problem. It appears that the Distributed Computing Environment (DCE) as produced by the Open Systems Foundation (OSF) is highly applicable to enabling the kind of software interchangability that the Navy cannot afford to be without.

Computer language design that allows separation of module declarations and implementations is also highly useful. This feature is common to Modula-2, C, and Ada amongst other languages as this feature enables the ability to distribute modules across clients and servers on a network using remote procedure calls.

Networking shortcomings

A somewhat more egregious NGCR shortcoming is the Navy's failure to come to grips with its multiple stovepipe communications link problem. There is no apparent analog to the JMCIS Figure 3 above for Navy communications systems:

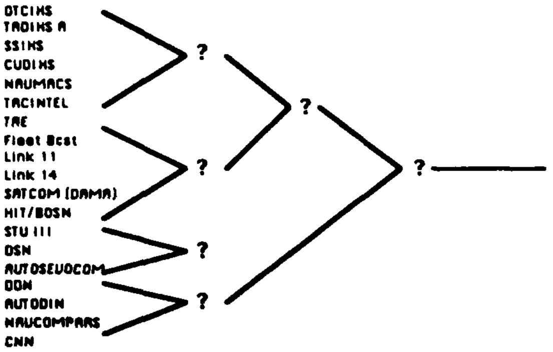


Figure 4. Convergence (??) of Navy Communications

This shortcoming is critical: as programs develop, requirements become better understood and articulated. And requirements change. And the need for the information systems of different systems to communicate with each other becomes more evident. All these requirements shortcomings have reasonably straightforward growth path solutions if the network is done right. On the other hand, if the communications is done wrong, this evolutionary growth path is forever tortuous and stunted.

A further goad should be the rapid evolution in radio-based communications from a situation where DoD owns the connectivity to one where the military will simply purchase the services from commercial vendors. The rapid rise in commercial satellite connectivity planned for the rest of this decade indicates that the Navy-owned circuits, and Navy-unique protocols, will rapidly become unaffordable relics of the past.

For the communications layer, the Navy's needs and the commercially used reference model — the ISO Reference Model — are highly coincident. The internet approach — a complex interconnection of networks of networks, yielding a highly survivable substrate that shares the scarce bandwidth resource — is ideal for a Navy.

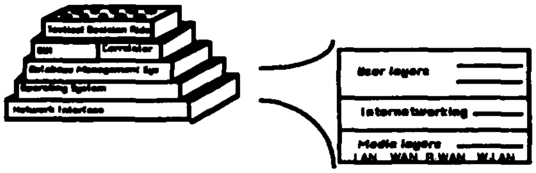


Figure 5. Internetwork reference model

SAFENET does a fine job within this context — for local area networks. But SAFENET's scope is inadequate for terrestrial and radio based Wide Area Networks (WANs) and for wireless LANs. SAFENET's scope and means also did not adequately cover upper layer protocols which provide critical logical interfaces for the operating system, database management system and decision aid layers.

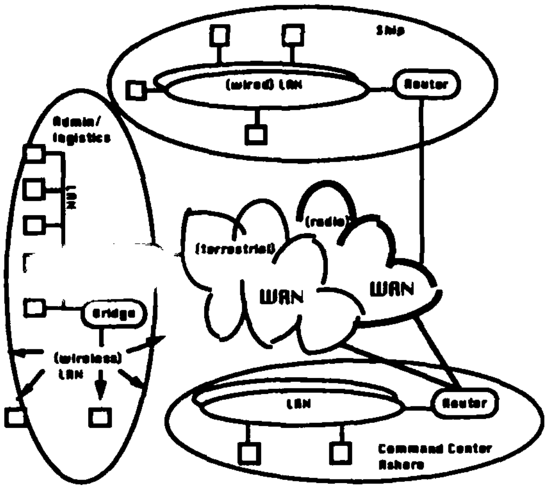


Figure 6. Internet Connectivity Development

Missing is the Navy's influence in Frame Relay and Asynchronous Transfer Mode (ATM) standards development. ATM appears destined to be the protocol of choice in both terrestrial WANs and radio WANs and DoD will probably have to use the protocol as industry provides it.

Missing also is Navy influence in the IEEE 802.11 committee which is attempting to produce standards for wireless LANs which have great potential in military applications.

And missing is Navy understanding and influence of the upper layer protocol development, particularly in the areas of secure electronic messaging.

Correcting these sins of omission would provide the basis for completing Figure 4 above.

Modularization — Part the Second

In the first half of this paper, we presented a 'reference model' or a layered architecture approach to the NGCR standards effort. In this second part, we address a more concrete problem, more typical of those that confront program managers producing information systems[2].

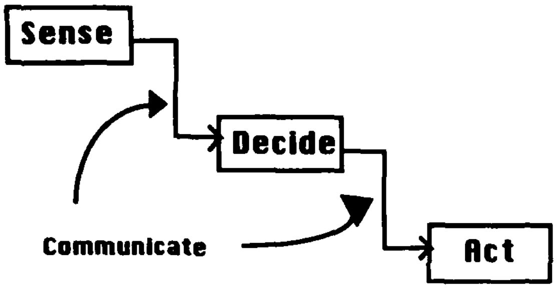


Figure 7. A Concrete Model of Information Systems

Sense-Decide-Act

Let's use this Sense-Decide-Act model as our modularization guide. A typical information system is made up of end systems that include:

- sensors or data collection nodes. These range from radars to human lookouts on a ship, to humans capturing data and filling in a form.
- decision support nodes. These nodes are in the business of transforming data into information into knowledge into wisdom. The term 'decision support' is used deliberately, connoting that the human decision maker remains an integral part of most systems.
- action nodes which carry out the decisions made in the decision support function.
- a network that connects the three kinds of nodes above together.

The interface definitions that get codified as standards are those between the modules.

Complexity

Complex information systems may find these models either:

- nested inside each other. A weapon may be the Act node in a combat direction system. It may also have an entire Sense-Decide-Act view within itself, particularly if it has some autonomy (i.e. fire & forget) requirements.
- chained together. The output of one Sense-Decide-Act sequence may be the input of another. What one information system views as 'action' may be providing the 'sensory' data to another.

The model also is neutral regarding numbers if we do the modularization job correctly. A decision support system will usually have many sensors feeding it; conversely a sensor may feed multiple decision support nodes.

Communications substrate

These end systems (Sense, Decide and Act) all are attached to a communications substrate that carries the output of one node to the input of the succeeding. Use of an internetworked communications substrate again renders the network neutral regarding numbers — any number of sensors, decision support nodes and action entities can be attached to the network providing it is sized appropriately.

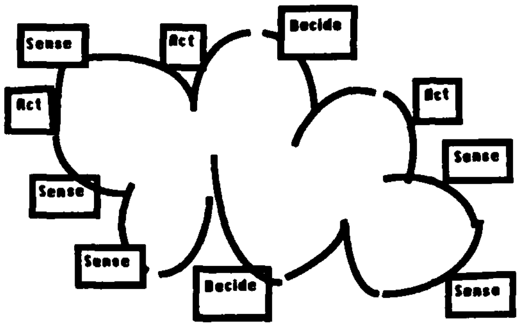


Figure 8. Network substrate

One of the advantages of current internetwork technology is that this capacity problem is fairly easily assuaged — it is usually quite practical to increase the bandwidth of the network arbitrarily and incrementally. Without requiring a system rebuild, and

[2] A purist would point out that the modularization notion — the reason for interface definitions — should preceed the standards which are the means for articulating the interfaces. Indeed, the two must go together and DoD, by concentrating on standards without the accompanying models has the chicken without the egg.

transparently to the end systems attached to it[3].

Abstract to concrete

The following chart is offered as a proposed allocation of function amongst the different modules.

Module Function	Sense	Decide	Act	Network
Decision Aids	Standalone ops only	Library w/ standard calls	Autonomous ops only	None
Correlator	Data reduction	Similar and dissimilar sensor fusion	None	None
GUI	Standalone ops only	CMW	Autonomous ops only	None
DBMS	Data out in form DBMS can use	Messaging savvy	None	None
Operating System	Invisible externally	Standardized interface	Invisible externally	None
Network Interface	- Messaging interface - LAN interface - Mgt MIB	- Messaging interface - LAN interface - Mgt MIB	- Messaging interface - LAN interface - Mgt MIB	- WAN interface - Net mgt

Figure 9. Function Mapping

The next few sections are amplifying discussion to the chart.

Sensor Module

GUI and decision aids. For integration purposes, a sensor need not contain any user interfaces and hence no GUI. In a practical sense, however, most sensors have some sort of command/display unit allowing autonomous local operation; there is no objection to this. Local operation often means that decision aids also reside in the sensor; again, there is no objection, but the issue is irrelevant to the larger systems problem.

Correlation. The sensor should contain a complete data reduction capability so that an updated sensor can be added to the system without also requiring upgrades within the decision support system.

Conversely, the multiple sensor integration functions should reside wholly within the decision support system so that improved correlation algorithms can be inserted in new software releases without requiring sensor changes. Avoid multiple-sensor integration functions within the sensors themselves.

Data management. The critical item is that the sensor must provide data in a format suitable for direct insertion in the decision support system's database. A data standardization problem.

Most sensors will contain some kind of operating system (and perhaps a DBMS). Many sensors lack the complexity needed for a general purpose operating system like Unix. In most cases, the OS used is up to the supplying vendor — since it is wholly contained within the sensor module and the supported code is the vendor's — not the government's — responsibility, this is quite acceptable.

Communications. Each end system, including sensors and the DBMSs within the decision support system, should contain a messaging interface. The ubiquitous choice should be an electronic mail one. If additional options are needed for special purposes, they can be added but any additions should be done without compromising the common denominator.

End systems should generally interface only to a LAN and leave wide area communications (both terrestrial and radio-based) as a function wholly encapsulated within the network module.

End systems, especially sensors, need a management interface. Management information can be categorized into:

- get-info (querying a sensor),
- set-info (controlling the sensor and its configuration),
- traps (the sensor initiating messages about itself).

Given the state of network management technology, it makes the most program development and logistics sense to extend the network management information bases to include the end systems, particularly as the network management protocols have been designed to allow this[4].

[3] The appropriate truism here is that amateurs talk about capacity while professionals discuss availability.

[4] If the communications interface is built, as described, then the backplane within any sensors is irrelevant. The backplane is of concern for logistics and sensor upgrade, but not interoperability reasons. For this reason, we judge the NGCR Backplane Working Group with high marks for execution, but low ones for relevancy.

Action entities

Those instruments effecting actions upon the environment, whether weapons or some other actuator, need a defined interface with the decision support system, but it can be somewhat sparser.

All of the functions above the network are either vacuous or wholly contained within the action entity and therefore not germane to our interoperability problem with the exceptions of the data elements which must be standardized.

The network interface should again be a LAN connection, a messaging interface and a management information base.

Decision support systems

The decision support system is likely to be the most complex module and is likely to contain most of the functions. Decision aids, a GUI, the DBMS, and a complex operating system will all be found here.

Correlator. The partitioning of the data fusion function as noted in the sensor paragraph should persist. This is a key division in maintaining an open system: it's one easily transgressed and hard to fix retroactively. But if the data reduction function remains in the sensor and the data fusion function stays wholly in the decision support system, both can evolve independently and incrementally.

Network interface. The same modularization guidance: a messaging interface, a LAN physical interface and a management information base should all be part of the network interface.

Networking substrate

The networking structure that connects all the above end systems should contain the local area and wide area networking segments required, a management information base to maintain control of the network, and robust messaging support.

The network should not have any requirements in any of the other layers as far as the specific information system is concerned (these elements may well appear internally to the network as part of the network management function). The network function should be confined to transporting data and the network should not be linked to the content of any particular data. In other words, we use the network to transport data and not this network to transport track data, that network to transport intelligence data and the other network to transport EW data.

This modularization allows:

- incremental additions to the network to extend connectivity, increase capacity or improve availability (robustness) of the network without affecting any of the attached end systems,
- allows the end systems to evolve independent of network developments.

Installed base

Updates to the installed base are often implemented with little thought to gaining or maintaining the modularity described[5]. One of the most persistent sins is to violate the correlation partitions — rather than update a sensor and equip it with improved data reduction processing horsepower, the path of least resistance may be to try to sandwich that improved data reduction into the next release of the decision support software.

Over the long term, such expediencies cost more than they save. This is especially true when a sensor is recycled and feeds data to multiple decision support systems — a common case with specialized military equipment.

A common understanding of how Navy information systems are modularized will help multiple, often rather independent, component managers produce and maintain components that can be assembled into usable systems.

Recommendation: Every time a radar or other device is field changed and every time a piece of decision support software is upgraded to a new release, this incremental improvement should include progress toward the partitioning of functions as outlined here.

These things we must fix

The Navy requires an authority with architectural responsibility. Included:

- defining between-program interfaces (e.g. between Aegis and NTCS-A),
- defining the interfaces between the Copernicus Pillars (which will place a clear modular separation between decision support systems and an underlying network substrate).

[5] The software industry is well aware of the effects of the Second Law of Thermodynamics; but it happens to hardware and to modularization too.

- focus on a 'Top Half Dozen' programs.

This focus on architecture should be distinct from the programmatic and budget bureaucracy because it all too easily gets subordinated to that exercise. This is particularly true when the architect is physically located in the same command as program managers, such as a SysCom. Additionally, locating this authority within a particul·r SysCom is insufficient as information systems are produced by all of them.

The scope must extend to at least the combat direction system, C^3I system, and management information system development communities. These unions tend to view themselves as isolated from the others but:

- each is dealing with the same information system architecture problems,
- information systems created and evolved by each community must interoperate with those of the others.

Authority

The authority must be seen as one improving the Navy's information systems capability (and thereby, its combat capability) and not as another stumbling block on a program manager's critical path:

- The authority must manage an **education** program, particularly of program personnel, in open systems, layered architectures, and appropriate modularization of systems. This education should focus on problem analysis (education) rather than solution building (too often training). The education should be career-oriented — not a one-time deal.

Note that I'm using the term *education* as distinct from *training*[6]. A training approach would be to say 'here's a standard, go use it'. An educational approach would provide the background to outline requirements and then select an appropriate mo·ularization and set of standards to define the module boundaries.

The target audience must include senior uniformed and civilian personnel on the N-staff and in the SysComs. It must also include, in greater depth, mid-grade personnel in the SysComs and engineering personnel at the Navy's labs[7]. And if the Navy intends to be serious about cooperation with industry, this educational opportunity must be made available to contractor and potential contractor personnel.

The educational approach should be applied throughout an officer or civil servant's career.

The executive agent must have management influence over an education budget and educational infrastructure to carry out this duty.

- **Technical intelligence**. The Navy too often reinvents things available COTS because it is unaware that solutions already exist in the commercial sector. The Navy needs a KGB — in the Andropov incarnation — to keep both it, and the program managers it services, current in our galloping information technology.

Such an intelligence system need not be clandestine or in any way covert. Open sources will do nicely.

Simple intelligence gathering is insufficient; an ability to evaluate, fuse, and effectively disseminate finished intelligence in a manner responsive to the customer's needs is vital. The intelligence community working in the national security arena has excellent tools and methodology, along with a good understanding of what finished, evaluated, intelligence means — we need to apply tha/ methodology here to our own national technology base.

- **Program planning**. Many of the architectural sins in Navy programs are committed before the Tentative Operational Requirement is ever signed out of the Pentagon. Many more of them are committed by the time the Navy's labs get their tasking. Like software engineering, errors committed the earliest are the hardest and most expensive to correct. Therefore, the architectural authority must be privy to program planning at its ear!iest stages — that is the most effective and leas\ painful place to outline an open systems pattern to build the system to.

The authority should apply the limited resources at his disposal to a half dozen programs that we judge as key to the Navy. This schwerepunkt approach says that if these half dozen programs are all delivered as open, evolvable systems based on the same model, then the rest of the Navy's information systems will come along of their own accord. (This

[6] Webster: education is a systematic study of problems, methods, and theories .... Training is instruction to make proficient or qualified.

[7] As a data point, Naval Postgraduate School provides a one-time shot (no continuing education) that is limited almost entirely to uniformed naval officers at the Lt/Lcdr level.

focused approach does not, of course, prohibit on-request help to other programs).

An informal poll suggests the following as a Top Half Dozen:

- AEGIS Combat System Mk7 — fleet and littoral area air defense
- ACDS Model 5 and NTU/ACDS Model 4 block 0 — combat direction systems
- CSS/C2P/Link 16 — the radio-WAN
- BSY-1/BSY-2 — submarine combat direction system
- JOTS/GOTS/NTCS-A/JMCIS — decision support system
- OSS — decision support system

Conclusion

Admiral Nelson and the Royal Navy achieved mastery of the seas when Nelson threw the doctrinaire Fighting Instructions overboard and replaced them with a shared corporate ethic and understanding of how to defeat the enemy. Open systems cannot be achieved by a directive-oriented, standards-only approach. We must build — through an educational process — a shared understanding of how program components are modularized if we wish to achieve a building-block open systems architecture.

Please note that we are dealing with a dramatically changed environment from twenty years ago in information systems — the bulk of the technology is dual use. If the US Navy doesn't do this first, the same technology is available to other navies.

Recommendation 1 — organizational commitment. Someone in the Navy needs to take charge of the Navy's information system architecture. A simple program advocating standards is not enough.

Recommendation 2 — education of our people. The Navy needs a strong educational program that teaches program management personnel about how to analyze, and modularize information systems problems.

Recommendation 3 — the Navy needs to adopt a modularization scheme common to all information systems expressly to foster the production of recyclable building blocks.

Reuse-based Reengineering: Notes From the Underground

Frank Svoboda
Unisys Government Systems Group
Reston, Va.

Abstract

This paper describes the reengineering process and lessons learned to-date for the Army/STARS/Unisys Demonstration Project (the Demo Project) Application Engineering Team (AET). The Demo Project combines elements of reengineering and reuse through simultaneous Domain Engineering (domain modeling and asset[1] creation) and Application Engineering (reengineering, maintenance, and new development).

Although traditionally assuming the passive role of consumer for Domain Engineering (DE) products, Application Engineering (AE) can actively support reuse (reuse-based reengineering). The concurrent nature of the Demo Project's DE/AE work affords unique possibilities for sharing of information and modeling techniques that might not otherwise occur when the two activities are distributed across space and time. The AET process defined in this paper addresses issues related to systematized reuse during reengineering and continuing software maintenance.

Introduction

Megaprogramming is the STARS vision of process-driven, domain-specific reuse-based, technology-supported systems development. The purpose of the Army/STARS/-Unisys Demo Project is to show the benefits of megaprogramming in an Intelligence/Electronic Warfare (IEW) environment. To this end, the Demo Project enacts simultaneous Domain Engineering and Application Engineering cycles to create and utilize software assets for reengineering and subsequent maintenance phases. A fundamental difference between these modes of operation is that Domain Engineering identifies applicable systems for a given domain, while Application Engineering identifies applicable domains for a given system.

Figure 1 depicts the traditional roles of Domain Engineering and Application Engineering. This model implies that Domain Engineering and Application Engineering are tightly coupled, executing in a strictly sequential manner.

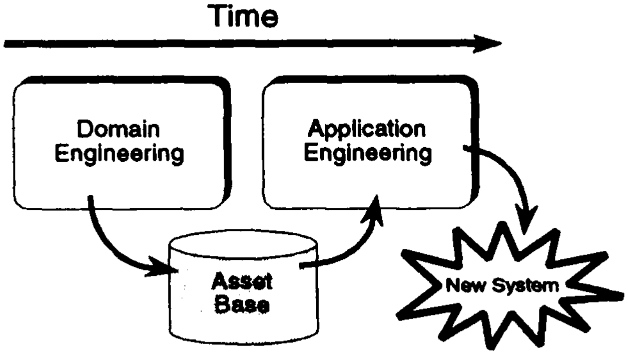


Figure 1: Tradition Domain/Application Engineering Roles

The Domain Engineering (DET) selected an IEW sub-domain — Emitter Location Processing and Analysis (ELPA) — as its *domain of focus* (DOF). The DOF provides the scope for domain modeling and asset creation. The AET, in turn, uses assets developed by the DET for reengineering within the DOF. Figure 2 depicts the Demo Project's model of concurrent Domain Engineering and

[1] An *artifact* is data existing on physical media that conveys information about a software system. A *work-product* is a planned artifact. An *asset* is an artifact of potential value within a given *domain* (i.e., an area of related knowledge or activity). Assets have either been developed for reuse, reengineered for reuse, or certified for reuse as-is.

Application Engineering. Here, AE can operate independently of DE in non-DOF areas, coordinating at asset hand-off.

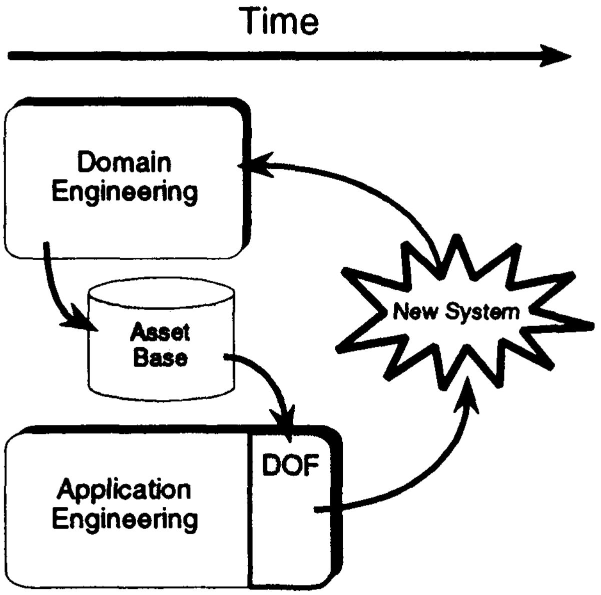


Figure 2: Concurrent Domain/Application Engineering

Additionally, the AET has chosen a broader reengineering scope that encompasses interfacing sub-systems and non-contiguous application areas as well. Within this broader scope, the AET is reengineering not only for near-term maintainability, but also to capture tradeoffs, decisions, and underlying rationales. Providing this information about non-DOF areas to future Domain Engineering completes the DE/AE cycle. It is envisioned that domain engineering and maintenance will eventually be integrated, so that the asset base will continue to evolve through successive maintenance iterations.

Application Engineering Goals

The AET's prime objective is the reengineering of a target IEW system — Improved Guardrail V (IGRV) — to improve maintainability, flexibility, and *evolvability* (the ability of a system to suffer modification). Incorporating the STARS megaprogramming vision into this effort expands these objectives to include the following goals that support systematic reuse for subsequent maintenance cycles:

a. the definition and enactment of a repeatable S/W maintenance process supported by an integrated software engineering environment (SEE)
b. the utilization and assessment of software assets
c. sharing of information and modeling techniques with the Domain Engineering Team

At the time this paper was drafted, the AET had completed Reverse Engineering of a selected non-DOF IGRV subsystem. The sequence of activities for this Reengineering phase and the conceptual underpinnings of the AET Reengineering process are described below.

AET Reengineering Process

The components of the AET Reengineering process are Reverse Engineering, Improvement, Restructuring, and Forward Engineering [1]. We distinguish *maintenance* as "modification of a system's *functionality*" from *reengineering* as "modification of a system's *form.*" In full-cycle reengineering, requirements serve as the linchpin between a legacy system and its reengineered counterpart. In this case, the first significant issues addressed during Forward Engineering are those relating to architecture. The identification and adoption of standard architectures are important components of Reengineering between the life-cycle phases of Requirements and Design.

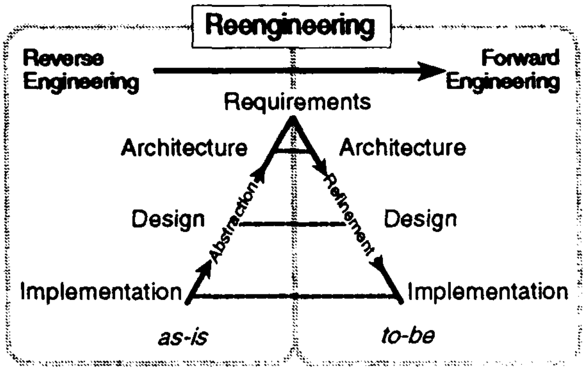


Figure 3: Reengineering Process Framework

Figure 3 (adapted from Byrne [2] and Chikofsky/Cross [3]) depicts the continuum of activities that comprise the AET's Reengineering process. As indicated, *abstraction* is the underlying mechanism for Reverse Engineering, while refinement serves the same purpose for Forward Engineering. The Reverse Engineering and Forward Engineering phases are "mirror images" of each other, progressing backwards and forward, respectively, through the life-cycle steps of Requirements, Architecture, Design, and Implementation. The AET Reengineering approach incorporates improvement and restructuring within the body of the Forward Engineering phase. Separate activities for Reverse Engineering are given in the following text.

Reverse Engineering Process

The AET Reverse Engineering process consists of activities that represent IGRV artifacts at increasing levels of abstraction. This process requires techniques that progress backwards conceptually from relatively concrete code artifacts to more abstract requirements artifacts.

The goal of Reverse Engineering is to model and understand an existing (as-is) system. Since Reverse Engineering is also part of Domain Engineering, both the DET and the AET can exploit this similarity through similar representations, tools, and processes. The scope of Reverse Engineering will vary between teams, however: Application Engineering models a single system, while Domain Engineering models commonality and variation across multiple systems in a given domain.

The first step in defining our reengineering process consisted of defining a procedural model that applies to each life-cycle phase. The Reverse Engineering Process Template below describes this model, which incorporates Byrne's Reverse Engineering Procedure [4] and the Plan-Enact-Learn Cycle of the CFRP [5]. The Reverse Engineering Process Scenario is an expansion of the template across the entire life-cycle. The Demo Project used this scenario to specify tool use and team interaction at given process steps.

Reverse Engineering Process Template

For each life-cycle step backwards from code through requirements, the following steps are performed:

a. Collect, organize, and understand system artifacts. "Hard" artifacts such as code and supporting documentation are gathered and organized to ease retrieval. System knowledge and informal feature descriptions are captured as artifacts.
b. Analyze and model system artifacts. Determine grouping criteria. Interpret artifacts and group into higher-level abstractions. Artifact models, when recorded, become the artifact input for the next life-cycle step in the reverse engineering process (e.g., Structure Charts, as code models, typically represent the functional decomposition of design).
c. Evaluate the artifact models. Determine evaluation criteria and evaluate the artifact models against these. For example, if Structure charts derived during code modeling do not convey any significant information, then it may be desirable to eliminate selected portions from the overall model.
d. Consolidate the artifact models. Reconcile the artifact models with any existing artifacts at the same level. For example, after creating Structure Charts, examine any legacy design artifacts and combine these to produce a complete, accurate, and understandable set.

Reverse Engineering Process Scenario:

I. Preparation

A. Perform line-of-business analysis. Define and scope reengineering effort. Set reengineering context and objectives.
B. Collect and catalog legacy system artifacts. This includes baselining code, design, architecture, and requirements models, trouble reports, test cases and procedures, and supporting documentation.
C. Interview domain experts and record system knowledge.
D. Interview users and record informal features of the system.

II. Code → Design

A. Collect code artifacts and organize according to reverse engineering needs (e.g., map task/function to directories/subdirectories). Derive and capture understanding of the artifacts.

B. Identify criteria for design partitioning and group code modules as design units. Record module invocation and data coupling among code modules (e.g., via Structure Charts, cross-reference, etc.).

C. Create representation of internal module processing and data access using textual description, tables, and/or graphic notation.

D. Update code documentation, including headers, embedded commentary, and other supporting documents.

E. Generate Data Dictionary. Perform balancing and consistency checks on data dictionary.

F. Determine evaluation criteria for design artifacts.

G. Evaluate design artifacts to determine usability, accuracy, and coverage. Hold internal reviews and walk-throughs. Record results and recommended actions for rework.

H. Consolidate design work-products with legacy design artifacts.

III. Design → Architecture

A. Collect design artifacts and organize according to user needs (e.g., map task/function to directories/-subdirectories). Derive and capture understanding of the artifacts.

B. Identify criteria for architectural partitioning and group code modules as architectural units that describe functionality, connections, and interface mechanisms. Create representation of architectural units using textual description, tables, and/or graphic notation.

C. Update design documentation.

D. Update data dictionary. Perform balancing and consistency checks on data dictionary.

E. Determine evaluation criteria for architecture artifacts.

F. Evaluate architecture artifacts. Hold internal reviews and walk-throughs. Record results and recommended actions for rework.

G. Consolidate architecture work-products with legacy architecture artifacts.

IV. Architecture → Requirements

A. Collect architecture and test artifacts and organize according to user needs. Assess informal features (implicit requirements) through user interviews. Derive and capture understanding of the artifacts. (Note: since legacy test artifacts [e.g., plans, procedures, cases, results] were used to validate/verify the original system, these artifacts may provide insight into system requirements.)

B. Identify criteria for requirements model partitioning and group architectural units into data transformations or control transformations (data/control flow diagrams). Identify external data sources/sinks.

C. Identify state/mode actions, events, and triggering mechanisms (behavioral modeling). Depict these via textual description, tables, or graphic notation (e.g., State Transition Diagrams, Statecharts, Event Diagrams, etc.).

D. Depict data organization via textual description, tables, or graphic notation (Data Structure Diagrams, Entity-Relationship Diagrams).

E. Update architecture documentation.

F. Update data dictionary. Perform balancing and consistency checks on data dictionary.

G. Determine evaluation criteria for requirements model artifacts.

H. Evaluate requirements model artifacts. Hold internal reviews and walk-throughs. Record results and recommended actions for rework.

I. Consolidate requirements model work-products with legacy artifacts, recorded domain knowledge, and informal features.

J. Update legacy requirements documentation.

Lessons Learned

The following items informally describe some lessons learned during the enactment of the Reverse Engineering process.

- Choose common terms (domain lexicon) and agree on definitions (glossary). When terms are adopted that take on a specific meaning, it is of critical importance to communicate the meaning of those terms to all stakeholders. This step should be performed early and often. A set of canonical domain terms should be defined and shared. Continuing efforts should feed back information into the lexicon.

- There is no substitute for domain/system expertise. The Demo Project was blessed with readily available domain/system experts and information. In assessing artifact quality, in modeling system functionality, and in identifying implicit requirements, expert help greatly enhanced productivity.

  As with typical legacy systems, additional IGRV information sources included code and requirements documentation only. Current executable code and its corresponding source code as well as accurate code documentation (headers, embedded commentary) were still intact. Neither design nor architecture artifacts existed. Requirements artifacts included the system requirements specification (of questionable accuracy) and the system operator's manual.

- Closer interaction between the DET and AET was anticipated than what occurred. Due to different goals (the DET needed to model commonality and variation of multiple systems within the same domain; the AET needed to model multiple subsystems within a single application), different tools were chosen and interaction diverged. Certain efforts have benefited from cross-fertilization. Design modeling of some low-level units was shared between the DET and AET. The DET architectural modeling group performed reverse engineering using representations to those used during AET Reverse Engineering.

  Use of similar tools and representations eases sharing of information across Domain Engineering and Application Engineering teams; different tools inhibit sharing. *Bridge* utilities (i.e., those that enable information sharing between tools) can help narrow the gap between tools, if the underlying representations are congruent to each other.

- Some typical reengineering steps did not prove as effective as originally thought. Structure Charts were not as useful to architectural grouping as originally thought. Architectural grouping criteria were extracted from requirements and supporting documentation and confirmed with domain experts. For a better designed system, the Structure Chart → Architectural unit progression may have been easier. The AET may use Structure Charts created during Reverse Engineering to identify and reduce module complexity during Forward Engineering.

- The AET did not choose to perform complexity analysis on legacy code. Although this analysis could identify maintenance problem areas, the AET's decisions on which areas to reengineer were based more on user needs and economic factors.

- By looking ahead at requirements as a goal, the AET used legacy requirements as heuristics for determining architectural and requirements grouping criteria. This technique enhanced productivity, but tended to inhibit traceability between life-cycle work-products.

- Dealing with scoping and boundary issues is an important part of Reengineering context setting. Although intuition suggests that coarse-grained architectural improvements may provide the best long-term return on investment, available resources may dictate an incremental approach. The scope of Reverse Engineering may differ from the scope of Forward Engineering.

- The following were identified as sources of requirements:
  a. "Hard" requirements from legacy documentation
  b. Reverse-engineered requirements
  c. Informal features - implicit requirements from a user's point-of-view; at a minimum, a reengineered system must provide the capabilities of its predecessor.

Conclusion

For any engineering effort, reuse and reusability should be embraced as first principles. New development should occur only as a last resort. Opportunities to share knowledge should be planned for and explored at every possible step. As Domain Engineering becomes more accepted, tools, processes, and training will evolve to support coordinated DE/AE efforts. At a human support level, our experience has shown the value of commonality in language, concepts, and representation. Common understanding is at the core of reusability. Through closer interaction and more effective communication among system/domain stakeholders, we can evolve our engineering practices and improve the quality and adaptability of the software-intensive systems we develop.

References

1. Thomas J. Remaley. "Reengineering of Software-driven Systems", *Proceedings: Paramax (Unisys) Systems and Software Symposium*, November 1992.

2. Eric J. Byrne. "A Conceptual Foundation for Software Re-engineering", *Proceedings: IEEE Conference on Software Maintenance*, 1992.

3. Elliot Chikofsky and James Cross. "Reverse Engineering and Design Recovery: A Taxonomy", *IEEE Software*, January, 1990.

4. Eric J. Byrne. "Software Reverse Engineering: A Case Study", *Software — Practice and Experience*, December 1991.

5. Software Technology for Adaptable, Reliable Systems (STARS). *Conceptual Framework for Reuse Processes (CFRP), Volume I: Definition, Version 3.0*, ST^RS-VC-A018/001/00, 25 October 1993.

Reengineering as an Engineering Problem: Conceptual Framework and Application to Community Problems

Peter Feiler, Walter Lamia, Dennis Smith
Software Engineering Institute[1]
Carnegie Mellon University

Abstract: This paper discusses a plan that addresses how the Software Engineering Institute (SEI) may assist the Department of Defense (DoD) in reengineering its large software-intensive systems. This plan is based on a view of reengineering as an engineering problem to improve the cost-effective evolution of large software-intensive systems. This view of reengineering, which takes the whole software engineering process into account, fosters a growth path by leveraging promising emerging software engineering technologies. The paper also highlights the results of an October, 1993 workshop conducted by the SEI, and discusses how the workshop themes relate to the issues discussed in the paper.

1 Introduction

In the last few years, the world has realized that the number of large systems being built from scratch is rapidly diminishing while the number of legacy systems in use is very high. New system capabilities are created by combining existing systems. At the same time, the context in which these systems have been built has changed. Changes range from changes in the application environment in which these systems operate (e.g., new sensors) to changes in hardware and software technologies (e.g., dramatic increases in processor speed and memory, high-level languages, improved methods). Some of the technologies used when these systems were built can hinder the system's ability to evolve to meet ever-changing demands in a cost-effective way. As a result of these problems, a number of technology solutions have sprung up under a variety of labels, including reengineering, reuse, recycling, modernization, renovation, reconstitution, reverse engineering, design recovery, redocumentation, respecification, redesign, restructuring, and retargeting. For a summary of software reengineering technology, the reader is referred to [Arnold 93].

1.1 Definition

In this paper we are building on Chikofsky's work on a taxonomy [Chikofsky 90], the results of the First Software Reengineering Workshop of the Joint Logistics Commanders Joint Policy Coordinating Group on Computer Resources Management [Santa Barbara 92], as well as insights from ARPA sponsored work including STARS, DSSA and from European efforts sponsored under the auspices of ESPRIT, Eureka (Eureka Software Factory), and the Institute for Systems and Software Technology of the Frauenhofer Gesellschaft [ISST 92].

Definitions for reengineering found in the literature include:

- the examination and alteration of an existing system to reconstitute it into a new form and the subsequent implementation of the new form;
- the process of adapting an existing system to changes in its environment or technology without changing its overall functionality;
- modification and possible further development of an existing system;

[1] The Software Engineering Institute is sponsored by the U.S. Department of Defense. This report was funded by the U.S. Department of Defense

- improvement of a system through reverse engineering (and restructuring) followed by forward engineering.

Figure 1-1 illustrates a taxonomy of terms related to reengineering by Chikofsky. In this commonly-accepted taxonomy, software system abstractions are represented in terms of life-cycle phases. Shown are requirements, design, and implementation. The traditional process of developing a system by creating these abstractions is referred to as *forward engineering*. *Reverse engineering* is the process of analyzing an existing system; identifying system components, abstractions, and interrelationships; and creating the respective representations. Redocumentation and design recovery are two forms of reverse engineering. Redocumentation refers to the creation and revision of representations at the same level of abstraction, while design recovery refers to the utilization of external information including domain knowledge in addition to observations of the existing system to identify meaningful higher levels of abstraction. The third process component of reengineering is restructuring. *Restructuring* is the transformation of representations at the same level of abstraction while preserving the system's external behavior. *Reengineering* is an engineering process to reconstitute an existing system into a new form through a combination of reverse engineering, restructuring, and forward engineering.

Reengineering relates closely to *maintenance*, which is generally viewed as consisting of corrective, perfective, preventive, and adaptive maintenance. According to ANSI/IEEE Std 729-1983, software maintenance is the "modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a changed environment." In this paper we use the term *system evolution* to include software maintenance.

For the purposes of this paper, we take an encompassing view of *reengineering* as *addressing the engineering problem of (improving) cost-effective evolution of large software intensive systems, both existing and future, through appropriate application of effective best-practice engineering methods and tools*. Evolution of many existing systems is considered as not being cost-effective and cannot keep pace with changes in the application (domain) environment and changes in the computing environment and software engineering technology. The term *legacy system* has been attached to systems with such characteristics. Changes in the application environment (the external environment the application system operates in) as well as in the implementation environment (the hardware/software platform) have

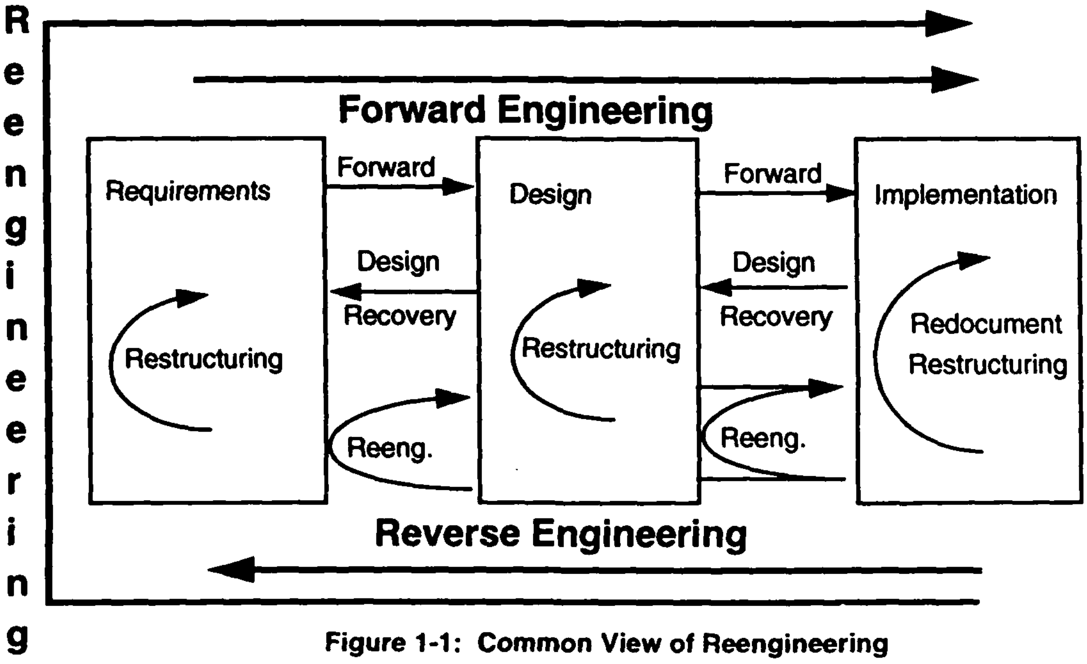


Figure 1-1: Common View of Reengineering

to be assumed as a given and have to be accommodated (*engineering for change*). This need for engineering for change applies to both existing systems and new (or future) systems.

1.2 Context

The focus of this paper is on technical aspects of reengineering. However, economic, management, and acquisition aspects play as important a role in the successful improvement of the capability to reengineer legacy systems.

The cost of incremental change to a legacy system needs to be reduced. Criteria for deciding on the need for reengineering range from heuristics such as age of code and excessive maintenance personnel training cost (as found in a 1983 NIST document) to parameterized cost models (see [ISST 92, Santa Barbara 92]). Improvement in this cost is anticipated by investing more than the minimal amount into reflecting the requested change. The additional investment would go into improving the way the system has been engineered with the result of smaller incremental cost in the future. If several legacy systems have to be reengineered, their similarities can be captured in a common reusable architecture, treating them as a family of systems rather than isolated point solutions. The cost models for reengineering, together with better understanding of the effectiveness of different engineering techniques, will allow software engineers to make reasonable engineering tradeoffs as they choose a particular evolutionary reengineering strategy for a legacy system.

Engineering effectiveness is influenced by how well an organization is able to manage its engineering process and improve its engineering capability. SEI has provided leadership for government and industry to improve these organizational software process capabilities through work on the Capability Maturity Model (CMM) and its use as an assessment and improvement tool. In the context of this paper we assume that the reader understands the relevance of such capabilities for an organization's ability to systematically, efficiently, and effectively reengineer legacy systems.

Successful improvement of legacy systems through reengineering also requires attention to improvements in the acquisition process and to legal concerns. The Joint Logistics Commanders Joint Policy Coordinating Group on Computer Resources Management is holding a workshop series to address acquisition issues at the policy level. For further discussion of these and other inhibitors to successful transition of improved software engineering practice see the work done on transition models by SEI and others [Przybylinski 91; Leonard-Barton 88].

2 A Reengineering Framework

In this paper we have cast reengineering as an engineering problem. Problem solving involves an understanding of the problem, i.e., a clear understanding of the root causes in terms of its existing state, an understanding of the desired state, and a path (plan) to evolve from the current state to the desired state. Figure 2-1 illustrates this. The current state reflects properties of the existing system and the process by which the system is engineered (developed and maintained). A subset of those properties is undesirable, reflecting the problem to be solved. System understanding reflects the process of creating and maintaining an understanding of a system (through analysis, elicitation, and capture). System evolution represents the engineering activity of migrating the existing system to the desired state. Based on an understanding of the current and desired system state and available (re)engineering technology, an analysis making engineering tradeoffs by considering technical, management, and economic risks and constraints results in a (re)engineering plan. During the execution of this plan (i.e., the actual evolution of the system through engineering activity), the plans may be reassessed taking into consideration changes in the context (e.g., technical changes such as promising new technologies or economic changes such as budget reductions or increases).

2.1 The Current System State

The root causes for the lack of cost-effective evolution fall into two categories: management of the engineering process and the engineering process itself. Management of the engineering process is addressed by SEI's work on CMM and will not be elaborated here. The second category represents technology root causes, i.e., the engineering process, methods, and tools. It will be the focus of further discussion.

The technology root causes manifest themselves in a number of ways. Some examples are:

- Data structures not cleanly implemented. Assumptions that a specific element of shared memory (e.g., Fortran COMMON) is used as the communication mechanism.
- System representations such as architectural and design descriptions reflecting the application domain and the implementation approach may never have been created or documented; the documentation (and sometimes even the source code) is out of date.

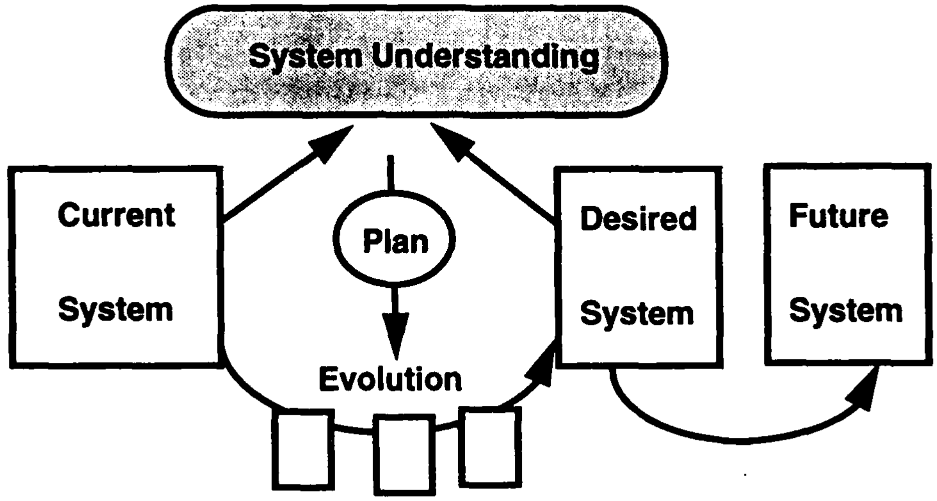


Figure 2-1: Reengineering Problem Solving

- Assumptions about the application environment have been hardcoded in the implementation. Examples include assuming a point solution including fixed number and types of real-world objects.

- The computing environment evolved through several generations. For example, early hardware platforms were memory-limited, resulting in a number of sometimes (in today's view) convoluted implementation "tricks," such as overlay, instruction reuse, and cryptic user interaction. No operating system support was assumed. Today's computing environments typically consist of COTS standard operating systems, DBMS, window systems, and networking support, and are geared toward a high degree of interactiveness and "user-friendliness."

- The implementation technology has evolved from machine code with absolute addressing; to symbolic assembler, high-level algorithmic languages (COBOL, FORTRAN, ALGOL); to languages supporting data abstraction, modularity, information hiding, concurrency support, data modeling capabilities, etc. Design and implementation methods have been coming and going, each leaving its trademark in the code of legacy systems. This code may or may not accommodate the changes demanded from systems today.

Legacy systems also have a number of properties that are worth preserving. Examples include:

- Legacy systems are deployed and have undergone the scrutiny of real users with respect to their functionality meeting their real needs.

- Nonfunctional properties such as performance and accuracy have been fine-tuned.

- Corrective maintenance has resulted in "hardened" code and a wealth of test and validation capabilities.

- System history exists in the form of original designers, current and past maintainers, as well as bug report and change order records.

In many cases some of the root causes and their implications may be understood by some experts, but are not documented and available to the majority of software engineers. Information about systems is quite limited, usually to the source code and/or executable, an operations manual, and people maintaining the system.

2.2 The Desired System State

The desired system state is a combination of properties of the existing system to be maintained, properties expected of a system as part of state-of-the-art software engineering practice and implementation technology, and properties that have their roots in changing environments and are reflected in the system history, but may not have been explicitly expressed by the system user. Examples of maintained properties are functionality, performance, and accuracy. Examples of properties resulting from best practice software engineering and implementation technology include portability, modularity, structure, readability, testability, data independence, documented system understanding, openness (open system), interoperability, and seamless integration. Properties that address continuous change and provide flexibility include localization of information regarding certain different types of change in both the application domain and the implementation, introduction of virtual machine abstractions, and parameterization (dynamic as well as generation technology), COTS, and reuse of components. Properties that encourage reuse of existing engineering know-how include the existence of domain models, domain-independent software architectural principles, domain-specific architectures, and adaptable components.

The desired system state may be known to system users, system maintainers, original system builders, and best software engineering practice experts. The customer (user) may not necessarily be aware of all the potentially desired properties and may only be willing and able to invest in some. Some desired properties can be provided with proven technology, while others depend on emerging technology whose maturity for practical application has not been demonstrated.

2.3 System Understanding

The current state of an existing system and its desired state represent an understanding of the system. This understanding is based on artifacts of the existing system; knowledge and experience with the system as it may exist in users', maintainers', and original builders' heads; and documented system history in the form of bug reports and change records. Figure 2-2 illustrates the sources of information for system understanding. The artifacts are source code, manuals, and the executing system. The knowledge and experience with the system include understanding of engineering decisions, rationale, and possible or considered alternatives, as well as undocumented history and (typically nonfunctional) properties such as performance, robustness, work-arounds, etc. History provides insight into robustness of system components, types, and frequency of changes in the environment (and implementation).

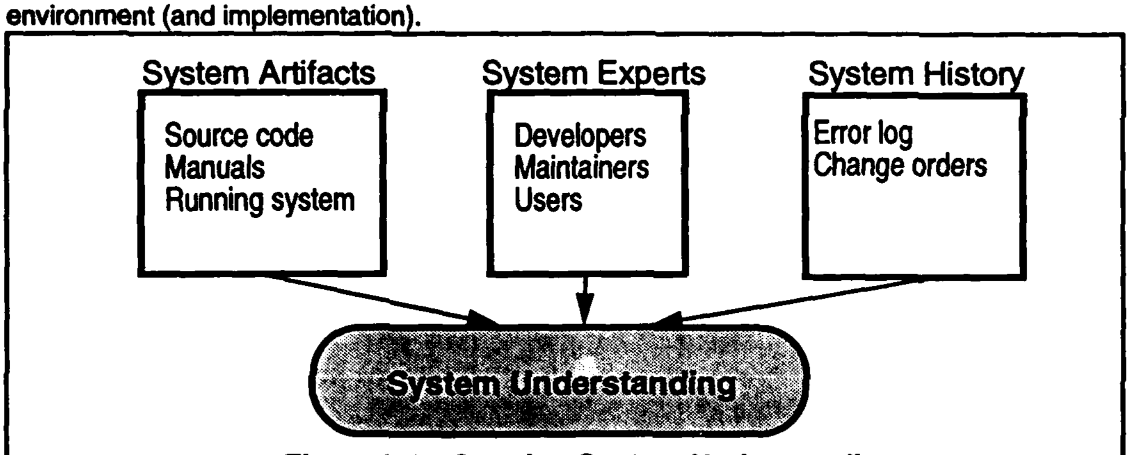


Figure 2-2: Creating System Understanding

Capture, representation, currency, and accessibility of this system understanding is a big challenge. Figure 2-3 illustrates a framework for representation of such system understanding. A central component of system understanding is the system design records that document system representations at different levels of abstraction. This is complemented with rationale for design decisions, the software engineering process and methods used, and the evolution history. Let us first elaborate on models of (software) systems.

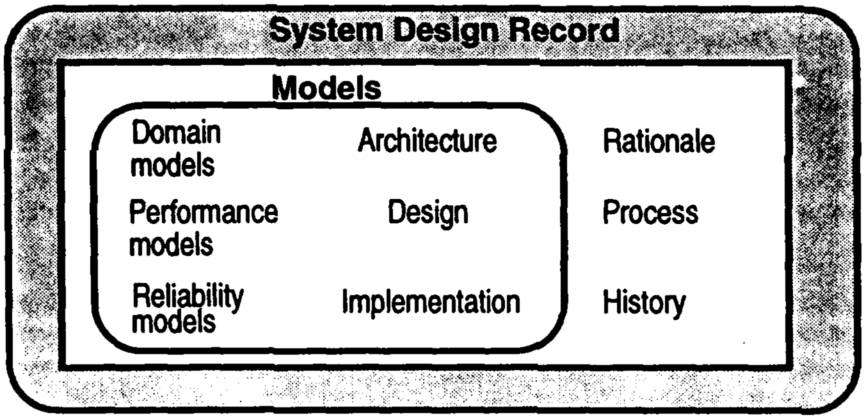


Figure 2-3: Representing System Understanding

These representations are *models* of the system. Models reflect views of the system focusing on certain aspects with different degrees of detail. The purpose of a model is to present a view that is understandable, i.e., not too complex. This is accomplished by the model capturing those abstractions that are relevant from a particular perspective. Some models focus on architectural issues while other models focus on data representation, behavioral, reliability and performance aspects of a system. Examples of models are domain models, domain-specific architectures, real-time timing models such as rate monotonic analysis (RMA), performance models based on queuing theory, etc.

Models have different degrees of formality and may have the ability to be executed. The models may reflect designs (i.e., the notation they are expressed in needs to be transformed into executable implementations), or they may be executable and capture all the desired user functionality and can act as prototype implementations, which can be made more robust or efficient through reimplementation (i.e., transformation into a modeling notation that more appropriately satisfies the need).

As more than one system is considered, models can show their similarities and differences. Systems can be grouped into families. Some models focus on information about the application domain (domain models) while others focus on the implementation architecture. Domain models and domain-independent architectural modeling principles are combined to create domain-specific architectures. Those architectures are populated with components and adapted to the particular application needs. The result is a technology base of models that can be (re)used for a number of systems, leveraging existing engineering know-how. Domain analysis and architectural analysis contribute to the population of this technology base, while application engineering can get adapted to utilizing these models (see Figure 2-4). Furthermore, the technology base can be expanded by the emergence of new modeling concepts, e.g., safety modeling.

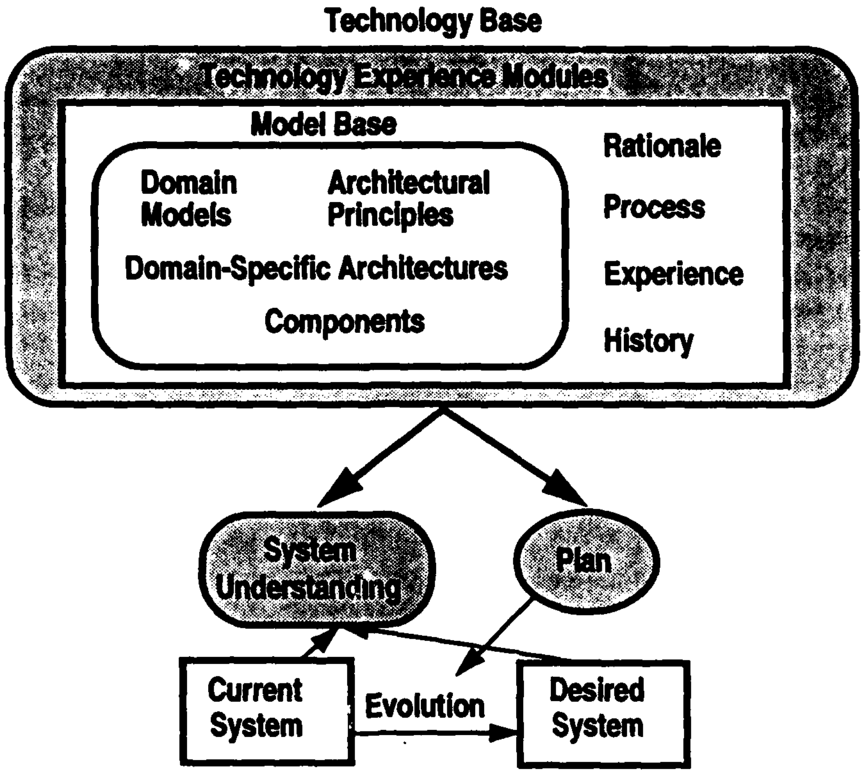


Figure 2-4: Engineering Technology Base

While some models represent the executing system itself, other models reflect constraints the system must satisfy. Those are models used to validate desired system behavior. Examples of such models are assertions validated in design reviews or verification, or translated into test suites and test data validating the behavior of the running system. When reengineering a legacy system, such test and validation models exist and have stood the test of time. They can be leveraged for verification and validation of the desired system. Depending on the particular migration path to the desired system, alternatives to full regression testing may be considered. One example is validation of functional equivalence at a certain level of abstraction through comparison of event traces [Britcher 90].

Engineering decisions, rationale, and alternatives complement these models. They may be captured through elicitation processes such as IBIS [MicroComputer Corporation (MCC)]. The models together with the engineering knowledge are known in other engineering disciplines as experience modules.

In this idealized view, the amount of engineering information available to the engineer grows tremendously, resulting in information overload. In order to cope with this situation an intelligent intermediary (intelligent engineering assistant or engineering associate) will become essential to the successful utilization of the system understanding. Technologies that are potential contributors to this notion of intelligent assistant include case-based reasoning and intelligent tutoring.

2.4 Evolutionary Migration Path

The understanding of the system, both the current and the desired system state, is the technical basis for determining the particular reengineering strategy to be chosen. It requires analysis, considering alternatives, and making engineering tradeoffs. Such a technical engineering analysis consists of two major components: choosing the degree of legacy leverage, i.e., what can be taken over and what has to be newly created; and choosing the approach for migrating over to the desired system, i.e., how to introduce the changes into the system. The reengineering case study by Britcher [Britcher 90] nicely illustrates that no single approach is appropriate, but engineering tradeoffs need to be considered.

Legacy leverage refers to the ability to utilize (recycle) as much as possible of the existing system in the process of evolving to the desired system. Both the existing and the desired system can be described in terms of a collection of models. For the legacy system, code exists. Other models may have to be derived from the code or other information sources. Certain abstraction may not exist in the legacy system or may reflect undesirable properties. The goal is to eliminate undesirable properties while at the same time introduce desirable properties. Choices have to be made as to which legacy system models to ignore, which ones to transform, and which ones to leave intact. This is illustrated in Figure 3-5. The choices are driven by our understanding of the legacy and desired system properties as well as their reflection in the different models. In concrete terms this means that in some cases, undesirable properties of legacy systems can be eliminated by massaging the code or transforming the data representation, while in other cases a new architecture or data model has to be developed and only a few system components can be translated into the new implementation language.

The change can be introduced in a number of ways. The following are three classic approaches, but hybrid approaches are possible:

- **Big Bang Approach:** The desired system may be built separately from the legacy system, although parts of the legacy system may have been recycled. Once completed the new system is put into operation while the old system is shut down.
- **Phase-out Approach,** also known as **Incremental Development:** The architecture of the desired system may be created and a skeleton implementation developed. A mapping between the data representation of the legacy and the desired system, implemented as a two-way transformation filter allows the skeleton desired system to run as a shadow of the "live" legacy sys-

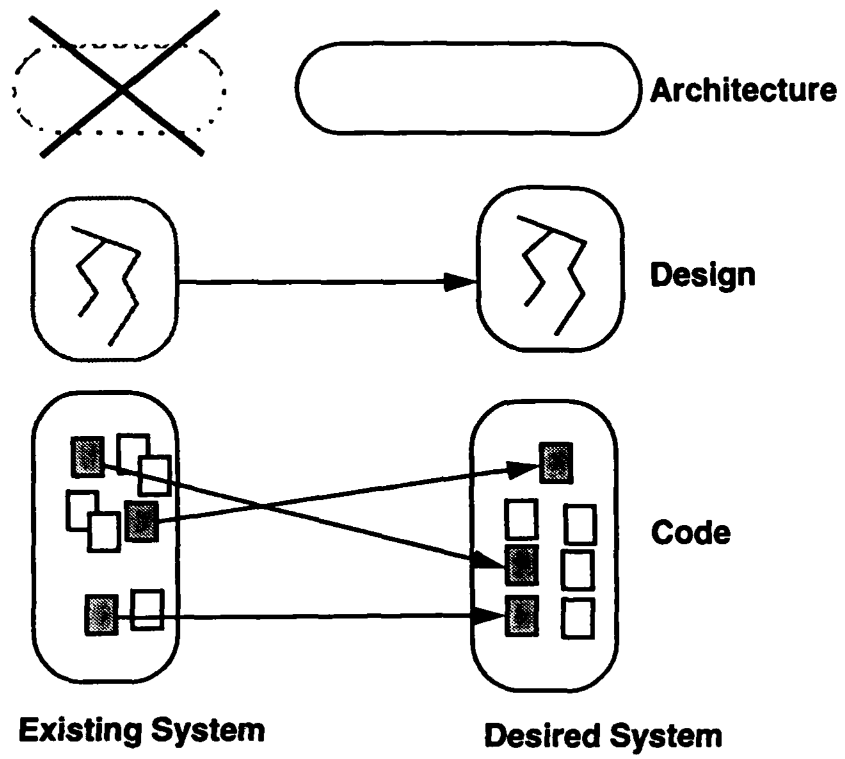


Figure 2-5: System Evolution

tem, while parts of the desired system implementation are completed and incrementally added to the skeleton. This approach incrementally phases out pieces of the legacy system.

- **Phase-In Approach**, also referred to as **Evolutionary Development**: The legacy system code may be restructured to introduce modularity and partitioning. Desired system properties are incrementally introduced into the existing system resulting in an incremental evolution of both the architecture and the system components.

Validation of the desired system can utilize existing testing capabilities. Validation can be decomposed into validating that the desired system still provides equivalent functionality and detection of bugs in the reimplementation.

The choice of the particular reengineering strategy is affected by the risks the alternative approaches. Risks to consider are:

- Perceived and actual undesirable and desirable system properties
- Ability to eliminate or reduce undesirable system properties
- Maturity of technology inserted into the system
- Introduction of new technology to system maintainers (reengineers)
- Impact of introduction of the reengineered system
- Impact of system changes on performance and robustness
- Cost and time of reengineering

In summary, reengineering is an engineering activity that involves system understanding and evolution through application of appropriate engineering practices. The framework outlined here does not promote particular techniques but accommodates emerging technologies as they mature.

2.5 An Engineering Framework

The framework presented above in the context of reengineering can be used as an engineering framework for software intensive systems. A full discussion of this point is beyond the scope of this paper. The following characterization of different software engineering processes and paradigms serves to quickly illustrate the validity of this claim:

- **New system development**: The system to be improved in the application environment may be a system performing without computer support. This legacy system has desirable properties to be maintained and undesirable properties to be overcome. For example, for many information systems the data model of the legacy system, though not documented, may be directly applicable to the desired system. In traditional life-cycle terminology this is referred to as the requirements phase. Software recycle is applicable only if parts of the legacy system are computer-based. For the introduction of the new system the same migration alternatives may be considered as discussed in the context of reengineering.

- **Reengineering of future systems**: Change in the application environment and the implementation environment are givens. When a new system is being defined, customers often focus on the functionality needed to address their particular problem at that time. Many of the types of changes that will occur over the lifetime of the system and their implications on desirable system properties are not considered during requirement definition. Reengineering of future systems implies that engineering for change and up-to-date maintenance of a system understanding (system design records) occurs from the outset. Engineering for change requires an understanding of commonly-accepted changes as well as an anticipation of paradigm shifts due to new technology and localization of assumptions about certain environment constants.

- **Open systems**: The open systems concept has gained momentum over the last few years, as reflected in organizations such as X-Open, Open Sysιems Foundation (OSF), and the User Alliance for Open Systems. This concept permits interoperability, allows rapid technology insertion and upgrade, encourages alternative solutions to be applicable, and provides one solution applicable in a number of systems. Characteristics of open systems are modularity and standard interfaces. These are desirable properties of both legacy and future systems as they reduce system cost.

- **Reuse**: Reuse is an engineering activity that focuses on the recognition of commonalities of systems within and across domains. It consists of the creation of models with different abstractions (ranging from code components to domain models) and their use during the engineering of an application. Thus, the focus is on the growth and utilization of the technology base.

- **Evolutionary development**: Evolutionary development focuses on designing the architecture of a system in such a way that the capabilities offered to the user can grow incrementally. New capabilities may be introduced through prototyping of new system components (possibly utilizing different implementation technology). Such prototypes interoperate with the operational system and may get hardened through incremental reengineering.

- **Megaprogramming**: Megaprogramming focuses on recognition of system commonalities at high levels of abstraction (e.g., architecture) and creation of system instances through parameterized automatic composition or generation.

- **Model-Based Software Engineering (MBSE)**: The objective of MBSE is to improve the effectiveness and efficiency of producing software intensive systems through better utilization of engineering experience and system understanding. MBSE focuses on the use of engineering product models as the primary means for improving the construction and maintenance of software.

3 SEI Workshop: October, 1993

The SEI conducted a reengineering workshop on October 26-27, 1993 to identify work which is underway, and to develop an understanding of needs which are currently unmet and which can be addressed by the SEI or others. Invited participants included broad representatives of the DoD, commercial and academic community. Although a large number of themes were discussed, there was broad consensus on a number of basic needs, together with some ideas on logical next steps.

The October workshop will be followed up by a larger workshop in May, 1994 at which a number of initial themes will be developed in greater detail, and at which additional community resources will be identified.

The needs expressed by the practitioner community were quite consistent with the overall approach outlined in this paper. Some of the major points discussed at the workshop are outlined below, together with preliminary ideas on where the work may be followed up.

The expressed needs included the following:

- **Definition of conceptual frameworks and reference models.** Several groups are beginning to develop a common conceptual understanding, including IEEE and follow-up work to the JLC Santa Barbara workshop (Santa Barbara 92).There is a need for continuing efforts by these groups to help clarify the fundamental conceptual issues of reengineering. The concepts of evolutionary migration path and the model based approach outlined in this paper can offer some contribution to these efforts.

- **Domain engineering perspective.** Consistent with the approach outlined earlier in this paper, the workshop identified the need for and potential benefit of incorporating the perspectives of domain engineering and architectures into an overall understanding of reengineering. These perspectives offer promise for helping to evolve from an ad hoc orientation to the planned, evolvable engineering of legacy systems. Among the groups which are currently focusing on these issues are SEI and DSSA. In addition, this is a promising area for cross fertilization with STARS, DoD reuse initiatives, PRISM and CARDS.

- **Engineering process models.** A number of organizations and companies have developed process models for reengineering. These process models offer a certain degree of commonality. An analysis of the coverage of the different process models, together with empirical data on experiences with these models can be of strong benefit to the rest of the community.

- **Economic and cost benefit analysis, and metrics.** There is a strong need for better data on cost benefits, a better understanding of reengineering economics issues, together with more refined and easily accessible metrics. A draft reengineering economics handbook (MIL-HDBK-REH, Draft, 93)has been developed by the Reengineering Economics Panel from the Santa Barbara workshop. This handbook provides an important baseline for economic issues and will be refined based on experience. In addition, empirical data on actual uses of cost benefit analyses and the application of metrics programs are urgently needed.

- **Decision support systems and project selection criteria.** A recurrent theme concerned the requirement for better decision support models including a consideration of technical, economic and transition issues. There is a need for comprehensive work to integrate these perspectives into a common model to help in decision making on when and how reengineering can provide benefit.

- **Case studies and lessons learned.** Practitioners cited the potential usefulness of being able to learn from the successes and failures of those who have attempted similar efforts in the past. The follow-up workshop will discuss the feasibility of organizing such a program. Systematic case studies can represent one step toward the development of an organized community of reengineering practitioners.

- **Management and transition issues.** A number of critical success factors which distinguish between successful and unsuccessful projects have been management and transition issues, including the management of expectations. A number of the insights gained from transition management and CASE adoption are relevant and can be codified for application to the reengineering domain.

4 Conclusion

Reengineering has been presented as an engineering problem. As such, reengineering draws from a number of software engineering technologies. Advances in best practice of (re)engineering benefit from innovative research (i.e., creation of new technology solutions); analytical research (i.e., recognition of promising technologies and technology trends); and engineering research (i.e., the maturation of technology into engineering use). Evolution of conceptual frameworks for subareas of software engineering can be successfully leveraged through technical leadership in selected community forums. In the context of reengineering, a model outlines a roadmap for improvement in the effectiveness and efficiency of reengineering systems.

References

[Arnold 93] Arnold, R.S., *Software Reengineering*, IEEE Computer Society Press, Los Alamitos, CA, 1993.

[Britcher90] Britcher, R.N., "Re-engineering software: A case study," *IBM Systems Journal*, Vol. 29, No. 4, 1990., pp. 551-567.

[Chikofsky90] Chikofsky, E.J. & Cross II, J.H., "Reverse Engineering and Design Recovery: A Taxonomy," *IEEE Software*, January 1990, pp. 13-17.

[Przybylinski91] Przybylinski, S.R., Fowler, P.J., and Maher Jr., J.H., "Software Technology Transition," a tutorial presented at the 13th International Conference on Software Engineering, Austin, TX, May 12, 1991.

[IEEE90] *IEEE Software*, Editor-in-Chief: Ted Lewis, Published by the IEEE Computer Society, Elsevier Science Publishing Co. Inc., New York, NY, May 1990.

[IEEENews] Software Engineering Technical Committee Newsletter, IEEE Computer Society/TCSE, Editor: Samuel T. Redwine, Jr., Vol. 11, No. 3, January 1993.

[ISST92] Witschurke, R., "Wiederverwendung in der Informationsverarbeitung: Re-use, Re-engineering, Reverse Engineering," ISSN 0943-1624, Institut fur Software und Systemtechnik, Universitat Dortmund, Germany, December 1992.

[Leonard-Barton88] Leonard-Barton, Dorothy, "Implementation as Mutual Adaptation of Technology and Organization." *Research Policy*, 17 (5), pp. 251-267, October 1988.

[MIL-HDBK-REH Draft93] Department of Defense, *Reengineering Economics Handbook*, Draft, September 1993.

[R-EvHa90] Rock-Evans, R.; Hales, K.: *Reverse Engineering: Markets, Methods and Tools.* London: Ovum LTD. 1990.

[Santa Barbara92] Santa Barbara 1, "Back to the Future Through Reengineering," Joint Logistics Commanders Joint Policy Coordinating Group on Computer Management, Santa Barbara, CA, November 1992.

[SEISymposium92] *Capability Maturity Model (CMM) Real-Time Extensions*, Daniel Roy, Software Engineering Symposium, Carnegie Mellon University, Software Engineering Institute, Pittsburgh, PA, September 1992.

[SPC/VCOE92] Reuse Adoption Guidebook, SPC-92051-CMC, Version 01.00.03, Software Productivity Consortium Services Corporation, 1992.

Current STSC Reengineering Projects:

MIL-HDBK-RAH Application Findings, Reengineering Project Planning Process, and STSC Reengineering Survey Results

Michael R. Olsem[1] and Chris Sittenauer[2]

[1]Science Applications International Corp. [2]USAF Software Technology Support Center

OO-ALC/TISEC
7278 4th Street
Hill AFB, UT 84056-5205

RAH Analysis

In early 1992, the STSC published a process for matching legacy softwa; systems with reengineering strategies. Since then, the DoD's Joint Logistics Commanders Workshop on Reengineering in Santa Barbara, California (September 1992) used this process as the basis for the reengineering strategy selection section of the draft DoD MIL-HDBK-RAH (Reengineering Assessment Handbook). Comptek is now under contract to the USAF Cost Agency to produce version 1.0 of the RAH by December 1993.

The authors have been using a draft version of RAH while assessing various legacy software systems for reengineering at several USAF and DoD organizations nationwide. Based upon our findings, we are proposing changes to the RAH. But overall, we find the results of the RAH analysis to be pretty consistent with those reengineering strategies we would have chosen without the handbook.

This paper won't dwell on the details of the RAH as that subject is being discussed in another paper at this conference.

Reengineering Surveys

Besides testing the draft RAH by assessing reengineering projects, the STSC has initiated an in-depth analysis of the correlation between reengineering strategy, maintenance environment, and potential return-on-investment (ROI). We have developed two surveys for DoD reengineering projects. Over 100 DoD reengineering projects have been identified and are participating in this survey.

The first survey targets reengineering projects still being planned or underway. This survey tries to understand the reasons an organization attempts a reengineering project. We correlate this information with the reengineering strategy the organization is most likely to use versus what RAH would suggest.

The second survey targets completed reengineering projects. This survey looks at how "successful" the reengineering project was in terms of maintenance efficiency and reduced costs.

Both surveys are now being mailed out and results should be firm by the February time frame of the NSWC conference. Copies of both surveys have been added as an addendum to this paper. The reader's comments would be most welcome.

Planning

The actual reengineering is easy. The planning for successful reengineering is hard.

Matching an appropriate reengineering strategy to legacy software systems is but one of the last steps necessary for a successful reengineering project. The planning and preparation prior to the actual reengineering is absolutely critical. Unfortunately, most of the USAF organizations we advise are far to eager to rush right out and buy a reengineering tool. This, we believe, is analogous to shopping for groceries without a list of projected meals. Not only do you rarely return with all you need, but you end up purchasing items you don't need at all.

In STSC's *1993 Software Reengineering Technology Report*, we propose a 9 step preparatory planning process for reengineering projects. (We've included a copy of this process as an addendum to this paper.) We will be delivering a half-day tutorial based on this process at the DoD's 1994 Software Technology Conference (STC) in Salt Lake City, Utah and at the 12th Annual Natiional Conference on Ada Technology (ANCOAT) in Williamsburg, Virginia.

In order, the reengineering preparation steps are:

1. Evaluate your organization's business needs

2. Define your organization's development/maintenance environment

3. Form a reengineering team

4. Create a standard set of software metrics

5. Ensure your standard testbed/validation suite is current and complete

6. Analyze your legacy systems (This is where the RAH becomes useful)

7. Analyze the available COTS/GOTS reengineering tools. (Something the STSC will gladly help you with.)

8. Create a reengineering implementation plan

9. Train your reengineering team, users, and maintenance staff

Preparing to Reengineer

(Extract from STSC's 1993 *Software Reengineering Technology Report*)

Introduction

There is a core process that every organization should follow when reengineering. Automation and tools can only support this process, not preempt it. Again, this comes back to the realization that reengineering is not a silver bullet that will solve all of your maintenance problems. The real work must come from you. Don't make the same mistake the software community made with CASE, AI, OOA, etc. Good technologies do not necessarily make for good software. Reengineering, without a target maintenance environment (i.e. well-defined and consistent with your organization's CMM goals), will waste your time and money, leaving you with the same problems you now face and possibly worse off if your experienced maintenance staff must now contend with code they no longer understand.

Reengineering tools will prove a great help in moving your legacy systems to a new, hopefully better, maintenance environment. But you, and your organization, must define this environment, and how to move your legacy systems there. We propose the following core process, or approach, to preparing for any software reengineering project.

Step 1 - Evaluate Your Organization's Business Needs

To begin with, you need to seriously look at your organization's business needs and goals. Although this is the beginning of business reengineering, it's also very important to your software reengineering projects. Your organization's business needs (or the outcome of your organization's business reengineering process) should dictate what your development/maintenance process will look like. Besides making logical sense, tailoring your process to meet your business needs is the only argument that will carry any weight with upper management. You know them. They're the ones that will ultimately fund your efforts and back you up during the initial dark days when the enemies of change will be charging at you from all sides.

Step 2 - Create Your Development/Maintenance Process

Input to this creation process should come from users, programmer/analysts, and managers. Consult the programmers working on the current development/maintenance process. Even if the current maintenance process is informal, it should have merit. Change always has a better chance at success when it fits in with the organization's cultural norms. But the users may want faster modification turn-around. And the managers are frustrated with their inability to make the software change as quickly as the business environment they are facing.

The creation of a development/maintenance process is clearly beyond the scope of this article. It is much better covered by SEI's CMM. But be aware that this step is time-consuming and must be done prior to reengineering. If your organization already has a well-defined, well-regulated development/maintenance process, then clearly this step is redundant.

Step 3 - Form Your Reengineering Team

Create a reengineering team composed of those programmer/analysts, users, and managers that are knowledgeable and open-minded enough to at least look at technical change to solve their maintenance problems. Choose carefully. They should be technically competent, credible, patient, possess good social skills, and know the organization's culture well enough to function successfully. This team will:

- Understand the current environment and business needs
- Establish goals, strategies, and action plans
- Provide cost justification
- Test new tools
- Purchase tools
- Provide internal marketing, consulting, and service
- Train others
- Continue to research and evolve technology
- Provide vendor liaison - partnerships to advance technology
- Improve their own processes using client satisfaction surveys

Be sure the user community is well-represented. Besides their expertise, you need their support and buy-in. In addition, they need to help in the validation of the post-reengineered system.

Step 4 - Select a Standard Set of Software Metrics

You must know where you are before you can get to where you want to be. Upper management will demand this. You must show them what they got in return for their money and support. For your own career's security, you need metrics to prove to the doubting Thomas's that process control and reengineering was worth all the pain you caused them.

But beyond the political concerns, metrics helps you find out what is wrong and how to fix it. Think of software as you would an ill patient in a hospital. No competent doctor would think of treating a patient without knowing all the symptoms and test results. Similarly, you shouldn't treat your ill and dying software without knowing what symptoms your software is suffering. Metrics and software analysis tools gather this data for you. But just as your family doctor keeps a record of your health over the years, you should continue to gather metrics on your systems for their entire lifecycle. You and your doctor need to know what is normal in order to react instantly to abnormal signs.

One last warning about metrics. They must be non-threatening to the troops. Don't use them for personnel appraisals, judgments, or budgets - just use metrics for process improvement and continued health of your software systems. Otherwise, they'll start reporting what they think you want to hear. In order to reward quality, we require honest, accurate, and consistent data from the troops.

Step 5 - Create a Standard Testbed or Validation Suite

This is a terribly overlooked part of the reengineering process. How do you prove the end product of reengineering is functionally equivalent to the input source code? Whether you reformatted, restructured, translated, or generated code from captured design data, you need to validate the functionality of the resulting software. Critical systems, in particular, need to demonstrate unchanged functionality after reengineering. (This is a key reason why it is extremely unwise to propose any functional changes until after the reengineering process is stabilized.) So check your current validation suite for the target software system. Ask yourself if it's current and complete. If not, take the time to create a complete testbed for the target software system. If the target software system is rather large and you decide to adopt an incremental approach to its reengineering, then you ca:: validate/create a validation suite incrementally based only upon those parts of the system to be reengineered.

Traceability is also important. Design functionality must be associated with the source code. This is your only link between the new, repository-based design representation and the old source code. When looking at reengineering tools, ask the vendors about traceability.

For a much more in-depth discussion of testing strategies, we strongly recommend you read two other STSC reports: *Test Preparation, Execution, & Evaluation Software Technologies Report* and *Source Code Static Analysis Technologies Report.*

Step 6 - Analyze Your Legacy Systems

Each target system will have different reengineering needs and a different reengineering strategy. Some systems may have sufficient documentation but your maintenance process requires capturing the design information to a repository. Another system may have missing or inaccurate documentation so it needs redocumentation and reverse engineering to a repository.

Another reason to analyze your legacy systems is to discover their unique aspects regarding eventual reengineering. Is the system comprised of several programs linked by JCL? Does it use any embedded DBMS or Assembler macro calls? Are other languages embedded or called? What about any online screens? Are there any implied "dynamic" (i.e. execution time) requirements such as response time constraints, available memory, etc.? This must all be catalogued prior to the next step of finding the right reengineering tool(s). Moreover, this information must also be captured within the design repository. When picking your repository or reengineering tools, find out if they can handle such data.

During this process step, gather whatever requirements data you can find. Currently, reengineering tools cannot supply you with requirements information from source code. But you need requirements data to effectively maintain your software systems. Talk to the maintainers, users, management, and anyone else who's worked with the system from the time it was developed. Then ask the vendor whether their repository can store requirements data.

There are numerous tools to help you

in gathering analysis data on your legacy systems. These tools will help you analyze the complexity of your software, check the system's degree of structuredness, evaluate the impact of a given modification, and identify standards violations (e.g. non-initialized variables, dead code, etc.). This data will help during the validation phase and supply objective data to support your selection of the most appropriate reengineering strategy applicable to the target software system. Program analysis tools do not modify source code. Running program analysis tools before and after reengineering provides a means of measuring the effectiveness of the reengineering. These tools are described in much more detail (along with a list of such tools) in the STSC's 1993 *Test Preparation, Evaluation, and Execution Software Technologies Report* and *Source Code Static Analysis Technologies Report.*

Step 7 - Reengineering Tools Analysis

All the preceding steps should be accomplished prior to your initial efforts at reengineering tools analysis. Only within such a context can reengineering tools help your maintenance process. "Buying software tools the way most software development organizations do is like going to the grocery store without a list - when you get home, not only do you not have everything you need, you have a lot of stuff you don't need" (remark made by Denis Meredith during the *8th International Conference and Exposition on Software Maintenance & Re-engineering,* Washington, DC, USPDI, Silver Spring, MD, 5-9 August 1991).

Find the reengineering tools that will accomplish your maintenance and reengineering goals. They should also fit the needs of your target systems as defined above. Be aware that there may not be a COTS reengineering tool that satisfies your requirements. Your source code language may not have a sufficiently large user base to warrant any vendor creating a reengineering tool. Or there may exist reengineering tools for your target source code language but they won't capture JCL, screen, DBMS calls, macros, etc. You may then need to decide whether the discrepancies between the tool's functionality and the legacy system's characteristics are sufficiently small to still make the tool usable.

You may have to "massage" your legacy software so that your reengineering tools can process it. This may take several forms:

- Most reengineering tools run on workstations or PC's. Thus, if your software resides on the mainframe, there is the issue of downsizing or porting your legacy software from your mainframe.

- If the tool will only process COBOL 85 and your legacy systems are written in COBOL 74 and COBOL 68, use a translator to upgrade your COBOL code to 85 first.

- If your legacy systems have embedded macros, DBMS calls, other language calls, etc., and your reengineering tool cannot process anything outside the primary source code language, then stub out these areas.

Tool integration is also an important consideration. As discussed above, no single reengineering tool will match all your requirements. But where one tool may fall short, another may pick up the slack. So check whether the reengineering tool can integrate with other reengineering tools, your chosen repository, the hardware platform/OS, and the target maintenance process/tools.

Step 8 - Create a Reengineering Process

Based on the preceding steps, you must now define your reengineering process. Step by step, decide what needs to be done, taking into account the legacy software characteristics (starting point), your new maintenance process (finish point), and all intervening steps such as validation, metrics, tool integration, training etc. Set up a viable schedule with milestones to report back to management.

One cautionary note: it is very tempting to include requirements modifications during the reengineering process. DON'T!!! This creates problems with functional validation and complicates the whole project immeasurably. Wait until the reengineering project is complete and stable (i.e. functionally validated) before changing and implementing modifications.

It is at this point that you must make a basic decision regarding your reengineering process - evolution vs. revolution. There are three fundamental strategies to approach reengineering:

- **Systems reengineering**

An entire system is reengineered. Systems reengineering can be used for one-time reengineering projects where you need to solve an immediate problem for a particular application. The advantage to this strategy is that the system is brought into the newly defined maintenance environment all at once. A disadvantage is the amount of risk your organization runs. The full system, after reengineering, must work flawlessly. But can you guarantee that? Is the functionality intact? Were any bugs introduced during the reengineering? Remember, there is a good deal of human intervention with any reengineering project. If you're confident you can control these potential problems, then systems reengineering will allow you to quickly switch over from the old system to the reengineered system. Otherwise, perhaps one of the following strategies would be better.

- **Incremental reengineering**

Parts of the system are reengineered as needed thus creating a new "version" of the system. When a system modification is required, take the opportunity to reengineer only those parts of the system that need to be changed. The advantage to this approach is a reduced risk when switching over to the new system. Only portions of the system were changed and these portions were clearly identified. Should a problem occur, it's origin can be clearly traced. The disadvantage lies in the number of interim versions of the system generated until the entire system is reengineered eventually. Each version would contain some reengineered code and some not. Configuration control considerations are critical for this strategy to work.

- **Partial reengineering**

Functionally cohesive sections of the system are reengineered as needed. This is similar to both systems and incremental reengineering. When a system modification is required, only those parts of the software affected by the modification (e.g. sub-program or called routine) are reengineered. But the rest of the system is left alone - until the next modification. The advantage to this strategy is the inherent advantage of modular design. If a problem occurs after switch over, just re-implement the old module. The disadvantage lies in interface issues. If one or more modules reside in a repository but the remainder of the system is executed the same old way, they must interact flawlessly. Other questions to consider include: will there be any response time degradation that is unacceptable (particularly for real time systems); and will the generated code (either from a repository or restructurer) interface correctly with the remainder of the legacy source code?

If reengineering is viewed as the path to long-term software maintenance improvement for your organization, then we recommend incremental or partial reengineering. But if you need a short-term solution to an immediate problem for a particular application, then systems reengineering might be the better strategy.

Step 9 - Train, Train, Train...

Train your reengineering team. They need to understand several key concepts:

- How to manage technological change
- The basics of reengineering
- How to use the selected reengineering tools
- How to use the target development/maintenance process

Because of the training and time investments in your reengineering team, you should seriously consider making the team permanent. Reengineering is an ongoing process. Any moderate to large software organization will have more than enough legacy software to keep such a team busy for the next several years at least.

Summary

Reengineering requires a core process of manual tasks prior to buying your first tool. There is no avoiding these tasks. They are common throughout all successful reengineering projects. Obviously, short-term, heavy-duty resources must be allocated up front to gain the cost savings over the long haul. Decide now whether you can gather the support for this level of commitment. Smaller pilot projects are usually a good way of proving reengineering concepts and their cost savings. But whatever political strategy you employ, don't attempt reengineering without each of the steps outlined above.

A Road Map Guide to Software Reengineering Technology

Robert S. Arnold

Purpose and Structure

This paper introduces the reader to software reengineering definitions and technology. Software reengineering technology supports three major themes: (1) understanding software, (2) improving software, and (3) capturing, preserving, and extending knowledge about software. This paper will help the reader see the boundaries of reengineering technology and appreciate some of its risks.

The reader is assumed to be interested in the maintenance, improvement, or understanding of existing source code. This interest may be of itself, or part of a larger task, such as converting software from one operating system to another. This paper will help reengineering nonexperts appreciate reengineering issues. Experts will see a fresh contemporary viewpoint.

The paper is structured as follows: "Reengineering Definitions" defines reengineering terms and places them in context. "The Significance of Reengineering" discusses why reengineering is significant and worthy of the reader's time and study. "Reengineering Technology" discusses reengineering technology themes and connects technology areas with the themes. It briefly discusses the importance and significance of the technology areas. "Reengineering Strategies and Risk Mitigation" discusses risks and cautions in using reengineering technology. "Future Advances" discusses future research issues for reengineering. Appendix A provides a procedure for classifying a software transformation consistent with the reengineering definition used here.

Reengineering Definitions

Software reengineering is any activity that

(1) improves one's understanding of software, or
(2) prepares or improves the software itself, usually for increased maintainability, reusability, or evolvability.

In this definition, the term "software" includes — in addition to source code — documentation, graphical pictures, and analyses. The analyses are about source code, designs, specifications, test data, and other documents directly supporting software development or maintenance.

Part 1 of this definition includes activities such as browsing, measuring, drawing pictures of software, documenting, and analyzing. Part 2 includes activities designed to improve static qualities of software, usually so the software is easier for people to work with.

Part 2 tends to exclude modifications whose purpose is not for maintainability, reusability, evolvability, or improving one's understanding of the software. For example, optimizing code or restructuring it purely for performance is not commonly thought of as reengineering.[1]

Reverse engineering pertains to part 1 of the reengineering definition. Reverse engineering generates information about a software representation (such as source code) to help one understand it or to facilitate its processing.

Other reengineering definitions

Different people or groups seem to have different meanings for reengineering. For example, GUIDE defines reengineering as

> the process of modifying the internal mechanisms of a system or program or the data structures of a system or program without changing its functionality [GUIDE89].

Chikofsky and Cross define reengineering as

> the examination and alteration of a subject system to reconstitute it in a new form and subsequent implementation of that form [Chikofsky90].

For reference in this discussion, the reengineering definition described in the previous section, the GUIDE definition, and the Chikofsky and Cross definition will be called definitions A, G, and C, respectively. Both definitions G and C are valuable and useful. The reengineering definition one uses depends on one's perspective. Reengineering definition A is used in this paper for several reasons.

First, definition A centers on the purpose of reengineering activities, rather than on their means or processes. This recognizes that reengineering activities can use technology that, in other contexts, may not be called "reengineering." For example, impact analysis and software testing are used in software maintenance, but not always in the context of reengineering.

Second, on close examination, definitions G and C allow different activities to be called "reengineering." Definition A is more inclusive. For example, under definition C one might classify a software functionality change as reengineering. But under definition G one would not. Under definition A one could classify a functionality change as reengineering, provided it was for the purposes mentioned.

As another example, definition G does not consider creating information about software as reengineering, unless the information is used to support modification. Definition C does define creating information about software as reengineering,[2] but only if it is on the path to reimplementation. Definition A allows the creation of information as an end in itself as part of reengineering. For example, definition A considers as reengineering the creation of information about software to facilitate understanding or to promote maintainability, reusability, or evolvability.

[1] The context of reengineering is expanding so rapidly, however, that some people include even these as reengineering — improving suitability for use — of the code.

[2] If we equate the information with "a new form" in definition C.

Third, definition G tends to focus on changes to source code. Improvements of non-source code items, such as documentation and specifications, may also be considered as reengineering. Such activities are allowed in definition A, and may be allowed in definition C.

This discussion should not be taken to imply that one reengineering definition is better than another. The definitions capture different perspectives. Perspective change is common in a rapidly evolving field such as reengineering. Because of the proliferation of reengineering definitions, the reader should ask people what they mean by reengineering when more than a general understanding is important.[3]

The discussion in this paper embodies an approach to classifying reengineering and similar activities, relative to reengineering definition A. In this approach, a written definition of reengineering is created. (This was done in the "Reengineering Definitions" section.) Then the context of reengineering and related terms is diagrammed. The diagram features various views of information and the transitions among the views. (Figure 1 is such a diagram, and is discussed in "The Context of Reengineering.") The transitions in the diagram are classified in a table. Finally, a decision procedure is created for classifying an activity using the table. (Table A2 and the decision procedure appear in Appendix A.)

Reengineering spellings and related terms

Just as there are no universally accepted reengineering definitions, there are no universally accepted spellings for reengineering. The most common spellings are reengineering and re-engineering. "Reengineering" is used in this paper.

Synonyms for reengineering abound, often with nuances reflecting a specific reengineering purpose:

- improvement
- renewal
- renovation
- refurbishing
- modernization
- redevelopment engineering
- reclamation
- reuse engineering

The terms *improvement, renewal, renovation, refurbishing, and redevelopment engineering* all have similar meaning: improving software for evolvability and further use. *Modernization* includes improvement of software, but may go beyond by improving software development and maintenance activities surrounding the software. *Reclamation* and *reuse engineering* refer to reengineering to make source code more reusable.

The context of reengineering

Figure 1 illustrates a framework for understanding reengineering and related terms. The figure, an update to a similar one in [Chikofsky90], reflects evolving connotations of terms. Five ideas are shown in Figure 1:

(1) views of software,
(2) information base,
(3) decomposition,
(4) composition, and
(5) transformation.

[3] As in negotiating contracts for reengineering work.

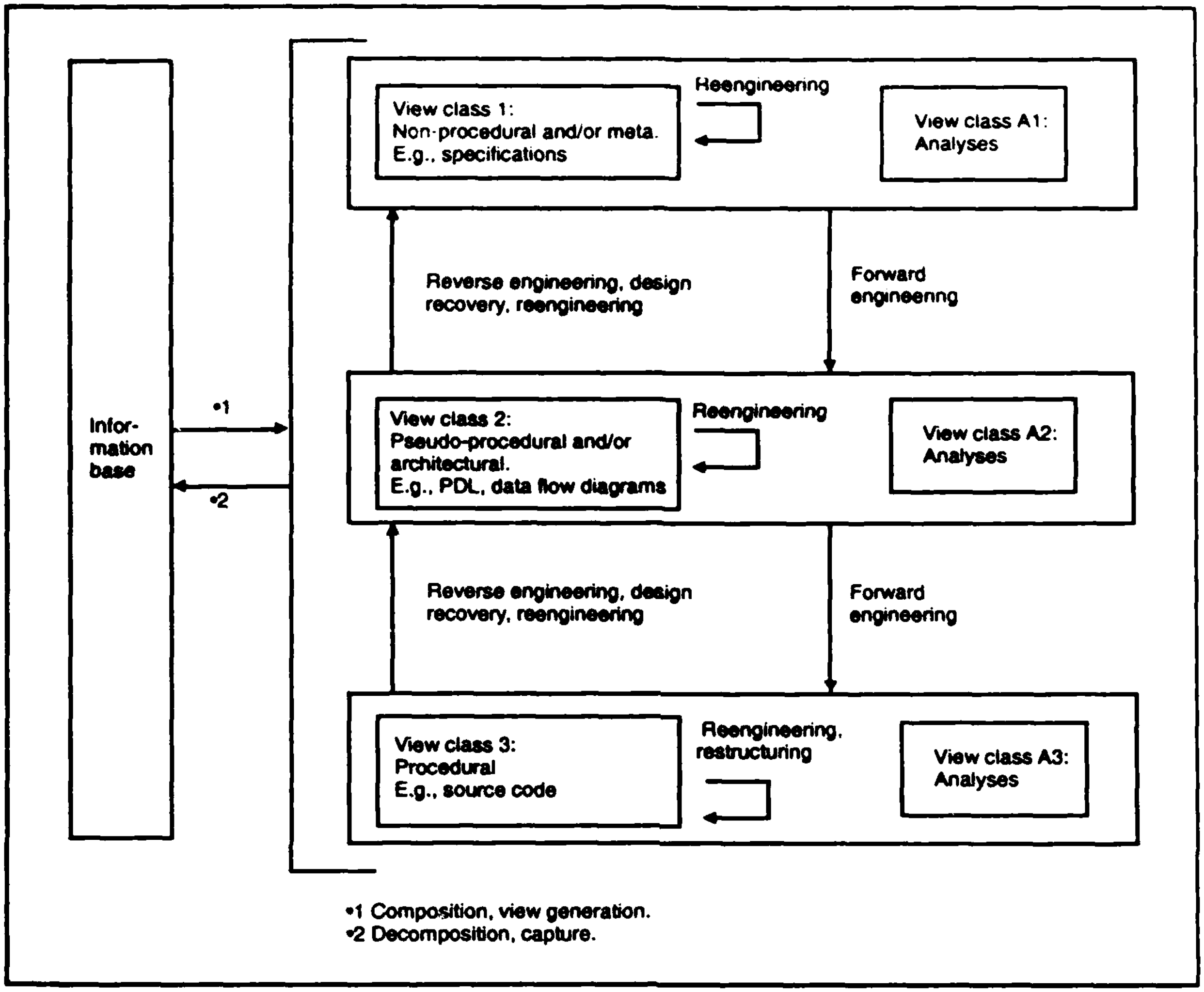

View class 1:
Non-procedural and/or meta.
E.g., specifications
Reengineering
View class A1:
Analyses
Reverse engineering, design recovery, reengineering
Forward engineenng
Infor-mation base
•1
•2
View class 2:
Pseudo-procedural and/or architectural.
E.g., PDL, data flow diagrams
Reengineering
View class A2:
Analyses
Reverse engineering, design recovery, reengineering
Forward engineering
View class 3:
Procedural
E.g., source code
Reengineering, restructuring
View class A3:
Analyses
•1 Composition, view generation.
•2 Decomposition, capture.

Figure 1. Reengineering and related terms. Reengineering and related technology may be viewed as transforming information in one software view to information in another software view. The transformation may move information into and out of the information base.

Understanding these ideas helps distinguish the terms presented in Figure 1.

A software *view* is a representation of software or a report about software. A software view may be for human viewing or not, but it typically is a significant interim representation of software that humans may want to see. In this discussion, the word view refers to the type of view (e.g., a data flow diagram). *View information* means the specific information in a view (e.g., a specific data flow diagram D), or the information base of knowledge decomposed from information in the view.

Examples of software views are specifications, source code, measurements, reports derived from static source code analysis, and test data used to characterize software behavior. Figure 2 has several sample views, with possible view information shown as pictures. When a view is supported by a tool, it nearly always comes with a view editor, to support entering, browsing, and changing view information.

As implied by Figure 1, views can be grouped into four classes:

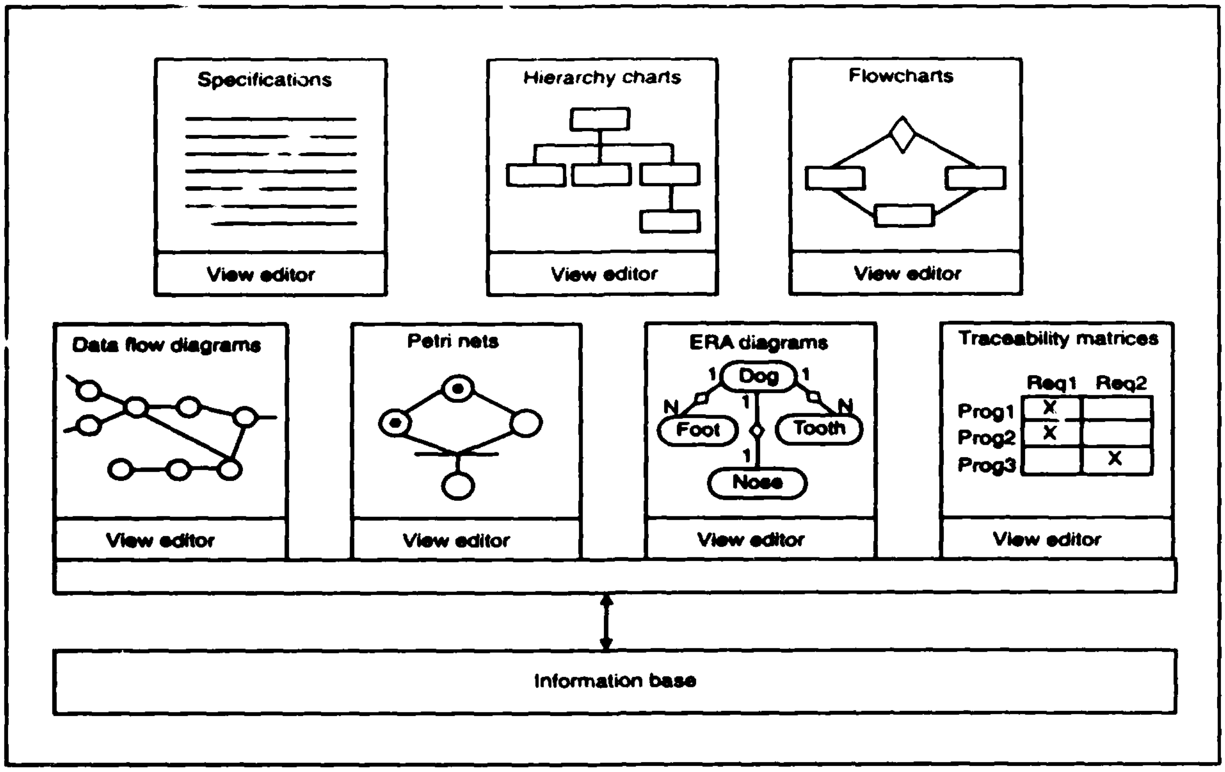


Figure 2. Multiple software views. Many reengineering tools, especially CASE toolsets, support several different views of software. No tool as yet supports all the views above, but this gives an idea of what can be found in a view.

Class 1:	Nonprocedural- and/or meta-oriented views.
Class 2:	Pseudoprocedural- and/or architectural-oriented views.
Class 3:	Highly procedural views, or close derivatives.
Class A:	Analysis views that may accompany any other view.

Class 1 contains views that are nonprocedural and/or *meta*.[4] Software specifications and conceptual schemas fall into Class 1. Class 2 contains views that are pseudo-procedural or architectural. Designs, program design language descriptions, and software architecture diagrams (such as calling hierarchies or data flow diagrams) fall into this class. Class 3 contains highly procedural information, information closely associated or derived from this information, or direct information about representations. Source code, program slices, data, data definitions in source code, objects and relationships decomposed from views, and syntax trees are in this class.

Class A contains analysis views derived from any of the other views. For example, software metrics are derived by analyzing software. An analysis view can appear with information in any of the other views. Class A can be divided as follows:

Class A1:	Analyses pertaining to Class 1 views. Example: Fog index of specification text.

[4] A "meta" view is a view *about* something. The intent here is *nonprocedural* meta views, such as data schemas or decision tables.

Class A2: Analyses pertaining to Class 2 views.
Example: Coupling levels of source modules.

Class A3: Analyses pertaining to Class 3 views.
Example: Number of modules in source code.

In practice, view classes 1, 2, 3, and A are not disjoint. For example, some people may place a program design language (PDL) description in Class 3, others in Class 2. However, it is assumed for the rest of the discussion that it is possible for an individual to create disjoint classes for himself or herself, if needed. For the rest of the paper, it is assumed that the classes are disjoint.

The *information base* is the repository of information about the software. It is loaded in three ways:

(1) Decomposing software into objects and relationships,
(2) Incrementally building up objects and relationships through tools that build on or add to knowledge in the information base, and
(3) Importing information from other information bases.

Decomposition is the process of transforming a view into objects and relationships stored in the information base. For example, compilers commonly decompose programs into abstract syntax tree representations.

Composition generates view information from information in the information base. The *composer* (the tool or person that does the composition) assembles view information by finding relevant objects and relationships in the information base, then adding view formatting as needed to display the view information. For example, the back end of a compiler commonly generates code by traversing a semantic graph of the program, or some equivalent.

The notion of *transformation* is central In Figure 1, reengineering transforms, in effect, information from one software view into information in another view, at the same or earlier view class.[5]

Examples of reengineering transformations are transformations from source code (Class 3) to restructured source code (Class 3), updated designs (Class 2), corrected specifications (Class 1), or computed static measurements (Class A3). The reengineering transformation usually "improves" the information in the view according to some criterion.[6] Software restructuring is reengineering centered on transforming the source code's structure (syntax and semantics).

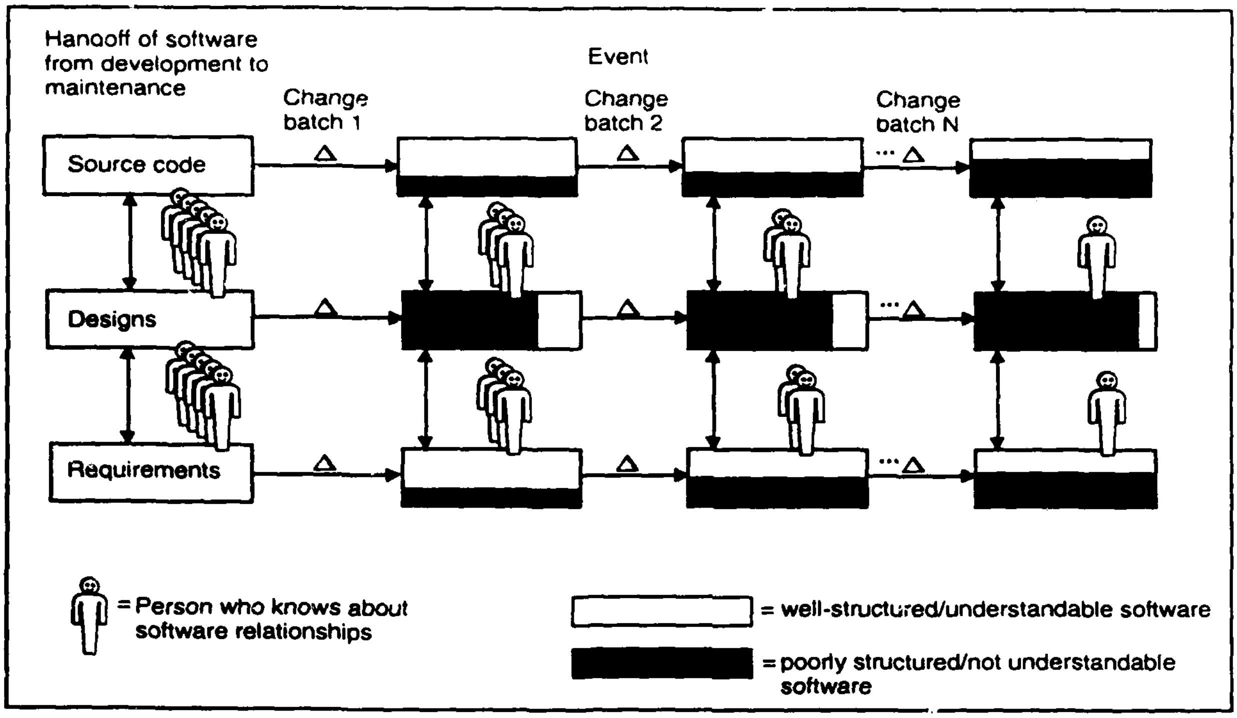
Transformation also underlies reverse engineering and design recovery. Reverse engineering is like reengineering, except the origin and target views are different, the target view normally being in an earlier view class. Transforming source code, for example, into structure charts may be either reengineering or reverse engineering. But transforming source code to restructured source code is reengineering or restructuring, not reverse engineering. Updating embedded source code comments is reengineering.

Reverse engineering has other meanings, though these are not used here. For example, it can mean to analyze a detailed representation to discover its inherent design. One sometimes hears that people perform reverse engineering to determine source code from object code. For other meanings of reverse engineering, see [Rekoff85].

Design recovery, a subset of reverse engineering, generates information about software. Often the information is

[5] Even though some reengineering tasks, such as purely manual inspection and improvement of source code, do not involve an online information base, they can still fit this model. Just substitute information in the person's head for information in the information base.

[6] A frequent criterion is that reengineering should change form but not function. For example, restructuring code to improve readability is also reengineering.


Handoff of software from development to maintenance
Event
Change batch 1
Change batch 2
Change batch N
Source code
Designs
Requirements
= Person who knows about software relationships
= well-structured/understandable software
= poorly structured/not understandable software

Copyright © 1992 by Robert S. Arnold. Used by permission.

Figure 3. Dynamics of maintenance and information loss. Maintenance tends to make software harder to change through loss of information about how to modify it. Fewer and fewer people know less and less about the software. This happens because people leave who understand the software, and because the software itself becomes harder for new programmers to understand. Mental connections are lost.

not easily extractable from the software and associated documentation, and requires considerable effort to deduce. Three examples are generating rationales describing why the software is in its current form, generating specifications from source code, and generating black box test data sets for software without documentation that is current. Appendix A gives another interpretation of design recovery in terms of views.

Forward engineering is a transformation, usually from an earlier to a later view class. For example, generating source code from a data flow diagram is usually a forward engineering activity.

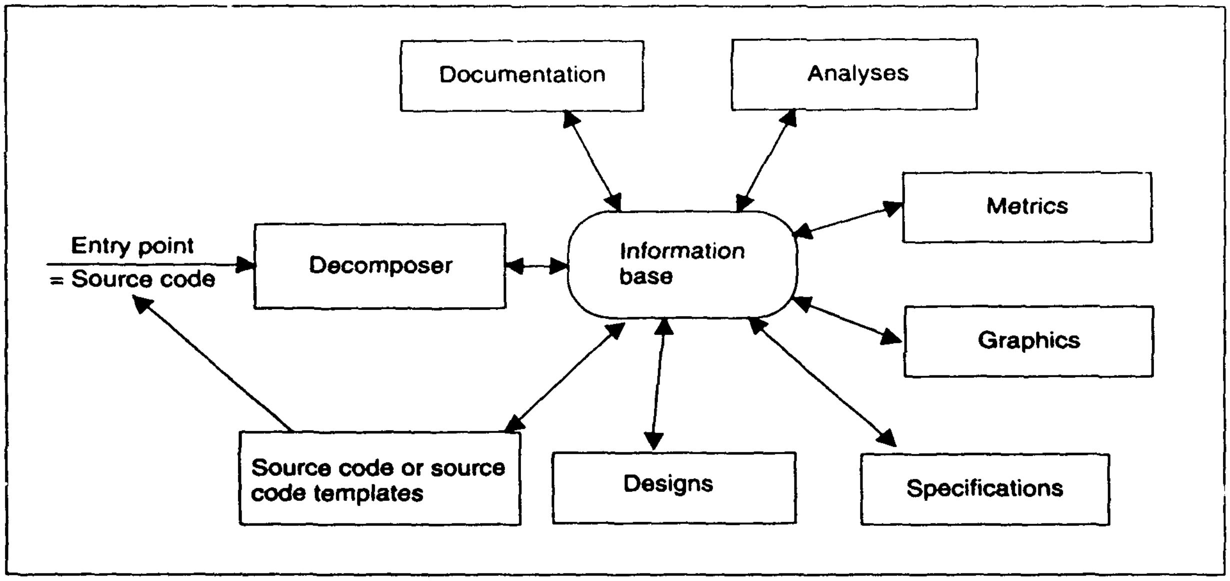
Reengineering can be considered in other ways besides transformations of view. The following section discusses the purposes behind reengineering.

Significance of Reengineering

Reengineering is important for several reasons:

(1) Reengineering can help reduce an organization's evolution risk.

To extend software capabilities, organizations can build new software, evolve existing software, reengineer and evolve existing software, use application generators, or obtain software parts or packages. When the latter two options are not available, organizations are faced with building new software or evolving existing software. Simply manually evolving existing source code tends, in current practice, to make the software harder to change, or less reliable when changed (see Figure 3). Building software from scratch can be expensive and uncertain. Reengineering software and evolving it sometimes offers less change risk. It can help safeguard an organization's software investment better than building software from scratch or simply evolving it through traditional maintenance.



Copyright © 1992 by Robert S. Arnold. Used by permission.

Figure 4. Reengineering and CASE toolsets. Most CASE toolsets started as forward engineering tools. For added value, CASE toolmakers have added reengineering and reverse engineering capabilities. Most entry points to these capabilities currently depend on source code only.

(2) Reengineering can help an organization recoup its investment in software.

Companies have spent hundreds of thousands of dollars building software. The software industry has spent billions. Rather than ignore existing software, companies can use reengineering to partially recoup their software investment. Reengineering helps organizations build on existing software.

(3) Reengineering can make software easier to change.

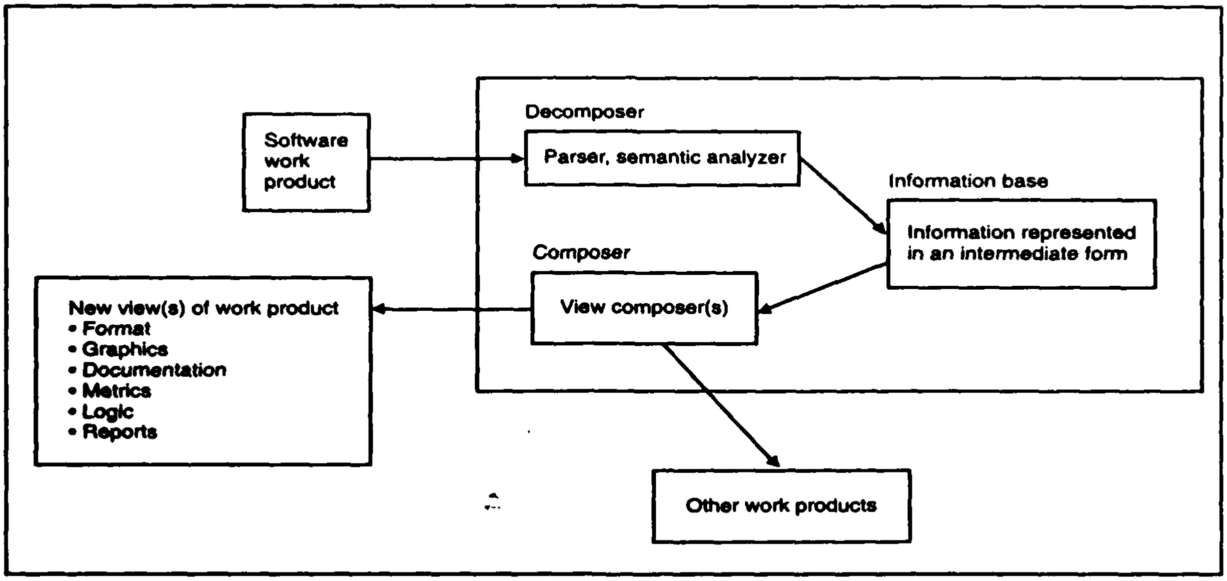
Improving software can pay rich dividends. It speeds productivity of the maintenance programmer by making code easier to understand or work with. It gives an organization more flexibility because its software can be modified more quickly to accommodate business changes. Reengineering extends an organization's options.

(4) Reengineering is big business.

The 1990's market for reengineering services and tools has been estimated to be in the billions. As reported in [*Computerworld* 91], the estimated expenditures allocated to reengineering *services* in 1990 was $4.6 billion. For 1995 the estimate is $11.9 billion. The estimated expenditures allocated to reengineering *products,* including back-end CASE tools, was $.8 billion in 1990. For 1995 the estimate is $2.7 billion. Many software systems, and system parts, need updating. Software contractors and service companies are pursuing work in this area. Many organizations are looking for reengineering techniques, tools, and processes to use.

(5) Reengineering capability extends CASE toolsets.

Reengineering helps new techniques and tools to be applied to old software. This has several benefits. For maintainers, it allows newer and more powerful tools to help them maintain software. Reengineering may be seen as a kind of technology transfer vehicle, allowing old systems to be brought into frameworks of new, more powerful tools. For CASE tool vendors, adding reengineering tools to their toolsets opens new markets of existing software. It also gives important added value to their previously forward-engineering-only toolsets (see Figure 4).



Copyright © 1992 by Robert S. Arnold. Used by permission.

Figure 5. Automatic reengineering process. Reengineering tools tend to follow a common framework. They parse information into an information base. Views are then generated from the information base to reflect different information about software.

(6) Reengineering is a catalyst for automating software maintenance.

Most reengineering tools follow the pattern of Figure 5. They are essentially repositories, with specialized ways to get information into and out of them. An important part of this is parsed information stored in the information base. The parsed information is valuable for analysis, automation, and research in software maintenance. The framework in Figure 5 also provides clear points of evolution within any of its parts.

(7) Reengineering is a catalyst for applying artificial intelligence (AI) techniques to solve software reengineering problems.

As discussed earlier, reengineering has a basis in transformations. Historically, the field of automated transformations is an outgrowth of work in AI, such as the rule-based production systems [Davis76] and language processing. Earlier, transformations grew out of formal logic and mathematical rewrite systems. Reengineering is supplying AI workers with a fruitful field in which to apply their work (e.g., [Rich90], [Hartman91]).

For the preceding reasons, reengineering has been significant and likely will remain so during the next 5 – 15 years. Experience with reengineering will provide ways to improve software development and maintenance. Ideally, progress in reengineering, development, and maintenance will lessen the need to reengineer in the first place.

Reengineering Technology

There are several themes underlying software reengineering technology: improving software, understanding software, and capturing, preserving, and extending knowledge about software. Table 1 shows these themes and their associated technology. The following sections discuss these technology areas. They briefly describe each technical area and its significance, and list two or more references for further study. The reader will find the discussion useful for quickly appreciating reengineering technology. This discussion, being brief, is not comprehensive of all reengineering technology or all relevant work.

Table 1. Reengineering Technology

Reengineering theme	Associated technology
Improving software	Restructuring Redocumenting, annotating, updating documentation Reuse engineering Remodularization Data reengineering *Business process reengineering* Maintainability analysis, portfolio analysis, economic analysis
Understanding software	Browsing Analyzing, measuring Reverse engineering, design recovery
Capturing, preserving, and extending knowledge about software	Decomposition Reverse engineering, design recovery Object recovery Program understanding Knowledge bases and transformations

Technology for improving software

Software restructuring. Software restructuring is the modification of software to make it easier to understand or easier to maintain [Arnold89b]. Nowadays the term connotes changing the source code control structure. Restructuring is significant because it is one of the oldest (e.g., [Bohm66]) and most refined reengineering techniques. Restructuring was one of the first reengineering tasks to be fully automated. Developing automated restructurers has led the way to other reengineering tools [Arnold90b]. Informative references on restructuring are [Arnold89b], [Miller87], and [Calliss88].

Redocumenting, annotating, and updating documentation. Redocumenting software is the creation of updated, correct information about software. Redocumenting code is a transformation from code (and other documents and programmer knowledge) into new or updated documentation about code. Normally this documentation is textual (e.g., embedded comments), but it can be graphical as well. Software improvement by updating documentation (embedded comments, designs, and specifications) is one of the older reengineering techniques ([Heninger78], [Sneed84]). Redocumentation is important because maintainers tend to depend on good inline comments [Glass81] as guideposts to what the code is doing.

Annotating connotes adding documentation to source code when there is little useful documentation to begin with. Annotating is particularly important for understanding assembly code. [Landis88] is an informative work on redocumentation. Success in redocumentation depends heavily on automated tools. [Philips84] tells a horror story about trying to perform redocumentation without tools.

Reuse engineering. Reuse engineering is the modification of software to make it more reusable, usually aiming to find software parts and rebuild them so that they can be put into a reuse library. Several authors describe processes for finding and reusing parts ([Arnold91b], [Caldiera88]). Prospecting metrics and heuristics are described in [Arnold90a], [Reynolds90], and [Caldiera91]. Specific techniques for making existing software reusable are described in [Bailey90]. Reengineering code into more object-oriented forms is related to reuse engineering, and is discussed below.

Legacy systems is a subfield of reuse engineering that decomposes existing systems into objects and relationships that may be reassembled (reused) in new systems. [Theby91] discusses finding and storing objects and relationships for legacy systems. The term "legacy system" may also refer to a system of enduring value.

Remodularization. Software remodularization is the changing of the module structure of a system. Often this depends on cluster analysis of system component characteristics and coupling measures. Recent work in this area is [Schwanke91] and [Sneed88]. Criteria for modularization are discussed in [Card85].

Data reengineering. Data reengineering improves a system's data. Schemas may be reorganized and updated, multiple schemas may be consolidated into one schema, data dictionary entries may be made semantically consistent, and invalid data may be removed. Data reengineering is often a prelude to other tasks, such as migrating data to another data base management system. Informative references on data reengineering are [Ricketts89] and [Hevner89]

Business process reengineering. With today's newer, flexible software architectures and information technology automation possibilities, there is a trend to make software fit the business, rather than business fit the software. Experience has shown that powerful productivity improvements can sometimes come from rethinking the business processes automated by software ([Hammer90], [Davenport90]). This rethinking may result in new software designs that can become the basis for reengineering, migrating, or evolving a software system

Maintainability analysis, portfolio analysis, economic analysis. Software maintainability analysis is important for discovering what parts of a system should be reengineered. Typically the majority of maintenance work is centered on a relatively few modules in a system. Maintainability analysis helps to locate the high maintenance system parts. These parts have the biggest initial impact on maintenance costs.

[Peercy81] defines a methodology for determining program maintainability. More recently, [Oman92a] and [Oman92b] describe what maintainability is and metrics for assessing it. To decide when and where to reengineer, [Husmann90] discusses the use of portfolio analysis, [Sneed91] discusses the use of cost-benefit models, and [Connell87] describes reengineering criteria.

Technology for understanding software

Browsing. Browsing of software, such as with a text editor, is perhaps the oldest means for understanding it. Recently browsing has become more advanced, with the use of hypertext [Conklin87] to make connections between related parts and multiple view systems [Cleveland89] to provide different views at the click of a mouse (see Figure 2). Cross-reference tools are another important part of browsing.

Analyzing, measuring. Analysis and measurement are also important technologies for understanding program properties such as complexity. A large literature on metrics has accumulated (such as [Cote88]). Relevant techniques for reengineering are program slicing [Weiser81], control flow complexity measurement [McCabe76], coupling measures [Myers75], and many others (for example, [Harrison82], [Rombach89]).

Reverse engineering, design recovery. As indicated above, reverse engineering and design recovery generate new information about software, usually in a different view. This technology has become popular, but determining some kinds of design information (like design rationales) is still quite risky [Corbi90].

More commonly, reverse engineering generates structure charts or data flow diagrams from source code. These tools rely heavily on information readily available or analyzable from the code itself. The January 1990 issue of *IEEE Software* has a good collection of papers on reverse engineering and design recovery.

Capturing, preserving, and extending knowledge about software

Decomposition. Program decomposition takes a program and makes objects and relationships out of it. These objects and relationships are stored in an information base. The objects and relationships facilitate analysis, measurement, and transformation and extraction of further information. Working on a decomposition rather than directly on the source code saves the work of having to parse the program and create objects and relationships for use by a tool. This task, though straightforward for most languages, is time-consuming to solve from scratch. For most languages it is easier to rely on off-the-shelf decomposers, or decomposer generators such as lex and yacc on Unix.

Decomposition is not confined to reengineering. It is used in integrated programming support environments and structured editors. Good references on decomposition are [Chen90], [Lyle88], and [Gopal88] — and there are many more.

Object recovery. Object recovery obtains objects from source code. This allows one to view previously nonobject-oriented source code in an object-oriented way. The object-orientation (classes, inheritance, methods, abstract data types, etc.) may be partial or complete.

Migrating source code to object-oriented form has been receiving attention. An object-oriented program structure may offer more program understanding, migration, and impact reduction possibilities than nonobject-oriented code. There has been previous experience with object orientation when converting systems from C to C++. [Breuer91], [Dietrich91], [Jacobson91], [Dunn91], and [Byrne91] describe recent experience and ideas for discovering objects in source code.

Program understanding. Program understanding takes several forms. One is manual or automated techniques for programmers to gain a better understanding of the software. The other is a body of work that stores information about programming and uses this information to find instances of programming knowledge in the code. Understanding is evidenced by the extent to which the software is matched with the tool's base of programming knowledge. [Robson91] gives a quick overview of both forms of program understanding. Instances of the latter work are [Hartman91], [Rich90], and [Harandi90].

Knowledge bases and transformations. Knowledge bases (for example, [Harandi90]) and program transformations (e.g., [Burson90]) are foundational to much reengineering technology today. The information base, associated transformation engine, and programmed transformations drive the power of the reengineering tool. The transformations work on program graphs and object graphs stored in the knowledge base. Object-based, transformational architectures for reengineering tools (e.g., [Burson90]) are attractive for building new reengineering tools (for example, see [Kozaczynski89]).

Reengineering Strategies and Risk Mitigation

Being able to apply reengineering technology is as important as knowing what it is and what it can do. Much experience has been accumulated.

Reengineering process

The reengineering process takes many forms, depending on its objectives. Sample objectives are code cleanup, redocumentation, migration, capture of information in an information base, and reengineering code for reuse.

Case studies are frequent sources of reengineering processes. For example, [Slovin91] describes the improvement of the modular structure and maintainability of a system having high maintenance costs. [Britcher90] describes a reengineering feasibility project in which a Federal Aviation Administration terminal approach control system was reengineered to operate on more modern hardware and in Pascal (instead of assembly code).

Others have tried to systematize the reengineering process. Such an approach has been described in [Ulrich90 – 91]. The major process steps here are inventory/analysis, positioning, and transformation. The inventory and analysis phase establishes a software components base and evaluates reengineering options based on this inventory. Positioning improves software quality without necessarily affecting existing functionality or architecture. Positioning improves the software to facilitate change, or analysis to support change. The transformation phase creates a new architecture from the existing one.

Reengineering evaluation

There is concern about the empirical effectiveness of reengineering. The evidence for reengineering falls into anecdotes, case studies, lessons learned, and experiments. A good discussion of cost-benefit analysis issues for reengineering can be found in [Sneed91]. Experiments such as [FSMSC87] and [Sneed90] imply that software restructuring and reengineering, respectively, can be helpful. An early case study ([Sneed84]) showed mixed cost-benefit results, but this is not unusual for major work being done for the first time. Later case studies (i.e., [Slovin91] and [Britcher90]) showed positive reengineering results.

Reengineering risk analysis and mitigation

Reengineering is not something that one simply "does." It is easy to waste time and dollars on ineffective approaches. As pointed out in [Ulrich90], one should approach reengineering with a plan. The plan can then be evaluated and risks assessed.

[Arnold91a] cataloged several typical reengineering risk areas, associated risks, and mitigations. Table 2 shows some of these areas and their associated risks.

Table 2. Reengineering Risks
(from [Arnold91a])

Risk Area	Risks
Process Risks	Extremely high manual reengineering costs. Cost benefits not realized in required time frame. Cannot economically justify the reengineering effort. Reengineering effort drifts. Lack of management commitment to ongoing reengineering solution [Ulrich90a].
Personnel Risks	Programmers inhibiting the start of reengineering. Programmers performing less effectively to make an unpopular reengineering project look less effective.
Application Risks	Reengineering with no local application experts available. Existing business knowledge embedded in source code is lost ([Koka91], [Ulrich90a]). Reengineered system does not perform adequately [Koka91].
Technology Risks	Recovered information is not useful or not used [Chikofsky91]. Masses of (expensive) documentation produced. Reverse engineering to representations that cannot be shared. Reengineering technology inadequate to accomplish reengineering goals. Reengineering where there is little reengineering technology support.
Tools Risks	Dependence on tools that do not perform as advertised. Not using installed tools.
Strategy Risks	Premature commitment to a reengineering solution for an entire system. Failure to have a long-term vision with interim goals [Ulrich90a]. Lack of global view: code, data, process reengineering. No plan for using reengineering tools [Ulrich90a].

The message here is to respect reengineering and have contingencies for mitigating risk. In many cases mitigation means trying reengineering on a small scale (e.g., a small subset of programs) and assessing risks before committing to a wholesale reengineering effort.

Future Advances

Several advances in reengineering technology can be expected, and are discussed briefly below. Some of the areas are already supported, and are being extended with CASE tools. Often CASE tool vendors are leading the way in (internal) research because of the product infrastructure that they own.[7]

Software maintainability measurement

There is interest in developing suites of metrics for finding hard-to-maintain code. This code could be a prime candidate for reengineering. Several maintainability metrics and frameworks exist (e.g., [Arnold83], [Arnold82], [Peercy81]). However, more recently, newer maintainability frameworks have emerged ([Oman92a], [Oman92b]).

Reengineering cost benefit models

A medium or large reengineering effort often requires cost-benefit justification. There is a need for organizations to input their maintenance parameters and get a reasonable estimate of the costs, benefits, and timeframe for payoff. More work is needed in making these estimation tools available ([Arnold89], [Sneed91], [Sittenauer92]).

Expert systems for reengineering tasks

Much expertise now exists about reengineering. A logical next step is to capture this expertise in an expert system rules base. The rules base can then be embedded into an existing CASE tool to enhance the value of the tool for users. Example areas where expert systems can be useful are prospecting for reusable parts in source code [Knight92], finding objects in source code, and remodularizing systems.

Reverse engineering into models

Model-based maintenance deals mainly with nonprocedural diagrams that together model an application. The source code is generated, usually with a CASE tool, from the diagrams. Maintenance of source code is done through the diagrams, not directly on the source code. This maintenance paradigm holds great promise for reducing maintenance costs and facilitating software evolution.

CASE tools already exist for model-based maintenance. Methodologies have been developed and are being refined for migrating existing systems into model-based systems. There is room for extending and automating parts of this work.

Software process instrumentation for maintenance

The software process needs to be instrumented to capture software change. This will allow change histories for software to be animated and "played back" to maintainers. This goes beyond configuration management. The change histories must be captured in ways that semantically make sense to maintainers that are browsing them.

[7] Trying to create a CASE tool infrastructure from scratch, without tool-building tools, can be time-consuming and expensive. However, powerful tool-building technology and off-the-shelf tools are now available that can speed the creation of a tool infrastructure.

Summary

The major points made here are

(1) Software reengineering is any activity designed

- to improve one's understanding of software, or
- to improve the software itself, usually for increased maintainability, reusability, or evolvability.

(2) There is no universally accepted definition of reengineering. Several definitions exist that reflect different perspectives on what reengineering is. The approach embodied in the section titled, "The Context of Reengineering," and Appendix A can be used to more precisely characterize definitions of reengineering, reverse engineering, and other terms.

(3) Transformation is central to reengineering and related terms. Key elements of transformations are views of software, an information base, decomposition of software information into objects and relationships in an information base, and composition of views from information in the information base.

(4) There are several reasons why reengineering is significant:

- Reengineering can help reduce an organization's evolution risk.
- Reengineering can help an organization recoup its investment in software.
- Reengineering can make software easier to change.
- Reengineering is big business.
- Reengineering capability amplifies CASE toolsets.
- Reengineering is a catalyst for automating software maintenance.
- Reengineering is a catalyst for applying artificial intelligence (AI) techniques to solve software reengineering problems.

(5) Reengineering technology has three basic themes:

- Improving software,
- Understanding software, and
- Capturing, preserving, and extending knowledge about software.

(6) Reengineering risks exist, but they can be planned for and mitigated.

Appendix A

A decision procedure for classifying a software transformation

To decide if a given transformation (or procedure), transforming information in view D into information in view C, is reengineering, reverse engineering, design recovery, or forward engineering, do the following:

1. Put view D and view C into one of the view classes (1, 2, 3, A1, A2, A3). (See Figure 1 and the discussion of it in the section titled, "The Context of Reengineering".) If D and C cannot be classified, then this decision procedure cannot be used.[8]
2. Classify the transformation according to the following table. A label of "open" in the table means the transformation is not classified. (A transformation not listed in the table is also "open.") It may or may not be an instance of the other terms. More information is needed.

Example

Each example is assumed to meet the necessary conditions of Table A2.

Table A1. Example Transformation Classifications

Transformation	View D, Class	View C, Class	Classification from Table
Source code--> Decision table	Source code, 3	Decision table, 1	Reverse engineering, design recovery, reengineering
Data flow diagram --> Textual specifications	Data flow diagram, 2	Textual specifications, 1	Reverse engineering, design recovery, reengineering
Source code --> Lines of code measurement	Source code, 3	Lines of code measurement, A3	Open
Source code --> Pretty printed source code	Source code, 3	Pretty printed source code (same program and language), 3	Restructuring, reengineering
Source code inline comments --> Revised source code inline comments	Source code inline comments, 3	Revised source code inline comments, 3	Reengineering
Raw data in database --> Logical data schema	Raw data in database, 3	Logical data schema, 1	Reverse engineering, design recovery, reengineering

[8] It is also assumed that the classification has intuitive validity. For example, source code is not placed view class 1, nonprocedural specifications.

Table A2. Classification of Transformations in Figure 1.

Classification	View class of D	View class of C	Other conditions*
Forward engineering	1	2, A2, 3, or A3	
Open	1 or A1	A1	
Reengineering	1	1	V
Forward engineering	A1	1, 2, A2, 3, or A3	
Reverse engineering, design recovery, reengineering	2	1 or A1	V
Reengineering	2	2	V
Open	2 or A2	A2	
Forward engineering	2	3 or A3	
Reverse engineering	A2	1	V
Open	A2	A1	
Forward engineering	A2	2, 3, or A3	
Reverse engineering, design recovery, reengineering	3	1, 2, A1, or A2	V
Reengineering	3	3	V
Restructuring	3	3	V, and both D and C are source code in the same language for the same program.
Open	3 or A3	A3	
Forward engineering	A3	3	
Reverse engineering, design recovery, reengineering	A3	2 or A2	V

*Note: V means that the transformation meets the following conditions:

(1) Improves one's understanding of software, or

(2) Prepares or improves the software itself, usually for increased maintainability, reusability, or evolvability.

The "or" is an inclusive "or."

References

The papers reprinted in this book are marked with an asterisk (*). Papers marked with a pound sign (#) were reprinted in [Arnold86].

#[Arnold82] R.S. Arnold, and D.A. Parker, "The Dimensions of Healthy Maintenance," *Proc. Sixth Int'l Conf. on Software Eng.*, IEEE Computer Society Press, Los Alamitos, Calif., Sept. 1982, pp. 10-27.

[Arnold83] R.S. Arnold, *On the Generation and Use of Quantitative Criteria for Assessing Software Maintenance Quality*, doctoral dissertation, Univ. of Maryland, College Park, Md., 1983.

[Arnold86] R.S. Arnold, *Software Restructuring*, IEEE Computer Society Press, Los Alamitos, Calif., 1986.

[Arnold89a] R.S. Arnold, "Software Reengineering," private seminar notes, Herndon, Va., 1989.

*[Arnold89b] R.S. Arnold, "Software Restructuring," *Proc. IEEE*, Vol. 77, No. 4, Apr. 1989.

[Arnold90a] R.S. Arnold, "Heuristics for Salvaging Reusable Parts from Ada Source Code," Tech. Report: Ada Reuse Heuristics-90011-N, Software Productivity Consortium, Herndon, Va., Mar. 1990.

[Arnold90b] R.S. Arnold, "Software Restructuring: Foundation for Reengineering," *Proc. Reverse Eng. Forum*, St. Louis, Mo., Apr. 1990.

[Arnold90c] R.S. Arnold, "Tools for Static Analysis of Ada Source Code," Tech. Report: Ada Static Tools Survey-90015-N, Software Productivity Consortium, Herndon, Va., June 1990.

[Arnold91a] R.S. Arnold, "Risks of Reengineering," *Proc. Reverse Eng. Forum*, St. Louis, Mo., Apr. 1991.

*[Arnold91b] R.S. Arnold and W.F. Frakes, "Reuse and Reengineering," Final draft (1991) of a paper appearing under the same title in *CASE Trends*, Feb. 1992.

[Arnold92] R.S. Arnold, "Software Reengineering," seminar notes, Software Evolution Technology, Herndon, Va., 1992.

*[Bailey90] J.W. Bailey and V.R. Basili, "Software Reclamation: Improving Post-Development Reusability," *Proc. Eighth Ann. Nat'l Conf. on Ada Technology*, U.S. Army Communications - Electronics Command, Fort Monmouth, N.J., 1990, pp. 477-498.

[Bohm66] C. Bohm and G. Jacopini, "Flow Diagrams, Turing Machines, and Languages with Only Two Formation Rules." *Comm. ACM*, Vol. 9, No. 5, May 1966, pp. 366-371.

*[Breuer91] P.T. Breuer and K. Lano, "Creating Specifications from Code: Reverse Engineering Techniques," *J. Software Maintenance: Research and Practice*, Vol. 3, 1991, pp. 145-162.

*[Britcher90] R.N. Britcher, "Re-engineering Software: A Case Study," *IBM Systems J.*, Vol. 29, No. 4, 1990, pp. 551-567.

*[Burson90] S. Burson, G.B. Kotik, and L.Z. Markosian, "A Program Transformation Approach to Automating Software Re-engineering," *Proc. COMPSAC*, IEEE Computer Society Press, Los Alamitos, Calif., 1990, pp. 314-322.

*[Byrne91] E.J. Byrne, "Software Reverse Engineering: A Case Study," *Software—Practice and Experience*, Vol. 21, No. 12, Dec. 1991, pp. 1349-1364.

[Caldiera88] G. Caldiera, and V.R. Basili, "Reusing Existing Software," Tech. Report CS-TR-2116, Computer Science Dept., Univ. of Maryland, College Park, Md., Oct. 1988.

*[Caldiera91] G. Caldiera and V.R. Basili, "Identifying and Qualifying Reusable Software Components," *Computer*, Vol. 24, No. 2, Feb. 1991, pp. 61-70.

[Calliss88] F.W. Calliss, "Problems with Automatic Restructurers," *SIGPLAN Notices*, Vol. 23, No. 3, Mar. 1988, pp. 13-21.

[Card85] D.N. Card, G.T. Page, and F.E. McGarry, "Criteria for Software Modularization," *Proc. Eighth Int'l Conf. Software Eng.*, IEEE Computer Society Press, Los Alamitos, Calif., 1985, pp. 372-377.

*[Chen90] Y.-F. Chen, M. Nishimoto, and C.V. Ramamoorthy, "The C Information Abstraction System," *IEEE Trans. Software Eng.*, Vol. 16, No. 3, Mar. 1990, pp. 325-334.

*[Chikofsky90] E. Chikofsky and J.H. Cross, "Reverse Engineering and Design Recovery: A Taxonomy," *IEEE Software*, Jan. 1990, pp. 13-17.

[Chikofsky91] E. Chikofsky, lecture notes on reverse engineering and design recovery, Feb. 1991.

*[Cleveland89] L. Cleveland, "A Program Understanding Support Environment," *IBM Systems J.*, Vol. 28, No. 2, 1989, pp. 324-344.

[*Computerworld* 91] *Computerworld*, Vol. XXV, No. 12, Mar. 25, 1991, p. 68.

[Conklin87] J. Conklin, "A Survey of Hypertext," *Computer*, Vol. 20, No. 9, Sept. 1987, pp. 17-41.

[Connell87] J. Connell and L. Brice, *The Professional User's Guide to Acquiring Software*, "Identifying Systems that Need Rework", Ch. 2, Van Nostrand, N.Y., 1987.

*[Corbi89] T. Corbi, "Program Understanding: Challenge for the 1990s," *IBM Systems J.*, Vol. 28, No. 2, 1989, pp. 294-306.

[Cote88] V. Cote, P. Bourque, S. Oligny, and N. Rivard, "Software Metrics: An Overview of Recent Results," *J. Systems and Software*, Vol. 8, 1988, pp. 121-131.

*[Davenport90] T.H. Davenport, and J.E. Short, "The New Industrial Engineering: Information Technology and Business Process Redesign," *Sloan Management Rev.*, Summer, 1990, pp. 11-27.

[Davis76] R. Davis, and J. King, "An Overview of Production Systems." In E.W. Elcock and D. Michie, eds. *Machine Intelligence*, 1976, Wiley, N.Y., pp. 300-332.

*[Dietrich89] W.C. Dietrich, Jr., L.R. Nackman, and F. Gracer, "Saving a Legacy with Objects," *Proc. OOPSLA*, Association for Computing Machinery, N.Y., 1989, pp. 77-83.

*[Dunn91] M.F. Dunn and J.C. Knight, "Software Reuse in an Industrial Setting: A Case Study," *Proc. 13th Int'l Conf. on Software Eng.*, IEEE Computer Society Press, Los Alamitos, Calif., 1991, pp. 329-338.

*[FSMSC87] Fed. Software Management Support Center, "Parallel Test and Evaluation of a Cobol Restructuring Tool," *Office of Software Development and Information Technology*, Falls Church, Va., Sept. 1987.

[Glass81] R.L. Glass and R.A. Noiseux, *Software Maintenance Guidebook*, Prentice-Hall, N.J., 1981.

*[Gopal88] R. Gopal and S. Schach, "Using Automatic Program Decomposition Techniques in Software Maintenance Tools," *Proc. Conf. on Software Maintenance*, IEEE Computer Society Press, Los Alamitos, Calif., 1988, pp 132-141.

[GUIDE89] "Application Reengineering," Guide Pub. GPP-208, Guide Int'l Corp., Chicago, 1989.

*[Hammer90] M. Hammer, "Reengineering Work: Don't Automate, Obliterate," *Harvard Business Rev.*, July-Aug. 1990, pp. 104-112.

*[Harandi90] M.T. Harandi and J.Q. Ning, "Knowledge-Based Program Analysis," *IEEE Software*, Vol. 7, No. 1, Jan. 1990, pp. 74-81.

[Hartman91] J. Hartman, "Understanding Natural Programs Using Proper Decomposition," *Proc. 13th Int'l Conf. on Software Eng.*, IEEE Computer Society Press, Los Alamitos, Calif., May 1991, pp. 62-73.

#[Harrison82] W. Harrison, K. Magel, R. Kluczny, and A. DeKock, "Applying Software Complexity Metrics to Software Maintenance," *Computer*, Vol. 15, No. 9, Sept. 1982, pp. 65-79.

[Heninger78] K. Heninger, J. Kallander, D. Parnas, and J. Shore, "Software Requirements for the A-7E Aircraft," NRL Memorandum Report 3876, Nov. 1978.

#[Hevner89] A.R. Hevner and R.C. Linger, "A Method for Data Re-engineering in Structured Programs," *Proc. 22nd. Hawaii Int'l Conf. on System Sciences*, IEEE Computer Society Press, Los Alamitos, Calif., Jan. 1989, Vol. 2, pp. 1024-1034.

[Husmann90] H.H. Husmann, "Re-Engineering Economics," Eden Systems Corp., Carmel, Ind., 1990. Also appeared in *System Development*, Feb. 1991.

*[Jacobson91] I. Jacobson, "Re-Engineering of Old Systems to an Object-Oriented Architecture," *Proc. OOPSLA*, Association for Computing Machinery, N.Y., 1991, pp. 340-350.

[Knight92] J. Knight, personal communication, telephone conversation between John Knight and Robert Arnold, June 1992.

[Koka91] R. Koka, "Mainframe Realities," *Software Magazine*, Jan. 10, 1991.

*[Kozaczynski89] W. Kozaczynski and J.Q. Ning, "SRE: A Knowledge-Based Environment for Large-Scale Software Reengineering Activities," *Proc. 11th Int'l Conf. on Software Eng.*, IEEE Computer Society Press, Los Alamitos, Calif., 1989, pp. 113-121.

*[Landis88] L.D. Landis, P.M. Highland, A.L. Gilbert, and A.J. Fine, "Documentation in a Software Maintenance Environment," *Proc. Conf. on Software Maintenance*, IEEE Computer Society Press, Los Alamitos, Calif., 1988.

*[Lyle88] J.R. Lyle and K.B. Gallagher, "Using Program Decomposition to Guide Modifications," *Proc. Conf. on Software Maintenance*, IEEE Computer Society Press, Los Alamitos, Calif., 1988, pp. 265-269.

[McCabe76] T. McCabe, "A Complexity Metric," *IEEE Trans. on Software Eng.*, Vol. SE-2, No. 2, Dec. 1976.

[Miller87] J.C. Miller, and B.M. Strauss, "Implications of Automatic Restructuring of Cobol," *SIGPLAN Notices*, Vol. 22, No. 6, June 1987, pp. 76-82.

[Myers75] G.J. Myers, *Reliable Software through Composite Design*, Van Nostrand Reinhold Co., N.Y., 1975.

[Oman92a] P. Oman, J. Hagemeister, and D. Ash, "A Definition and Taxonomy for Software Maintainability," Tech. Report #91-08 (revised version), Software Eng. Lab., Univ. of Idaho, Jan. 1992.

[Oman92b] P. Oman and J. Hagemeister, "Metrics for Assessing Software Maintainability," Tech. Report #92-01, Software Eng. Lab., Univ. of Idaho, Mar. 1992.

#[Peercy81] D.A. Peercy, "A Software Maintainability Evaluation Methodology," *IEEE Trans. on Software Eng.*, Vol. SE-7, No. 4, July 1981, pp. 343-352.

#[Philips84] J.C. Philips, "Creating a Baseline for an Undocumented System — Or What Do You Do with Someone Else's Code?" in *Record of the 1983 Software Maintenance Workshop*, R.S. Arnold, ed., IEEE Computer Society Press, Los Alamitos, Calif., 1984, pp. 63-64.

[Rekoff85] M.G. Rekoff, Jr., "On Reverse Engineering," *IEEE Trans. Systems, Man, and Cybernetics*, Mar.-Apr. 1985, pp. 244-252.

[Reynolds90] R.G. Reynolds and J.C. Esteva, "Learning to Recognize Reusable Software by Induction." White paper, Wayne State Univ., Detroit, Mich., 1990.

*[Rich90] C. Rich, and L.M. Wills, "Recognizing a Program's Design: A Graph-Parsing Approach," *IEEE Software*, Vol. 7, No. 1, Jan. 1990, pp. 82-89.

*[Ricketts89] J.A. Ricketts, J.C. DelMonaco, and M.W. Weeks, "Data Reengineering for Application Systems," *Proc. Conf. on Software Maintenance*, IEEE Computer Society Press, Los Alamitos, Calif., 1989, pp. 174-179.

*[Robson91] D.J. Robson, K.H. Bennett, B.J. Cornelius, and M. Munro, "Approaches to Program Comprehension," *J. Systems and Software* Vol. 14, Feb. 1991, pp. 79-84.

[Rombach89] D.H. Rombach and B.T. Ulery, "Improving Software Maintenance through Measurement," *Proc. IEEE*, Vol. 77, No. 4, Apr. 1989, pp. 581-595.

*[Schwanke91] R.W. Schwanke, "An Intelligent Tool for Re-engineering Software Modularity," *Proc. 13th Int'l Conf. on Software Eng.*, IEEE Computer Society Press, Los Alamitos, Calif., May 1991, pp. 83-92.

[Sittenauer92] C. Sittenauer, M. Olsem, and D. Murdock, "Software Re-engineering Tools Report," Software Technology Support Center (STSC), Hill Air Force Base, Utah, Apr. 1992.

*[Slovin91] M. Slovin and S. Malik, "Reengineering to Reduce System Maintenance: A Case Study," *Software Eng.*, July/Aug. 1991, pp. 14-24.

*[Sneed84] H.M. Sneed, "Software Renewal: A Case Study," *IEEE Software*, Vol. 1, No. 3, July 1984, pp. 56-63.

[Sneed88] H.M. Sneed and G. Jandrasics, "Inverse Transformation of Software from Code to Specification," *Proc. Conf. on Software Maintenance*, IEEE Computer Society Press, Los Alamitos, Calif., 1988, pp. 102-109.

*[Sneed90] H.M. Sneed and A. Kaposi, "A Study of the Effect of Reengineering upon Software Maintainability," *Proc. Conf. on Software Maintenance*, IEEE Computer Society Press, Los Alamitos, Calif., 1990, pp. 91-99.

*[Sneed91] H.M. Sneed, "Economics of Software Re-engineering," *J. Software Maintenance: Research and Practice*, Vol. 3, No. 3, Sept. 1991, pp. 163-182.

*[Ulrich90a] W.M. Ulrich, "The Evolutionary Growth of Software Reengineering and the Decade Ahead," *Am. Programmer*, Vol. 3, No. 10, Oct. 1990, pp. 14-20.

*[Ulrich90-91] W.M. Ulrich, "Reengineering: Defining an Integrated Migration Framework," *CASE Trends*, 4-part series. Nov./Dec. 1990; Jan./Feb. 1991; Mar./Spr. 1991; Summer, 1991.

[Theby91] S. Theby, "Mapping Cobol to Objects," *Proc. Reverse Eng. Forum*, St. Louis, Mo., Apr. 1991.

[Weiser81] M. Weiser, "Program Slicing," *Proc. Int'l Conf. on Software Eng.*, IEEE Computer Society Press, Los Alamitos, Calif., Mar. 1981.

Authors' Index