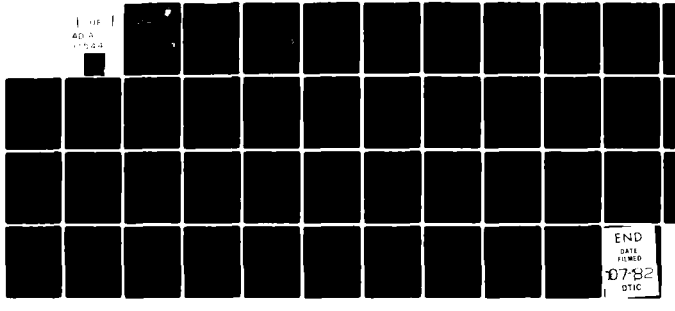
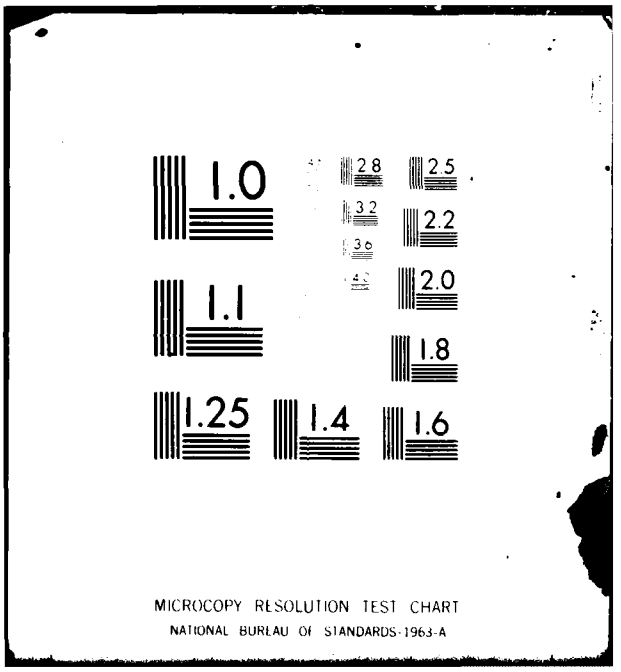
AD-A115 441 ROME AIR DEVELOPMENT CENTER GRIFFISS AFB NY F/G 9/2
SOFTWARE DESIGN METHODOLOGIES - SOME MANAGEMENT PERSPECTIVES,(U)
MAR 82 W E RZEPKA
UNCLASSIFIED RADC-TR-82-50 NL




1.0
2.8
2.5
3.2
2.2
3.6
2.0
1.1
1.8
1.25
1.4
1.6
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

RADC-TR-82-50
In-House Report
March 1982

SOFTWARE DESIGN METHODOLOGIES - SOME MANAGEMENT PERSPECTIVES

William E. Rzepka

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, New York 13441

This report has been reviewed by the RADC Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RADC-TR-82-50 has been reviewed and is approved for publication.

APPROVED:

SAMUEL A. DINITTO, JR.
Chief, Computer & S/W Engr Branch
Command and Control Division

APPROVED:

JOHN J. MARCINIAK, Colonel, USAF
Chief, Command and Control Division

FOR THE COMMANDER:

JOHN P. HUSS
Acting Chief, Plans Office

If your address has changed or if you wish to be removed from the RADC mailing list, or if the addressee is no longer employed by your organization, please notify RADC (COEE) Griffiss AFB NY 13441. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document requires that it be returned.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER RADC-TR-82-50	2. GOVT ACCESSION NO. AD-A115 441	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) SOFTWARE DESIGN METHODOLOGIES - SOME MANAGEMENT PERSPECTIVES		5. TYPE OF REPORT & PERIOD COVERED In-House Report 6. PERFORMING ORG. REPORT NUMBER N/A
7. AUTHOR(s) William E. Rzepka		8. CONTRACT OR GRANT NUMBER(s) N/A
9. PERFORMING ORGANIZATION NAME AND ADDRESS Rome Air Development Center (COEE) Griffiss AFB NY 13441		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS J.O. 55810282
11. CONTROLLING OFFICE NAME AND ADDRESS Rome Air Development Center (COEE) Griffiss AFB NY 13441		12. REPORT DATE March 1982 13. NUMBER OF PAGES 44
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) Same		15. SECURITY CLASS. (of this report) UNCLASSIFIED 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

Same

18. SUPPLEMENTARY NOTES

None

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Software Design Methodology	Software Requirements
Program Design Language	Verification
Software Design Specification	Software Acquisition Management

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The purpose of this report is to provide general information concerning software design methodologies to technical management personnel. In particular, the report is aimed at managers who are in the planning stages of a software development and are facing the decisions of whether to use a formal design methodology and which methodology to use.

To accomplish this objective a definition of design methodology is

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

presented, explained and illustrated. Major design philosophies are discussed and the kinds of applications which they address are described. Following this introductory information, various management perspectives on software design methodologies are presented. In general, these perspectives pertain to the current state of software design methodology development and support tools, as well as relevant application experiences.

TABLE OF CONTENTS

SECTION 1

INTRODUCTION

The development and use of software design methodologies over the past several years has emerged as a result of a combination of factors. The field of software development has been in existence for some 25 years and sufficient maturity has been gained to foster an environment conducive to the development of its theoretical foundations. This maturity, with its many positive and negative experiences, has provided a practical impetus to the development of an engineering discipline for software. These factors have led to the synthesis of several software design methodologies with different software development philosophies, goals and techniques. From all indications, this trend will continue.

It is the purpose of this report to attempt to provide some general information concerning software design methodologies to software acquisition managers. In particular, this report is aimed at managers who are in the planning stages of a software development and are facing the decisions of whether to use a formal design methodology and which kind of methodology to use. In order to do this, a definition of methodology will be presented, explained and illustrated. Major design philosophies will be discussed and the kinds of applications which they address will be described. After this introductory material, various management perspectives on software design methodologies will be presented. In general, these perspectives relate to the current state of design methodology development and support tools as well as relevant application experience.

SECTION 2

METHODOLOGY DEFINITION

A concise and lucid definition of methodology, and the one to be used in this paper, is taken from [BOYD78b]. It defines methodology as

> a complex of work procedures supported by a body of scientific principles which are made effective by an appropriate set of notational and organizational tools.

To place this general definition within the context of software design, it is necessary to understand the phases through which software progresses during its life time. These are described in [ZELK78] as consisting of:

> requirements analysis - an acceptable solution to the problem and how a computer system contributes to the solution is defined.
>
> specification - what the computer and its software are to do are precisely defined.
>
> design - the system structure and functional algorithms are implemented in a programming language.
>
> testing - the implemented system is presented with data representative of its actual use in order to insure that the system satisfies its requirements.
>
> operation and maintenance - the continuous modification of the delivered system to accommodate errors uncovered during testing and changing requirements.

The software design phase primarily involves three activities:

> identification of the functions which will realize the requirements as stated in the specification;

decomposition of these functions into a system structure that consists of small, understandable units which manage discrete data entities; development of algorithms which manipulate these data.

A software design methodology must address these kinds of activities. It will be based on established principles of software architecture. It will contain procedures and guidelines which can be followed in a step-by-step manner to insure the proper application of the principles. Its work procedures will entail the use of notations, such as design languages, and support tools, such as text editors and documentation systems, to enhance both the production process and its final product. The following paragraphs explain the various components of a software design methodology in greater detail and give some examples.

2.1 Software Design Principles

Over the last ten years of software development experience, a set of ideas has emerged and gained acceptance within the software community as standards of good design. These principles are small in number and have been discussed elsewhere [BOYD78b, ROBI78b, DIJK76, HOAR69]. The principles of hierarchical structuring, abstraction, modularity, formal specification and program verification will be briefly discussed here.

Hierarchical structuring of software systems creates a special, tree-style relationship among the constituent components of a software system. The tree has its root node at the top of the structure. This node represents a major function of the design. The tree's end nodes

or leaves are at the bottom and represent sub-functions that together accomplish the major function. The intermediate nodes in the tree provide the control necessary to accomplish the major function. All of the nodes in the tree have the property that if one depends upon another for some function or data then the reverse situation is not also true. This is significant because it allows unrelated functions to be isolated from each other, thereby reducing design complexity.

Abstraction is a concept that has been used by a number of engineering disciplines for a number of years. Its basic concept is to select certain, important properties of a complex entity for the purpose of creating a model which is simpler and easier to understand than the entity itself. In software system design, abstraction is used with both functions and data. In the case of the design of a function, it may be found that another fairly complex function is needed. This sub-function is invoked as if it actually existed, but its details are deferred from the function under design. The required sub-function is designed separately, thereby reducing the complexity of the original function.

During the design of a software function it often occurs that the design is facilitated and made simpler if it is thought of in terms of a data entity which is conceptually richer than the available primitive data entities. Such abstract data types can be conceptualized as needed and then represented by using or combining available primitive data types (integer, array, character, etc.). In addition, the legal operations on the abstract data types are defined in a manner completely analogous to the operations on the primitive

types (addition, concatenation, etc.).

Modularity is the concept of confining specific design decisions or functions into a distinct design element which is as independent of other elements as possible. The independence requires a precisely defined interface for the module. The advantage of such a modularly designed system is that modules can be replaced without disturbing system functions so long as their interface meets the stated requirements. Such replacements are desirable when implementation strategies are changed (storage structure) or to take advantage of newer, faster more economical hardware devices. In each case, such changes could be made to a modularly designed system with the user unaware of their occurrence.

Experience has shown that natural language descriptions of complex, abstract entities like software systems are noted for their ambiguity and lack of precision. Formal language specifications of software design provide a precise, unambiguous and implementation independent statement of design. Such specifications are represented in a formal notation which is capable of expressing powerful design concepts such as abstract data types, hierarchical structuring and modularization in addition to capturing, independent of implementation, the state changes which occur in a system as a result of its invocations. This kind of formal specification of software systems is essential to understanding and communicating design ideas among even small groups of people.

Program verification refers to a formal procedure for establishing that a program correctly transforms its input state to a

desired output state and then terminates. This is done by describing the output and input states and the semantics of the program transformations in a formal and implementation independent manner. Verification conditions are established for each transformation and are then proved correct. Termination is then established for iteration sequences.

The concepts and procedures of program verification are equally applicable to software system designs if the system can be structured and formally specified as a number of small, hierarchically related programs or modules. In this manner, the complexity of a large system is reduced and the proof process becomes one of establishing the correctness of one design level, termed an abstract machine, at a time. This is done by proving the correctness of the machine's components (modules) and by showing the relationships of the data and operations to those of some lower level machine. Once established the proof of the next lower level can proceed independent of other, higher levels.

2.2 Software Design Work Procedures

The primary purpose of a software design methodology's work procedures is to enable the practitioner to apply the methodology's principles in a straightforward step-by-step manner. The work procedures provide the user with guidelines for structuring, specifying and verifying a software design. The work procedures will not insure a creative design process or an elegant and correct end product, but they do serve to structure and unify this difficult

development. They are typically stated using a combination of formal steps or stages, informal guidelines and small, illustrative examples.

The work procedures of the Rational Design Methodology (RDM) [BOYD78b] consist of eight distinct, formal steps: requirements interpretation, requirements analysis, static (preliminary) design and verification, external review, detailed (procedural) design and verification, documentation, technical review, and design implementation. Because of the highly creative nature of the activities involved, it is impossible to provide a "cookbook" formulation for any of these steps. Instead, the general objectives of each step are stated along with the products that are their expected outputs. Guidelines for achieving the objectives and for producing the products are also included. The goal of RDM's static design step is described as being the formalization of "the informal identification of storage and functionality" that was recognized in the first two requirements analysis steps. Typical guidelines include checking that all assumptions are made explicit, structuring design specifications to read similar to the requirements statements and testing design specifications against the informal requirements. Numerous succinct, lucid examples are provided. They complement the written work procedures by bringing many of the individually stated procedures together in one place and in a complete manner. They also answer the many questions about design practice which inevitably arise from the independent descriptions of the work procedures.

Yourdon, Inc., describes five work procedures involved in the data flow oriented, Structured Design [YOUR75b] technique of transform

analysis. They are:

restating the problem as a data flow graph;

identifying the most abstracted forms of the input/output data;

first-level factoring or identification of major system functions;

factoring of these functions until lowest level modules are identified;

accommodating exceptions or departures from transform analysis.

Guidelines include suggestions for developing data flow graphs, such as, avoiding control logic and ignoring initialization and termination conditions. Yourdon provides many and varied examples mostly taken from business applications.

2.3 Notational Tools

Notations for expressing design are one aspect of design methodology where there is certainly no lack of development. Indeed, the number of design languages exceeds accompanying methods. This is because each formal methodology has a preferred specification approach which includes a notation, and also because program design languages have been developed for use as a documentation tool independent of any methodology.

In general, design notations may be classified as graphic or linguistic and the latter category may be further partitioned into procedural and non-procedural languages. The graphic notations, like flowcharts, HIPO diagrams [RADC75b] and SADT diagrams [SOFT76b] primarily depend upon pictures of data or control flow (or both) to express design. Their obvious advantage is the mental associations that can be devised to show abstract, informational concepts. Their

disadvantages include their ambiguity, lack of preciseness, small amount of per page information and large maintenance costs. The language based notations overcome the problems of ambiguity, imprecision and maintenance. The ease with which their information may be comprehended is sacrificed to a certain extent, because of their formal syntax. However, this is a short term problem, usually associated with the casual reader, which training and practical experience can resolve.

Among the procedurally oriented languages, there are those like Honeywell's P-Notation [BOYD78b] which provides language constructs, such as input and output assertions and formal semantics for its transforms, so that program verification can be performed. This, of course, does not preclude informal applications. Other procedural languages, such as the ISDOS Problem Statement Language (PSL) [TEIC74] and SREM's Requirement Statement Language (RSL) [ALFO77] do not actively support formal verification. Instead they provide a process oriented design description with the goal of being able to check its consistency, ambiguity, traceability and in the case of RSL its testability. In addition, English text is allowed for documentation purposes. Still other procedurally oriented languages, such as IBM's PIDGIN [RADC75b], Caine, Farber and Gordon's PDL [CAIN75], Jet Propulsion Laboratory's Software Design and Documentation Language [KLEI77] and Chu's Software Design Language - 1 [CHU76] have as their primary goal the documentation and communication of design information. In general, these languages utilize the basic forms of structured programming to express design logic. Language support for

data structures is not, in general, as strong. They all allow the use of free form English text to supplement and document design decisions.

A non-procedural language which is being actively developed and used is SRI International's "Specification and Assertion Language" (SPECIAL) [ROUB77]. It is an extension and refinement of the software specification language first introduced by Parnas [PARN72]. Its primary goal is an implementation independent statement of design decisions. This is accomplished by treating the design as a set of modules which are finite state machines, and by specifying the design in terms of the state change effects that module functions cause when they are activated. This language has been used to specify and prove the correctness of several software designs [ROBI78a] [WENS76].

2.4 Organizational Tools

The purpose of these tools is to provide an environment for the effective application of the principles, procedures and notations of a software design methodology. The tools are intended to be computer-based and to include all documentation on the methodology and its use. The generic types of tools needed to create a hospitable methodology environment include aids for generating and maintaining design notation and documentation, support for managing the resources and activities associated with the design process and software tools to assist in design analysis.

Tools for creating and maintaining design notations and documentation are typically available on general purpose computer systems in the form of text editors and report formatting and

production systems. Developments here continue to be motivated by general requirements for word processing capabilities.

Support tools for managing the resources and activities associated with a software system development have been suggested as part of the Rational Design Methodology. These tools would allow management to allocate and track design resources such as personnel, expenditures and schedules, and activities such as assignment of responsibility for parts of the design and conduct of various formal and informal reviews. The information upon which the management tools would operate would be stored in a centralized design data base.

Design analysis has received a great deal of attention because of the amount of useful information that can be provided by machine assisted checking of formally specified designs. Tools ranging from specification syntax checkers to formal verification systems have been constructed. Most developmental activity has occurred in the area of language syntax and semantics checking and analysis of design consistency and completeness.

The ISDOS Project at the University of Michigan developed the Problem Statement Language and Analyzer (PSL/PSA) [TEIC74] as a prototype of design specification and analysis systems. It provides the capability of specifying designs in a formal, process oriented language and of entering this information into a design data base. Design analysis includes checks for functional completeness and naming and interface consistency as well as selective displays of the data base. This information is presented using a fixed report format.

A successor to PSL/PSA, the Software Requirements Engineering

Methodology [ALFO76], employs a Requirements Statements Language (RSL) to specify the stimulus/response activity across system/sub-system interfaces. RSL has capabilities for defining new design concepts, for enhanced retrieval capabilities and for expressing system process dynamics. The Hierarchical Development Methodology [ROBI78b] developed at SRI International has as its objective the formal specification and verification of software designs. SRI has developed a set of computer-based tools to support this goal. A formal specification language (SPECIAL) and analyzer have been developed. This language allows an implementation-independent, non-procedural specification of design to be constructed, and the analyzer assists the designer in checking the syntactic and semantic consistency of the formal specifications and data representations. A language (Hierarchy Specification Language) has also been implemented for specifying the structure of a software system, i.e., the module inter-relationships. Associated tools analyze the specification, checking for the consistency of the hierarchy. A language for specifying implementation decisions as abstract programs has been proposed. This language would be based on an existing programming language, such as Ada or PASCAL. Tools for assisting in the formal verification of the abstract programs could be based on similar verification aids developed by SRI for other high-level programming languages. Tools for simulating software designs using only their design specifications are in the planning stages.

SECTION 3

MANAGEMENT PERSPECTIVES

With the foregoing definition of a software design methodology, it is the intent of the following paragraphs to provide additional, detailed management perspectives on how such methodologies may be profitably used. In particular, these discussions are intended to assist the software acquisition manager who is contemplating the use of a design methodology on a large-scale, system development. Although small projects (less than 5 people) would benefit from the use of a methodological approach to software design, it is in the larger software development efforts where the advantages of structure, formalism and organized procedures result in major improvements in the consistency, understandability and confidence level of the design.

3.1 State of Development

Probably the most noticeable attribute of software design methodologies is their condition of constant change. This evolutionary process is a natural condition in any emerging technology. It exemplifies the developer's ambitions to improve their product and usually occurs as a result of some experience gained in actual use of the methodology. If the experience is positive, the resulting change in the methodology may reflect refinements made to reinforce that experience. Negative results will require changes to correct the situation.

The evolutionary changes in design methodologies are evidenced in

the literature. In 1975 SRI developed, and later published [ROBI76b], a 5 stage, unnamed methodology for the design and proof of large operating system software. It consisted of five stages: hierarchical decomposition, formal specification, data representation, abstract implementation and coding followed by a formal verification stage. At the Second International Conference on Software Engineering, SRI's methodology [NEUM76] had been reformulated into the following stages: interface definition, hierarchical decomposition, module specification, mapping functions (data representation), (abstract) implementation followed by a formal verification stage. A more recent report [ROBI78b] expands the now named Hierarchical Development Methodology into eight stages: conceptualization, external interface definition, intermediate interface definition, formal specification, formal (data) representation, abstract implementation, coding and verification. Much of the motivation for these changes is derived from applications of the methodology to the design of a secure operating system [NEUM77], a fault-tolerant computer system [WENS76] and a message processing system [ROBI78a]. The changes result in a clearer view of software system design which improves future applications and increases the level of design confidence. The drawback is, of course, the difficulty one encounters in attempting to learn or use such a "moving target." This evolutionary trend will continue through the mid-1980's as the technology of software engineering continues to mature. Until a steady state is reached, the project manager must make choices based upon a "frozen" version of the methodology. This is possible with even the most volatile of methodologies. For

example, a version of SRI's HDM which is oriented toward secure software design and verification is being established and maintained by SRI in conjunction with another software company. Other methodologies which have been commercially marketed, such as Yourdon's Structured Design and SofTech's SADT, have reached stability as a marketing necessity. By the late-1980's, the knowledge base of these methodologies will expand sufficiently so that the trial and error of the 70's will result in a well-developed, maturing software engineering discipline.

3.2 Hidden Costs

One immediately evident payoff in the use of software design methodologies is a marked increase in understanding of the design by the software designers and programmers engaged in the system development work. Although this increase in understanding has not been scientifically measured, most people utilizing design methodologies or program design languages report a positive effect on understanding. These effects are best explained in terms of the way in which systems are currently understood.

Current systems are usually expressed with a combination of informal English textual descriptions and pictorial representations, such as hierarchy charts and flowcharts. English text is ambiguous and imprecise, fostering inconsistencies and lulling the reader into a false sense of understanding. He thinks he understands the intended meaning of the design representation when in fact his own background, education and experience usually provide him with a slightly different

interpretation of the same words. Similarly, the use of flowcharts can cause the same kind of misunderstandings because of their emphasis on control flow, coupled with a corresponding lack of data representation, the lack of standards for the level of detail presented and their partial reliance on English text.

It is the lack of formalism and structure that is responsible for the lack of understanding on the part of specification users/readers, that is, software engineers. The employment of a medium for design expression which incorporates these formal qualities will improve this situation. The precision of formal specifications make explicit the design decisions which otherwise would be imprecisely specified in English. Any imprecision forces the reader to make assumptions which often conflict with the intended design. If such inconsistencies go unresolved they become implemented in software and are not discovered until much later - in system test and integration, or worse, in actual use.

Since formal specifications are written in an artificial language, there is no context upon which to base faulty assumptions. Each decision is precisely written down. The specification language forces the reader to think about these decisions and either understand or question them. Formal specifications do not completely eliminate doubts and assumptions. However, their terseness greatly reduces this problem because it does not provide the background against which unrelated personal preferences can influence intended meanings.

Software design methodologies which incorporate techniques for simplifying complex systems can improve the capacity of an individual,

working as part of a group of designers/implementors or reviewers, to better understand that system. Examples of such specifications exist [NEUM77, BOYD78b], and on-going work in the computer security area will further demonstrate their usefulness. However, project managers must realize that there is a subtle cost involved in the benefits of increased project understanding resulting from use of formal specifications. The cost is associated with the overhead of learning to skillfully use the expressive powers of the language. This is non-trivial, because powerful concepts, such as hierarchical structuring, modularity, abstraction and stepwise refinement, are expressed in concise notation. Also the mathematical tools of the predicate calculus, such as predicates, quantifiers, assertions and invariants, are often used in formal specification languages. These notions and their use require significant formal training and experience. Finally, the technique of non-procedural description of state transitions is relatively unfamiliar to the control flow orientation of most people whose initial encounters with computers occurred as a result of programming or logic design.

In summary, formal specifications are not intended for casual reading. They do exact a start-up price for their increased understanding benefits. Also, because of this factor it is important that the formal specifications be accompanyed with adequate informal (English language) commentary to provide the intuitive understanding of the larger design goals which are difficult to grasp when formal specifications are read one line at a time.

3.3 Guidelines and Procedures

The guidelines and procedures of a software design methodology are a critical part of its total design approach. They represent the mechanics used to apply the principles of the methodology to a system design. These mechanics are intended to guide the designer through the various steps of the design process and to produce a high quality design product expressed in accordance with the notational conventions of the methodology.

There is an important distinction between what the methodology guidelines and procedures can and cannot do. They cannot instruct or guide the designer in any of the creative parts of the design process. Thus, the actual design decisions made are solely those of the designer, reflecting his intellectual capabilities, educational background and design experience. As stated by SRI in [ROBI78b], their Hierarchical Design Methodology "is a complete medium for stating design and implementation decisions once they are made, but it does not - in and of itself - provide the criteria for making these decisions". A methodology's procedures are intended to improve the quality of design in the sense that it is well-structured, precisely and unambiguously specified, consistent and as complete an understanding of the system as the requirement specifications allow. The value of the application of the procedures, then, is in the refinement, structure and representation of initially stated design decisions and in the improvement potential that these design qualities motivate.

3.4 Documentation Quality

Documentation on the theory and practice of software design methodologies varies in kind and quality. The kind of documentation appears to reflect the interests or current market of the writers. One group of methodology developers, including those working in the academic/research environments, tend to produce a series of technical papers concerning major aspects of their developments. The papers will represent the methodology documentation, until a major application is undertaken. At this point a comprehensive description of the methodology is usually generated.

Another writing trend seems to be the documentation package or book produced by a group which is eager to market their design strategy. This approach is common to the business data processing world. The documentation tends to be voluminous [YOUR75b, DEMA78] and is usually written with many examples. The package usually includes a formal course which presents the large quantity of material in summary form, so as to provide a general understanding of the overall methodology. An accompanying workshop provides specific skills in applying the methodology's procedures to a small problem. Under this schema, the documentation becomes a reference source for further, more complex applications of the methodology.

Honeywell's WELLMADE is an example of a methodology which has been presented in several papers [BOYD75, BOYD76, BOYD77, BOYD78a] and then definitized in an Air Force sponsored demonstration [BOYD78b] involving the design of a programming support library. Many aspects of the SRI Hierarchical Development Methodology were presented in papers

[NEUM76, ROBI76a, ROBI76b] prior to a general description which was included as part of the design approach for a secure operating system [NEUM77]. Yourdon's Structured Design [YOUR75b] is an example of the business world documentation trend.

In general, the quality of software design methodology documentation is affected by the evolutionary nature of the methods themselves. Most noticeable is the lack of a standard set of terminology. Mills and Ferrentino [MILL77] describe an abstract machine as a synchronized set of sequential processes where localized steps are carried out by finite state machines. This abstract machine is specified using a procedural language. SRI International [ROBI78b] defines an abstract machine as a level of a hierarchically structured system providing a set of operations to the programs or users of that machine. It is specified as a finite state machine using a non-procedural language. Honeywell [BOYD78b] considers an abstract machine to be the basic component of a system. It includes not only a set of programs which provide a procedural description of the machine's data and operations but also including the machine's functional specification, usage information, acceptance criteria, data types, predicates, initialization conditions, data invariants and permanent data.

Another common problem of contemporary documentation is a lack of detailed procedures by which the principles of the methodology are applied. The primary reason for this omission is the difficulty in describing guidelines for thought processes which, to a large extent, deal with the creative aspects of design. Even where guidance is given

it tends to be quite general; for instance, this suggestion for design is taken from the Honeywell WELLMADE methodology [BOYD78b]: "Structure the specifications by using abstract predicates in such a way that the formal specifications read similar to the informal statements of requirements."

The extent to which the available work procedures are documented varies considerably. The data flow oriented methodologies, whose procedures consist of relatively informal, graph oriented techniques, appear to be well-documented [YOUR75a, YOUR78] with numerous examples. The formal methodologies, whose procedures concentrate upon the application of formal specification languages, are, in comparison, not nearly as well documented. Although the specification languages are adequately described [ROUB77, BOYD78b], their usage is only generally described [ROBI76a] and examples are scarce. This situation appears to be improving with the late 1978 publication of comprehensive documentation on the procedures of the Hierarchical Development Methodology [LEVI78].

As methodologies evolve, documentation will undoubtedly improve. This will occur as a result of their application to various design problems and the developer's attempts to bring their methodologies to as complete a state as possible.

3.5 Design Evaluation

Between the time that a software system is designed and specified and the time its actual implementation begins, several reviews or evaluations are normally conducted. The approaches for conducting such

reviews vary, depending on the desired goals of the reviews. The goals range from an informal understanding that a designed system meets its requirements to a formal, mathematical verification that a design completely and consistently satisfies its requirements. Satisfaction of design review requirements usually implies that both functional (what a system does) and performance (how well a system does) considerations be made.

The various contemporary methodologies have adopted evaluation approaches that are either formal or informal - recognizing that these are mutually exclusive. They primarily concentrate on the evaluation of whether a system meets its functional requirements. Performance evaluation at the detailed design level is just beginning to be explored. Discrete event simulations of systems remains as the primary means of obtaining performance data on system designs.

Among the informal evaluation approaches, the methodology of Structured Design [YOUR75b] concerns itself with attempting to improve design quality by verifying that its cohesion (the degree to which a system is decomposed on the basis of functional relatedness) and coupling (a measure of module interdependencies) characteristics are optimal. Optimal here means highly related functions and few intermodule connections. Top-down design methods [RADC77] have adopted the "structured walk through" as a means of program design review. More formal methodologies, such as Honeywell's WELLMADE methodology, rely on a precise and unambiguous specification of the design as the basis for a logical and convincing argument that the design satisfies its requirements. SRI's Hierarchical Development Methodology

incorporates a separate verification phase as part of its methodology. This consists of a formal demonstration that a system has no missing parts (completeness) and that there is an absence of conflicts among the parts (consistency). In the case of HDM, this means that the system's data representations and programs correctly implement its formal specifications.

Hughes Aircraft has integrated a system functional requirements methodology based on PSL/PSA and a set of general computer hardware and software models into a Design Analysis System [WILL78] for evaluation. This system utilizes inputs from a requirements data base to provide parametric data to a set of generalized computer system performance analysis models. The models expose system bottlenecks and indicate design areas where performance could be improved.

Recent research in the area of design evaluation indicates that usable tools for assessment of a system's functional completeness are still several years away. A technique [ROBI78c] under development at SRI is attempting to simulate a system's functional behavior using only its design specification. On-going work at USC/ISI [BALZ77] also provides a basis for functional simulations based upon a high-level, procedural design language. In both cases, the work relies on the emerging technology of symbolic execution and so must be considered high risk and long range but with potentially high payoff.

In summary, design evaluation techniques vary from the informal walk-throughs to the formal verification procedures. Clearly the quality of the design will improve as the formality and intensity of the review increases. Correspondingly, the costs and schedules for

the reviews will expand. The choice of evaluation approach must be a function of the degree of software reliability required. Most commercial systems can receive adequate evaluation scrutiny using informal techniques. High reliability applications, such as required for computer security or life dependent systems like airplane autopilots, must look toward the formal verification techniques.

3.6 Tool Spectrum

An important aspect of software design methodologies is the set of tools which assist users in their application of the methodology to a design problem. In general, the tools assist in the documentation and evaluation of software designs. These two functions provide a convenient framework for discussing the tools.

Some documentation tools assist the user in applying a formalized language to express the decisions made during the design process. The primary assistance is in the form of syntactic and semantic analysis. Syntactic checks are indispensible for insuring that errors of form involving declarations, expressions and simple punctuation as well as naming conflicts and overall design module structure are detected and eliminated. Semantic analysis can expose hierarchy and interface inconsistencies, and through the mechanism of type checking provides powerful assistance to the concepts of abstract data types and information hiding. SRI has implemented an analyzer which performs these functions for their Specification and Assertion Language SPECIAL [ROUB77], as has HOS Inc. for their specification language AXES [HOS76]. Honeywell has proposed a similar analyzer for their design

language, P-Notation [BOYD78b].

CADSAT [TEIC77] and SREM [ALFO76] both have specification languages and analyzers based upon the prototype Problem Statement Language and Analyzer, PSL/PSA, developed by the ISDOS Project at the University of Michigan [TEIC74]. PSL/PSA stores a system's functional requirements in a relational data base and utilizes various retrieval and report generation routines to access and present desired results. Typical reports include lists of information in the data base organized by class of object and displays of data and processes, their structure and their interrelationships.

Other documentation tools, such as those associated with the Structured Design [YOUR75b], SADT [SOFT76b] and top-down design methodologies rely on the use of graphic depictions, such as bubble charts, data flow diagrams and HIPO diagrams as a means of design expression. These diagrams have a small set of symbols, limited conventions and are supported by automated tools. In the case of SADT, automated graphics tools are being developed as a result of its use on the Air Force sponsored, Integrated Computer Aided Manufacturing (ICAM) Project. The non-trivial problems associated with flowchart maintenance have prompted some groups [RICH77, GODD78] to develop automated flowcharters.

A third kind of documentation tool is the program design language (PDL). It is best characterized as a medium for the expression of detailed software design and is most frequently used as an alternative to flowcharts. Program design languages are not associated with any specific design methodology and so do not support any particular

design philosophy, such as hierarchical or data flow. Their language forms are quite uncomplicated, usually following a format that has become known as "Structured English" - free form English text placed within the syntax of the sequence, if-then-else, do while and case constructs of structured programming. IBM's PDL [RADC75b], Caine, Farber and Gordon's PDL [CAIN75] and Jet Propulsion Laboratory's Software Design and Documentation Language [KLEI77] are examples of this trend in design languages.

Program design languages have been used extensively on various design problems including the use of IBM's PDL on a large Air Force radar system development. In general, feedback from their usage is very positive [HAZL78] with users indicating a high degree of satisfaction with the PDL as a means of informing technical management, programmers and new project personnel of the nature of the detailed software design. In addition, since PDLs are stored, processed (syntax analysis), changed and printed from computer based tools, their maintenance costs are substantially less than those associated with flowcharts.

General approaches for analyzing designs have already been described in the section on Design Evaluation. This section will focus on tools which assist the designer in carrying out those approaches. SRI's Hierarchical Development Methodology has a comprehensive, integrated set of tools for design evaluation [ROBI78b]. Processors which check the internal (syntax, type) and external (type, completeness) consistency of the specification language (SPECIAL) and abstract programming language (ILPL) have been constructed. In

addition, SRI has implemented a Hierarchy Specification Language which is used for describing a design hierarchy. Processors exist to check the design hierarchy and interface consistency of systems described in HSL.

Honeywell has proposed the design of a complete set of tools to support its WELLMADE design methodology [BOYD78b]. These include general text processing tools (text entry, retrieval, update, report generator, etc.), management support tools (access control, resource management, status reporting, etc.) and design analysis (hierarchy, module, procedure analyzers, static performance analysis). TRW has designed and implemented an analyzer for the SREM's Requirements Statement Language. The Air Force has developed a similar type analyzer for its CADSAT User Requirements Language. These were described in the section on Design Evaluation. HOS Inc. has implemented a processor for its AXES specification language. Hughes Aircraft has developed extensive simulation models for its Design Analysis System [WILL78]. These tools are based on generalized computer system models which are parametrized for a specific application. This approach greatly reduces the costly and time-consuming software development required by most discrete event type simulations of large systems.

Draper Labs has developed a methodology for requirements specification and preliminary design of real-time systems [DEWO77]. A specification language and a set of support tools, Design-Aids for Real-Time Systems (DARTS), has also been developed to support design expression. DARTS supports specifying the design, saving the

specification in a data base, editing and displaying the specification and performing various types of data flow analyses upon the specification.

Design tools are extremely useful for performing the kind of rapid mechanical checks of large and complex specifications which computers do so well and which humans do so poorly. Despite these advantages, project managers must keep in mind that use of such tools does not guarantee a flawless design. It is entirely possible for a design to be considered complete and consistent as a result of tool analysis and for it to be simultaneously flawed because the designer did not sufficiently understand the problem or did not make wise design decisions.

Project managers must also be wary of the maturity of software design tools. As discussed in the section on the state of tool development, many methodologies and their tools are undergoing continuous change. Using tools which are evolving can result in frustration as well as cost increases and schedule delays. Project managers must assure themselves that the tools and their associated documentation and training materials are stable, mature and bug-free, before placing their use on a project's critical path.

3.7 Performance Representation

The development of software design methodologies has been toward an integrated approach for better specifying and understanding the functions of complex system designs. The equally interesting questions of how well a given design performs and how this degree of performance

can be improved have not been as intensively studied. The primary reason for this is that performance assessment requires detailed knowledge of how a system works and this conflicts with modern design principles which stress the separation of design and implementation considerations. As a result, performance assessment procedures are only infrequently found as parts of software design methodologies.

The Software Requirements Engineering Methodology [ALFO76] incorporates design validation points where performance criteria can be associated with the resource usage at that stage. Other methodologies are just beginning to introduce performance assessment as a part of design. For example, C.S. Draper Laboratory has a Real-Time System Design Methodology under development which utilizes exchange functions [FITZ77] to express inter-process communication and coordination. This process structure provides a basis for real-time performance analysis. Honeywell has investigated space requirements analysis and execution time performance as part of its WELLMADE methodology. The storage space requirements are written as relations expressing the limitations on abstract data types. Execution time performance is based on Knuth's frequency analysis techniques [KNUT]. Probably the best developed performance analysis system in existence today is the Hughes Design Analysis System (DAS). As discussed above, the DAS bases performance measurement on discrete event simulation models which are parameterized for a particular problem. Models for system operations and data processing subsystem analysis have been developed and have been successfully used in many applications. A more recently developed performance analysis tool is the Performance

Oriented Design (POD) system developed by BGS Inc. POD enables analysis of existing computer system configurations on the basis of queuing network models [BUZE75].

In summary, software design methodologies are lacking in performance assessment procedures. Some promising techniques are beginning to be developed, but much work remains before this aspect of design will mature into an engineering discipline. Until that time simulation models provide the best source of performance information prior to the implementation stage of system development.

3.8 Applications

The application of software design methodologies to practical problem situations is a vital part of methodology development. The information gained from these experiences is the primary source of "debugging" a methodology. As one might expect the teaching of a methodology to a group of people previously unexposed to formal design methods also produces illuminating feedback on the methodology itself.

Feedback from methodology application experiences tends to impact the organizational and notational tools as opposed to its principles and procedures. This is because the principles, such as hierarchical structure, data abstraction and modularity are becoming increasingly understood, used and accepted by software engineers. For example, design procedures like the cohesion and coupling strategies of Constantine [STEV74], the constructive approach of WELLMADE [BOYD77], the transform analysis techniques of YOURDON [YOUR75b] and the abstract machine approach of SRI [ROBI76b] have been developed to a

relatively high degree of stability, whereas specification languages and tools which analyze them for completeness and consistency, present a set of syntax and semantics which tend to reflect the personal preferences of their originators. Once subjected to the rigors of usage by individuals outside the development group, these languages and tools are quite likely to change. For example, Frances and Friedman [FRAN77] report 21 distinct problems with the User Requirements Language and its analyzer, both parts of the Computer Aided Design, Specification and Analysis Technique (CADSAT), as a result of applying it to the development of the system requirements specification (MIL-STD-490 Type A) for an Air Force surveillance system. Command inconsistencies, inadequate control over report formats, cumbersome interfaces to the data base and excessive use of reserve words are among the problems documented.

The other major benefit of applying software design methodologies to practical problem situations lies in the transfer of their technology from the laboratory to the system development environment. In general, software engineers, like their counterparts in other disciplines, solve design problems by adapting models of existing solutions that are known to work. Applications of software design methodologies to actual system developments represent models of the design solution as well as instructive case studies in using the methodologies.

To date most applications of software design methodologies [HOS77, BOYD78b, SOFT76a] have been on purposely constrained problems. The rationale for this has been the desire to show the methodology

without complicating the demonstration with an overly complex problem. This is useful for introductory learning purposes but is not the kind of full-scale "live" development that results in a working solution and a useful model. In this latter category there have been relatively few applications of design methodologies. Probably the most well known is the use of the SRI Hierarchical Development Methodology to design a secure UNIX operating system kernel [FORD78]. This development will also utilize the Honeywell WELLMADE methodology to verify that the implementation accurately reflects the detailed design. Other major exercises of software design methodologies include the application of HDM to the design of a secure operating system [NEUM77] and the use of SofTech's Structured Analysis and Design Technique [SOFT76b] on the Air Force Air Materials Laboratory's Integrated Computer Aided Manufacturing (ICAM) Project.

As more large scale applications of software design methodologies are performed, more information on their utility and effectiveness will become available. It must be remembered, however, that although initial results will be worthwhile, especially with respect to the effectiveness of procedures and tools, it is the long term, system life cycle effects which are most interesting. Only these kinds of data will be able to verify whether the increased resources expended in applying these methods to the early stages of system development will significantly reduce implementation, testing and operational maintenance costs.

3.9 Verification

In certain systems, requirements for information security and high reliability may require that the system design be formally proven correct. Until recently, such requirements could not be satisfied because of a lack of formal verification technology. At this time, SRI'S Hierarchical Development Methodology [ROBI75b], System Development Corporation's Formal Design Methodology (FDM) and the University of Texas's GYPSY methodology all incorporate formal verification as an integral design step.

The SRI approach consists of a combination of design tactics which together present a verification strategy. Large and complex computer software systems (such as operating systems) are functionally decomposed into a hierarchy of machines, where the operations upon the data of any machine depends upon the primitive operations of another, lower level machine. This system layering begins with the user interface and progresses through various supporting machines until the actual target machine hardware is reached. Examples of some of the abstract machines found in an operating system are the user command language, the paging machine and the segmentation machine.

Each machine is decomposed into a set of functional modules, where each module defines a data object and the operations upon it. The only access to the data object is via the module's operations. This approach hides design decisions from other modules and other machines which need not be aware of them. This simultaneously reduces system complexity and enhances system modularity. These design techniques are applied in an effort to simultaneously design a system and decompose it into a set of separate entities each of which can be

separately analyzed for correctness. The results are specified using a formal language which is capable of expressing design decisions without simultaneously specifying their implementation. The language SPECIAL [ROUB77], is non-procedural; it describes a design in terms of the effects caused by invocation (execution) of its components, their interactions, and how they change the states of the data objects. The data are represented as a set of expressions and the relationships between data at adjacent levels are formally represented. In addition, each module is procedurally implemented as an abstract program which although not executable, does describe how the design is to be implemented.

Using this technique, the process of verification consists of assuring that the module specifications are correctly realized by the data representations and the abstract programs. This means not only that the completeness and consistency of the representations and programs be established with respect to the module specifications, but also that the abstract programs be proven to correctly implement the state transformations prescribed by the specification. The program proof process involves showing that a set of program verification conditions are true and that the program terminates. The verification conditions are generated from assertions which are made about the input and output states of the program and the kinds of state transformations which the program makes. This process is a difficult and time-consuming one demanding precise consistency checking of complex expressions. It is described in detail in [ROBI77].

Because the verification process is so complex, it is important

to have as much of it as possible performed with machine assistance. SRI has constructed tools for HDM which automatically check the consistency of the system design hierarchy, its interfaces, the module specifications, data representations and abstract programs. An automatic verification system has been under development for several years and has been successfully used in the verification of moderately sized programs. A specialized security verification system has been developed [FEIR80] and is being used by Ford Aerospace on the DARPA sponsored, Kernelized Secure Operating System (KSOS) project.

Applications of the verification step of the HDM methodology have been made in the area of secure operating system design [NEUM77], a software implemented fault tolerant system design [WENS76] and the design of a DEC PDP-11 UNIX based secure operating system. Plans currently exist for the implementation of only the UNIX based system.

Probably the most important issues faced by project managers who require verification as part of their software development are the availability of a stable methodology and tool set, the qualifications of the personnel and the limitations of verification technology.

Of the three verification methodologies referred to above, the HDM appears to be the most accessible. Documentation and a formal training course are available, as is a stable set of tools. Access to and maintenance of these resources on a continuing basis is currently being established.

Unfortunately, information concerning the usability of the FDM is not available because of SDC's proprietary policies concerning it. It has been used in the retrofit of a security kernel for the VM/370

operating system.

The GYPSY system has not yet been applied to major system developments. As a result, availability information cannot be reliably stated.

The second issue, that of personnel qualifications, is very important, because two distinct requirements exist. First, the designer must be experienced in software system design and must be capable of expressing that design in accordance with the methodology being used. In the case of the HDM, this means a hierarchy of abstract machines. The designer must also be capable of expressing the design in the non-procedural specification language, SPECIAL. Second, verification of the design and the subsequent implementation requires an additional skill. A strong mathematical background in the predicate calculus is necessary to prepare input to and interpret the results of automatic verification systems, that is, theorem provers. Both of these qualities are not readily available in typical software development project personnel.

Finally, project management must realize that use of verification systems does consume considerable computer resources. This is because of their size and the complexity of the problems with which they deal. Also, it is important that application program size be constrained, since the largest programs currently processed by theorem provers are in the range of 350 source lines of a higher order language [NEUM78].

SECTION 4

SUMMARY AND CONCLUSIONS

From the perspective of a software development manager, the use of a modern software design methodology presents much less risk and much more potential for improvement than it did just several years ago. Certainly, troublesome aspects still exist. The constantly changing nature of design methods and their user interfaces will likely continue for some time. Documentation with varying styles, inconsistent terminology and lack of detailed application procedures is also a problem. But despite these annoyances, the majority of evidence indicates that the application of design methodologies in the software development process is, without question, beneficial. The formalism employed by design methodologies exposes problems and encourages questions, thereby naturally increasing understanding of designs. A methodology's procedures and guidelines improve design quality by increasing the likelihood of well-structured, precise and unambiguous design specifications. Design evaluation techniques provide consistency checking of individual software development phases, like requirements and design, simulation of a design's functional capabilities and even formal verification of design properties over several development phases including implementation. Tools are available to assist in the mechanical aspects of design specification, evaluation and documentation. The specification and analysis of a design's performance properties can also be done using simulation techniques. Unfortunately, not all of these advances are embodied in a single, unified methodology.

Although the results of methodology applications have shown that user interfaces need improvement, the general conclusion has been that the principles which serve as the basis for the methods are sound. Hierarchial structuring, abstraction, modularity, formal specification and verification have been demonstrated to be a satisfactory foundation for a software engineering methodology. Their continued, systematic use will result in improved software developments. The methodology applications also serve as models of how the software engineering of large, complex problems should be carried out. This is a significant mechanism for the transfer of these ideas into practice.

The evidence gathered to date indicates that the best advice for software acquisition managers who are contemplating the use of a modern software design methodology is to certainly proceed. However, this advice must also be accompanied with the caution that, because the methods are relatively new, close management of the process must be maintained.

BIBLIOGRAPHY

[ALFO76] M.W. Alford, "A Requirements Engineering Methodology for Real-Time Processing Requirements," Technical Report TRW-SS-76-06, TRW, Redondo Beach CA (September 1976).

[ALFO77] M.W. Alford, L.W. Berrie, C.L. Pasine, "Requirements Development using SREM Technology - Volume I," BMD Report 27332-6921-029, Ballistic Missile Defense Advanced Technology Center, Huntsville AL (October 1977).

[BALZ77] R. Balzer, N. Goldman, D. Urle, "Informality in Program Specification," Technical Report ISI/RR-77-59, USC/Information Sciences Institute, Marina Del Rey CA (1977).

[BOYD75] D.L. Boyd, A. Pizzarello, "Introduction to the WELLMADE - Design Methodology," Proc. 3rd Intl. Conf. on Software Engineering (May 1975).

[BOYD76] D.L. Boyd, G.J. Gustafson, "The Design Methodology WELLMADE and its Relationship to the Software Generation Process: An Overview," Honeywell CRC Report (October 1976).

[BOYD77] D.L. Boyd, "WELLMADE Design Methodology," Proc. Honeywell Software Productivity Symposium, Minneapolis MN (April 1977).

[BOYD78a] D.L. Boyd, A. Pizzarello, "Introduction to WELLMADE - A Rational Software Design Methodology," Proc. 16th IEEE Computer Society International Conference, Spring Compcon 78 (March 1978).

[BOYD78b] D.L. Boyd, A. Pizzarello, S.C. Vestal, "Rational Design Methodology," Report HR-78-257:17-38 Honeywell Inc., Corporate Computer Sciences Center, Minneapolis MN (June 1976).

[BUZE75] J.P. Buzen, "Cost Effective Analytic Tools for Computer

Performance Evaluation," Proceedings COMPCON 75, Washington DC (September 1975).

[CAIN75] Program Design Language Reference Guide, Caine, Farber & Gordon, Inc. Pasadena CA (July 1975).

[CHU76] Y. Chu, "Introducing a Software Design Language," Proc. 2nd International Conference on Software Engineering (October 1976).

[DEMA78] T. DeMarco, Structured Analysis and System Specification, (Yourdon, Inc., New York NY 1978).

[DEWO77] J.B. DeWolf, R.N. Principato, "A Methodology for Requirements Specification and Preliminary Design of Real-Time Systems," Draper Lab Report C-4923, C.S. Draper Laboratory, Cambridge MA (July 1977).

[DIJK76] E. Dijkstra, A Discipline of Programming, (Prentice-Hall Inc., Englewood Cliffs NJ, 1976).

[FEIR80] R. Feiertag, "A Technique for Proving Specifications are Multilevel Secure," Technical Report, SRI International, Menlo Park CA (January 1980).

[FITZ77] D.R. Fitzwater, "The Formal Design and Analysis of Distributed Data Processing Systems," Computer Sciences TR-295, Univ. of Wisconsin, Madison WI (March 1977).

[FORD78] "Secure Minicomputer Operating System (KSOS) Executive Summary - Phase I Design of the DOD Kernalized Secure Operating System," Ford Aerospace and Communication Corp., Palo Alto CA (March 1978).

[FRAN77] J.W. Francis, M.P. Friedman, "Application of URL/URA to the GEODSS System Specification," MITRE WP-21223, MITRE Corp., Bedford MA (April 1977).

[GODD78] G. Goddard, M. Withworth, E. Strovink, "JOVIAL Structured

Design Diagrammer (JSDD)," RADC-TR-78-9 Vol II (AD AO52730), Rome Air Development Center, Griffiss AFB NY (February 1978).

[HAZL78] M. Hazle, "Program Design Languages and the Air Force Requirement for Software Design Documentation," M78-203, MITRE Corp., Bedford MA (February 1978).

[HOAR69] C.A.R. Hoare, "A Axiomatic Approach to Computer Programming," Comm ACM, 12, 10 (Oct 1969).

[HOS76] "AXES Syntax Description," HOS Technical Report TR-4, HOS Inc., Cambridge MA (December 1976).

[HOS77] "The Application of HOS to PLRS," HOS Technical Report No. 12, HOS, Inc., Cambridge MA (November 1977).

[KLEI77] H. Kleine, "Software Design and Documentation Language," NASA Jet Propulsion Laboratory, California Institute of Technology, Pasadena CA (July 1977).

[KNUT] D.E. Knuth, The Art of Computer Programming, Vols 1, 2, 3.

[LEVI78] K.N. Levitt, L. Robinson, "A Detailed Example of the Use of HDM," TRCSL-72, SRI Project 4828, Contract N00123-76-C-0195, SRI International, Menlo Park CA (April 1978).

[MILL77] H.D. Mills, A.B. Ferrentino, "A Semantics-Based Design Method," IBM/FSD Internal Memo (April 1977).

[NEUM76] P.G. Neumann, R.J. Fiertag, K.N. Levitt, and L. Robinson, "Software Development and Proofs of Multi-Level Security," Proc. 2nd Intl Conf. on Software Engineering, 13-15 Oct 76, San Francisco CA (October 1976).

[NEUM77] P.G. Neumann, R.S. Boyer, R.J. Fiertag, K.N. Levitt, and L. Robinson, "A Provably Secure Operating System: The System, Its

Applications and Proofs," Final Report, SRI Project 4332, Stanford Research Institute, Menlo Park CA (February 1977).

[NEUM78] P.G. Neumann, "NewsFlash : the Automatic Verification of a 327-Line Program", Software Engineering Notes, 3,4 (October 1978).

[PARN72] D.L. Parnas, "A Technique for Software Module Specification with Examples," Comm ACM, 15,5 (May 1972).

[RADC75] RADC, "Structured Programming Series, Chief Programmer Team Operations Description," RADC-TR-74-30 Vol 10 (AD AO08861), Rome Air Development Center, Griffiss AFB NY (January 1975).

[RADC75b] RADC, "Structured Programming Series, Program Design Study," RADC-TR-74-300 Vol 8 (AD AO16415), Rome Air Development Center, Griffiss AFB NY (May 1975).

[RADC77] RADC, "An Application of Formal Inspections to Top-Down Structured Program ," RADC-TR-77-212 (AD AO41645), Rome Air Development Center, Griffiss AFB NY (June 1977).

[RICH77] P.K. Richards, "Developing Design Aids for An Integrated Software Development System," General Electric, Information Systems Programs, Sunnyvale CA (November 1977).

[ROBI76a] L. Robinson, "Specification Techniques," Proc. 13th Design Automation Conference (June 1976).

[ROBI76b] L. Robinson, K.N. Levitt, P.G. Neumann, A.R. Saxena, "A Formal Methodology for the Design of Operating System Software," in Current Trends in Programming Methodology, Vol 1, R.T. Yeh, ed. (Prentice Hall, New York NY, May 1977).

[ROBI77] L. Robinson, K.N. Levitt, "Proof Techniques for Hierarchically Structured Programs," Comm. ACM Vol 20, No. 4, pp. 271-283 (April 1977).

[ROBI78a] L. Robinson and O.M. Roubene, "The Preliminary Design of a Family of Message Processing Systems Using HDM," Technical Report CSL-71, SRI Project 4828, Contract N00123-76-C-0195, SRI International, Menlo Park, CA (April 1978).

[ROBI78b] L. Robinson, "HDM-Command and Staff Overview," Technical Report CSL-49, SRI Project 4828, Contract N00123-76-C-0195, SRI International, Menlo Park CA (February 1978).

[ROBI78c] L. Robinson, K.N. Levitt, "A Basis for Module Simulation," Technical Report CSL, SRI Project 4828, Contract N00123-76-C-0195, SRI International, Menlo Park CA (November 1978).

[ROUB77] O.M. Roubine, L. Robinson, "The SPECIAL Reference Manual," TR-CSL- 45, SRI Project 4828, Contract N00123-76-C-0195, Stanford Research Institute, Menlo Park CA (January 1977).

[SOFT76a] SofTech, "Functional Area Requirements and Functional Description of Prototype System (Engine Inventory and Control)," SofTech, Inc., Waltham MA (August 1976).

[SOFT76b] SofTech, "An Introduction to SADT - Structured Analysis and Design Technique," SofTech, Inc., Waltham MA (November 1976).

[STEV74] W.P. Stevens, G.J. Myers, L.L. Constantine, "Structured Design," IBM Systems Journal 13, 2 (February 1974).

[TEIC74] D. Teichroew, E.A. Hershey, M.J. Bastarache, "An Introduction to PSL/PSA," ISDOS Working Paper No. 86, University of Michigan (March 1974).

[TEIC77] D. Teichroew, A. Hershey, S. Spewak, "User Requirements Language (URL), User's Manual Part 1 (Description), H6180/MULTICS/Version 3.2," USAF Electronics Systems Division, Hanscom

AFB MA (March 1977).

[WILL78] R.R. Willis, "DAS - An Automated System to Support Design Analysis," 3rd International Conference on Software Engineering (May 1978).

[WENS76] J.H. Wensley, M.W. Green, K.N. Levitt and R.E. Shostak, "The Design Analysis and Verification of the SIFT Fault Tolerant System," Proc. 2nd Intl Conf. on Software Engineering, 13-15 Oct 76, San Francisco CA, pp. 458-469 (October 1976).

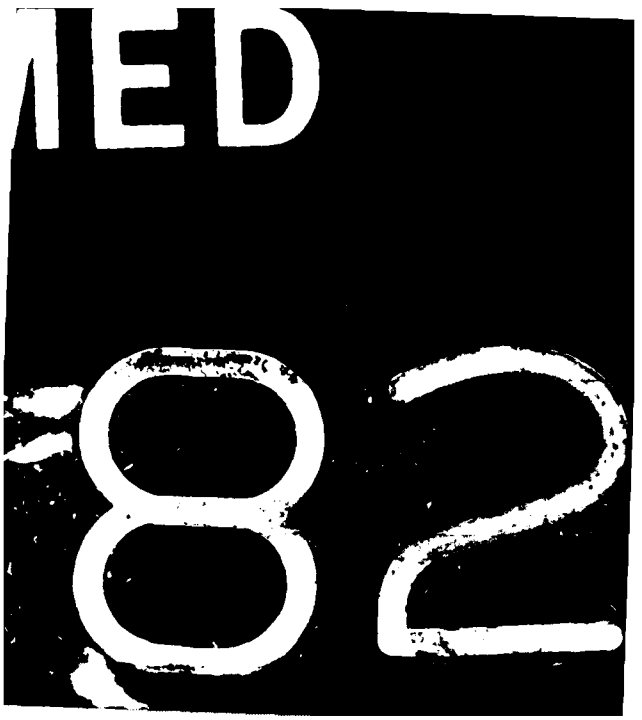
[YOUR75a] E. Yourdon, L.L. Constantine, Structured Design, (Yourdon, Inc., New York NY 1975).

[YOUR75b] Yourdon, Structured Design, Third Edition, (Yourdon, Inc., New York NY December 1975).

[ZELK78] M.V. Zelkowitz, "Perspectives on Software Engineering," Computing Surveys, 10, 2 (June 1978).

MISSION
of
Rome Air Development Center

RADC plans and executes research, development, test and selected acquisition programs in support of Command, Control Communications and Intelligence (C^3I) activities. Technical and engineering support within areas of technical competence is provided to ESD Program Offices (POs) and other ESD elements. The principal technical mission areas are communications, electromagnetic guidance and control, surveillance of ground and aerospace objects, intelligence data collection and handling, information system technology, ionospheric propagation, solid state sciences, microwave physics and electronic reliability, maintainability and compatibility.


ED
82