UNCLASSIFIED

AD NUMBER
ADB102655
LIMITATION CHANGES
TO: **Approved for public release; distribution is unlimited.**
FROM: **Distribution authorized to DoD and DoD contractors only; Administrative/Operational Use; MAY 1986. Other requests shall be referred to Air Force Armament Lab., Eglin AFB, FL 32542.**
AUTHORITY
AFSC/MNOL ltr dtd 13 Feb 1992

THIS PAGE IS UNCLASSIFIED

AFATL-TR-85-93

Common Ada® Missile Packages (CAMP)

Volume II: Software Parts Composition Study Results

Daniel G. McNicholl
Constance Palmer, et al.

McDONNELL DOUGLAS ASTRONAUTICS COMPANY
POST OFFICE BOX 516
ST. LOUIS, MISSOURI 63166

DTIC
ELECTE
JUN 1 8 1986
S D
D

MAY 1986

FINAL REPORT FOR PERIOD SEPTEMBER 1984 - SEPTEMBER 1985

DISTRIBUTION LIMITED TO DOD AND DOD CONTRACTORS ONLY; THIS REPORT DOCUMENTS ~~TEST AND EVALUATION~~; DISTRIBUTION LIMITATION APPLIED SEPTEMBER 1985. OTHER REQUESTS FOR THIS DOCUMENT MUST BE REFERRED TO THE AIR FORCE ARMAMENT LABORATORY (FXG), EGLIN AIR FORCE BASE, FLORIDA 32542-5000.

DESTRUCTION NOTICE: DESTROY BY ANY METHOD THAT WILL PREVENT DISCLOSURE OF CONTENTS OR RECONSTRUCTION OF THE DOCUMENT.

WARNING: This document contains technical data whose export is restricted by the Arms Export Control Act (Title 22, U.S.C. 2751 et seq) or Executive Order 12470. Violation of these export - control laws is subject to severe criminal penalties. Dissemination of this document is controlled under DOD Directive 5230.25

® Ada is a registered trademark of the U.S. Government, Ada Joint Program Office

AIR FORCE ARMAMENT LABORATORY
Air Force Systems Command*United States Air Force*Eglin Air Force Base, Florida

DTIC FILE COPY

86 6 17 004

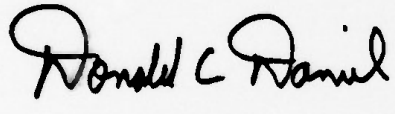
AD-B102 655

NOTICE

When Government drawings, specifications, or other data are used for any purpose other than in connection with a definitely related Government procurement operation, the United States Government thereby incurs no responsibility nor any obligation whatsoever; and the fact that the Government may have formulated, furnished, or in any way supplied the said drawings, specifications, or other data, is not to be regarded by implication or otherwise as in any manner licensing the holder or any other person or corporation, or conveying any rights or permission to manufacture, use, or sell any potential invention that may in any way be related thereto.

This technical report has been reviewed and is approved for publication.

FOR THE COMMANDER



DONALD C. DANIEL
Chief, Aeromechanics Division

Even though this report may contain special release rights held by the controlling office, please do not request copies from the Air Force Armament Laboratory. If you qualify as a recipient, release approval will be obtained from the originating activity by DTIC. Address your request for additional copies to:

Defense Technical Information Center
Cameron Station
Alexandria, Virginia 22314

If your address has changed, if you wish to be removed from our mailing list, or if the addressee is no longer employed by your organization, please notify AFATL/FXG, Eglin AFB, FL 32542

Copies of this report should not be returned unless return is required by security considerations, contractual obligations, or notice on a specific document.

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: Distribution limited to DOD and DOD Contractors only; this report documents ~~test and evaluation~~; distribution limitation applied September 1985.
2b DECLASSIFICATION/DOWNGRADING SCHEDULE	
4 PERFORMING ORGANIZATION REPORT NUMBER(S)	5 MONITORING ORGANIZATION REPORT NUMBER(S): AFATL-TR-85-93
6a NAME OF PERFORMING ORGANIZATION: McDonnell Douglas Astronautics Company; 6b OFFICE SYMBOL (If applicable)	7a NAME OF MONITORING ORGANIZATION: Aeromechanics Division, Air Force Armament Laboratory
6c ADDRESS (City, State and ZIP Code): P. O. Box 516, St. Louis, MD 63166	7b ADDRESS (City, State and ZIP Code): Eglin AFB, FL 32542-5434
8a NAME OF FUNDING/SPONSORING ORGANIZATION: STARS Joint Program Dffice; 8b OFFICE SYMBOL (If applicable)	9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER: F08635-84-C-028D
8c ADDRESS (City, State and ZIP Code): Room 3D139 (1211 Fern St.), The Pentagon, Washington, D.C. 20301-3081	10 SOURCE OF FUNDING NOS.

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT NO.
63756A			

11 TITLE (Include Security Classification): Common Ada® Missile Packages (CAMP), Volume II:

12 PERSONAL AUTHOR(S): McNicholl, Daniel G., Palmer, Constance, Cohen, Sanford G., Whitford, William H., Goeke, Gerard O.

13a TYPE OF REPORT	13b TIME COVERED	14 DATE OF REPORT (Yr., Mo., Day)	15 PAGE COUNT
FINAL	FROM Sep 84 TO Sep 85	May 1986	185

16 SUPPLEMENTARY NOTATION: SUBJECT TO EXPORT CONTROL LAWS.
Availability of this report is specified on verso of front cover.

17 COSATI CODES		
FIELD	GROUP	SUB GR.

18 SUBJECT TERMS (Continue on reverse if necessary and identify by block number): Reusable Software, Missile Software, Software Generators, Ada, Parts Composition, Systems, Software Parts.

19 ABSTRACT (Continue on reverse if necessary and identify by block number): The objective of the CAMP program is to demonstrate the feasibility of reusable Ada software parts in a real-time embedded application area; the domain chosen for the demonstration was that of missile flight software systems. This required that the existence of commonality within that domain be verified (in order to justify the development of parts for that domain), and that software parts be designed which address those areas identified. An associated parts cataloging scheme and parts composition system were developed to support parts usage.

→ See p 1

20 DISTRIBUTION/AVAILABILITY OF ABSTRACT	21 ABSTRACT SECURITY CLASSIFICATION
UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ OTIC USERS ☒	Unclassified

22a NAME OF RESPONSIBLE INDIVIDUAL	22b TELEPHONE NUMBER (Include Area Code)	22c OFFICE SYMBOL
Christine Anderson	(904) 882-2961	AFATL/ FXG

DD FORM 1473, 83 APR — EDITION OF 1 JAN 73 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

11. TITLE (CONCLUDED)

Software Parts Composition Study Results

3. DISTRIBUTION/AVAILABILITY OF REPORT. (CONCLUDED)

Other requests shall be referred to the Air Force Armament Laboratory (FXG), Eglin Air Force Base, Florida 32542-5434.

PREFACE

This report describes the work performed, the results obtained, and the conclusions reached during the Common Ada Missile Packages (CAMP) contract (F08635-84-C-0280). This work was performed by the Computer Systems & Software Engineering Department of the McDonnell Douglas Astronautics Company, St. Louis, Missouri (MDAC-STL) and was sponsored by the United States Air Force Armament Laboratory (FXG) at Eglin Air Force Base, Florida. This contract was performed between September 1984 and September 1985.

The MDAC-STL CAMP program manager was Dr. Daniel G. McNicholl (McDonnell Douglas Astronautics Company, Computer Systems and Software Engineering Department, P.O. Box 516, St. Louis, Mo. 63166) and the AFATL CAMP program manager was Christine M. Anderson (Air Force Armament Laboratory, Aeromechanics Division, Guidance and Control Branch, Eglin Air Force Base, Florida 32542).

This report consists of three volumes. Volume I contains overview material and the results of the CAMP commonality study. Volume II contains the results from the CAMP automated parts engineering study. Volume III contains the rationale for the CAMP parts.

Commercial hardware and software products mentioned in this report are sometimes identified by manufacturer or brand name. Such mention is necessary for an understanding of the R & D effort, but does not constitute endorsement of these items by the U.S. Government

ACKNOWLEDGEMENT

Special Thanks to the Armament Division Standardization Office and to the Software Technology for Adaptable, Reliable Systems (STARS) Joint Program Office for their support of this project.

(The reverse of this page is blank)

Accesion For	
NTIS CRA&I	☐
DTIC TAB	☑
Unannounced	☐
Justification	
By	
Distribution /	
Availability Codes	
Dist	Avail and / or Special
D-16	

QUALITY INSPECTED 3

TABLE OF CONTENTS

Section	Title	Page
I	INTRODUCTION	1
II	CATALOGING OF THE CAMP PARTS	2
	1. Introduction	2
	2. Review	2
	3. Issues	6
	4. Catalog Definition	10
	5. Documentation Requirements	13
	6. Organizational Factors	13
III	EVALUATION OF SOFTWARE GENERATION TECHNOLOGY	18
	1. Introduction	18
	2. Review	18
	3. Assessment	36
	4. Conceptual Framework	47
	5. Recommendations	54
IV	THE ROLE OF EXPERT SYSTEMS	61
	1. Expert System Overview	61
	2. Schematic Part Constructors	63
	3. Generic Instantiator	67
	4. Parts Identification	67
	5. Parts Catalog	68
	6. AMPEE System	69

TABLE OF CONTENTS (CONCLUDED)

Section	Title	Page
V	EVALUATION OF AN EXPERT SYSTEM	72
	1. Introduction	72
	2. Means of Evaluation	74
	3. Overview of ART	74
	4. Evaluation of ART with respect to the Problem Domain (AMPEE).....	81
	5. Conclusions	85
VI	SOFTWARE PARTS COMPOSITION CONCLUSIONS	86
Appendix		
A	DEFINITION OF THE CAMP PARTS CATALOG ATTRIBUTES.....	89
B	CAMP CATALOGING FORM	103
C	SAMPLE DBMS IMPLEMENTATION OF THE CAMP PARTS	107
D	THE FINITE STATE MACHINE CONSTRUCTOR	111
	REFERENCES.....	169

LIST OF FIGURES

Figure	Title	Page
1	Graphical Representation Methods	8
2	Catalog Attributes	11
3	Information Flow through Catalog System	12
4	Organizational Factors	14
5	Forms of Motivation	14
6	Goals of Automating the Software Generation Process	19
7	Summary of Review	21
8	Specification Techniques	27
9	Issues/Criteria of a SGS	37
10	Software Generation without Parts Reuse	39
11	Software Generation with Parts Reuse	39
12	Summary of Specification Techniques	44
13	Facilities of a Software Generation System	48
14	Overview of the Ideal Software Generation System	49
15	The Ideal Software Generation System	50
16	Major Component Requirements of an Ideal SGS	51
17	Parts Identification	56
18	An Example of Parts Identification with an Expert System	57
19	Parts Construction with an Expert System	57
20	Near-Term Technology Requirements	58
21	Mid-Term Software Generation System	60
22	The Structure of an Expert System	62
23	Overview of a Schematic Part Constructor	64
24	The Lateral/Directional Autopilot Schematic	65
25	The Navigation Schematic	66
26	Sample Parts Identification Rules	68

LIST OF FIGURES (CONCLUDED)

Figure	Title	Page
27	Overview of the AMPEE System	70
28	Operations Provided by the AMPEE System	71
29	Why ART was Selected for Evaluation	73
30	ART Features	76
31	Issues/Criteria of a SGS	83
A-1	Catalog Attributes	91
A-2	CAMP Parts Taxonomy	101
B-1	The Cataloging Form	105
C-1	Database Schema	109
D-1	A Finite State Machine	112
D-2	Architecture	117
D-3	Control Flow	118
D-4	Data Flow	119
D-5	Overview of the Proof-of-Concept Implementation	120

SECTION I
INTRODUCTION

This volume contains the results of the work performed on CAMP in the development of a software parts catalog and in the design of a prototype software parts composition system.

Section II describes the results of the CAMP software parts cataloging study and the cataloging scheme recommended for CAMP. The goal of the software cataloging task was to develop a method of describing and managing software parts to increase the productivity of the parts user. In addition to providing the structure for a textual catalog, the cataloging scheme developed on CAMP is suitable for automation. Appendices A through C present more detailed information on the CAMP cataloging scheme.

Section III contains the results of the CAMP software generation study and presents our view of the functionality of a software parts composition system. Although our major area of study was the automatic generation of software using parts, this examination included an investigation of software generation systems which did not handle parts.

Section IV discusses the role of an expert system in the construction of a software Parts Composition System (PCS). The prototype software parts composition system we designed during CAMP was based on incorporating all functions within an expert system to maximize the sharing of data between components of the PCS.

Section V describes the particular expert system tool, the Automated Reasoning Tool (ART), used on CAMP and discusses its applicability in the software parts composition system application. This tool was used to construct a proof-of concept implementation of a schematic part. Appendix D contains a description of this schematic part constructor.

Section VI discusses the major conclusions of the software cataloging and software parts composition system studies.

SECTION II
CATALOGING OF THE CAMP PARTS

1. Introduction 2
2. Review 2
3. Issues 6
4. Catalog Definition 10
5. Documentation Requirements .. 13
6. Organizational Factors 13

1. INTRODUCTION

The objective of this portion of the CAMP study was the development of a procedure to facilitate storage and retrieval of software parts for use on other projects. To achieve this objective, an Ada parts cataloging scheme was developed which provides a means for organizing, indexing, describing, and referencing these parts.

In Paragraph 2 existing software cataloging schemes are reviewed. Major issues related to software parts description are discussed in Paragraph 3. A detailed description of the cataloging scheme that was developed is provided in Paragraph 4. In addition to developing the cataloging scheme, the documentation required to support it has also been identified (see Paragraph 5). Paragraph 6 contains recommendations for the organizational structure needed to support the implementation and use of this scheme.

2. REVIEW

For many years people have advocated the use of existing software parts as a way to reduce the cost of software development and maintenance. Although there is currently a great deal of research being conducted in this area, significant inroads have not been made in the workplace; notable exceptions include Raytheon Missile Systems (References 1 and 2), and The Hartford Insurance Group (References 3 and 4), which have achieved a significant level of reuse in their business data processing departments.

Reuse of software has been successful in the area of mathematical and statistical packages such as the standard routines (e.g., sine, cosine)

usually supplied by compiler vendors, or the International Math and Statistical Library (IMSL) and Numerical Algorithms Group, Inc. (NAG) software libraries. The primary reason for this success is that users understand the functioning of the routines and have confidence in their quality. For instance, in the case of the IMSL, the routines undergo extensive testing; they are developed with strict adherence to standards; the code is robust, efficient, and accurate; technical support is provided; and the documentation (i.e., the catalog) is comprehensive and standardized.

One of the most significant reasons for the failure of many past reusability efforts is that a disproportionate emphasis has been placed on development of software components, while little or no effort was expended on developing a method for describing and identifying available parts.

As a prelude to developing an Ada parts catalog, existing catalogs, cataloging standards, and descriptive techniques were examined. On-going research efforts in software descriptive techniques were also reviewed. None of the catalogs, techniques, and standards reviewed pertained specifically to Ada (some did not even pertain specifically to software), but they provided useful background and insight into what is required of an Ada parts catalog. A survey of our findings follows.

a. Catalog Standards

Of the standards reviewed, those that had the greatest applicability to this task were the ones dealing with computer program abstracts. These were useful in determining the attributes and level of support necessary to develop a viable Ada parts catalog.

One such standard was developed by the National Bureau of Standards (Reference 5); it was intended to be applicable to all programs developed or acquired by Federal departments and agencies. In this scheme, an abstract provides a synopsis of capabilities, environmental requirements, and other relevant information that can assist a user in determining the functionality and appropriateness of a particular program.

The American National Standards Institute has also developed a standard for computer program abstracts--ANSI X3.88-1981 (Reference 6). In addition to the usual items (i.e., name, date, contact), the standard recommends the inclusion of a category field, keywords, program status within the lifecycle (e.g., requirements definition phase, in development,

operational), and assumptions and limitations (e.g., assumptions about the form and range of the input data).

b. Existing Software Catalogs

Although there are a number of software catalogs in existence today, many use basically the same attributes (i.e., name, id, abstract, etc.) to describe the software parts. In the summaries that follow, the features of existing catalogs that may be of particular interest in developing an Ada missile parts catalog are highlighted.

The Data and Analysis Center for Software (DACS) (Reference 7) software catalog makes use of a software engineering thesaurus to determine the proper classification of a software part both at the time of cataloging and at the time of retrieval. The thesaurus contains a listing of major areas, called cluster terms, and separate listings of subjects within the major area (e.g., MODELS is a major topic area that is found in the list of cluster terms; listed separately under MODELS, the user finds the field further subdivided into AVAILABILITY MODELS, BEHAVIOR MODELS, RELIABILITY MODELS, etc.). The individual subject items may be further decomposed. Attributes not found in most other catalogs examined include stage of development, purpose of development, target computer, documentation, and references.

The National Technical Information Service (NTIS) (Reference 8) utilizes a standard form to collect data for each software part that will come under NTIS control. The parts are classified by category; there are 40 major categories (e.g., Aeronautics and Aerodynamics, Astronomy and Astrophysics, Computers, Control, and Information Theory), and 342 subcategories. A subject classification booklet (Reference 9) is used to categorize incoming software, and to assist the user in locating cataloged software. Each category is given together with the titles of all of its subcategories (e.g., within Computers, Information, and Control Theory there are subcategories such as Computer Hardware, Computer Software, Control Systems and Control Theory, Information Processing Standards). Each subcategory is presented with a listing of the subject areas covered by that subcategory (e.g., within Computer Software are the subjects Computer Programming, Programming Languages, Compilers, etc.); a cross reference to related sections is also provided.

The NTIS catalog entries are not characterized by any unique information; but the catalog does have multiple indices that allow access via a number of different keys. The entries are indexed by subject (keyword, title, id), producing agency (agency name and location, agency id, title, product number, id), id number (either NTIS id or originating agency id, title), hardware (entries are in alphanumeric order by hardware type), and language (alphanumeric order by source language).

The IMSL (International Math and Statistical Library) Catalog (Reference 10) consists of mathematical and statistical routines, and has gained widespread acceptance and use because of the consistent quality of the routines. All modules conform to established coding and documentation standards, and contain both in-line and external documentation. The algorithms are categorized by the area of mathematics or statistics to which they apply, and the categories are alphabetized and organized into chapters in the documentation. The routines are kept alphabetically within category. Within each documentation chapter there is a quick reference guide to the purpose of each routine.

For each category, modules are described by the following information: routine name (label), brief statement of purpose, precision/hardware, and other required IMSL routines. Documentation for each routine contains routine name, purpose, call line, arguments (argument name; type; usage, i.e., input, input-output, output, work arrays, error parameters), precision/hardware, required IMSL routines, notation, remarks, algorithm, an example, and optionally, notes and accuracy.

The Numerical Algorithms Group, Inc. (NAG) (Reference 11) has a library of subroutines for mainframes and a small package of routines for the personal computer. The documentation provided for the routines is available both in hard-copy form and on-line. As with the IMSL routines, extensive, standardized documentation is provided, and the routines go through an extensive validation and certification process before being released. Naming conventions for the routines are strictly adhered to; this is important in increasing readability. Updates to the collection of routines are published well in advance of the effective date in order to allow users time to accommodate these changes. The routines are supported on a wide range of hardware.

The Collected Algorithms of CACM (Reference 12) provide extensive certification information for each algorithm. This includes 'certified by' information, explanatory remarks, test procedures, results, and comments.

Some other catalogs examined include the Micro Software Report (Reference 13), and the International Computer Programs Software Directory (Reference 14).

c. Descriptive Techniques

The Naval Research Laboratory, as part of its Software Cost Reduction Project (References 15 and 16), developed a module descriptive technique for software parts. Reusability was specifically addressed in this project. The researchers determined that reusability is promoted by well-defined and well-documented software. Information hiding and data abstraction are two techniques that were used to achieve this goal. An abstract interface specification technique was developed to allow interfaces to be specified without requiring internal details. Modules were designed to be flexible (i.e., easily modifiable) rather than general. Ada, through its package facility, provides many of the features desired by the researchers on the Software Cost Reduction project.

Much attention was given to documentation and the form it should take. Module documentation is precise and detailed; it is collected into module guides that serve as a software catalog. The documentation explains how requirements are allocated among the modules, and defines the scope and contents of individual design documents. A precise abstract is also provided.

3. ISSUES

Three major categories of issues arose during the investigation of cataloging Ada missile software parts. These issues address the following areas: the cataloging scheme, the cataloging mechanism, and organizational requirements. These areas are not independent of each other and the interdependencies will be pointed out in the discussion that follows.

a. The Cataloging Scheme

Reuse of software parts can be implemented at a number of different levels. Reusability can be defined as reuse of analysis (e.g., systems analysis, domain analysis), reuse of design, or reuse of code. The implementation level affects both the structure of a parts catalog (i.e., the attributes needed to describe the part), and the organizational and procedural requirements needed to support the use of such a catalog. There are several views as to the appropriate level at which reuse should take place. For example, Neighbors (Reference 17) indicates that to be meaningful, reuse should encompass analysis and design in addition to code.

A number of researchers, have pursued the idea of parts as software modules (e.g., mathematical and statistical routines, or special function modules such as data conversion routines). The issue then arises as to whether a part is reused only if it is taken 'as is' and used in another application, or if it can still be considered to be reused if it undergoes a series of transformations or modifications before it can be used elsewhere. It has also been suggested that parts consist of code templates that can be filled in by the user. A combination of approaches has proven successful in several applications, e.g., The Hartford Insurance Company (Reference 4), Raytheon Missile Systems (Reference 1).

The effectiveness of a parts catalog is heavily dependent upon the selection of attributes that will be kept for each part. If the catalog entry does not contain sufficient information for a user to determine, either manually or via an automated system, the appropriateness of a particular part, reuse of existing parts becomes virtually impossible.

It is inevitable that eventually there will be several parts in the catalog that appear similar in functionality but whose internals result in quite different execution or storage efficiency. The catalog must contain attributes that enable a user to differentiate between these parts, or there must be an automated means of determining the appropriate part for the user. Lack of an efficient means of differentiating between parts can lead to user dissatisfaction and a failure of the reusability effort.

b. The Cataloging Mechanism

The presentation of catalog information to the user can have a bearing on the success of the parts catalog. The information can, of course, be presented textually. Graphical representation of a part may be of value in clarifying the part's definition, and thus help the user to select the appropriate part. There are several different graphical representation methods that could be used; they are summarized in Figure 1.

The technology requirements for the user interface techniques must be considered when determining the feasibility of any particular system. For example, database technology and query language interfaces are well developed areas in computer science, but graphical and natural language interfaces are still in the early stages of practical development.

To gain acceptance and use, a parts catalog must provide adequate user support. It must be well documented and easy to use for both novice and experienced users. It must provide a reasonably fast response to inquiries; users become frustrated with slow, cumbersome systems. It must also provide some form of access control, although this would not necessarily meet DOD security requirements for trusted systems; research in this area is beyond the scope of this project.

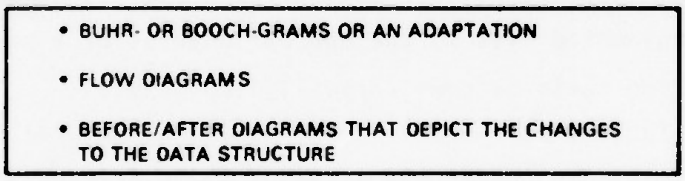


Figure 1. Graphical Representation Methods

c. Organizational Requirements

One of the most important organizational issues surrounding a parts catalog is who will mandate its development and use. Previous studies have recommended the establishment of a catalog and library of parts, but without the authority to enforce such a recommendation, reuse of software

parts has not become a reality. Reuse of existing parts requires discipline on the part of software developers; until the benefits of reuse of parts become obvious to all involved, there must be a means of enforcing that discipline.

The scope issue must be addressed in both the macroscopic and microscopic sense. The macroscopic view addresses the applicability of a parts catalog to an organization, i.e., the catalog can have an inter-company scope (e.g., the catalog may contain Ada missile software parts developed by all Air Force missile contractors) or intra-company scope (e.g., the catalog may contain Ada missile software parts developed only by McDonnell Douglas).

The microscopic view addresses the domain of the catalog (e.g., the domain could be very broad and include all Air Force Ada software development projects, or it could be narrow and include only Ada missile flight software). With respect to the microscopic view of scope, a decision must be made whether to have a single library, or to have several libraries based on the application or task area.

The organizational applicability of the catalog (i.e., is it inter-company or intra-company), affects several aspects of the catalog development and maintenance. For instance, standards become very important when parts from different developers are cataloged together, but it is easier to establish standards within a single company than across all Air Force contractors. Additionally, certain entries in the catalog should contain different levels of information depending on the catalog's scope. For example, information on the developer would vary depending upon the scope of the catalog. If the catalog is intra-company, information about the individual(s) actually involved in the development may be of use, whereas if the catalog is inter-company, merely identifying the company performing the development is probably sufficient. Additionally, if the catalog is inter-company, the question of proprietary rights may become an issue.

Procedures for maintenance must be established prior to the implementation of a parts catalog. Guidelines are needed for the addition and removal of items from the catalog. A consistent classification scheme should be developed and enforced when a part is added to the catalog. Security controls must be implemented to prohibit unauthorized access to both catalog entries and to the parts they identify. There must be a way to ensure that all required information is provided with a part when it enters the catalog system.

Although there may be a way of differentiating among parts that appear similar, an attempt should be made to limit their proliferation. Confronted by too many choices, a user may find it simpler to develop a part from scratch rather than wade through the descriptions of existing parts. This problem can be ameliorated by the imposition of procedural controls on the maintenance of the parts catalog (i.e., additions to the parts catalog should follow a standardized and carefully monitored procedure).

As mentioned earlier, quality assurance (QA) is essential to the success of reusable parts. The exact nature of the QA structure is at least partially dependent upon the scope of the catalog.

4. CATALOG DEFINITION

The purpose of a software parts catalog is to facilitate reuse of existing software parts by providing a mechanism for rapidly identifying relevant parts to a software developer. To that end, the software parts catalog must contain sufficient information to permit selection of components, but not so much information that it is cumbersome to use. This requires careful selection of attributes for inclusion in the catalog.

The investigators have developed a set of attributes to describe each catalog entry which provide the catalog user with sufficient but not overwhelming information about individual software parts. Figure 2 summarizes these attributes. Each of these attributes is discussed in greater detail in Appendix A. Figure 3 graphically depicts the flow of information into and from the parts catalog system.

MDAC-STL developed a prototype Ada parts catalog as a proof of concept. The catalog was developed using a relational database system (ORACLE™) on a VAX 11/780. A description of this catalog is contained in Appendix C.

PART ID	REVISION ID
VERSION	NAME
ABSTRACT	CATEGORY
TYPE	LEVEL
CLASS	INLINE
OPERATION	PARAMETER NAME
KEYWORDS	DATE CATALOGED
OEVELOPED BY	OEVELOPED FOR
OEVELOPMENT STATUS	VERIFICATION STATUS
CATALOG UNITS WITHED	WITHING UNITS
USAGE	LOCATION OF COOE
SECURITY CLASS (PART)	SECURITY CLASS (CATALOG ENTRY)
LINES OF CODE (SOURCE)	FIXEO OBJECT COOE SIZE
REQUIREMENTS OOCUMENTATION	OESIGN DOCUMENTATION
HARDWARE OEPENOENCIES	OTHER RESTRICTIONS
ACCURACY	TIMING CHARACTERISTICS
REMARKS	

Figure 2. Catalog Attributes

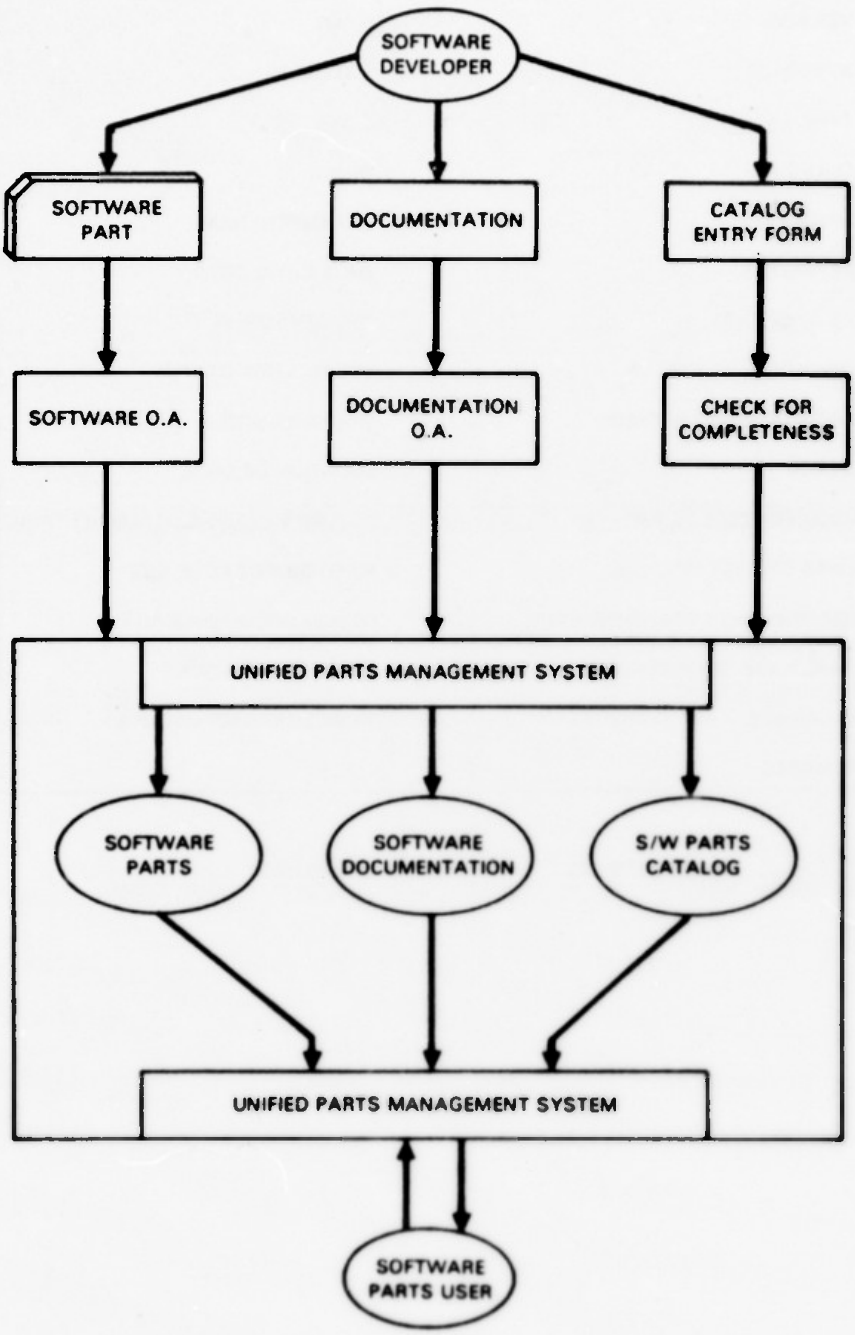


Figure 3. Information Flow through Catalog System

5. DOCUMENTATION REQUIREMENTS

In order to ensure that all necessary information is supplied when a part enters the parts catalog system, a standard form should be developed and utilized. Appendix B provides an example of such a form; this is the form used in the development of the MDAC Ada parts catalog. Some of the items on the form could be supplied automatically as a software part enters the system (e.g., Lines_of_Code, Units_Withed, Withing_Units). This form can be used for both intra-company and inter-company cataloging, although the level of detail of information provided in certain fields will vary depending upon the scope. Appendix A discusses the scope-dependent differences in attribute information.

Parts documentation is crucial to the success of any software reusability effort; information indicating the type of documentation and its availability should be provided for each part in the catalog.

Although reusable software has been discussed for a number of years, its implementation is fairly recent, therefore, the user must be provided with documentation and training material for use of the parts catalog, both from the viewpoint of a catalog user and as a developer of software that will be cataloged for future reuse.

6. ORGANIZATIONAL FACTORS

Organizational factors play a critical role in the success of any attempt to implement reuse of pre-built software parts. Although the scope of the catalog has a direct bearing on the exact nature of the organizational factors that must be addressed, the issues remain essentially the same. Figure 4 summarizes the organizational factors required to support a viable parts identification effort.

• MOTIVATION FOR REUSE	• PROCEDURAL CONTROLS
• CENTRAL REPOSITORY	• TRAINING
• STANDARDS	• USER SUPPORT

Figure 4. Organizational Factors

a. Motivation for Reuse

As stated earlier, software development that makes use of pre-built software parts requires discipline on the part of the developer. It also requires an initial investment of time and effort to establish a reusability program. Until the benefits of reuse become apparent to all involved, there must be motivation for organizations and individuals to reuse existing software in a structured way. Although in the long-run reuse of existing software parts can produce significant economic gains, some form of motivation will be required initially. The form that this motivation may take is scope-dependent (see Figure 5).

MOTIVATION BY CONTRACTOR	MOTIVATION BY CUSTOMER
• COMPANY STANDARD	• DOD MANDATE
• SUGGESTED COMPANY PRACTICE	• CONTRACT REQUIREMENT
• COMPETITIVE EDGE	• CONTRACT INCENTIVE

Figure 5. Forms of Motivation

At one extreme, motivation could be provided in the form of a DOD mandate similar to that for the use of Ada. Due to the extremely broad scope of such a mandate, this is not considered an optimal form of motivation at this time.

The Air Force could provide motivation in the form of contract incentives or contract requirements. Incentives could take the form of giving companies that have reusability programs in place greater consideration in proposal reviews. Motivation could also take the form of only considering companies that have reusability programs in place.

Although cost-plus contracts provide little incentive for the contractor to economize on development costs, some form of economic incentive may be appropriate to contractors who initiate or have in place a functioning reusability program. For example, a bonus could be tied to the amount of reuse on a project.

If individual contractors are expected to set up their own programs, the Air Force should provide guidelines in order to ensure comparability between programs, and to lay the foundation for the implementation of reuse of software on a more global scale.

Motivation for software reuse within a company can range from a corporate suggested practice to a strictly enforced company standard. A suggested practice may recommend software parts reuse as a sound software engineering practice and provide guidelines for developing reusable software. When adherence to reusability guidelines is required, a system of checks and audits must be established in order to determine compliance.

b. Central Repository

Regardless of the scope of the catalog there should be a central repository for both the catalog and the parts. This eliminates redundancy, reduces overhead, and facilitates maintenance, control, and use of the catalog and parts. Ideally users would be able to access the catalog and parts database from remote locations. The need for a central repository has been supported by a number of researchers, including DeRoze (Reference 18) who performed a study of defense software.

c. Standards

Adherence to coding and documentation standards is important to the success of the reusability concept. The development of a set of usable standards is a non-trivial task whose importance should not be

underestimated. Although the development of those standards is not within the scope of this study, we have recommended a set of attributes with which to characterize software parts.

d. Procedural Controls

Entry of parts into the system must be carefully controlled. Part of the control procedure involves ensuring that all required information is entered into the catalog in the appropriate format; this can be facilitated by the use of a catalog entry form similar to the one discussed in Paragraph 5.

Before entering the system, each part should be screened for conformance to coding and documentation standards. A determination of what these standards should be are not within the scope of this study, but they should be comprehensive and quantitatively enforceable.

Quality assurance is critical to the success of parts reuse; this was found to be a recurring theme in the literature reviewed (References 1 and 19). Poor quality parts cause two problems:

- Encounters with a few poor quality parts can destroy user confidence in all of the parts.
- Poor quality parts negate the benefits of reusability (i.e., reduced cost of development, greater reliability of the systems developed).

Verification of correctness of software parts is a complicated issue. Ideally, each part entering the system should be independently tested and certified as meeting its requirements. If the catalog is to be for all Air Force development, independent certification will require extensive resources in terms of both personnel and equipment. If the scope is intra-company, the QA procedures that are currently in place could be used as the basis for developing a verification and certification process for parts. At the very least, it should be required that the provider of the part identify the type of verification performed and who (or what organization) performed that verification.

Configuration control is another important aspect of procedural controls required for a software parts catalog. Users should not be allowed to make random additions of new parts, or indiscriminately create new versions of existing parts. There must be adequate justification for a new version of an existing part (e.g., correction of an error, major enhancement). Instantiations of meta-parts should not, in general, be included in the parts catalog because these are application-specific (i.e., tailored to a specific application) software components rather than general or domain specific parts.

As stated earlier, new parts must go through adequate quality control procedures before entering the catalog system. If possible, users should be notified well in advance of any updates to the cataloged parts; this provides the user with time to plan for the new or updated part. This procedure is followed by the Numerical Algorithms Group (see paragraph 2b).

e. Training Requirements

Training may be required in the use of the catalog and associated documentation procedures. Additional training may be required to teach personnel how to develop software that incorporates existing parts.

f. User Support

Extensive user support should be provided in addition to the training discussed in Paragraph 6e. For example, guidelines for selecting and using parts should be developed and published. Major updates and enhancements, and other information concerning the catalog should also be publicized.

SECTION III
EVALUATION OF SOFTWARE GENERATION TECHNOLOGY

1. Introduction 18
2. Review 18
3. Assessment 36
4. Conceptual Framework ... 47
5. Recommendations 54

1. INTRODUCTION

The objective of the Software Generation System (SGS) study was to determine the feasibility of an automated or semi-automated means of developing missile software that makes use of existing software parts. The approach taken by the investigators was to first survey current and past research efforts in this and related areas (see Paragraph 2). This led to an identification of issues and the development of a set of evaluation criteria that could be applied to existing and proposed systems (see Paragraph 3). Next, a conceptual framework was developed to facilitate the determination of technology requirements for an ideal software generation system (see Paragraph 4). Finally, based on an evaluation of current and state-of-the-art technology, recommendations for systems with current and mid-term feasibility were developed (see Paragraph 5).

2. REVIEW

Over the years there has been a wide range of views on the nature of software generation systems. As technological advances are made, researchers' expectations of these systems also advance (e.g., FORTRAN was originally thought of as an automatic program generator). The current desire for software generation systems is motivated by the same forces that motivated development of high order languages (HOL's), and assembly languages before them--better software faster and cheaper (References 20 and 21). Although there is still no general consensus on the exact nature of such a system, there is a consensus that their use will significantly reduce software development time and cost.

In researching the feasibility of automating the software generation process, the goals of that automation must be kept in mind. These goals are summarized in Figure 6.

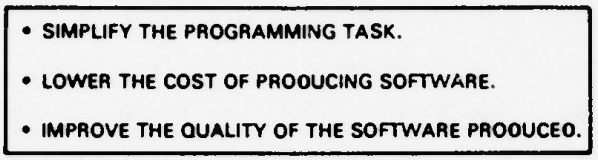


Figure 6. Goals of Automating the Software Generation Process

An automated software generation system simplifies the programming process by reducing the need for detailed programming knowledge. This is achieved by allowing the software to be specified at a much higher level of abstraction than is possible with manual programming, and/or requiring the 'programmer' to be less precise. The ultimate goal of a software generation system is to allow programming to be performed by domain engineers rather than software engineers; this would mean fewer programmers would be required resulting in a significant cost savings. Automated software generation that involves reuse of pre-built parts would realize additional cost savings by reducing the amount of software that would have to be developed.

Improved reliability is another goal of automation of the software generation process (e.g., fewer coding errors). The use of pre-built software parts would yield benefits in this area also; the parts would have been previously tested and verified, thus less time and effort would be required for testing and debugging of new software systems.

Many researchers have tried to develop universal software generation systems (i.e., systems that are applicable to all problem domains) with the result being that they are not particularly well suited to any given domain. Because of their generality, software specifications required by these systems often necessitate nearly the same level of detail as that associated with ordinary programming. Additionally, the lack of domain specific knowledge often results in significantly less efficient code than could be produced by a human coder. A number of researchers have noted that most success in the automation of software generation has come from systems with

modest goals, i.e., systems attempting to deal with limited application domains and a limited range of user proficiency (e.g., Prywes, Reference 22).

As mentioned earlier, the CAMP study is interested in software generation systems that make use of existing software parts; this requires a way to describe existing parts, store, manage, and retrieve them, and integrate them into future software development projects. In the past, it was almost as difficult to determine if an existing software part would meet a user's requirements as it was to (re-)develop the part. Generally, little or no documentation was available. The documentation that was available was poorly written making it difficult for the (potential) user to determine the functionality and appropriateness of a software part. Additionally, the quality of available components was unreliable. (Software parts cataloging and its associated problems were discussed in Section II.)

As part of the investigation into the feasibility of an automated software generation system, existing work in the technology areas that are relevant to software generation systems (i.e., automatic programming, expert system applications, formal specification systems, natural language interfaces, and text generation) was surveyed. The literature surveyed contained a great deal of ambiguity in the usage of terms such as automatic programming, program generator, and software generator. Some researchers use the terms automatic programming and program generator interchangeably, while others distinguish between the two. Program generators are often considered more mechanical in nature, not involving the expert system reasoning capabilities often associated with automatic programming. In our view, automatic programming and automatic software generation are equivalent, although the term automatic programming appears to be used more frequently in recent research; we will use the phrase software generation.

The remainder of this paragraph contains a summary of our findings. We will first look at automatic software generation systems in the large, and then provide a detailed look at three fairly recent systems. Next, we will look at the major architectural components of such a system (i.e., the specification technique, the method of operation, and output generation). An alternative to software generation is the use of expert system assistance in the development process. This is of particular interest to us, as one of the goals of the CAMP study was to investigate the feasibility of an automated or semi-automated means of developing missile flight software. Expert system

assistance has often been incorporated into proposals for fully automated systems. Two representative systems are presented in Paragraph 2e. Figure 7 summarizes our presentation.

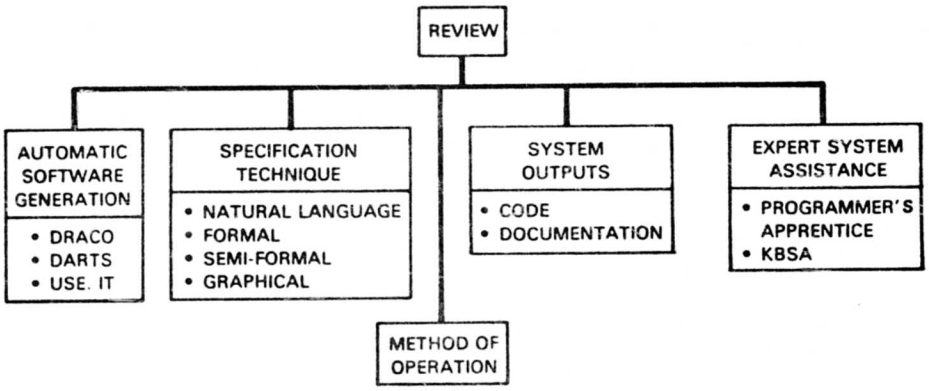


Figure 7. Summary of Review

a. Automatic Software Generation

An Automatic Software Generation System is a software system that automatically generates software when given a requirements specification in a very high order language (VHOL). VHOL's allow specifications to be provided at a higher level of abstraction than HOL's, just as HOL's provided a higher level of abstraction than assembler languages. The form of the VHOL can range from very formal specification languages to natural language; specification techniques are discussed in Paragraph 2b.

In addition to the specification technique, software generation systems can be characterized by their method of operation, their target language, and the problem domain (Reference 23). The method of operation is the technique employed to change the initial specification into a software part. There are a number of operational methods that a software generation system can incorporate; they are discussed in Paragraph 2c. The target language is the language in which the software will be generated. In the case of the CAMP study, the target language of interest is Ada. The problem

domain refers to the application area for which software will be generated (e.g., missile flight software). It can be seen that a wide range of technology areas are covered by software generation systems.

The scope of software generation systems can vary significantly. Some systems are designed to generate single program units while others are intended to generate entire software systems. Still others are designed to generate only specific parts of program units (e.g., data structures).

Most software generation system implementations are in the research phase, or at the stage that only relatively small programs can be developed. According to Neighbors (Reference 17), specification and requirements analysis present the major impediments to the development of complete software generation systems.

Software generation systems do not necessarily involve reusable software parts, and most systems developed to date do not. A few recently developed systems incorporate reusability of some form; they include DRACO, DARTS™, and USE.IT; these systems are discussed in the following paragraphs.

(1) DRACO

DRACO (Reference 17) is an interactive software generation system developed by Jim Neighbors at the University of California at Irvine. The system allows solutions to classes of problems to be developed. Once a solution to a particular class of problems has been developed, individual systems can be developed by personnel who are not necessarily software engineers.

Development begins with a determination of the existence of an appropriate modeling domain for the problem area (e.g., missile flight software). A modeling domain is essentially a model of the type of system the user wishes to develop. If a modeling domain does not exist, a domain expert must perform a domain analysis. The domain analysis takes a high-level look at the objects and operations that are used (required) in the problem domain. Domain analysis differs from systems analysis in that domain analysis examines the objects and operations that are required by systems of a particular class rather than looking at the requirements for one particular system.

If it is not likely that a number of similar systems will be built, domain analysis should not continue; instead a custom system should be constructed. Domain analysis is an expensive, time-consuming task that requires extensive knowledge of the problem domain. For this reason, DRACO is not well suited for the development of one-of-a-kind systems.

Domain analysis results in the development of a domain model and a domain language. The domain language encapsulates the design aspect of the application, and is intended to allow users to communicate in a language with which they are familiar rather than requiring them to learn an ordinary high order language for programming. Each object and operation in the domain language is represented by a software component (i.e., a part). Most domain languages are quite different from ordinary programming languages (e.g., a domain language may take the form of a table). It is through use of the domain language that reuse of design takes place.

The user specifies the problem in a domain language program; the domain language program then undergoes a series of refinements that are guided by the user or by a predefined strategy, to produce an executable program. The refinement history is saved along with the executable code that is produced.

We have identified several aspects of the DRACO system which make its widespread usability in the missile flight software domain questionable.

- The DRACO system is still in the early stages of development, and considerably more work is required to make it a production-quality system.
- The specification technique of the DRACO system is designed for ease of use, but the user still must learn a formal specification language and technique in order to use the system.
- Considerable detail is required on the part of the user when specifying requirements.
- Efficiency is another concern with the DRACO system. The code produced is claimed to be very efficient, but DRACO is a universal software generation system, and, as we have previously pointed out, the efficiency of the code produced by these types of systems is frequently inadequate for the types of applications with which we are concerned (i.e., real-time embedded systems).

(2) DARTS

DARTS™ (Development Arts for Real-Time Systems) (References 24, 25, and 26), developed by General Dynamics, also allows solutions to be developed for classes of problems. The goal of the research leading to the development of the DARTS technology was that once end users had a working system in place, they would be able to generate similar systems without programmer assistance. The user would enter the system specifications in some domain language, and through a series of transformations, the specifications would get translated to source code.

One premise upon which the technology was developed is that creativity is only really required in the development of the first implementation of a particular class of applications; significantly less creativity is required for the development of each successive system of a given class. Thus, after the initial system is developed, it should be possible to generate additional systems of the same class automatically using the original system as a prototype.

Efficiency was an important consideration in the development of the DARTS technology, just as it is in the CAMP study. The developers of the DARTS technology wanted the automatically generated systems to be at least as efficient as custom systems. As with DRACO, DARTS is a universal software generation system, and it is not clear that the code it can produce is efficient enough for the missile flight software domain.

When a problem is initially identified as being a candidate for solution by the DARTS technology, an analyst must perform a domain analysis and design a general software solution to the problem; a working system may already exist in the problem class. Once the initial software system is in existence, it must be genericized, or in the DARTS terminology, made into an archetype. During this time it is also necessary to develop a domain language and translator. AXE, the language component of the DARTS system, is extensible, and should be extended to incorporate the domain language. The

domain language is intended to facilitate user interaction with the system, but it still requires the learning of another specification language. The requisite knowledge bases for the application must also be developed. The end result of the domain analysis phase is that an environment is created that allows users to completely specify software systems without programmer assistance; code is generated automatically once the specifications are determined to be complete.

Existing software is genericized by embedding AXE statements in the source code; these statements are used to direct software generation by referencing the system knowledge bases. Actual software or code generation takes place through a series of transformations. Each class of system (or application area) essentially has its own software generator (i.e., its own archetype). AXE statements can also be embedded in documentation to allow the automatic generation of new documentation along with a new system.

DARTS provides a way to generate a family of modules. The domain analysis and language development are time consuming and relatively expensive tasks that require extensive knowledge of the domain. Prior to developing a general solution for an application area, an assessment must be made as to the likelihood that many similar systems will be needed or if the required system will probably be one-of-a-kind. Because of the costs involved, this technology should not be applied unless there is a foreseeable need for several systems of the same type.

DARTS is currently being marketed by General Dynamics, but at the time of our study, we were unable to obtain conclusive evidence from General Dynamics concerning its appropriateness to the missile flight software domain.

(3) USE.IT

USE.IT (References 27 and 28), a commercial system developed by Higher Order Software, Inc. (HOS), allows a user to specify unit requirements via a graphic specification technique. The specifications take the form of a hierarchical tree structure which is referred to as a control map. The leaf nodes of the control map are system primitives or external routines developed by a programmer. HOS provides the system with only very low-level primitives; it is left to the user (or installation) to develop higher level primitives. It is only through the development of additional primitives and

external routines that programming with USE.IT is raised to a higher level of abstraction than ordinary programming.

The requirements specifications are analyzed, and if found to be incomplete or inconsistent, the specification-analysis phase is reiterated. Once the specification is finalized, the control map can be used to automatically generate code, or it can be used as a specification for manual coding. English-language documentation can be produced as a by-product.

Reusability is manifested through the reuse of primitives. This is really reuse of both design and code (if the code for the primitives is also reused via automated or manual means).

We have identified a number of problems associated with the use of this system for the development of Ada missile flight software:

- -- USE.IT does not generate Ada code and there is no definite date in the future for the generation of Ada.
- -- The user must be aware of which primitives exist and be able to choose which would best suit his needs (this may require a primitives administrator position which would be similar to a database administrator).
- -- The primitives need to be developed at a sufficiently high level otherwise specification must be at as low a level as required for manual coding.

b. Specification Techniques

The specification technique employed by a software generation system has a significant impact on the system's usability and even its feasibility. The techniques range from natural language (NL) to code-like program design languages. When considering a specification technique, the intended user must be taken into consideration; some techniques require a substantial investment in time and effort to achieve effective use. The specification techniques covered are summarized in Figure 8.

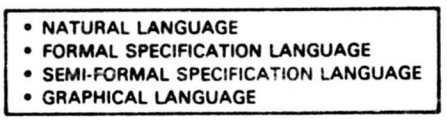


Figure 8. Specification Techniques

(1) Natural Language

For years it has been the goal of researchers to develop Natural Language (NL) man-machine interfaces. Although such interfaces could be used for a wide variety of man-machine interactions, we are particularly interested in natural language interfaces to databases and software generation systems. Natural language interfaces to software generation systems could alleviate many software development problems by allowing the user to communicate his requirements directly to the system rather than requiring him to work through a software engineer who must interpret and analyze his requirements.

Due to the wide range of possible inputs and their interpretations, the development of an NL interface for software generation systems is a more complicated problem than providing a natural language interface to a database. Unrestricted NL interfaces have not yet been realized, but some progress has been made, particularly within limited domains and with a restricted set of users; this finding is supported by several researchers including Biermann (Reference 20) and Hendrix and Sacerdoti (Reference 29).

Domain dependent specification languages are a special type of natural specification. These are specification languages that incorporate the jargon of the application domain; they are intended to facilitate user-system interaction by providing a simpler form of communication than a high order programming language. They are part of a trend towards natural language interfaces.

Hendrix and Sacerdoti (Reference 29) distinguish between natural language systems that utilize an explicit world model (i.e., a knowledge base containing information on the world as the system needs to see it) and those that do not. Systems that do not require an explicit world model are simpler to implement and are generally used for applications such as database interfaces. Systems that do use an explicit world model have been developed in the laboratory, but have not yet progressed into readily available production-quality systems.

One natural language system, SAFE (Skills Acquisition from Experts), developed by Robert Balzer, is concerned primarily with the transformation of a limited English specification into a formal specification. SAFE is part of a larger project under development by the Information Sciences Institute at USC, to develop a comprehensive software generation system.

Greater success has been realized in the implementation of natural language database interfaces (Reference 29); several projects have implemented NL interfaces of various types. LADDER (Language Access to Distributed Data with Error Recovery), developed at SRI (Reference 23), is an NL interface to a naval database; it makes use of the LIFER NL system (Reference 30). LIFER is a utility system that facilitates the development of natural language interfaces. LUNAR, a system developed at Bolt, Barenek, and Newman (Reference 30) to aid in geologic analysis of material brought back on the Apollo-11 space mission, also makes use of a natural language database interface. Natural language interfaces have been successful in these cases for two reasons: the goals have been relatively modest, and the application domain has been limited.

A natural language interface can also be used to assist a user in the development of database queries. RENDEZVOUS (Codd, 1978) (Reference 23) is one such system. It carries on a clarifying dialog via a series of menus that provide the user with options for further input and output. At the conclusion of the dialog, the system produces a natural language summary of its interpretation of the user's request.

(2) Formal Specification Languages

Formal Specification Language systems make use of very high order languages to specify requirements. The complexity of these systems varies greatly; they can be used to specify everything from entire systems to individual program units. The nature of the specification language has a significant impact on the system in which it is incorporated.

Specification languages (SL) can be classified as procedural or non-procedural. Procedural languages describe not only what to do, but how to do it; most ordinary programming languages fall into this category. Non-procedural languages merely describe what needs to be done (e.g., database query languages); they generally require less skill to use than procedural languages, and are at a higher level of abstraction. Specification languages can be further classified as domain independent or domain specific. Some systems incorporate extensible languages that allow the development of specification languages tailored to a particular domain area (e.g., DARTS).

Stoegerer (Reference 31) has partitioned specification languages into three classes: requirements specification languages, (system) design specification languages, and program design languages (the CAMP investigators have classified program design languages as a Semi-Formal Specification Technique). In reality, the distinction between the classes tends to be somewhat hazy. Stoegerer and others have suggested integrating a cohesive set of specification languages into a software development environment.

Two examples of requirements specification languages are RSL, the requirements specification language associated with the Software Requirements Engineering Methodology (SREM) developed by TRW for the Army Ballistic Missile Advanced Technology Center, and PSL (Problem Statement Language), the requirements specification language portion of the tool PSL/PSA (Problem Statement Analyzer). Both RSL and PSA are tailorable, structured English specification languages (i.e., the languages can be extended or tailored to fit the needs of a particular project) but both suffer from a relative lack of use. This emphasizes the fact that formal specification languages are typically difficult to work with. Training in either technique can take 1-2 months (Reference 32), and training must be

provided not only for those who will be writing requirements specifications, but also for those who must read them.

Both the DRACO and DARTS systems provide extensible specification languages that can be tailored to form domain specific specification languages. Many other systems utilize formal specification languages for user input; two of them are described briefly here.

MODEL (Module Description Language), developed by Noah Prywes of the University of Pennsylvania (References 22 and 33), is part of an experimental software generation system. MODEL is non-procedural and similar in structure to PL/1. The user must supply a fairly detailed specification of the input and output data. Assertions, or equations, which describe relationships between data objects, are also supplied by the user. The MODEL program undergoes analysis for inconsistencies, ambiguities, and incompleteness. After checking and correction, either PL/1 or COBOL can be generated. Although the use of a non procedural language does ease the programming burden, the user is still required to learn a PL/1-type language, and provide detailed specification of inputs and outputs.

Protran, the user interface to the IMSL library, is an extension of FORTRAN, and is not a part of a software generation system. Programs written in Protran are much smaller than equivalent programs written in FORTRAN, but the specifications are not any less complete than those required for a FORTRAN program.

(3) Semi-Formal Specification Languages

Program design languages (PDL's) and specification by example are two forms of semi-formal specification techniques. Program design languages can take many forms; the ones of particular interest to us are those that are Ada based. McDonnell Douglas Astronautics Co. has developed one such language, referred to as ADL (Ada Design Language) (Reference 34). It consists of a subset of Ada and is intended to be used for the design of software systems. Numerous other versions of Ada-based PDL's have been developed. There is currently an effort under way by the IEEE (Reference 35) to establish guidelines for their development.

One form of specification by example consists of the user providing the system with input-output pairs; the system then generates the code that would result in the given output when supplied with the specified input. A user must carefully construct examples that completely specify the requirements. The development of a comprehensive example for more than a trivial problem is not a simple task, but for simple problems, it has been found that users can converge on the correct solution fairly quickly simply by providing successive examples; this has been noted by several researchers (References 23 and 36).

Specification by example has been incorporated into PSI, a software generation system developed by Cordell Green at the University of Southern California.

(4) Graphical Specification Languages

Studies have indicated that both clarity and speed of information transfer are greater with graphic-based languages than with other types of languages (e.g., formal specification languages, natural languages (Reference 31)). The use of graphical languages for both input of specifications and other man-machine interactions (e.g., requests for further information from the user, summaries of specifications) has been proposed. Graphical representations allow information to be presented concisely.

A Graphical Specification Language requires both an appropriate set of symbols and a method for processing it. The development of automated graphical specification techniques is still in the early stages.

MIT had a project underway to develop such a technique (Reference 37). A preliminary step in the development process was the development of an appropriate set of symbols to represent various programming constructs and concepts. HOS's USE.IT system makes use of a graphical specification technique, although it is not at a very high level.

Both Booch (Reference 38) and Buhr (Reference 39) have proposed manual graphical representation schemas for Ada software parts.

c. Methods of Operation

A software generation system takes some form of requirements specification as input, and generates some form of software part as output. The technique used to change a requirements specification into code is referred to as the method of operation. There are, of course, many ways to do this, but, there do not appear to be any clean cut lines that clearly delineate the methods and thus facilitate classification; this is often the case with technologies that are in the early stages of development. This is not to imply that classification schemes have not been proposed. Some categories that have been suggested are deductive techniques, transformation techniques, expert system techniques, and custom tailoring.

Custom tailoring is often thought of as using parameterized software to generate unique configurations from a standard software system. This process has been used in telecommunications systems, and is also the method used to transform generic Ada parts into concrete usable instances; it is a way to generate families of concrete software systems (or programs) from an abstract system (or program).

An expert system can be used in conjunction with any method of operation, thus, a strict classification as expert system technique is not really meaningful.

Deductive or theorem proving techniques incorporate transformations that are usually in the form of predicate calculus statements. These techniques start with a theorem to be proven, and attempt to find a series of transformations which lead to that conclusion. A program is produced as a by-product of the proof.

The problem that we see with an attempt to classify the methods of operation at this stage of development is that almost all methods of operation can be forced into the category of transformation systems (i.e., they all transform a specification into a software part). The transformations can take a number of forms: they can be in the form of predicate calculus statements, they can be in the form of rules, or they can be simple substitutions.

A mechanism for selection and application of the transformations is required. The amount of user assistance required to guide the application of the transformations varies considerably between systems. Some systems

require no user input other than the provision of the initial specification, while others require a significant amount of human guidance (e.g., Kestrel's proposed Knowledge Based Software Assistant). An expert system may be used to aid in selection of the transformations, or the transformations may be applied in an arbitrary manner or with the aid of heuristics. The steps in the transformation process are often saved so that the transformation can be replayed later if the need arises to re-implement the software.

The range of problems that can be solved by any given method of operation varies considerably depending upon the particular implementation. As with specification languages, the trend in methods of operation has been towards greater domain specificity (Reference 40).

Several software generation systems (DRACO, DARTS, USE.IT) have previously been discussed, but we will briefly summarize how they produce programs. In the DRACO system, a program specified in a domain language undergoes a series of refinements (or transformations) that follow a pre-defined strategy or are guided by the user. In the DARTS system, the archetype system has AXE language statements embedded in them that reference various knowledge bases. The user's program supplies application specific information that is used in conjunction with the information from the knowledge bases to guide the transformation of a system from a model solution into a specific instantiation. In HOS's USE.IT system, code modules are substituted for primitives in the control maps; module interconnections on the control map require the generation of code.

PSI, a software generation system developed by Cordell Green at Stanford University in the 1970's, uses a number of cooperating experts (e.g., a domain expert, coding expert, efficiency expert) to transform the specification (which may be in the form of a series of examples) into code.

The DEDALUS system developed by Manna and Waldinger at SRI (References 23 and 36), has been referred to as a deductive system. It uses a modified form of predicate calculus (i.e., more English language text is allowed) for the specification, and generates programs in a language similar to LISP. The transformation rules contain knowledge about both general programming principles and the specific implementation language. Successive application of the rules leads to the transformation of the original specification into the final program.

d. Text Generation

In addition to producing code, it is desirable for a software generation system to also produce documentation. Text generation poses basically the opposite problem of natural language specification. Text generation requires the transformation of an internal representation of information (i.e., program specifications) into English text. A few systems incorporate some rudimentary form of text generation (e.g., HOS's USE.IT generates documentation that the developer claims meets military standards, but it appears to be at a fairly low level). The DARTS system is able to generate documentation from genericized documentation provided with the archetyped system. As was mentioned earlier, the Rendezvous system generates natural language summaries of user specifications. Automated text generation is not highly developed.

e. Expert System Assistance

An Expert System is a software system designed to exhibit human like reasoning behavior (i.e., such systems are able to form inferences based on factual knowledge, data, and rules of thumb). Expert systems have been proposed that would assist in the programming task rather than perform it automatically.

One such system is the Programmer's Apprentice (References 23 and 41), proposed in 1976 by researchers at MIT. The system is intended to provide assistance in the areas of documentation, verification, debugging, and modification management. The system incorporates general programming knowledge; this knowledge is stored in the form of plans. The programmer can either provide plans for the solution of a problem or provide code. The Apprentice uses the plans to form an understanding of the problem; it tries to determine if the code implementation corresponds to a valid plan, and if there is no correspondence, the programmer is notified. The Programmer's Apprentice can also provide assistance in determining the ramifications of modifications. A combination of plans and user supplied information are used to generate documentation. Research and development work on prototype systems has proceeded over the years.

Another knowledge-based programming assistant that has been proposed is the Knowledge-Based Software Assistant (KBSA). In a study performed by the Kestrel Institute for Rome Air Development Center (Reference 42), researchers proposed the development of a system that would provide assistance in all areas of a software development project, from requirements analysis to project management. It is proposed that the system be developed incrementally over the next 10 to 15 years, with work proceeding on a number of areas concurrently. Formalization of development practices is a key factor in automating the program development process.

The proposed system would interact with different types of users at the appropriate level (e.g., project managers would not be burdened with programming details, but a programmer would be able to get the information he needs from the system). An interesting aspect of this study is that the researchers chose not to include as goals of the KBSA two important goals of other proposed systems: automatic program generation and natural language interfaces. Natural language interfaces were omitted because it was felt that such interfaces would require the same underlying formalisms proposed for development as part of the KBSA, but that the amount of research required to effectively implement a NL interface is so vast that to do so would detract from the development of the remainder of the KBSA. Automatic programming was not included as a goal because it was felt that the user could be allowed to interact with the system at a higher level of abstraction if he was also required to assist in the code generation process (i.e., there is a technology gap between what is fully automatable and what is semi-automatable). For example, the user could be provided with the capability to partially specify software requirements and have the system assist with their completion.

3. ASSESSMENT

When considering a particular software generation system, it should be examined carefully in light of relevant issues and evaluation criteria. Two levels of issues and evaluation criteria were identified during the CAMP feasibility study. The top level relates to the system as a whole (i.e., reusability issues, issues related to Ada and the problem domain, technology issues, system maintenance and initialization issues, and issues relating to physical attributes of the system), while the second level looks at specific facilities and parts of the system (i.e., the specification technique and the specification itself, user support, and system outputs). Figure 9 summarizes these issues and evaluation criteria; each category is discussed in detail in the following paragraphs.

a. Reusability Issues

The CAMP study is concerned specifically with the reuse of existing software, therefore, any system examined must be evaluated in light of its ability to incorporate reusable software parts. Few existing software generation systems have such facilities. Figures 10 and 11 depict two views of an SGS--one system does not involve reuse of existing software and the other does.

The level at which reuse will take place is important to the structure of a software generation system. Reuse can be at the analysis, design, or code level. HOS's USE.IT system implements reuse at the code level through the reuse of primitives (i.e., pre-built software parts), but fails to provide an automated parts management system for these primitives. The DARTS system essentially reuses previously developed software systems (i.e., the archetype is used to generate new software systems). In the DRACO system, the emphasis is on the reuse of design and analysis (through the reuse of domain analysis and the domain language). Each component and operation in the domain language is a software component, and thus, reuse at the code level also takes place.

REUSABILITY

- IS THE REUSE OF PRE-BUILT PARTS SUPPORTED?
- AT WHAT LEVEL IS REUSE SUPPORTED (e.g., REQUIREMENTS, DESIGN, CODE) AND MAINTENANCE PERFORMED?
- IS REUSE OF PRE-BUILT PARTS ENFORCED?

ADA AND THE PROBLEM DOMAIN

- IS ADA SUPPORTED? (i.e., CAN ADA PARTS BE GENERATED?)
- IS THE PROBLEM DOMAIN (e.g., MISSILE FLIGHT SOFTWARE) ADDRESSED?
- IS THE CODE PRODUCED EFFICIENT ENOUGH FOR THE PROBLEM DOMAIN?

TECHNOLOGY

- IS THE TECHNOLOGY OF SUFFICIENT MATURITY FOR INCORPORATION INTO AN AUTOMATED SOFTWARE GENERATION SYSTEM?
- WHAT DEGREE OF AUTOMATION IS PROVIDED?

SYSTEM INITIALIZATION MAINTENANCE

- WHAT IS REQUIRED WHEN THE SYSTEM 'COMES IN THE DOOR'? (i.e., IS DOMAIN ANALYSIS REQUIRED? MUST A DOMAIN-SPECIFIC LANGUAGE BE DEVELOPED? DOES EXISTING CODE NEED TO BE RESTRUCTURED? DO SOFTWARE PARTS NEED TO BE PRE-BUILT FOR LATER USE?)
- IS THE SYSTEM EASY TO MAINTAIN?
- CAN THE SYSTEM EVOLVE AS TECHNOLOGICAL ADVANCES ARE MADE?

PHYSICAL ATTRIBUTES OF THE SYSTEM

- IS THE SYSTEM A REASONABLE SIZE? (i.e., WHAT ARE ITS BASIC STORAGE REQUIREMENTS?)
- IS THE SYSTEM EFFICIENT IN TERMS OF BOTH STORAGE AND RESPONSE TIME?

Figure 9. Issues/Criteria of a SGS

SPECIFICATION TECHNIOUE AND THE SPECIFICATION

- WHAT TYPE OF SPECIFICATION TECHNIQUE IS AVAILABLE? (e.g., FORMAL SPECIFICATION LANGUAGE? NATURAL LANGUAGE? PROCEDURAL OR NON-PROCEDURAL?)
- IS THE SPECIFICATION TECHNIOUE APPROPRIATE TO THE USER? ARE MULTIPLE SPECIFICATION TECHNIOUES PROVIDED SO THAT THE MOST APPROPRIATE ONE CAN BE USED?
- WHAT LEVEL OF EXPERTISE/TRAINING IS REOUIRED TO EFFECTIVELY INTERFACE WITH THE SYSTEM?
- IS THE INTERFACE TECHNIOUE APPROPRIATE TO THE PROBLEM DOMAIN?
- CAN THE SPECIFICATION BE AUTOMATICALLY TRANSFORMED TO A FORM THAT IS COMPREHENSIBLE TO ALL PARTIES WHO NEED TO KNOW?
- CAN THE SPECIFICATION BE PUT IN A FORM THAT IS ANALYZABLE (e.g., FOR COMPLETENESS, CONSISTENCY, CLARITY)?
- IS THE SPECIFICATION MAINTAINABLE (IF THE SPECIFICATION IS TO FUNCTION AS A FORM OF DOCUMENTATION AND CONTROL, IT MUST BE MAINTAINED IN A CURRENT STATE THROUGHOUT THE SOFTWARE LIFE CYCLE)?

USER SUPPORT

- IS THE USER ASSISTED WITH SPECIFICATIONS (i.e., IS PARTIAL SPECIFICATION SUPPORTED?)?
- DOES THE SYSTEM SUPPORT AN INCREMENTAL OR ITERATIVE APPROACH TO DEVELOPMENT?
- ARE THE SPECIFICATIONS CHECKED FOR COMPLETENESS, CONSISTENCY, CLARITY?
- CAN THE USER INTERFACE DIRECTLY WITH THE VARIOUS COMPONENTS OF THE SYSTEM (e.g., CAN HE DIRECTLY OUERY THE PARTS CATALOG?)?

SYSTEM OUTPUTS

- IS OPTIMIZED CODE PRODUCED?
- IS THE CODE VERIFIABLY CORRECT?
- ARE FACILITIES PROVIDED TO VERIFY CORRECTNESS OF RESULTING MODULES (e.g., AUTOMATIC GENERATION OF TEST PROCEDURE, CORRECTNESS PROOFS)
- ARE SUPPORTING DOCUMENTS (e.g., ADL, SYSTEM DOCUMENTATION) PRODUCED?

Figure 9. Issues/Criteria of a SGS (Concluded)

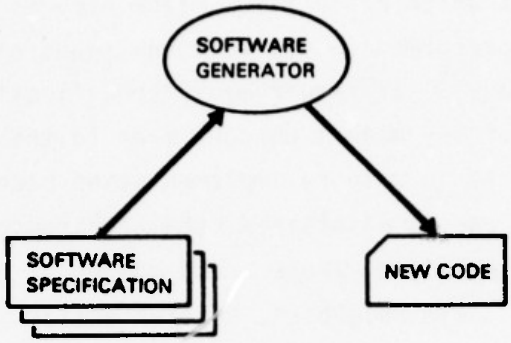


Figure 10. Software Generation without Parts Reuse

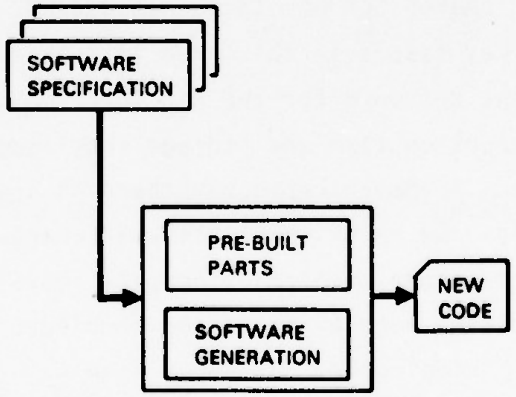


Figure 11. Software Generation with Parts Reuse

The level at which reuse takes place affects the level at which maintenance will be performed (e.g., will the requirements specification or the code be maintained?). If requirements specifications are maintained, a record must be kept of any manual changes made to the part (i.e., deviations from the standard part) in case re-implementation becomes necessary at a later time. If the code is maintained, the requirements must be changed as changes are made to the code. There are a number of advocates of maintenance of requirements (e.g., Jim Neighbors, Reference 17).

Enforcement of, and motivation for, reuse is critical. Motivation may take many forms. In addition to organizational motivation for reuse, the software generation system itself may incorporate a mechanism to prevent an engineer from building a part that already exists (i.e., a redundant code detector).

b. Ada and the Problem Domain

The CAMP study requires Ada as the implementation language for both the pre-built parts and the generated parts; the effect of this on the feasibility of an automated SGS has to be considered.

One of the key issues in this area is that any system used to develop missile flight software for the Air Force must produce efficient code (both in terms of execution time and storage requirements). Efficiency is crucial in this area. Although currently there is some degree of efficiency lost just by using Ada, we think that this will change in the near future. As more Ada compilers become available, compiler developers will strive to improve their competitive edge by producing compilers that generate increasingly more efficient code. This was seen to be the case with FORTRAN in its early stages of development. Initially there were objections to its use because it was claimed to be inefficient in comparison to the language with which most programmers were familiar (i.e., assembler), but over time, the efficiency of the code produced by FORTRAN compilers was increased to an acceptable level. We expect this to be the case with Ada compilers also.

Mandating Ada as a common language to be used for all DOD software development does have the advantage of providing an incentive to both improve Ada and develop optimizing compilers that will eliminate the inefficiencies currently found in compiled Ada code. Ada itself incorporates certain

features that lessen the effects of constructing software systems from pre-built parts (i.e., pragma IN_LINE).

The problem domain (i.e., missile flight software) has a bearing on the structure and acceptability of any given software generation system that might be considered. One certain effect is that any software generated for this application area must be highly reliable. Although no known software generators exist for the domain of interest in the CAMP study, some systems claim to be tailorable to any domain (e.g., DARTS, DRACO). It is not clear that the efficiency of the code produced by these systems is efficient enough for the problem domain under consideration; realistic demonstrations are required to prove their acceptability.

c. Technology Issues

The maturity of the technology required for any given part of a software generation system is of prime importance in determining the feasibility of the system as a whole. The stage of development of the technology should be determined (i.e., is it in the production stage, laboratory use, or research phase?).

The degree of automation provided by a software generation system is an important point to consider. Generally, there are trade-offs between the degree of automation provided and other technologically advanced features incorporated into the system (e.g., the Knowledge Based Software Assistant described in Paragraph 2e, trades off higher levels of abstraction in the specification technique against a lesser degree of automation in the software generation process).

d. System Initialization and Maintenance

Initialization (i.e., what is required to make the system operational for the end user) and maintenance of a software generation system are not strictly related to the feasibility of such a system, but they have important implications for its actual use. As we saw in the survey, some systems require an extensive amount of work before the system is operational for a particular application area. For example, DRACO requires domain analysis and the development of a domain language. DARTS (General Dynamics) requires that an archetype system be developed or that an existing system be

archetyped; the specification language also has to be extended for each domain. If a system requires this type of work, it must be determined who will perform it (e.g., will the work be performed by the Air Force with each contractor being required to install identical systems, will there be a central facility which can be accessed by all contractors as needed, or will each contractor be required to perform the work on their own). This is really an issue of scope of the system; many other issues will arise from any decision made in this area but an examination of them is not within the purview of the current study.

Ease of maintenance of the software generation system is important to its continued use. Because it is clear that technological advances will be made over time, it is desirable to be able to extend the capabilities of a software generation system as it becomes feasible to do so.

e. Physical Attributes

Physical attributes of the system also impact its feasibility. Its size and efficiency affect both where it can be used and by whom.

f. Specification Techniques and the Specification

The form of the specification (i.e., natural language, formal specification language, semi-formal specification language, graphical specification, or some combination) is important not only to the usability of the system, but also to its feasibility. The specification technique should be appropriate to the user of the system and to the problem domain. A minimum of training should be required in order to interface with the software generation system.

The specification should be in a form (or be readily convertible to a form) that is comprehensible to both the developer and the customer. It should also be in a form that is analyzable for completeness, consistency, and clarity. Finally, the specification should be maintainable throughout the software lifecycle.

Figure 12 presents a summary of how the specification techniques that were presented earlier stack up in light of the issues and criteria discussed here.

Natural language interfaces are easy to use and have the advantage that no new language is required; generally the specifications can be incomplete with the system prompting for more information as needed (this is the partial information issue).

The major drawback of natural language specification is that the technology required to support such an interface technique is not as mature as that needed to support formal specification languages. Another drawback of NL specifications is that they are not as concise as specifications in some other forms (e.g., formal specification languages) and may become voluminous for large systems.

A natural language interface makes the system easier to use, but does not negate the need for any of the underlying formalisms required by specification systems (i.e., the natural language specification will require translation into some type of formal specification in order for the system to be able to analyze it and generate code); this was the point made by researchers at Kestrel Institute who developed the plan for the Knowledge Based Software Assistant (Reference 42).

	UNRESTRICTED NL	SUBSET NL	FORMAL SL	SEMI-FORMAL SL (PDL)	GRAPHIC SL
APPROPRIATE TO USER_.	OE/SE	DE/SE	SE	SE	DE/SE
LEVEL OF TRAINING NEEOEO	L	M	H	M	M
APPROPRIATE TO PROBLEM OOMAIN	M[1]	H[2]	H[3]	M	H
COMPREHENSIBILITY OF SPECIFICATION	H[4]	H	L	M	H
ANALYZABILITY OF SPECIFICATION	L	M	H	H[5]	H
MAINTAINABILITY OF SPECIFICATION	H[6]	H	L	L	H
TECHNOLOGICAL FEASIBILITY	L	M	H	H	M

LEGEND
- OE – DOMAIN ENGINEER
- SE – SOFTWARE ENGINEER
- L – LOW
- M – MOOERATE
- H – HIGH

NOTES
1) SPECIFICATION OF SYSTEM REOUIREMENTS MAY BECOME VOLUMINOUS ANO WOROY WITH UNRESTRICTEO NL
2) RESTRICTING THE SPECIFICATION TO A NL SUBSET MORE NARROWLY FOCUSES THE STATEMENT OF THE SPECIFICATION SO THAT IT DOES NOT BECOME RAMBLING PROSE
3) FACILITATES A PRECISE STATEMENT OF REOUIREMENTS BY A KNOWLEDGEABLE SOFTWARE ENGINEER
4) COMPREHENSIBILITY MAY OECLINE AS SPECIFICATIONS BECOME VOLUMINOUS
5) IF COMPILABLE
6) TEXTUAL NATURAL LANGUAGE SPECIFICATIONS CAN BE MAINTAINEO ON A WORO PROCESSOR

Figure 12. Summary of Specification Techniques

It is probably not feasible at this time to expect that an entire set of missile software specifications can be entered via a natural language interface, although it may be possible for some of the interaction to be carried out in NL. For example, after analyzing the specifications for completeness, etc., the system could interact with the user in some limited natural language in order to obtain clarifying information. To date, most success with natural language interfaces has been with systems that have a limited domain of discourse (Reference 20).

The use of formal specification languages (i.e., VHOL's) avoids the technological problems associated with natural language interfaces, and generally avoids the need to deal with partial knowledge (specifications are generally required to be complete). Although these are advantages for the implementor of the software generation system (the technology required for these types of systems is, for the most part, more mature than that for natural language systems), they are generally viewed as disadvantages for the user of the system.

The use of formal specification languages necessitates the learning of yet another language in order to specify component requirements (e.g., even "state-of-the-art" systems such as DARTS and DRACO require the use of a formal specification language). Because of the large number of people who must be able to understand the specification, this may not be feasible (Reference 43), e.g., Stoegerer (Reference 31) states that

> " Specifications written in formal notations are largely incomprehensible to the vast majority of persons who contract for the design and development of software systems."

This idea is supported by the general lack of use of formal specification languages such as RSL and PSL (Reference 32).

Formal specification languages facilitate a precise statement of requirements; this can be both an advantage and a disadvantage. On the one hand, forcing precise requirements from the user helps ensure that the problem is well thought out in advance. It is also a step in the direction of developing verifiably correct specifications. The disadvantage of this precision is that the development of precise specifications requires a more educated and sophisticated user.

Because of the level of detail required when using most universal specification languages (i.e., a single specification language for all application areas), the benefits of programming this way as opposed to programming in an ordinary HOL may not be significant enough to warrant a change to a formal specification language. Special-purpose systems (i.e., those directed to a particular application area) may be somewhat easier to use effectively, but they still require an investment of time and effort for additional training.

General purpose VHOL's typically result in less efficient code than that produced by HOL's that are human-coded. The reason for this is that, unlike a human coder, the VHOL processor cannot take full advantage of domain-specific knowledge (Reference 20). Some systems have directly addressed the efficiency issue (e.g., DARTS, PSI) but we have not seen conclusive evidence to indicate that they have been successful in their attempts at producing code that is efficient enough for the missile flight software domain. More recent systems stress the importance of domain-specific specification languages and domain knowledge.

PDL's are semi-formal, general purpose specification languages, and as such, suffer from the same drawbacks as general purpose formal specification languages. PDL's can be used at varying levels of abstraction, and this should be viewed as an advantage to their use as an input medium. Additionally, PDL's based on Ada have been developed, and their use for specifications reduces the variety of languages a software engineer must know.

As mentioned previously, graphical languages have advantages over other types of languages in terms of both clarity and speed of information transfer. They permit the concise representation of large amounts of information. The software and hardware technology required to support graphical input of requirements is still in the early development stages; graphical specification languages cannot be easily processed into a machine-comprehensible form. Booch (Reference 38) and Buhr (Reference 39) have both developed manual graphical representation schemes for depicting software parts at a high level.

g. User Support

The quality and quantity of user assistance directly impacts the usability of any system, and thus is of concern when evaluating a software generation system. Specifically, the system should be viewed in light of the amount of assistance provided when the user is specifying requirements. Ideally, the user should be provided with an iterative approach to requirements specifications. System checking for completeness, consistency, and clarity of requirements is another desirable feature of a software generation system.

h. System Outputs

The two major outputs from a software generation system are code and documentation. Because of efficiency concerns, optimizing procedures within the software generation system may be desirable. Correctness of missile flight software is critical; therefore, facilities for verifying correctness are also desirable.

The system should be further evaluated in light of its ability to generate supporting documentation. Text generation is, as yet, an immature technology area. As mentioned earlier, a few systems generate textual output, but for the most part, it is done at a rather mechanical level.

4. CONCEPTUAL FRAMEWORK

The CAMP investigators found it useful to develop a view of an ideal software generation system to serve as a framework for developing near-term and mid-term recommendations for automation of the software generation process; we refer to this as our Conceptual Framework. There are several versions of a software generation system that can be envisioned as we proceed from the near-term to the long-term, but in this paragraph we will concentrate on presenting a single ideal system without emphasizing its technological feasibility.

a. The Ideal System

An ideal software parts generation system should have the ability to manipulate pre-built Ada parts, as well as the ability to generate new software parts. The major facilities of such a system are summarized in Figure 13.

1. PARTS IDENTIFICATION	THE PROCESS OF SELECTING A PART, OR SET OF PARTS, FROM A SET OF PRE-EXISTING PARTS FOR A SPECIFIC APPLICATION.
2. COMPONENT CREATION	THE PROCESS OF CREATING A SPECIFIC COMPONENT.
2a COMPONENT INSTANTIATION	THE PROCESS OF CONSTRUCTING AN INSTANTIATION OF A GENERIC SOFTWARE PART.
2b COMPONENT GENERATION	THE PROCESS OF CONSTRUCTING A SPECIFIC COMPONENT FROM A SCHEMATIC PART BY MEANS OF A PARTS CONSTRUCTION SCHEME.
2c COMPONENT CONSTRUCTION	THE PROCESS OF MANUALLY BUILDING A SPECIFIC SOFTWARE COMPONENT.
3. PARTS COMPOSITION	THE PROCESS OF INTEGRATING PARTS INTO A SOFTWARE SYSTEM UNDER DEVELOPMENT.

Figure 13. Facilities of a Software Generation System

Before discussing the technology requirements for an ideal SGS, a scenario of the system's use will be provided. Figure 14 depicts a high-level view of the system; Figure 15 goes into more detail.

The software generation system should have an intelligent interface; expert system assistance should be provided for all system facilities and processes. Requirements specification should be an iterative process performed at a high level of abstraction. In order to accommodate users with a wide range of backgrounds and needs (i.e., the user should not have to be a computer scientist), a variety of interface techniques should be provided (e.g., natural language, graphical language, formal (machine readable) specification language).

The specification should not have to be complete; the system should have facilities for dealing with partial knowledge. Analysis of the specification should be performed and should include checking for completeness, consistency, and clarity. Information should be solicited from the user as needed.

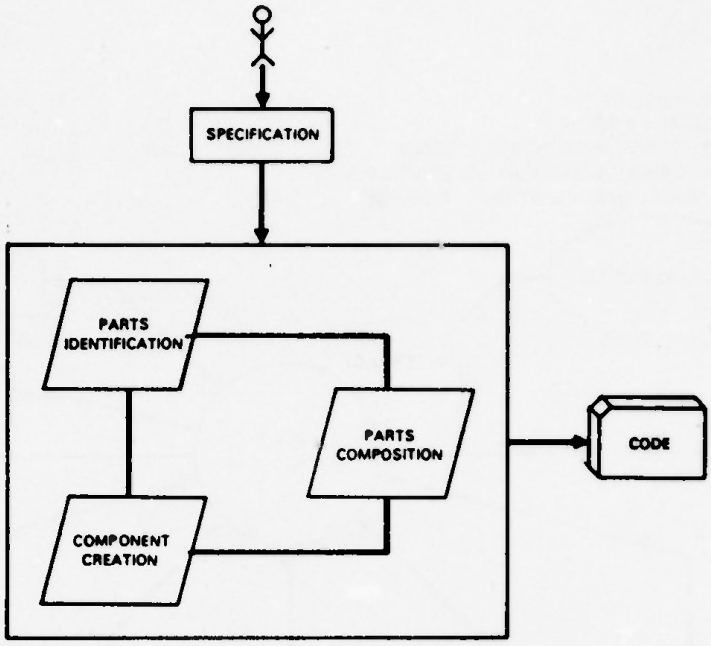


Figure 14. Overview of the Ideal Software Generation System

Once the specification phase is complete, it should be determined if a pre-built part exists that meets the user's requirements (this requires facilities for automatic location of existing software parts). Parts may be simple or meta-parts (see Volume I, Section II for a discussion of software parts), and one or more parts may be located that meet the requirements. If a part is located, the user will be notified in order to prevent redevelopment, otherwise, a new part will be built.

The user should be able to recall the specification in any of a number of forms--textual, graphical, formal specification language. Documentation should be generated as needed.

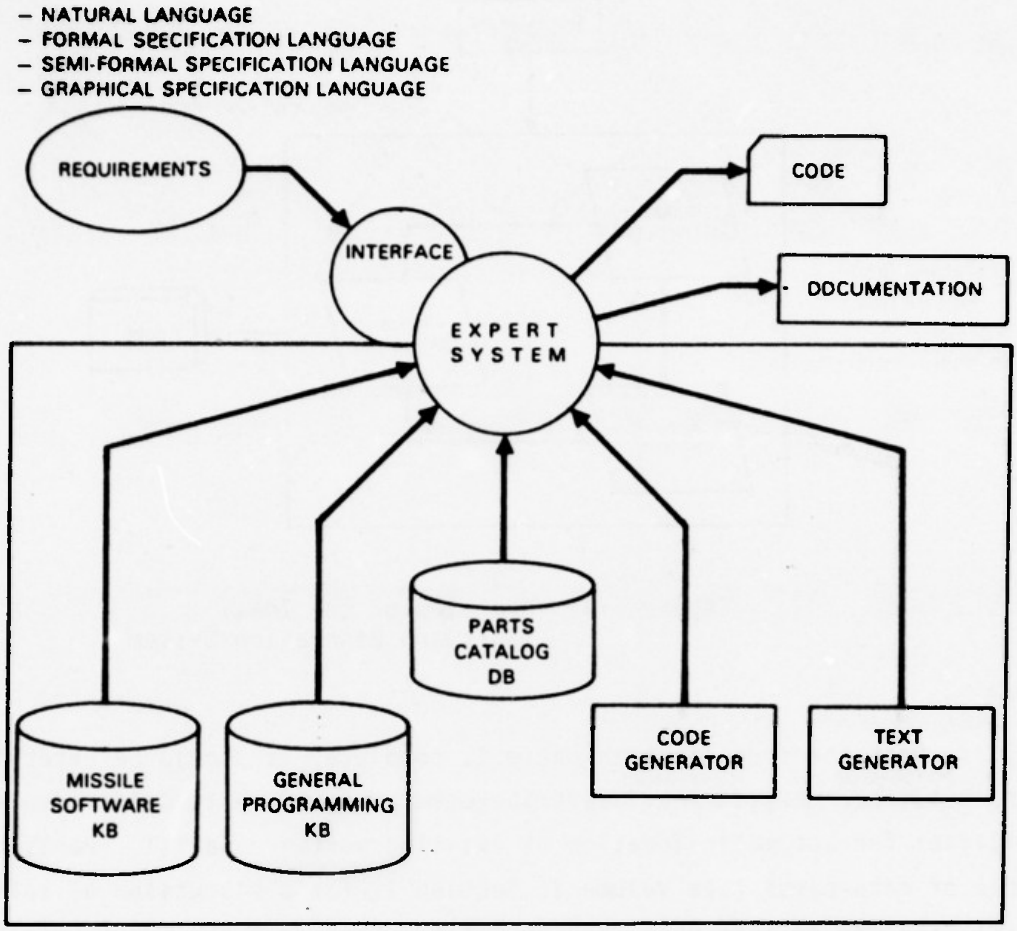


Figure 15. The Ideal Software Generation System

b. Components and Requirements

Based on an analysis of the scenario depicted in Paragraph 4a, the high level component requirements can be determined; Figure 16 summarizes these requirements. Some of the areas have previously been discussed (e.g., specification techniques; parts identification--see Section II); the other areas are discussed in the following paragraphs.

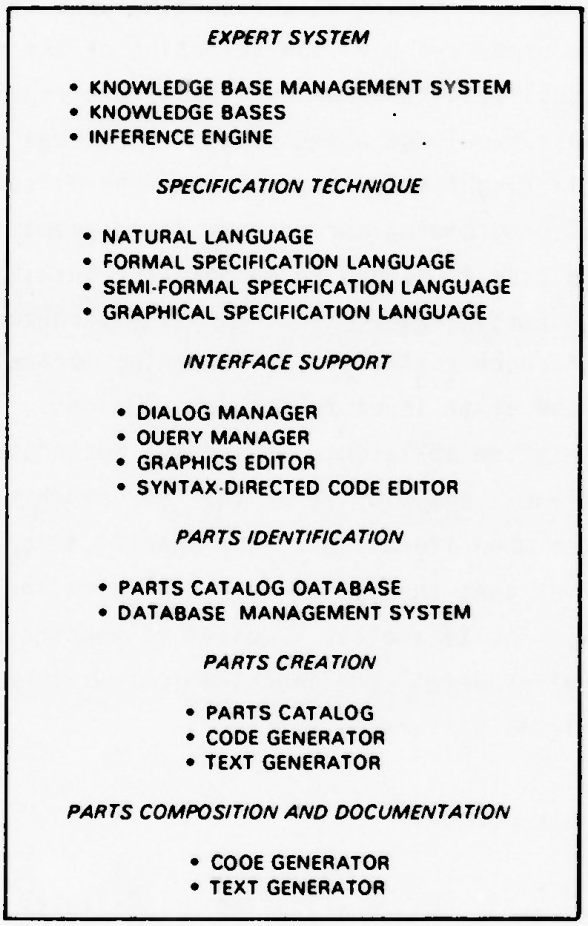


Figure 16. Major Component Requirements of an Ideal SGS

(1) Expert System Assistance

Expert system assistance should be provided throughout the system. This requires a knowledge base management system, several knowledge bases, and an inference engine.

A knowledge base management system (KBMS) is similar to a database management system in that it manages and coordinates activities within the knowledge bases. KBMS's vary depending upon the knowledge representation scheme used, and the sophistication of the system.

Conceptually, three knowledge bases are required: (a) the Missile flight software knowledge base contains knowledge specific to the development of missile flight software, (b) the General software knowledge base contains general programming and program development knowledge, and (c) the General knowledge base is needed to support the intelligent interface (e.g., support of a natural language specification technique).

The inference engine is the reasoning mechanism that utilizes the knowledge bases and other input to draw conclusions.

Expert system assistance includes a mechanism to analyze the completeness, consistency, and clarity of the requirements provided by the user. This determines when iteration of the specification-analysis phase terminates, and implies that the system must deal with incomplete (i.e., partial) information. The technology required to support this mechanism depends upon the level of detail and checking that will be performed and the way in which the analysis will be performed.

(2) Interface Support

Natural language specification necessitates the presence of a Dialog Manager. The Dialog Manager is responsible for managing (i.e., analyzing, processing, and conducting) the natural language dialog with the user.

The Query Manager handles queries directed to the database. This function would generally be performed by the database management system. Queries must be translated into a machine comprehensible form. The technology requirements for this component are dependent upon the query specification technique.

Interface support for an automated software generation system includes a Loader/Unloader for formal specifications in a machine-readable form. The Loader is needed to input machine-readable specifications directly into the system; this is similar to loading a HOL program. The Unloader outputs machine-readable specifications; this is analogous to unloading object code for an HOL program.

A graphical editor is required to support a graphical specification technique; it provides an easy way to manipulate the components of the specification.

A code editor is another requirement of the interface support. Ideally, a syntax directed editor should be part of the automatic programming environment. Ada syntax directed editors are currently under development in the commercial sector.

(3) Parts Identification

The Parts Identification facility requires a parts catalog which was discussed in detail in Section II. Expert system assistance should be provided in locating parts. Additionally, an automatic code locator should be provided to determine the existence of a software part; this mechanism would prevent the development of redundant code. Such a mechanism requires the system to be able to translate the user's requirements specifications into a form that would allow formulation of a query to the parts catalog. If the user was attempting to build a part that already existed, he should be notified of the existence of the part. It may not always be possible to accurately ascertain the existence of a part.

(4) Component Creation

Component Creation consists of Component Instantiation, Component Generation, and Component Construction. Parts Identification plays a role in determining whether a part exists that can be instantiated (i.e., a generic part) or generated (i.e., a schematic part), or has to be constructed (either automatically or manually). Automated component construction requires a code generator and other supporting mechanisms at a lower level. Expert system assistance should aid in the creation of software parts.

(5) Parts Composition

Parts Composition involves the integration of software parts. Ideally, software parts composition would be an automated process based on expert system knowledge and the user's requirements specification. Parts composition requires code generation to combine the individual software parts. The degree of automation of this facility has a significant impact on the supporting mechanisms required. HOS's USE.IT system has an automated parts composition element, but this does not appear to be at the level of sophistication desired for the ideal system presented here. Research is continuing in this area.

5. RECOMMENDATIONS

The recommendations presented here take into account the ideal software generation system presented in Paragraph 4, and temper it with what appears to be technologically feasible. Recommendations are presented for both the near-term and mid-term. We define near-term to be in the range of 0 to 3 years, and mid-term to be from 4 to 7 years. The recommendations start with a very basic system that handles parts identification, management, and generation, and proceed to a progressively more sophisticated and fully automated software generation system. At each stage of development, increasingly more sophisticated technology is required, thus the design must allow for evolution over time as technological advances are made. This is a very important aspect of our recommendations. Given the fast pace with which technological advances are made, users should not be burdened with a system made obsolete by its inability for progressive development.

a. Near-Term Recommendations

The system with the greatest potential for near-term payoff is a relatively simple system consisting of parts identification and parts management facilities, and a parts generation facility that makes use of the meta-parts (generic and schematic Ada parts) discussed earlier. The parts identification facility would be as described in Section II. The parts management facility would keep track of parts usage (i.e., where and by whom parts are being used).

The basic scenario for using such a system involves the user interfacing with the system to determine the existence of a simple or meta-part that will meet his requirements. The user could either specify his needs via some query language, or browse through the catalog. Zero or more parts may be found that fit the user's described needs. If a part is found, further information about it may be obtained by accessing the complete catalog entry. If a user selects a part for use, the Parts Management system must keep track of this.

Simple parts may be used as is (i.e., merely by providing the requisite parameters). If a generic meta-part is found, the user must provide instantiating information in order to generate a usable software component. Schematic meta-parts provide information on how to build the required software part; the user would perform the actual construction.

Based on this scenario, several systems can be envisaged that are currently technologically feasible. One version of such a system could be developed with limited technological requirements (e.g., a database management system, a query language, and minimal additional supporting software); Figure 17 depicts a simple Parts Identification facility.

The system could also be implemented with a limited natural language or domain specific specification language, and some rudimentary form of graphical specification technique. As we have stated previously, the technology required for unrestricted natural language dialog is not currently of sufficient maturity for production quality systems. Limited forms of NL interfacing are feasible; some success has been realized with limited natural language database interfaces. As mentioned earlier, the technology required to support a full-blown graphical specification technique is still in the early stages of development, but it appears feasible to provide an elementary graphical technique. These additions, while currently technologically feasible, impose considerable additional technology requirements upon the implementor.

Figure 17. Parts Identification

Expert system assistance could also be provided for the parts identification and creation facilities. Using a Parts Identification Expert (PIE) and a limited natural language (or domain specific specification language), the user would specify his requirements, and PIE would transform the specification into an internal form that could be used to access the parts catalog to determine the existence of the requested part(s). A Parts Construction Expert (PCE) could be used to aid in the instantiation and generation of components. Figures 18 and 19 depict an expert system approach to parts identification and creation, respectively. The main differences in approaches to near-term systems is in the system interface technique and the technology required to support it.

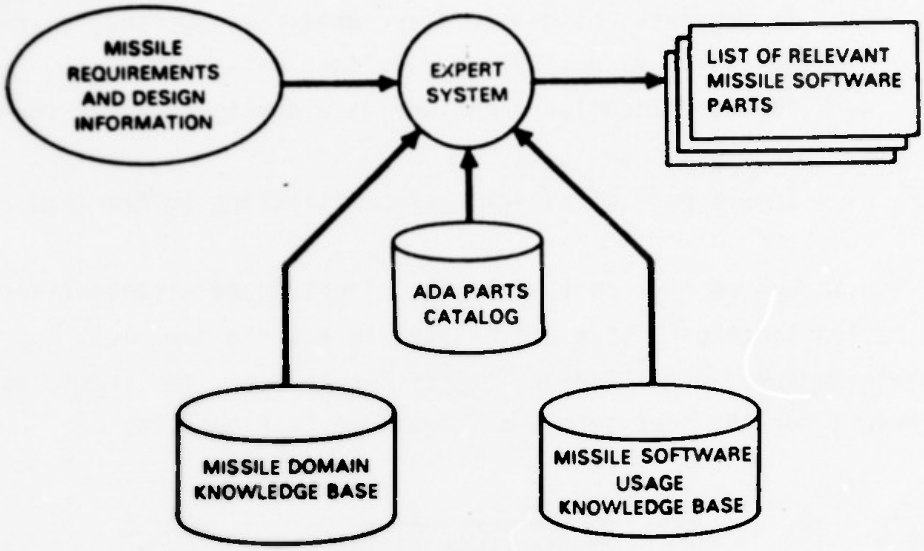


Figure 18. An Example of Parts Identification with an Expert System

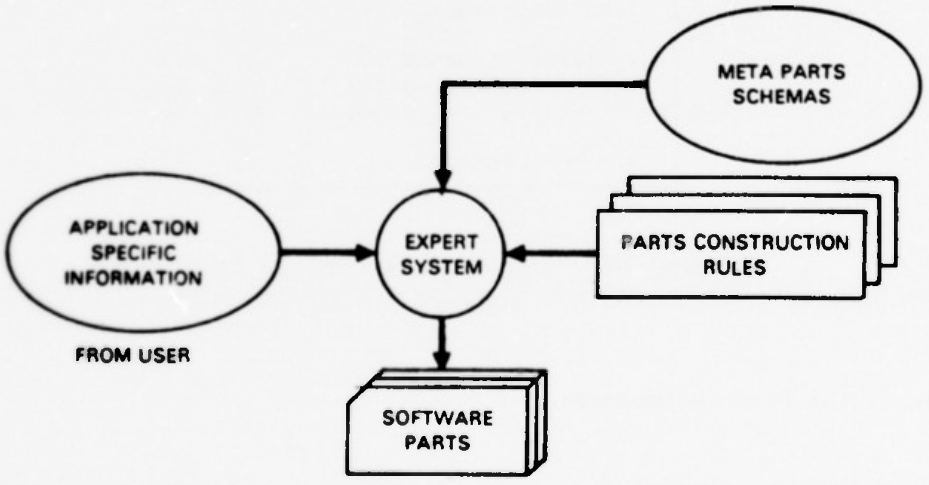


Figure 19. Parts Construction with an Expert System

A near-term system may be characterized by the following:

-- It generates single software units (as opposed to entire software systems) via pre-defined meta-parts.
-- The specification technique is a domain specific specification language.
-- Some degree of expert system assistance is provided.

Although we have continually highlighted the disadvantages of formal specification languages, they are relatively easy to implement and thus contribute to the overall feasibility of the system. The technology requirements for the near term are summarized in Figure 20.

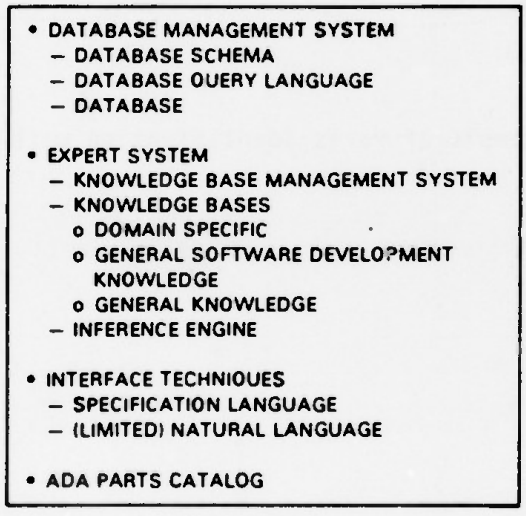


Figure 20. Near-Term Technology Requirements

b. Mid-Term Recommendations

Most of the technological advances will affect the specification technique and the component creation facility. In the mid-term we may expect the software generation system to allow the specifications to be provided at a higher level of abstraction than was previously possible. Additionally, we may now expect the parts creation facility to progress beyond merely

supplying the user with parts constructors to actually generating code for some parts.

The basic scenario in this stage of development involves having the user, via some high-level specification technique and/or natural language dialog, specify his requirements. The software generation system would analyze the requirements for clarity, consistency, and completeness. The specification-analysis phase would be an iterative process. Once the specifications were finalized, an automatic code locator would determine if a simple or meta-part exists that would satisfy the user's requirements. If a simple part existed, it would be retrieved for the user. The user would be provided with expert system assistance for the instantiation and generation of meta-parts. Automated construction of some parts will be feasible. Figure 21 depicts a view of this system.

The automated parts composition system designed during CAMP is currently feasible (and thus fits the near-term classification). It can generate complex software components from predefined meta-parts but cannot generate entire systems. It will make use of a limited natural language interface and specification method, and it will incorporate expert system assistance. Sections IV and V contain more information on this system. The reader is also directed to References 44 and 45 for a description of the CAMP parts composition system.

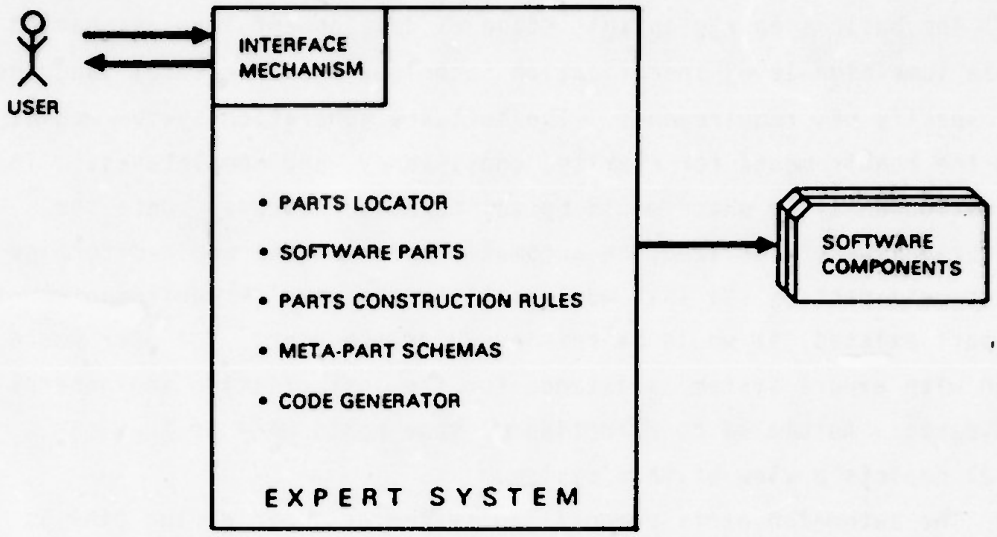


Figure 21. Mid-Term Software Generation System

SECTION IV
THE ROLE OF EXPERT SYSTEMS

1. Expert System Overview 61
2. Schematic Part Constructors 63
3. Generic Instantiator 67
4. Parts Identification 67
5. Parts Catalog 68
6. AMPEE System 69

1. EXPERT SYSTEM OVERVIEW

An Expert System is a software system which emulates the manner in which human experts solve problems. A particular expert system is a software system which has been given a body of knowledge about some finite domain (i.e., application area) and a method of applying this knowledge to problems within this domain. When presented with data about a specific problem within the domain, the expert system is able to draw conclusions about the problem and possibly take some actions based on the conclusions it has reached.

Conceptually, an expert system has a very simple structure (see Figure 22). It's knowledge base contains all the knowledge about the domain over which it is intended to be an expert. The inference engine or inference generator is the mechanism by which the knowledge is used in light of a given set of problem data to infer conclusions. If an analogy is made between humans and expert systems, the knowledge a human possesses would correspond to the knowledge base of the expert system, and the physical, electrical, and chemical mechanisms of the brain would correspond to the inference engine.

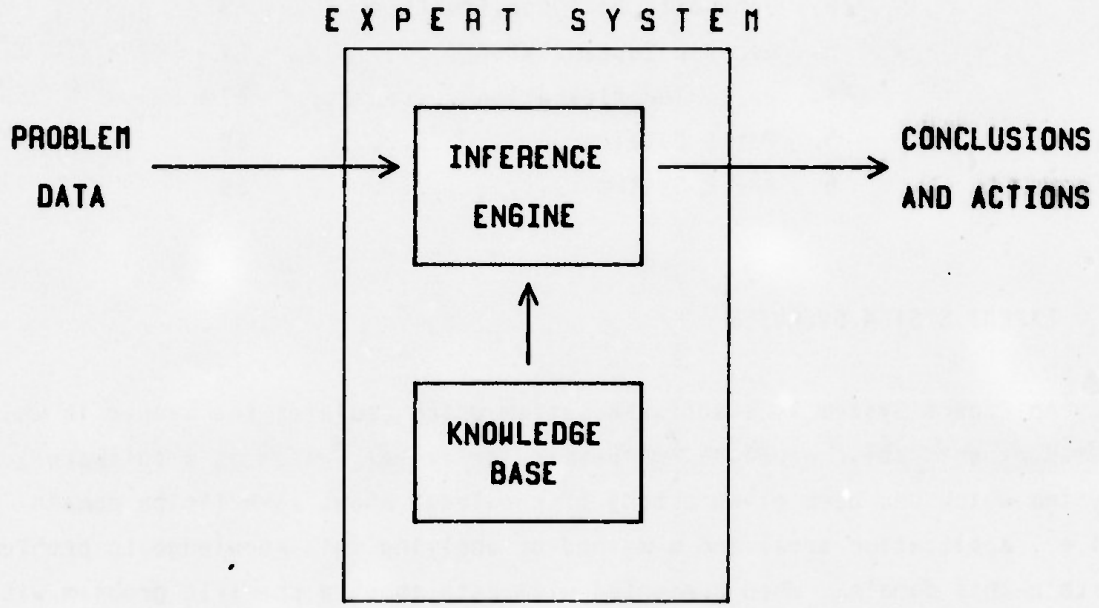


Figure 22. The Structure of An Expert System

A great deal of research has been performed over the past two decades into the structure of knowledge. Much of this work is still in the conceptual stage but some of it has been incorporated into commercially available products. For the purposes of this report, the knowledge structuring mechanism of one such product, the Automated Reasoning Tool (ART), will be used to illustrate a typical expert system knowledge base (it should be noted that ART is not typical when compared with older systems). Section V contains more information on ART. ART's knowledge base consists of three types of knowledge--facts, rules, and schemata.

A fact is a statement of truth within the domain of expertise. For example, the statement "1 is the identity for multiplication" is a fact likely to be found in an expert system designed to manipulate mathematical equations. Likewise, the statement "steel is heavier that wood" is a fact.

A rule is a statement of inference. An inference statement can be conceptualize as a statement which says "If I know A, then I can infer B". For example, the statement "If R is transitive and aRb and bRc, then aRc" would be a rule typically found in an expert system designed to manipulate mathematical equations. Likewise, the statement "If X is the lightest available material and X is sufficient strong, then use X in the product" is a rule.

A schemata is a mechanism used to structure facts. A schemata is very similar to a data structure in classical programming languages in that it allows the aggregation of data into a single entity. A structure which contains all the information about a software part in a software parts catalog would be an example of a schemata.

In the remaining portions of this section, the utility of expert systems will be discussed in various software parts composition areas. At the end of the section, a system will be presented which encompasses all these areas into one tool.

2. SCHEMATIC PART CONSTRUCTORS

During the CAMP domain analysis, it was determined that there were some types of commonality which could not be implemented by means of the Ada generic facility alone. In other words, we identified software design paradigms which we believed could be automated but the Ada generic facility was not sufficient for this process. We called these types of parts schematic parts. A schematic part is a design template together with a set of construction rules used to build application-specific software components from the template. After examining various methods of automating these schematic parts we decided that an expert system would be the best vehicle for building the schematic part constructors (see Figure 23).

These constructors would allow the user to specify his application's needs for a specific software component and would build the Ada code to satisfy these requirements. This process is best illustrated with examples.

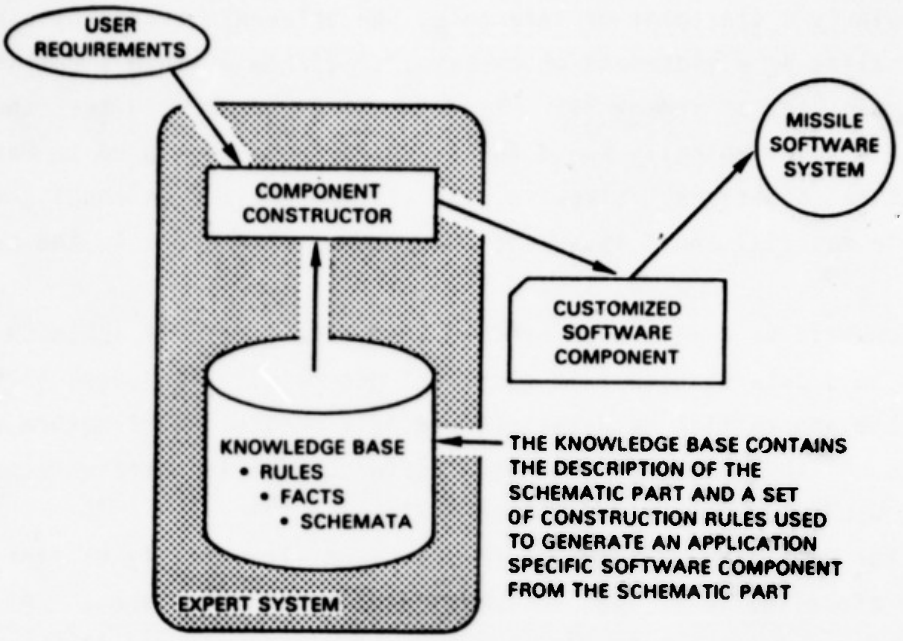


Figure 23. Overview of a Schematic Part Constructor

Figure 24 depicts the structure of a typical missile's lateral directional autopilot subsystem. This subsystem was identified as a schematic part because it can be mechanically constructed given basic requirements from the user such as the type of digital filters to use, the required range and precision of the data, and the type of limiters to use. Given this information, it is possible for an expert system to construct the application-specific Ada code for this subsystem. It should be noted that the expert system will use quite a few non-schematic CAMP parts in this construction. For example, it will use CAMP parts to construct the digital filters and limiters.

Another example of a schematic part is illustrated in Figure 25. The construction of a navigation subsystem is dependent upon what the user wants the navigation subsystem to compute, what data is provided as raw input to the navigation subsystem, and what navigation coordinate system (e.g. wander azimuth, north pointing) the navigation computation are to work within.

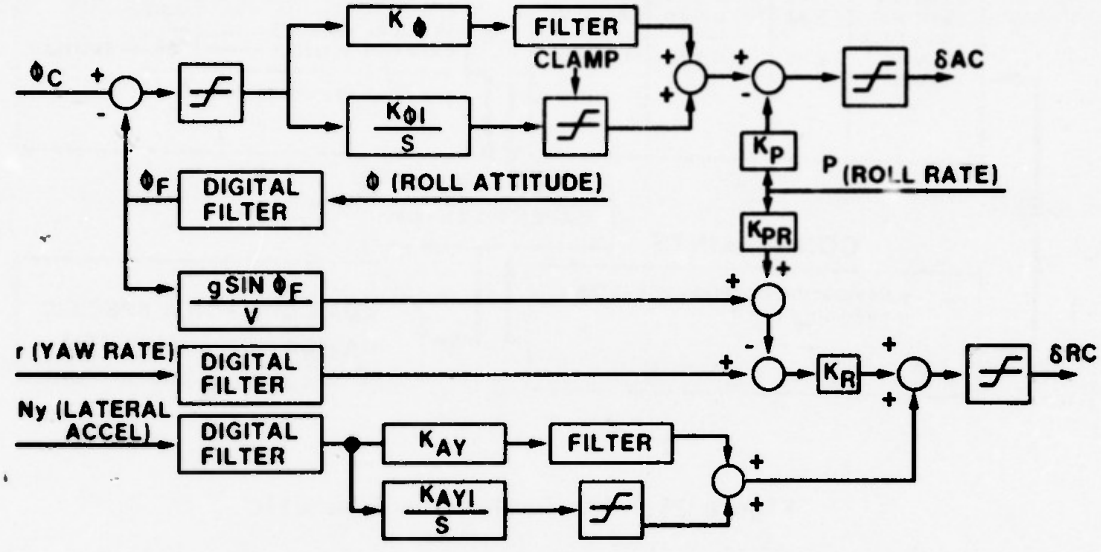


Figure 24. The Lateral/Directional Autopilot Schematic

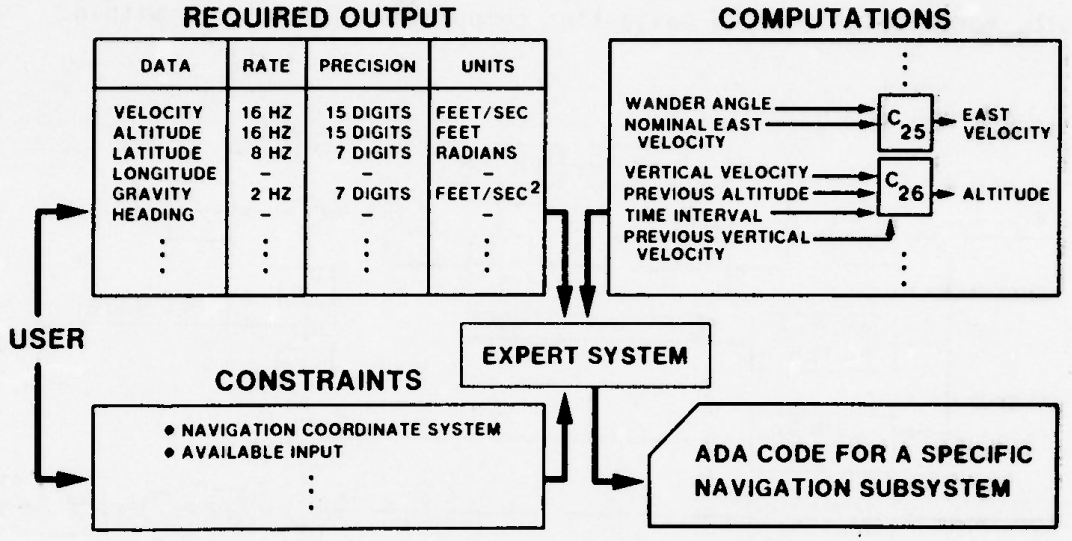


Figure 25. The Navigation Schematic

Given this data, the actual computations to transform the input to the required output are relatively standard. Figure 25 illustrates a schematic part constructor whose knowledge base would contain the standard computations such that when told the required output, available input, and coordinate system, the constructor would be able to select the correct computations for performing the navigation functions required to produce the output.

Appendix D in this volume presents a much more detailed example of a schematic part and its constructor. Appendix D is the result of actually building a proof-of-concept implementation of one of the schematic part constructors.

3. GENERIC INSTANTIATOR

In order to make the CAMP parts as reusable as possible while still protecting them against misuse, many of the CAMP parts were designed as generic subprograms or packages with relatively complex generic declaration sections. Fortunately, by using defaults for many of the generic functional parameters, this complexity can be hidden from the user. But, when the user wants to have more control over the operation of the part (e.g., what sine routine it should use) he will need to be able to properly instantiate these generics so that the defaults are overridden. For these reasons, we believe some type of general purpose generic instantiator is needed as part of the software parts composition system. This constructor will have the ability to construct the Ada code for correctly instantiating any generic based on data it obtains from the user by means of a dialog. In effect, this generic instantiator will allow the part designer to specify what questions should be asked of the user to allow the proper instantiation of the generic part.

4. PARTS IDENTIFICATION

A key aspect to an effective software parts program is to provide a mechanism for the early identification of appropriate software parts. Software parts need to be identified very early in the system development process (even before the completion of the software requirements activities) in order to facilitate trade-off analyses, cost estimates, software sizing and timing analyses, and other activities. In many cases, the functions provided by a software parts catalog (to be discussed in the next subsection) are not sufficient for this task. What is needed is the ability to relate product characteristics to software parts.

The software parts identification function provides the user with the ability to find appropriate parts for a new application based only on high level missile requirements and design information. In effect this function maps missile requirements to software parts. Figure 26 depicts some sample rules for this function. In this sample, by knowing that an anti-ship missile is being constructed the expert system can infer the need for a terminal seeker interface package part.

RULE 1: IF the missile's target type is a ship
THEN the missile will contain some type of terminal seeker

RULE 2: IF the missile contains a hardware component X
THEN the missile software system will need a software interface package to X

RULE 3: IF the missile needs a software interface package to X
THEN ask the parts catalog if one exists

RULE 4: IF the catalog confirms the existence of part X
THEN ask the user if part X is satisfactory for his application

Figure 26. Sample Parts Identification Rules

5. PARTS CATALOGING

Examined in isolation, there is little evidence that an expert system needs to be used for a software parts catalog. Although expert systems are well suited for this type of application, existing mature tools such as database management systems can implement a software parts catalog quite well. But, when one considers the close interaction between the schematic part constructors, the generic instantiator, the part identification function, and the parts catalog, there are benefits to implementing the parts catalog in the same expert system as the remaining functions. The parts catalog provides several functions for managing parts and examining all information about the parts. These functions have been discussed elsewhere in this volume. the next subsection also presents some details on the role of the parts catalog.

6. AMPEE SYSTEM

During CAMP, a software parts composition system was designed based on the use of an expert system that would provide all the aforementioned capabilities in one tool. This system was entitled the Ada Missile Parts Engineering Expert system and is summarized in Figures 27 and 28.

The advantages of use using an expert system for this tool are:

a. Expert systems are useful when the process being implemented are evolutionary in nature. In other words, when the knowledge changes rapidly, a classical program would have to be recoded. An expert system needs only its knowledge base changed.

b. Expert systems are very powerful symbolic processors. Our implementation of a schematic constructor showed that with a small number of rules, a very powerful system can be constructed.

c. Expert systems allow the construction of a very simple user interface. Because the interaction between man and machine has been a primary focus of artificial intelligence since its inception, most expert systems have very powerful facilities for building interfaces which allow the system to be used with minimum training and/or expertise.

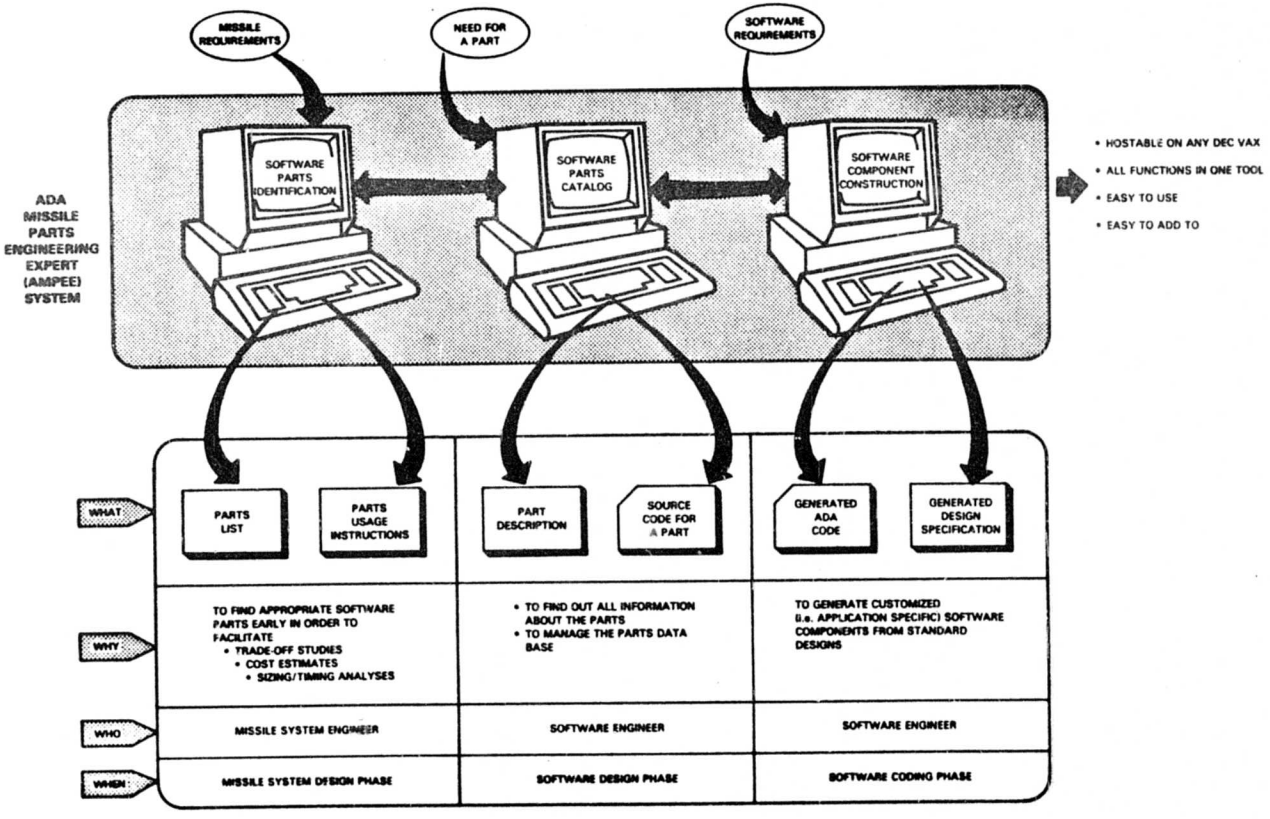


Figure 27. Overview of the AMPEE System

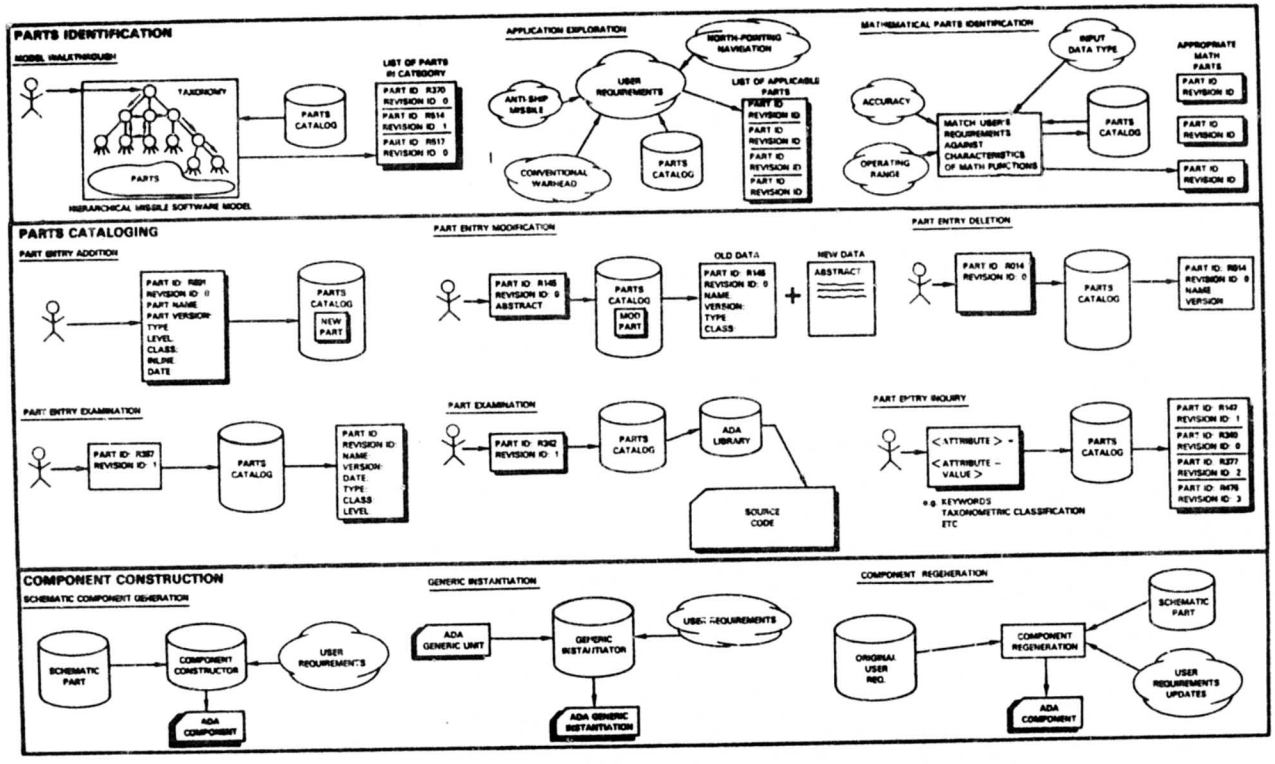


Figure 28. Operations Provided by the AMPEE System

SECTION V
EVALUATION OF AN EXPERT SYSTEM

1. Introduction 72
2. Means of Evaluation 74
3. Overview of ART 74
4. Evaluation of ART with respect to the Problem Domain (AMPEE) 81
5. Conclusions 85

1. INTRODUCTION

The CAMP program was tasked to evaluate the role that could be played by an expert system in support of software reusability in the missile flight software domain (see Section IV), and to evaluate a commercially available (i.e., an off-the-shelf product) expert system tool relative to this domain. ART™, available from Inference Corp., was selected for evaluation.

ART is a commercially available expert system development tool that falls into the category of an expert system shell. An expert system shell is a software system that provides the inference engine and infrastructure for an expert system thus greatly simplifying the development of an expert system. The expert system developer need only supply the domain specific information for his application (i.e., he provides the rules, facts, etc., which are specific to his domain). Although this is not a trivial task, it does bring the development of expert systems within the grasp of many more people.

It must be emphasized that ART is neither an expert system nor a software generation system, rather it is a tool that can be used in the development of expert systems. The CAMP project utilized ART as the basis for developing a software parts engineering expert system, but ART is not limited to that domain (i.e., it can be used as the basis for development of expert systems in virtually any domain).

ART was selected from among the available products for a number of reasons; they are summarized in Figure 29, and discussed in the following paragraphs.

- No hardware procurement under CAMP contract
- Available on a widely used processor
- Lower cost to end-user by utilizing VAX
- Sufficient functionality

Figure 29. Why ART was Selected for Evaluation

The CAMP contract called for the procurement of no additional hardware, thus it was almost mandatory to find a product that was available on a VAX. Additionally, it was desired to evaluate a product that was hostable on a widely available processor. Again, this pointed to a product that was available on the VAX.

VAX equipment is in widespread use throughout the DOD and defense contractor communities, thus the cost of adopting the expert-system approach to software parts engineering that is recommended here is much lower than if specialized hardware (e.g., LISP machines) were required. Cost can be measured in terms of both time and money. The monetary cost is much less because specialized hardware is not required. The time cost is also less because personnel are already familiar with the VAX. Familiarity has the additional advantage that, although there are many new ideas associated with a parts engineering approach to software engineering, at least the tool is based on a familiar computer system and thus may not appear as radical. These factors can contribute substantially to the probability of success of this type of software reusability effort.

Many of the expert system development tools that are commercially available have been developed for specialized LISP hardware such as the Symbolics machine. Although these specialized machines are optimized for LISP, and generally provide a comprehensive development environment, the additional cost of acquiring such hardware was unacceptable at this time both from the viewpoint of developing the system, and from the viewpoint of expecting others to acquire and use such a system. ART provides functionality that is at least on a par with many of the products that are available only on specialized LISP hardware; this functionality is discussed in Paragraph 3. (Note: ART is also available on both the Symbolics and LMI LISP machines.)

Thus, the cost, functionality, and availability of ART made it a feasible product for evaluation in the CAMP study.

2. MEANS OF EVALUATION

In order to evaluate ART, it was used as the basis for a proof-of-concept implementation of a software parts engineering expert system known as the Ada Missile Parts Engineering Expert (AMPEE) System, and as the foundation for the requirements and design of a prototype of the AMPEE System. The requirements and top-level design of this prototype system are described in References 44 and 45, respectively. The AMPEE System is an expert system to promote software reuse in the area of Ada missile flight software. It provides capabilities to catalog, identify, and locate reusable Ada software parts. It also provides a component construction facility to assist the user in the instantiation of meta-parts. The proof-of-concept implementation involved primarily the Finite State Machine Constructor which is described in detail in Appendix D of this volume.

The application did not entail the use of all of the features provided by ART, but many of them were tried with smaller sample applications. Additionally, one member of the CAMP team attended two weeks of training in the use of ART and was able to try and discuss features that were not used in the CAMP application. The proof-of-concept implementation made use of the following ART features:

- facts
- relations
- schemata
- rules
- case statement, if-then statement
- LISP interface

3. OVERVIEW OF ART

In the following paragraphs a number of ART features that facilitate the development of expert systems are discussed. Some other features that would be useful, although they are not currently available, are also identified. Finally, some problems and design issues that arise from the use of ART to develop an expert system are discussed.

It should be noted that during the evaluation period, only Beta versions of ART were available on the VAX. Although an improvement was noted between Beta versions, initially functionality was somewhat limited, and development was hindered by system bugs.

a. ART Background

An ART program does not contain functions or subprograms as are found in traditional programming languages such as Ada or Fortran. Instead, actions (which traditionally are performed by subprograms, etc.) are performed by rules which fire (i.e., execute) when all of their conditions are satisfied. Rules generally take the following form:

if <condition(s)> then <perform specified action(s)>

Conditions take the form of patterns and pattern restrictions which must be matched by data within the current state of the expert system's knowledge base. Data is represented as facts which take a specific syntactic form (see the ART Reference Manual, Reference 46, for details).

ART automatically runs through all of the rules that have been supplied in an effort to determine which rules have their conditions met (or satisfied) by facts currently in the knowledge base. Rules whose conditions are met are said to be instantiated. Instantiated rules are placed on what is referred to as an agenda. Only rules that are on the current agenda can be fired (i.e., executed).

The determination of which rule on the agenda will fire is based in part on priority. The expert system developer can assign priorities to rules which will cause an ordering of rules on the agenda (i.e., rules that have been instantiated). If a rule has not been assigned a priority, either explicitly or based on the type of rule, it will automatically be assigned the default priority. Rules with the same priority are ordered on the agenda in essentially a random manner.

If a rule firing causes a change in the current state of the knowledge base, the agenda will be recalculated. In general, rules do cause changes to the current state of the knowledge base, thus one can effectively think about the agenda being updated after every rule firing.

At any given point in the running of the expert system, a particular fact will only cause one firing of a given rule. The implication of this is that although a rule may still be instantiated by a particular fact, it will not remain on the agenda or fire repeatedly in what would essentially be an endless loop.

Retrieval of information from an ART knowledge base differs from retrieval of information from a traditional database. Within ART, searching is merely an outgrowth of the pattern matching process, i.e., specific searching routines need not be developed because this function is performed automatically by the pattern matching process which is an integral part of the ART environment. Specific rules are needed to direct ART to attempt pattern matching in search of specific data within the knowledge base, but this differs from writing a traditional search routine.

b. Features Provided

ART consists of both a programming language, and a development and run-time environment for expert systems. ART incorporates many features that facilitate the development of an expert system; these are summarized in Figure 30. Detailed information can be found in the ART reference material (References 46 - 49).

- Forward and Backward Chaining
- Viewpoint Mechanism
- Schema Structure
- Relation Facility
- LISP Interface
- Development Environment
- Debugging Aids

Figure 30. ART Features

The reasoning mechanism of an expert system is referred to as the inference engine. The inference engine typically works through either forward or backward chaining; ART incorporates both reasoning methods. Forward chaining starts with a basic premise and reasons forward to a particular conclusion. A forward chaining rule could be represented in the form 'if an anti-ship missile is to be developed, then a terminal seeker will be needed'. Backward chaining begins with a conclusion to be proven (or a goal) and then attempts to find a series of logically consistent facts and rules that support that conclusion. For example, if the goal is to compute navigation coefficients and not all of the information is immediately available, subgoals will be established to compute the required information. If these subgoals can be satisfied, then the original goal will also be satisfied. Forward chaining reasoning has been referred to as being data driven, while backward chaining reasoning has been referred to as being goal driven (Reference 50).

Historically, expert systems have utilized a backward chaining reasoning mechanism. The parts engineering application involves the potential search through many parts in an attempt to determine all of the parts required for a particular application. Backward chaining through such a search space will not be as efficient as establishing a set of forward chaining rules to accomplish the same task, thus ART's dual reasoning mechanism is a desirable feature for incorporation into the AMPEE System.

ART provides a powerful viewpoint mechanism for use in modeling both temporal and hypothetical reasoning. The temporal viewpoint mechanism allows the expert system to reason about a situation over time, while the hypothetical reasoning mechanism allows the system to reason about hypothetical alternatives. One CAMP related area in which hypothetical reasoning could be utilized is component construction. For example, if the construction of a component requires the composition of several software parts from the Ada missile parts catalog, and more than one part meets the initial criteria for inclusion in the composite, hypothetical reasoning could be used to have the constructor pursue the use of these alternative parts. Through the use of the hypothetical reasoning mechanism, all alternative paths to the construction could be pursued essentially simultaneously. If timing, size, or accuracy constraints had been established by the user, this information could be utilized to select the appropriate instantiation for the user.

The two types of reasoning can be combined in a single application. The viewpoint mechanism also provides a method of reducing the search space. For example, if it is known that it is a contradiction for a certain set of events to occur in the same viewpoint, that viewpoint can be pruned from the search space; ART will never allow that viewpoint to be created again. ART also provides a mechanism for merging viewpoints in order to reduce redundancy.

A schema structure is provided that allows automatic inheritance of information between related schemata. A schema is a collection of facts about a particular object. The Ada software parts catalog that forms a part of the AMPEE System can be implemented via the schema structure provided by ART. This is done by establishing a template for a catalog entry; there is a place for each catalog attribute in the template. Default values or known properties about an attribute can be specified in this template. Each specific catalog entry is an instantiation of the template; the individual catalog entries will inherit the default values and specified properties.

ART also has a relation structure that provides a simplified means of enforcing certain consistency constraints. For example, if a relation 'upstream' was defined, and its inverse was defined to be 'downstream', then any time an 'upstream' fact occurred, a 'downstream' fact would be added to the knowledge base automatically.

ART provides a means of interfacing to LISP routines. This allows the expert system developer to write special-purpose LISP routines to perform functions not provided by ART and to access those routines from within ART rules. Examples of the types of routines that might be developed in LISP include data conversion routines and special-purpose input-output.

The Symbolics version of ART provides facilities for the development of graphic end-user interfaces; the facility is known as the ARTIST. The VAX version does not currently have this feature but it is expected to be available in the near future.

The ART development environment has several features to facilitate system development. For instance, the Studio interface provides menus for accessing ART facilities; these facilities can also be accessed via commands. Facilities are provided for watching changes as they occur in the knowledge bases, and for watching the firing of rules as the application is run. A graphical interface is available on the Symbolics version that allows

the user to watch the viewpoint structure during the running of the expert system. There are also facilities for inspecting the knowledge bases both before and after execution, and for viewing the static viewpoint structure. The Symbolics version has (and the VAX version will have) facilities for the display of multiple windows; this allows the user to view several aspects of system operation simultaneously.

c. Facilities not Provided

One facility not currently provided is for the use of variable names to reference schema slots, although this is a feature that is under consideration for incorporation in a future release of ART. This feature may not be needed on a day-to-day basis, but on occasion, it would allow rules of a more general nature to be written.

Additionally, ART does not provide a mechanism for permanently updating the initial state of the knowledge base by facts generated during a run. Currently the knowledge base is reset to its initial state each time the expert system is loaded or reset. This can cause difficulties in certain applications, but it is possible to work around this constraint by providing knowledge base updating routines in LISP and accessing those routines from the ART application.

ART does not currently support the use of rule sets. Although more than one ART application file can be loaded, if they are not all loaded at the time the expert system is initiated, the 'reset' that is required to make an ART application file accessible, will cause all of the ART files that have been loaded to be reset, thus restoring the knowledge bases to their initial state.

d. Problems Encountered with the VAX Version

The VAX version of ART is lacking many of the 'nice' run-time debugging features that are available on the Symbolics version of the product. As mentioned earlier, there are currently no end-user graphic interface capabilities, although work is currently in progress at Inference to make this feature available on the VAX.

Execution speed has also been somewhat of a problem with the VAX version of the product. Achievement of any type of reasonable response time on a VAX 11/780, required operation as a single user. Even then development and debugging proceeded at an agonizingly slow pace. ARI was rehosted on a VAX 8600 with a significant improvement in response time even with many (up to 15) users on the system running all types of applications. A three and a half fold decrease in CPU time was noted for the loading of some ART files. The really dramatic improvement came in actual elapsed time. ART is scheduled to be targeted for the Micro-VAX II; no figures are yet available concerning expected response time on this machine.

Improvements in speed will come in two areas: from the optimization of LISP by DEC, and from the optimization of ART by Inference.

e. Design Issues

The incorporation of ART into an expert system has a direct impact on the design of that system. For example, the AMPEE System will require a non-trivial amount of time to load and reset on the VAX, therefore it would be desirable to load in smaller portions of the system as they are needed. The problem with this is that in order for an ARI application program to be run, it must first go through the load and reset steps. Load causes the ART source code to be compiled and certain data structures to be built. Reset causes any initial facts and schemata to be asserted into the knowledge base. It also calculates the initial agenda. If, during the execution of some portion of the AMPEE System, it became necessary to load in another portion of the system, a reset would cause all of the Art-based expert system that had previously been loaded, to be reset. The implication of this is that all of the previously fired rules would become eligible for firing again as the old facts were re-asserted into the knowledge base. Additionally, the reset wipes out any interim facts that may have been added to the knowledge base by means of actions on the RHS of rules.

One possible solution to this problem is to precede the load and reset sequence of commands with a clear command. Then previously fired rules would not fire again, but, the clear would eliminate any intermediate facts from the knowledge base. Additionally, the rules that had previously been loaded would no longer by available, as they too would have been cleared from the knowledge base.

Another possible solution is to maintain the AMPEE System as a LISP suspended image. Thus, as part of the logout procedure from the AMPEE System, a new suspended image would be created that would be written to a new version of the same file that was resumed. If for some reason, an abnormal termination occurred during the execution of the AMPEE System, and normal end of processing was not performed, the work of the entire session would be lost because the new suspended image would not be created.

Depending on the type of usage that is foreseen (especially for the prototype version), this may not be such a drawback. For instance, if little updating will be performed to global knowledge bases (e.g., the catalog or the requirements database) then generally not much data would be lost in the event of an abnormal termination.

4. EVALUATION OF ART WITH RESPECT TO THE PROBLEM DOMAIN (The AMPEE System)

Because ART is an expert system development tool, it is suitable for the development of expert systems in any problem domain. Thus, in addition to evaluating ART itself, the expert system developed using ART must also be evaluated for its suitability to the problem domain. In the paragraphs that follow, the evaluation criteria and issues identified in Section III will be applied to the system proposed under the CAMP study. These issues and evaluation criteria are summarized in Figure 32.

The AMPEE System does support the reuse of pre-built Ada software parts for use in the area of missile flight software, but the prototype design does not call for an automatic means of enforcing reuse. Reuse is supported at the requirements and design level via the use of schematic parts. Reuse at the code level takes place through the reuse of simple and generic parts. Code efficiency is very important in the AMPEE System. There are two places where this comes into play:

- The simple and generic parts are coded as efficiently as possible.
- Efficiency rules are incorporated into the part constructors to facilitate the production of efficient code.

The technology used in the AMPEE System has emerged from the laboratory and is now commercially available, but it is still considered an emerging technology area. The system is designed to be flexible and easy to maintain in order to incorporate future technological advances.

Automation is provided in the areas of part identification and location, and in the generation of tailored software components from meta-parts. Further automation can be incorporated as it becomes feasible.

The AMPEE System is targeted for the missile flight software domain, thus, little is required when the system is delivered. The catalog of parts is easily updated, thus the addition or deletion of parts to the system is easily accomplished.

The user will interface to the AMPEE System via menus and a limited natural language dialog. Each major facility (i.e., the catalog, parts identification, and component constructors) will be directly accessible to the user. It is expected that the system will be usable by both software engineers and domain engineers. User training requirements will be developed in the next phase of CAMP.

The user will be prompted for specifications for the software component under construction. The specifications provided by the user will be analyzed for consistency and completeness. The transformation of these specifications into different forms (e.g., program design language, text, graphical form) is not a feature that will be provided initially, although it is a desirable feature of this type of system.

The component constructors will produce correct Ada code that will have efficiency built into it, but facilities are not provided for proving the correctness of the code. The size of the AMPEE System and the storage and response times are indeterminate at this time (see References 44 and 45).

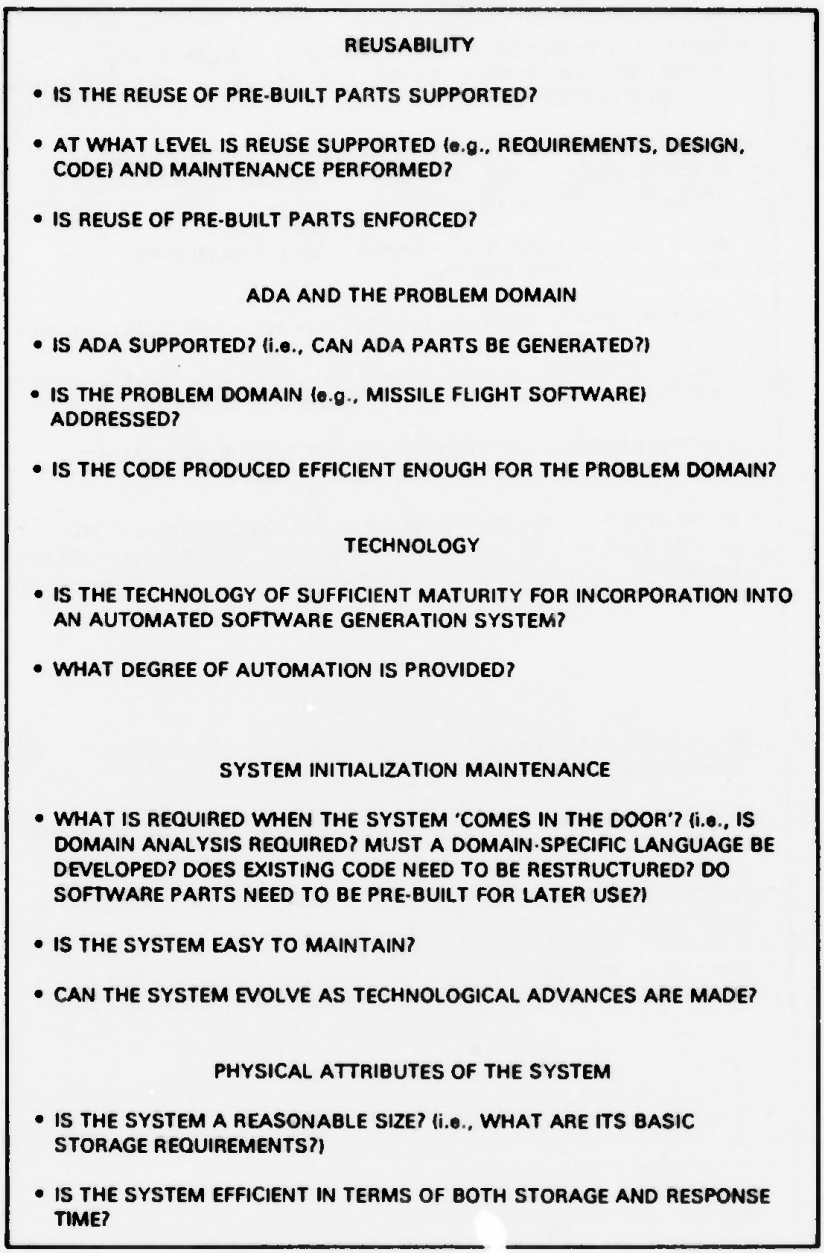


Figure 31. Issues/Criteria of a SGS

SPECIFICATION TECHNIDUE AND THE SPECIFICATIDN

- WHAT TYPE DF SPECIFICATIDN TECHNIDUE IS AVAILABLE? (e.g., FORMAL SPECIFICATION LANGUAGE? NATURAL LANGUAGE? PRDCEDURAL OR NDN-PROCEDURAL?)
- IS THE SPECIFICATION TECHNIOUE APPROPRIATE TD THE USER? ARE MULTIPLE SPECIFICATIDN TECHNIDUES PRDVIDED SD THAT THE MDST APPRDPRIATE ONE CAN BE USED?
- WHAT LEVEL DF EXPERTISE/TRAINING IS REQUIRED TO EFFECTIVELY INTERFACE WITH THE SYSTEM?
- IS THE INTERFACE TECHNIOUE APPRDPRIATE TO THE PROBLEM DDMAIN?
- CAN THE SPECIFICATION BE AUTOMATICALLY TRANSFORMED TO A FORM THAT IS COMPREHENSIBLE TO ALL PARTIES WHO NEED TO KNDW?
- CAN THE SPECIFICATIDN BE PUT IN A FORM THAT IS ANALYZABLE (e.g., FOR COMPLETENESS, CONSISTENCY, CLARITY)?
- IS THE SPECIFICATION MAINTAINABLE (IF THE SPECIFICATIDN IS TO FUNCTION AS A FORM OF DOCUMENTATION AND CONTROL, IT MUST BE MAINTAINED IN A CURRENT STATE THROUGHDUT THE SOFTWARE LIFE CYCLE)?

USER SUPPDRT

- IS THE USER ASSISTED WITH SPECIFICATIONS (i.e., IS PARTIAL SPECIFICATIDN SUPPORTED?)?
- DOES THE SYSTEM SUPPORT AN INCREMENTAL DR ITERATIVE APPROACH TO DEVELOPMENT?
- ARE THE SPECIFICATIONS CHECKED FOR COMPLETENESS, CONSISTENCY, CLARITY?
- CAN THE USER INTERFACE DIRECTLY WITH THE VARIDUS COMPONENTS OF THE SYSTEM (e.g., CAN HE DIRECTLY OUERY THE PARTS CATALOG?)?

SYSTEM OUTPUTS

- IS OPTIMIZED CDDE PRDDUCED?
- IS THE CDDE VERIFIABLY CDRRECT?
- ARE FACILITIES PROVIDED TO VERIFY CDRRECTNESS OF RESULTING MODULES (e.g., AUTOMATIC GENERATION OF TEST PROCEDURE, CORRECTNESS PRDOFS)
- ARE SUPPORTING DOCUMENTS (e.g., ADL, SYSTEM DDCUMENTATION) PRODUCED?

Figure 31. Issues/Criteria of a SGS (Concluded)

5. CONCLUSIONS

During the evaluation period only Beta versions of ART were available on the VAX (ART was released on the Symbolics in March, 1985), but a consistent increase in quality has been noted during that time. Although ART is not yet a mature product, and as such suffers from some of the drawbacks of systems that are newly developed, it provides a high degree of functionality for the application under consideration in the CAMP study. It is anticipated that there will be an improvement in efficiency and speed of the system which will facilitate system development and make ART an even more attractive choice for expert system development on VAX-based systems. Thus, ART's functionality coupled with its availability on VAX equipment and the interest of Inference personnel in improving the product, lead us to conclude that ART is a tool that should continue to be used in the CAMP project.

SECTION VI
SOFTWARE PARTS COMPOSITION SYSTEM CONCLUSIONS

This section discusses some of the conclusions reached during the software composition/generation study portion of the CAMP project. Volume I contains conclusions that relate to missile software commonality and the design of software parts.

The development of a universal software generator system is not practical in the foreseeable future. Although there are several research efforts underway to develop application independent systems which can generate software from requirements, these systems have several major drawbacks.

a. They are still in the research phase of development.
b. They are very complex to use.
c. The code they generate is not efficient enough for real-time embedded applications.

Few existing software generation systems are capable of handling software parts. Most of the work being done in the area of software generation assumes that a new software system will be generated from scratch. In those systems which do account for reusable components (e.g., Use.It), the parts are restricted to simple functional black boxes. No provision is made for complex parts (e.g. generic and schematic parts).

Formal specification languages have severe drawbacks as interface mechanisms to a software parts composition system. Although a formal specification language is a sound technical approach to specifying software requirements and design information, past experience has shown that this type interface mechanism is very poor in terms of comprehensibility. In effect, only experts can read and understand the data being described. Since the goal of a software parts composition system is to simplify the use of parts, we believe that this approach is not fruitful.

An automated software parts catalog is an essential component of any successful software reusability effort. Given that there is a significant number of parts, an automated tool will be needed to help manage the parts and to help the user of the parts analyze them for suitability for his

application. If parts are constructed from other parts (as they should be) the interrelationship of the set of parts can become quite complex. An automated tool can help the user manage this complexity and increase the productivity gained from using the parts.

A textual software parts catalog is an essential aspect of any successful software reusability effort. The existence of an automated software parts catalog does not preclude the need for a textual version of the catalog. There will be organizations which for some reason or another will not have access to the automated tool and will need information about the parts. Ideally, the automated software parts catalog will be able to generate the textual software parts catalog.

The early identification of software parts can be facilitated by an automated tool. A critical factor in the successful use of software parts will be the introduction of the parts into the software system early in the software development process. In other words, the knowledge that software parts will be used (and the knowledge of what parts will be used) will have an effect on the design of the software. In addition, the use of software parts might impact the requirements for the software. This case might arise when the existence of certain parts facilitates a certain algorithmic approach (e.g., if the missile guidance engineer is aware of the existence of a wealth of parts to perform an operation in a certain manner, he might select that approach to reduce costs). The parts identification process will map system (e.g., missile) requirements to existing software parts thereby allowing early identification of applicable software parts. This identification will also facilitate trade-off analyses, cost estimates, and sizing and timing analyses.

The technology exists for automating the construction of software components from design paradigms. The concept of software design parts has been discussed for a number of years within the software engineering community. In the past, most researchers have adopted a template view of this type of part. In other words, the design would be implemented by means of a template which the user would manually complete. Our experience on CAMP indicated that if the template is supplemented by a set of construction rules then the construction of the software component from user-specified requirements could be completely automated. We have termed these types of parts (template plus construction rules) schematic parts.

Expert systems have a high potential in the automation of the software parts engineering process. Expert systems are typically beneficial in areas which have eluded solution by classical programming techniques and which are currently being solved by human experts. This is most definitely the case in the use of schematic parts and in the identification of applicable software parts. During CAMP we demonstrated that schematic parts can be effectively and efficiently generated using an expert system. We also designed a system for parts identification using an expert system. Although the software parts catalog need not use an expert system (e.g., classical data base management systems can be used), the incorporation of all three functions into one tool facilitates information sharing.

APPENDIX A

DEFINITION OF THE CAMP PARTS CATALOG ATTRIBUTES

APPENDIX A

DEFINITION OF THE CAMP PARTS CATALOG ATTRIBUTES

This appendix provides a detailed explanation of each attribute of the Ada missile software parts catalog developed under the CAMP contract. For each attribute the following information is provided (as applicable):

(a) The name of the attribute.

(b) The data type of the attribute. The type of an attribute can be STRING (e.g., the value of 'Part Id' is a string), TEXT (e.g., the value of 'Abstract' is of type TEXT), ENUMERATION (e.g., the 'Level' attribute must have a value of 'simple', 'generic', or 'schematic'), or NUMERIC (e.g., the value for 'Source Size' must be the number of lines of code).

(c) The domain of an ENUMERATION type.

(d) The status of the attribute. This is either REQUIRED (i.e., all parts must be supplied a value for this attribute) or RECOMMENDED (i.e., the attribute is recommended for completeness but not required).

(e) Where useful, an example of an attribute value is shown.

(f) The description of the attribute's meaning.

In addition to the above information, attributes whose value is dependent upon the scope of the catalog are identified, and the differences in content are elaborated. Figure A-1 enumerates the catalog attributes.

PART ID	REVISION ID
VERSION	NAME
ABSTRACT	CATEGORY
TYPE	LEVEL
CLASS	INLINE
OPERATION	PARAMETER NAME
KEYWORDS	DATE CATALOGED
DEVELOPED BY	DEVELOPED FOR
DEVELOPMENT STATUS	VERIFICATION STATUS
CATALOG UNITS WITHED	WITHING UNITS
USAGE	LOCATION OF CODE
SECURITY CLASS (PART)	SECURITY CLASS (CATALOG ENTRY)
LINES OF CODE (SOURCE)	FIXED OBJECT CODE SIZE
REQUIREMENTS DOCUMENTATION	DESIGN DOCUMENTATION
HARDWARE DEPENDENCIES	OTHER RESTRICTIONS
ACCURACY	TIMING CHARACTERISTICS
REMARKS	

Figure A-1. Catalog Attributes

ATTRIBUTE NAME ..... Part Id
TYPE ..... String
STATUS ..... Required
EXAMPLE 1160
DESCRIPTION ..... The Part Id is a non-semantic code which together with the value of the Revision Id attribute uniquely identifies a catalog entry. The Part Id is not required to be unique (e.g., the same code would be used for all revisions of a given part). This type of code will facilitate catalog implementation by providing a way to identify software parts independently of their names (e.g., different developers may develop parts with the same name); by assigning a Part Id to each part, all of these parts can be kept in the same catalog. There are currently several coding schemes proposed or currently in use to identify software; these codes are used to identify the developer and the software product (Reference 15). We propose that the Part Id merely be a sequential identifying number assigned to the software part with other fields being used to convey descriptive information.

ATTRIBUTE NAME ..... Revision Id
TYPE ..... String
STATUS ..... Required
EXAMPLE A5
DESCRIPTION ..... The Revision Id is a non-semantic code used to uniquely identify revisions of a particular part. This code together with the Part Id form a unique key.

ATTRIBUTE NAME ..... Version
TYPE ..... String
STATUS ..... Required
EXAMPLE ..... Wander angle, North pointing
DESCRIPTION ..... This attribute contains a brief description used to differentiate between parts that have the same name.

ATTRIBUTE NAME ..... Name
TYPE ..... String
STATUS ..... Required
EXAMPLE ..... Missile Launch Platform
DESCRIPTION ..... This attribute provides a brief, but not necessarily unique, descriptive name of the part (e.g., a package may have more than one body, in which case both bodies would have the same name but they would be uniquely identifiable by the combined key consisting of Part Id and Revision Id).

ATTRIBUTE NAME ..... Abstract
TYPE ..... Text
STATUS ..... Required
DESCRIPTION ..... The abstract is a brief (300-500 words) explanation of the purpose and functioning of the part, and the reason for original development (including design rationalization). The Naval Research Laboratory's Software Cost Reduction Project has a separate entry for design issues. An alternative that we recommend is to include a brief reference to design issues in the abstract, and if it is thought that the user will require further information, he should be referred to an external design document. If the part is being revised, the originating component may be referenced for this information, but the abstract must contain the reason for revision. Information on reason for original development may provide insight into the appropriateness of a unit for a particular application, and thus facilitate reuse of parts; the DACS software catalog contains a separate entry for this. We think this too can be briefly stated in the abstract and if greater detail is required, the user should be referred to an external document. The level of detail in the abstract will depend upon the scope of the catalog; it is intended to provide the user with a quick overview of the unit. If the catalog has been incorporated into an automated system, the abstract can be scanned to pick up keywords or phrases when the system is performing a search for requested parts.

ATTRIBUTE NAME ..... Category
TYPE ..... Enumeration
DOMAIN ..... see Figure A-2
STATUS ..... Required
DESCRIPTION ..... This attribute specifies the taxonometric classification of the part.

ATTRIBUTE NAME ..... Type
TYPE ..... Enumeration
DOMAIN ..... (package, subprogram, task)
STATUS ..... Required
DESCRIPTION ..... The TYPE attribute specifies the Ada program unit type of the software part.

ATTRIBUTE NAME ..... Level
TYPE ..... Enumeration
DOMAIN ..... (simple, generic, schematic)
STATUS ..... Required
DESCRIPTION ..... The LEVEL specifies the abstraction level of the part. See Volume I, Section II for more details.

ATTRIBUTE NAME ..... Class
TYPE ..... Enumeration
DOMAIN ..... (specification, body)
STATUS ..... Required
DESCRIPTION ..... Ada specifications and their associated bodies have separate entries in the parts catalog; this attribute is used to identify that aspect of a part.

ATTRIBUTE NAME ..... Inline
TYPE ..... Enumeration
STATUS ..... Required
DOMAIN ..... (yes, no, N/A)
DESCRIPTION ..... This attribute specifies whether the part has been set up to be 'inlined' or not.

ATTRIBUTE NAME ..... Keywords
TYPE ..... Set of 0 or more Strings
STATUS ..... Recommended
DESCRIPTION ..... This attribute contains one or more keywords or phrases that can be used to locate a part. Keywords can be used to describe functionality of the part, or task area. The purpose of a keyword is to narrow the search for a desired component. If an automated catalog scheme is utilized, words that appear in the abstract need not be repeated here as they can be automatically extracted and added to the keyword list.

ATTRIBUTE NAME ..... Date Cataloged
TYPE ..... String
STATUS ..... Required
EXAMPLE 02-22-85
DESCRIPTION ..... This attribute provides the date that the original part or revision was cataloged. A standard format for the date should be established.

ATTRIBUTE NAME ..... Developer
TYPE ..... String
STATUS ..... Required
EXAMPLE ..... McDonnell Douglas Astronautics Co.
DESCRIPTION ..... The exact information contained in this entry is dependent upon the scope of the catalog. For instance, if the catalog is intra-company, knowledge of the actual individual(s) may be useful, whereas if the catalog is inter-company, knowledge of the organization may be sufficient. This entry should contain at least the name of the developing organization. Other information that might be useful includes the address of the developer and a phone number for a contact person. If the entry is for a revision, the modifier should be identified.

ATTRIBUTE NAME ..... Developed For
TYPE ..... String
STATUS ..... Recommended
EXAMPLE ..... Tomahawk (BGM-109AS) Flight Software
DESCRIPTION ..... This attribute should identify the project and type of software.

ATTRIBUTE NAME ..... Development Status
TYPE ..... Enumeration
DOMAIN ..... (in development, complete, verified)
STATUS ..... Required
DESCRIPTION ..... This attribute indicates the development status of the unit. The usefulness of such an entry is dependent upon the scope of the catalog. For instance, if the catalog is for all Air Force software projects, the usefulness of knowing the stage of development of a particular component diminishes greatly, whereas if the catalog is being used within a single project or for a particular contractor, such information may be of value. The DACS software catalog contains an entry for this, and ANSI X3.99-1981 recommends the inclusion of this attribute in program abstracts.

ATTRIBUTE NAME ..... Verification Status
TYPE ..... Enumeration
DOMAIN ..... (internal, external)
STATUS ..... Recommended
DESCRIPTION ..... Verification of the units increases user confidence and promotes reuse of existing parts. This is illustrated by the contrast in usage of parts supplied by a computer users group which are not validated, and those supplied by an organization which performs extensive testing, e.g., IMSL. The entries for algorithms presented in the Collected Algorithms of the CACM also provide information on verification; the name of the certifying individual or organization, the certification method, results, and remarks are supplied. The major issue surrounding verification and validation of parts, is who should perform this operation. User confidence is increased when an independent or external organization performs the verification, but

Verification Status (concluded)

the task of verifying all parts may become monolithic for a single organization. Our proposed solution is to provide information on whether the part was verified internally or by an external organization. This issue is discussed in greater detail in Section II, paragraph 6, Organizational Factors.

ATTRIBUTE NAME ..... Catalog Units Withed
TYPE ..... String
STATUS ..... Required
DESCRIPTION ..... This attribute contains an enumeration of other units within the catalog that this unit 'withes' (units identified by Part Id, Revision Id, Name, and Version.

ATTRIBUTE NAME ..... Withing Units
TYPE ..... String
STATUS ..... Required
DESCRIPTION ..... This attribute contains an enumeration of other units within the catalog that 'with' this unit.

ATTRIBUTE NAME ..... Usage
TYPE ..... String
STATUS ..... Recommended
DESCRIPTION ..... This attribute contains an enumeration of the projects and systems that use this particular part. This should also include the places where parts generated via schematics are used. The usage attribute aids in the tracking of which systems have 'checked a part out of the library'. Such an entry facilitates maintenance in the event that an error is found in a part.

ATTRIBUTE NAME ..... Location of Code/Constructor
TYPE ..... String
STATUS ..... Recommended
DESCRIPTION ..... This entry should specify the file name, library, and computer system where the part is located; the part 'level' determines whether it will be source code or a parts constructor.

ATTRIBUTE NAME ..... Security Classification of Part
TYPE ..... Enumeration
DOMAIN .....
(Unclassified, Confidential, Secret, Top_Secret)
STATUS ..... Required
DESCRIPTION ..... This attribute specifies the DOD security classification of the part.

ATTRIBUTE NAME ..... Security Classification of Catalog Entry
TYPE ..... Enumeration
DOMAIN .....
(Unclassified, Confidential, Secret, Top_Secret)
STATUS ..... Required
DESCRIPTION ..... This entry specifies the security classification of a part's catalog entry; this may be different from the security classification of the part itself.

ATTRIBUTE NAME ..... Operation
TYPE ..... String
STATUS ..... Recommended
DESCRIPTION ..... This attribute identifies the operations that are exported by the part.

ATTRIBUTE NAME ..... Parameter Name
TYPE ..... String
STATUS ..... Recommended
DESCRIPTION ..... This attribute identifies the parameters associated with each operation identified in the 'Operation' field. The parameters shall be identified as to whether they are 'in', 'out', or 'in/out' parameters.

ATTRIBUTE NAME ..... Source Code Size
TYPE ..... Numeric
STATUS ..... Recommended
DESCRIPTION ..... This attribute provides the size of the Ada part in terms of lines of source code (LOC). The definition of LOC must be provided when the catalog is established.

ATTRIBUTE NAME ..... Fixed Object Code Size
TYPE ..... Numeric
STATUS ..... Recommended
DESCRIPTION ..... This attribute provides the fixed (static) size of the Ada part in terms of bytes of object code.

ATTRIBUTE NAME ..... Hardware Dependencies
TYPE ..... Text
STATUS ..... Recommended
EXAMPLE ..... 1553B data bus
DESCRIPTION ..... This entry contains an elaboration of any hardware dependencies of the part which would limit its transportability.

ATTRIBUTE NAME ..... Requirements Documentation
TYPE ..... Text
STATUS ..... Recommended
DESCRIPTION ..... This attribute identifies the requirements documentation and indicates its availability.

ATTRIBUTE NAME ..... Design Documentation
TYPE ..... Text
STATUS ..... Recommended
DESCRIPTION ..... This attribute identifies the design documentation and indicates its availability.

ATTRIBUTE NAME ..... Restrictions
TYPE ..... Text
STATUS ..... Recommended
DESCRIPTION ..... This attribute indicates any usage restrictions such as proprietary rights and copyrights.

ATTRIBUTE NAME ..... Remarks
TYPE ..... Text
STATUS ..... Recommended
DESCRIPTION ..... This field is for any general remarks concerning the part, or for continuations of other fields.

ATTRIBUTE NAME ..... Accuracy
TYPE ..... Text
STATUS ..... Recommended
DESCRIPTION ..... This field contains information on the accuracy or precision of numerical results computed by the part. If this information is not relevant, it should be left blank.

ATTRIBUTE NAME ..... Timing
TYPE ..... Text
STATUS ..... Recommended
DESCRIPTION ..... This field contains information on execution time for sample invocations or instantiations of the part. The run-time conditions that produced the timing results must be specified in order to make this information relevant.

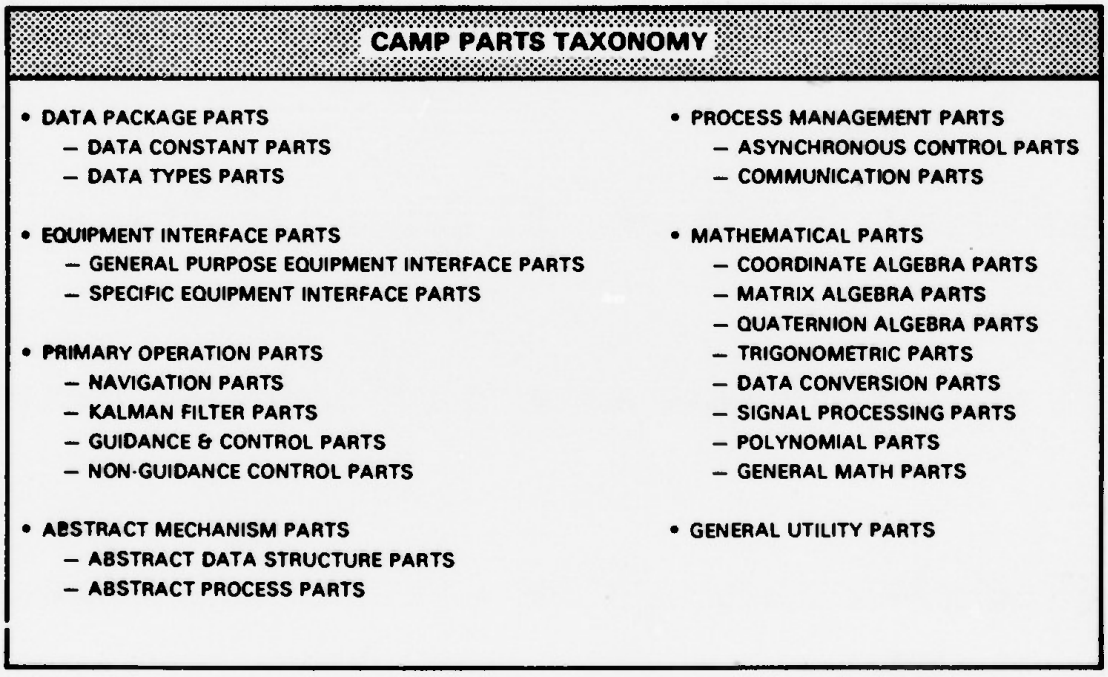


Figure A-2. CAMP Parts Taxonomy

(The reverse of this page is blank)

APPENDIX B

CAMP CATALOGING FORM

APPENDIX B
CAMP CATALOGING FORM

This appendix contains a form for use with the Ada parts catalog described in Appendix A. This form is shown on the next page.

ID ..... _____ Revision ... _____

Name ..... _____
Version .. _____
Type ..... __ Subprogram __ Package __ Task
Level .... __ Simple __ Generic __ Schematic
Class .... __ Specification __ Body
Inline ... __ Yes __ No __ N/A

Abstract . _____
_____

Category . _____
Keywords . _____

Operation	Parameter Name	In/Out
_____	_____	_____
_____	_____	_____

Development Status .... __ In Progress __ Completed
Verification Status ... __ None __ Internal __ External
Date Cataloged ..... _____
Developed By ..... _____
Developed For ..... _____

Requirements Documentation .... _____
Design Documentation ..... _____
Location of Code ..... _____

Code Size (loc) ..... _____ Object Size (bytes) ...
_____

Accuracy Characterization ..... _____
_____
Timing Characterizations ..... _____
_____

Hardware Dependencies ..... _____
_____
Other Restrictions ..... _____
_____

Withed Parts	Withing Parts
_____	_____
_____	_____

Remarks ..... _____
_____

Security Classification (of part) ..... _____
Security Classification (of catalog) ... _____

Figure B-1. The Cataloging Form

(The reverse of this page is blank)

APPENDIX C

SAMPLE DBMS IMPLEMENTATION OF THE CAMP PARTS CATALOG

APPENDIX C

SAMPLE DBMS IMPLEMENTATION OF THE CAMP PARTS CATALOG

1. Database Schema 106
2. Database Usage 108

As a proof-of-concept, MDAC-STL constructed a parts database using ORACLE™, a state-of-the-art relational database management system.

1. DATABASE SCHEMA

The parts database consists of 6 tables. They and their associated attributes are shown in Figure C-1. The purpose of each table is described in the following paragraphs.

The Parts Table contains the attributes unique to each cataloged part (i.e., there exists a one-to-one mapping between attribute values and entities). This is the primary table in the database, containing the majority of the items described in Appendix A. The Part ID and Version ID together form the key for this relation.

The Developer Table contains information about each engineer developing software. The Developer ID is the key for this relation. This information is separated from the Parts Table because an engineer can develop more than one part; data redundancy would result from including this information in the Parts Table. Parts and Developers are bound together by means of the Developer ID attribute in the Parts Table.

The Project Table contains a list of parts which are in use by one or more projects, and an indication of which projects are using which parts. There is a many-to-many relationship between parts and part users, thus, the project information is kept separate from the Parts table to avoid data redundancy.

TABLES	ATTRIBUTES	
PARTS	Id	Version
	Name	Abstract
	Category	Type
	Level	Class
	Date of Development	Developer
	Project	Software
	Development Status	Verification Status
	Security Class of Part	Security Class of Entry
	Source Size	Object Size
	Hardware Dependencies	Documentation
	Restrictions	Remarks
	Accuracy	Timing
DEVELOPER	Id	Name
	Department	MDC Component
PROJECT	Id	Software
	Part	Version
USAGE	Usage Mechanism	Used Part
	Used Part's Version	Using Part
	Using Part's Version	
KEYWORD	Word	Part
	Version	
NOISE	Word	

Figure C-1. Database Schema

The Usage Table tracks parts that either 'with' other parts in the catalog, or are generated from another part in the catalog. This table is separated from the Parts Table because there is a many-to-many relationship between 'withing' and 'withed' parts.

The Keyword Table contains a list of keywords found in the abstracts of cataloged parts. Entries in the Keyword Table can be generated in two ways:

(1) Explicit entry: A user can specify a keyword to be included in the table by specifying that word in the 'Keywords' section of the Missile Software Ada Parts Cataloging Form (see Appendix B).

(2) Automatic entry: A Keyword Table generation program will examine the abstract of each cataloged part and make an entry for each keyword found. A keyword is any word which is not found in the Noise Table. The Noise Table is a list of words which should not be included in the Keyword Table when found in an Abstract (e.g., "and", "the", "a", "not").

2. DATABASE USAGE

Two primary interfaces can be developed for database report generation. The first is a menu-driven mechanism for generating standard reports, based on ORACLE's Interactive Application Facility (IAF). The second interface is command driven and uses Structured Query Language (SQL) commands to access any information in the database. The following are some examples of the use of SQL to retrieve information from the Ada parts catalog.

EXAMPLE 1. List all parts which are currently in development:

```
SELECT ID, VERSION_ID, NAME
FROM PARTS_TABLE
WHERE DEVELOPMENT_STATUS = "IN DEVELOPMENT"
```

EXAMPLE 2. List all parts which 'with' part called BINARY_TREE:

```
SELECT USING_PART_ID, USING_VERSION_ID
FROM USAGE_TABLE
WHERE USED_PART_ID"USED_VERSION_ID =
      (SELECT ID"VERSION_ID
       FROM PARTS_TABLE
       WHERE NAME = "BINARY_TREE")
```

APPENDIX D

THE FINITE STATE MACHINE CONSTRUCTOR

APPENDIX D
THE FINITE STATE MACHINE CONSTRUCTOR

1. Introduction 110
2. FSM Constructor Requirements 112
3. FSM Constructor Top-Level Design 114
4. Implementation of the Proof-of-Concept FSM Constructor 117

1. INTRODUCTION

A finite state machine (FSM or finite automaton) is an abstraction that can be used to model software systems or portions of software systems that consist of a number of distinct states and stimuli or events that cause a change in or transition between those states. An FSM has one state that is designated as the initial or beginning state. This is the state at which processing begins. A terminal or end state is a state at which processing ends (i.e., there are no transitions out of that state). Transitions between the states of the FSM are caused by stimuli or events. A transition is dependent upon the current state at the time the event occurs (i.e., the same stimuli applied in two different states in the same FSM may result in two different transitions). Figure D-1 depicts a typical graphical representation of a finite state machine.

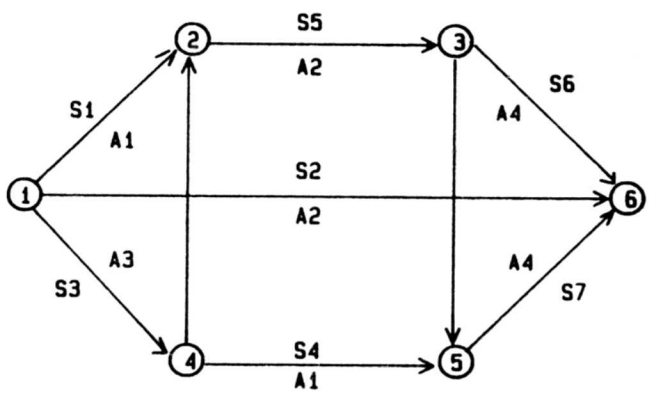


Figure D-1. A Finite State Machine

There are several variations of the basic finite state machine. For example, actions can be associated with the transitions between states or with the states themselves.

Finite state machines are a useful representation for a number of different types of software that arise in the missile flight domain (e.g., launch control software, signal processing).

As part of the CAMP study, MDAC developed the requirements specification and top-level design of a prototype software parts engineering expert system known as the Ada Missile Parts Engineering Expert (AMPEE) System (see References 44 & 45). This system incorporates a facility for component construction that provides the user with the ability to construct (i.e., tailor or instantiate) meta-parts (i.e., schematic or generic parts) found in the Ada missile software parts catalog. The Component Construction facility is intended to consist of a constructor for each schematic part and for some generic parts (constructors will only be developed for generic parts that are sufficiently complex). The Finite State Machine Constructor is one constructor that will form a part of the AMPEE System.

A number of reasons exist for developing a schematic part and part constructor for a finite state machine.

- Finite state machines occur frequently within the operational missile flight software domain.
- The part is very straight-forward to build, but certain variations cannot be captured via the Ada generic facility (e.g., actions associated with state transitions).
- Providing a schematic part relieves the software developer of the tedium of building a fairly simple piece of software, and provides an error-free implementation based on his specifications.

R.J.A. Buhr (Reference 39) summarized the need for a standardized implementation in the following way:

> " Finite state machines are ubiquitous in many types of embedded systems. Accordingly, their explicit, consistent, and uniform representation in the Ada program text seems desirable, both for verifiability and readability."

The implementation of the Finite State Machine Constructor served two purposes:

- It served as a proof-of-concept for providing a semi-automated means of generating missile flight software.
- It provided a means for evaluating ART, the expert system development tool discussed in Section V.

In the paragraphs that follow, the requirements, design, implementation, and efficiency considerations are discussed.

2. FSM CONSTRUCTOR REQUIREMENTS

The Finite State Machine Constructor forms a part of the Component Generation function of the Ada Missile Parts Engineering Expert System. It is a domain-independent schematic part that provides the user with an automated means of generating a finite state machine software component. An Ada package is created that contains a procedure to process incoming stimuli. A function is also provided that allows the current state to be determined.

The Finite State Machine Constructor requires the use of both the ART programming language and LISP.

a. Interface Requirements

The FSM Constructor is required to interface to the VAX file system in order to access the fixed portions of code used in the component construction process, and write the FSM component that is output from this constructor. File access is handled via LISP input/output facilities.

b. Functional Requirements

The functional requirements of the FSM Constructor are discussed in the following paragraphs.

(1) Inputs

The user supplied inputs are enumerated below.

- File Name: The name of the file where the component is to be written; must be a valid file name.
- Process Name: An Ada identifier that will identify the package to be constructed.
- Initial State: The initial state of the finite state machine.
- States: The valid states within the finite state machine.
- Transitions: The transitions associated with each state.
- Stimuli: The stimuli that result in the transitions associated with each state.
- Actions: The actions (if any) that are associated with the transitions between states; this is in the form of a package name and the procedure within the package that performs the requested action.

All states, stimuli, and actions provided by the user must be valid Ada identifiers.

System supplied inputs are as follow.

- Ada Missile Parts Catalog: This is used to determine the location of the fixed portions of code that are used in the construction of a component for the user.
- FSM Construction Rules: These are the rules that guide the construction of the component.

(2) Processing

The FSM Constructor prompts the user to enter the required inputs. These inputs are edited for conformance to format and other constraints. If the input data passes all consistency and format checks, an Ada component is constructed and written to the file specified by the user.

(3) Outputs

The FSM constructor outputs the Ada code that implements the FSM specified by the user. This output is directed to the file specified by the user.

c. Quality Factors

There are several areas that must be addressed when considering the quality requirements for the FSM Constructor. Correctness of the Ada code produced is of the utmost concern in the development of part constructors within the AMPEE System. As was pointed out in the main portion of this report, a few encounters with bad parts could destroy much of the reusability effort. This constructor must not only produce correct Ada code, it must do so consistently. Although it is important that the FSM Constructor operate efficiently, it is of greater importance that the code produced by the FSM constructor be efficient.

Usability, flexibility, and maintainability are other quality concerns that must be addressed. The Finite State Machine Constructor is designed to be easy to use; the user will be prompted for the required inputs and will be provided with the appropriate format. Flexibility and maintainability are two concerns of the entire AMPEE System.

3. FSM CONSTRUCTOR TOP-LEVEL DESIGN

This paragraph discusses the top-level design of the Finite State Machine Constructor. A top-level view of the architecture is presented along with the functional and data flow for this portion of the AMPEE System. Global and local data are also discussed.

a. Architecture

Figure D-2 depicts the top-level architecture for this portion of the system. As can be seen in the diagram, the AMPEE System Executive (see Reference 45), which is invoked when a user logs into the system, invokes the Component Construction subfunction. At this point the user can invoke any of

the available component constructors. Control is transferred to the Finite State Machine Constructor when the user requests this part and it is verified that a constructor exists.

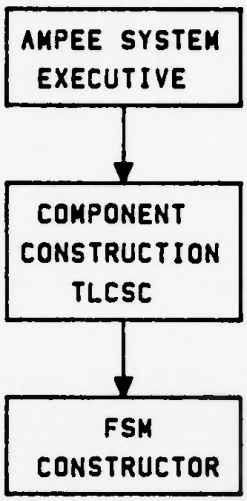


Figure D-2. Architecture

b. Functional Control and Data Flow

Figure D-3 depicts the functional control flow and Figure D-4 depicts the data flow for this constructor.

c. Global Data

The following global data is utilized by the Finite State Machine Constructor:

- Ada Missile Parts Catalog: This is used to obtain the location of fixed portions of code used in the construction of the component.

- User Id: This is used to tag the set of requirements provided by the user.

d. Local Data

The Finite State Machine Constructor makes use of the following local data:

-- FSM-User-Requirements: This is a schema that is used to capture and store the user's requirements for a specific instance of the finite state machine part. These requirements are tagged with the user's id and are time-stamped in order to facilitate their retrieval at a later time (e.g., to perform Component Regeneration - see References 44 and 45).

-- Other local data includes intermediate data structures used in constructing the software component and local facts used to control the firing of rules.

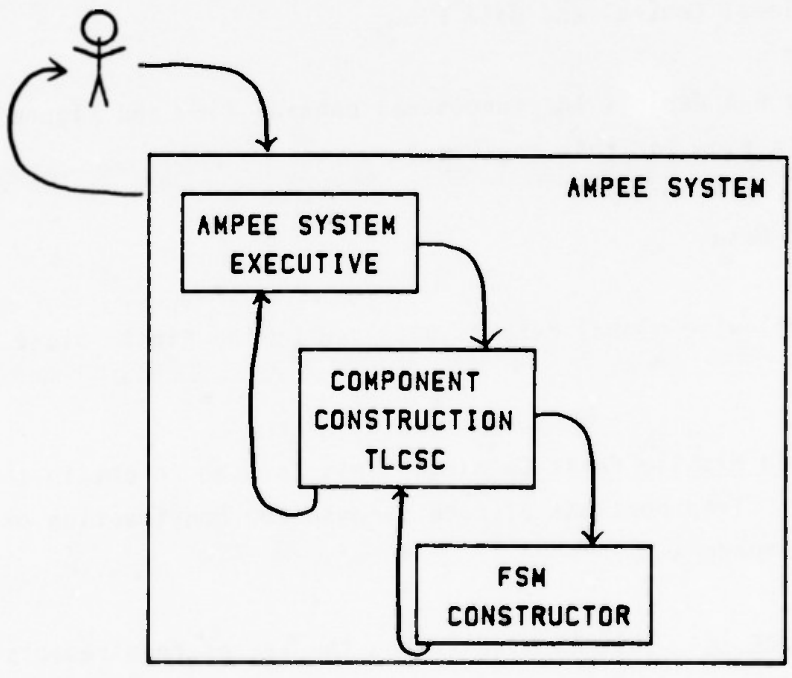


Figure D-3. Control Flow

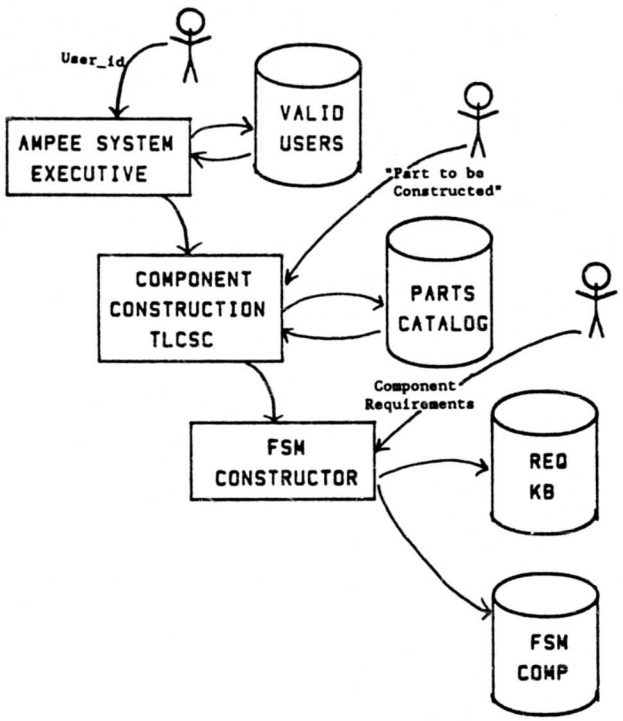


Figure D-4. Data Flow

4. IMPLEMENTATION OF THE PROOF-OF-CONCEPT FSM CONSTRUCTOR

The purpose of this part constructor is to provide a standard design for finite state machines. It was desired to build a part that is flexible (e.g., actions can be associated with the state transitions if the user desires), efficient (e.g., dead code will not be introduced), and simple to use (e.g., the user doesn't have to learn a high-order specification language in order to use this part constructor successfully). A proof-of-concept implementation of this part constructor was undertaken to prove the feasibility of both the approach (i.e., the use of an expert system) and the tool (i.e., ART).

a. The Implementation

The implementation consists of both ART and LISP files. The individual components are discussed below; Figure D-5 provides an overview of the implementation.

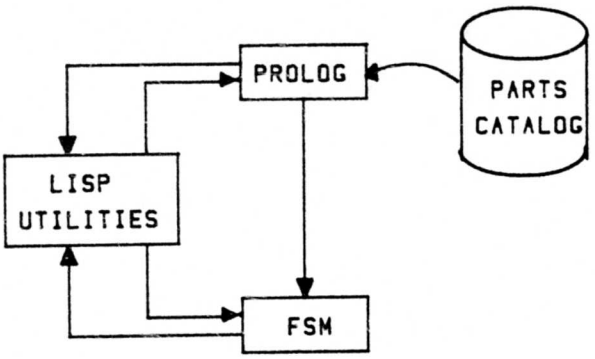


Figure D-5. Overview of Proof-of-Concept Implementation

-- PROLOG.ART: This is an ART application file that encompasses a bare-bones version of the AMPEE system executive. This component assumes initial control when the system is brought up. It performs the following functions:

(1) Verifies the validity of the user requesting AMPEE System services
(2) Solicits identity of the part to be constructed
(3) Verifies the existence of the part in the Ada missile parts catalog
(4) Obtains the fixed code locations from the catalog entry
(5) Obtains the name of the file where the component is to be written

-- CATALOG.ART: This component contains the Ada missile parts catalog. For this implementation, it contains only the basic catalog schema and a schema for the Finite State Machine schematic part. No processing is performed by this component.

-- FSM.ART: This is the ART file that contains the actual FSM Constructor; it performs the following functions:

(1) Solicits user input to construct an instantiation of the finite state machine part
(2) Creates a schema to store the component requirements specified by the user
(3) Performs consistency checks and constructs intermediate data structures (through the invocation of LISP routines)
(4) Generates the FSM component specified by the user

-- LFSM.LSP: This is a LISP file that contains utilities that construct various intermediate data structures used in the construction of the finite state machine component, perform error checking of the data provided by the user, and write out the majority of the Ada component.

b. Expert Features

The Finite State Machine Constructor incorporates a number of features (i.e., the smarts or optimizations) that contribute to the efficiency and flexibility of the Ada code that is produced. These features contribute to the generation of Ada code that is as good as that produced by an expert Ada programmer.

-- The user is provided with the expected format of the input data. This type of assistance makes the part usable by a wide range of personnel.

-- Redundant state transitions are eliminated from the input (i.e., If the beginning state and ending states are the same for two sets of transitions, then the stimuli will be combined and only one state transition will appear). This is one means of preventing the introduction of redundant code in the component.

-- Checks are made for non-determinism in the state transitions (e.g., If for two sets of state transitions, the beginning states are the same and the stimuli are the same, but the ending states are different, then a data error is signaled).

-- A decision to use a case statement or an if-then else statement is based on the number of alternatives. Some work is currently in progress to determine at what point a case statement becomes more efficient than an if-then-else statement. As yet, no results have been obtained, but the decision point is easily changed.

-- A check is made to determine if a stimuli does not result in a transfer out of the present state. If a transfer does not occur, then a re-assignment of the current state will not be made. This prevents the introduction of extraneous code into the constructed component.

c. User-System Interaction

The following sections provide a sample of an interactive session with the system, and the output from that session (i.e., the generated component). The compilation listing of the generated Ada component is provided in paragraph (3).

The scenario for this interactive session is as follows.

-- The user, desiring to build a finite state machine representation for missile firing, elects to use the AMPEE System Finite State Machine Constructor to facilitate the implementation.

-- The user logs in and is presented with a series of menus for facility selection.

-- After selecting the Finite State Machine Constructor, the user is prompted to provide state-transition information, i.e., the beginning state, the stimuli that cause transitions from that state, the ending state, and any actions associated with the transition.

-- When the user has entered all the state-transition information associated with the FSM that he is building, he enters 'quit'. The Ada component is then generated if all data checks are passed.

(1) A Sample User Session

```
Dribbling to USERDISK3:[PALM03]FTR.LSP;1


Enter User Id: u207215

Ada Missile Parts Engineering Expert System

     1)  Parts Catalog
     2)  Parts Identification
     3)  Component Construction

  Please enter choice:
     # 3


Component Construction

     1)  Component Generation
     2)  Component Regeneration

  Please enter choice:
     # 1


Component Constructors

     1)  Finite State Machine

  Please enter choice:
     # 1



Part ID: a001

Revision ID: 0

Enter file name (pathname) for component: "miasile.ada"

Enter Component Name: miasile

Enter Initial State: s_0

Enter states and tranaitions as prompted below.
Events are to be entered in the following format:
   (event_1 event_2 ... event_n)

Actions are to be in the following format:
   (<action_package> <action_procedure>)
If no actions are asaociated with the transition enter NIL

States are to be entered as symbols, e.g., state_1


Beginning State: s_0

Events: (intent_to_launch_cmd_recd)

Ending State: s_1
```

```
Action: (ap launch_countdown_seq)

Beginning State: s_1

Events: (test_failure_1)

Ending State: s_n

Action: (ap shutdown_abort_seq)

Beginning State: s_1

Events: (all_tests_passed_1)

Ending State: s_2

Action: (ap first_motion_seq)

Beginning State: s_2

Events: (test_failure_2_
)

Error - Invalid Ada identifier entered as an event.
Events: (test_failure_2)

Ending State: s_n

Action: (ap shutdown_abort_seq)

Beginning State: s_2

Events: (passed_launcher_clear_test)

Ending State: s_3

Action: (ap fin_deploy_seq)

Beginning State: s_3

Events: (all_test_passes_2)

Ending State: s_4

Action: (ap wings_out_early)

Beginning State: s_4

Events: (thrust_decay_detected)

Ending State: s_5

Action: (ap system_off_seq)
```

```
Beginning State: quit

Data passed nd check
Data passed check for unreachable states.
Constructing component MISSILE
No applicable rules.
Ending run.
NIL
 ART1.0 Lisp> (dribble)
```

(2) The Generated Ada Component

```
with AP;
package MISSILE is
  type States is (S_2, S_3, S_0, S_5, S_4, S_1, S_N);
  type Stimuli is (TEST_FAILURE_2, INTENT_TO_LAUNCH_CMD_RECD, PASSED_LAUNCHER_CLEAR_TEST, THRUST_DECAY
_DETECTED,
      ALL_TESTS_PASSED_1, TEST_FAILURE_1, ALL_TEST_PASSES_2);
  function Current_State return States;
  procedure Signal (Event : in Stimuli);
  Invalid_Stimuli : EXCEPTION;
end MISSILE;

package body MISSILE is
  Present_State : States := S_0;
  Event : Stimuli;
  function Current_State return states is
  begin
    return Present_State;
  end Current_State;


  procedure Signal (Event : in Stimuli) is
  begin

    case Present_State is
      when S_0      =>
        if (event = INTENT_TO_LAUNCH_CMD_RECD) then
          AP.LAUNCH_COUNTDOWN_SEQ;
          Present_State := S_1;
        else
          raise Invalid_Stimuli;
        end if;
      when S_2      =>
        if (event = PASSED_LAUNCHER_CLEAR_TEST) then
          AP.FIN_DEPLOY_SEQ;
          Present_State := S_3;
        elsif (event = TEST_FAILURE_2) then
          AP.SHUTDOWN_ABORT_SEQ;
          Present_State := S_N;
        else
          raise Invalid_Stimuli;
        end if;
      when S_4      =>
        if (event = THRUST_DECAY_DETECTED) then
          AP.SYSTEM_OFF_SEQ;
          Present_State := S_5;
        else
          raise Invalid_Stimuli;
        end if;
      when S_1      =>
        if (event = ALL_TESTS_PASSED_1) then
          AP.FIRST_MOTION_SEQ;
          Present_State := S_2;
        elsif (event = TEST_FAILURE_1) then
          AP.SHUTDOWN_ABORT_SEQ;
          Present_State := S_N;
        else
          rsise Invalid_Stimuli;
        end if;
      when S_3      =>
        if (event = ALL_TEST_PASSES_2) then
          AP.WINGS_OUT_EARLY;
```

```
            Present_State := S_4;
          else
            raise Invalid_Stimuli;
          end if;

        when others => raise invalid_stimuli;
      end case;
      end Signal;

end MISSILE;
```

(3) The Compiled Ada Component

```
MISSILE                                                  10-Sep-1985 08:57:44     VAX Ada V1.
                     Page   1
D1                                                       10-Sep-1985 08:43:51     USERDISK3:[
M03]MISSILE.ADA;1          (1)

     1   with AP;
     2   package MISSILE is
     3     type States is (S_2, S_3, S_0, S_5, S_4, S_I, S_N);
     4     type Stimuli is (TEST_FAILURE_2, INTENT_TO_LAUNCH_CMD_RECD, PASSED_LAUNCHER_CLEAR_TEST, T.
ST_DECAY_DETECTED,
     5         ALL_TESTS_PASSED_1, TEST_FAILURE_1, ALL_TEST_PASSES_2);
     6     function Current_State return States;
     7     procedure Signal (Event : in Stimuli);
     8     Invalid_Stimuli : EXCEPTION;
     9   end MISSILE;

PSECT MAP

Psect Hex Size     Dec Size     Name
    0 00000006            6    MISSILE_.$CODE
    1 00000010           16    MISSILE_.$CONSTANT

%ADAC-I-CL_ADDED, Package specification MISSILE added to library



LIBRARY SUMMARY

USERDISK3:[PALM03.CAMP.LIB]

        Unit name                  Nodes   Percent   Blocks          Unit kind
                                   read              read
AP                                     4     40          4     Package specification
```

```
10
11  package body MISSILE is
12    Present_State : States := S_0;
13    Event : Stimuli;
14    function Current_State return states is
15    begin
16      return Present_State;
17    end Current_State;
18
19
20    procedure Signal (Event : in Stimuli) is
21    begin
22
23      case Present_State is
24        when S_0      =>
25          if (event = INTENT_TO_LAUNCH_CMD_RECD) then
26            AP.LAUNCH_COUNTDOWN_SEQ;
27            Present_State := S_1;
28          else
29            raise Invalid_Stimuli;
30          end if;
31        when S_2      =>
32          if (event = PASSED_LAUNCHER_CLEAR_TEST) then
33            AP.FIN_DEPLOY_SEQ;
34            Present_State := S_3;
35          elsif (event = TEST_FAILURE_2) then
36            AP.SHUTDOWN_ABORT_SEQ;
37            Present_State := S_N;
38          else
39            raise Invalid_Stimuli;
40          end if;
41        when S_4      =>
42          if (event = THRUST_DECAY_DETECTED) then
43            AP.SYSTEM_OFF_SEQ;
44            Present_State := S_5;
45          else
46            raise Invalid_Stimuli;
47          end if;
48        when S_1      =>
49          if (event = ALL_TESTS_PASSED_1) then
50            AP.FIRST_MOTION_SEQ;
51            Present_State := S_2;
52          elsif (event = TEST_FAILURE_1) then
53            AP.SHUTDOWN_ABORT_SEQ;
54            Present_State := S_N;
55          else
56            raise Invalid_Stimuli;
57          end if;
58        when S_3      =>
59          if (event = ALL_TEST_PASSES_2) then
60            AP.WINGS_OUT_EARLY;
61            Present_State := S_4;
62          else
63            raise Invalid_Stimuli;
64          end if;
65
66
```

```
   67          when others => raise invalid_stimuli;
   68        end case;
   69        end Signal;
   70
   71    end MISSILE;
```

```
PSECT MAP

Psect Hex Size     Dec Size     Name
    0 00000154          340    MISSILE.$CODE
    1 00000004            4    MISSILE.$DATA


%ADAC-I-CL_ADDED, Package body MISSILE added to library
    Corresponds to package specification MISSILE compiled 10-Sep-1985 08:57



LIBRARY SUMMARY

USERDISK3:[PALM03.CAMP.LIB]

        Unit name                  Nodes   Percent  Blocks          Unit kind
                                   read             read
MISSILE                              27     100                Package specification
AP                                   10     100       10       Package specification
```

COMMAND QUALIFIERS

```
ADA/LIS MISSILE.ADA
```

QUALIFIERS USED

```
/CHECK/COPY_SOURCE/DEBUG=ALL/ERROR_LIMIT=30/LIST/NOMACHINE_CODE
/NODIAGNOSTICS/LIBRARY=ADA$LIB
/NOTE_SOURCE/OPTIMIZE=TIME/NOSHOW/NOSYNTAX_ONLY
/WARNINGS=(NOCOMPILATION_NOTES,STATUS=LIST,SUPPLEMENTAL=ALL,WARNINGS=ALL,WEAK_WARNINGS=ALL)
```

COMPILER INTERNAL TIMING

Phase	CPU seconds	Elapsed seconds	Page faults	I/O count
Initialization	0.16	1.44	368	24
Parser	0.15	0.18	167	1
Static semantics	0.16	0.89	183	6
IL generation	0.25	1.47	242	21
Segment tree	0.13	1.11	130	21
Annotate tree	0.02	0.02	11	0
Flow analysis	0.02	0.02	7	0
Linearize tree	0.07	0.31	86	0
Code generation	0.37	1.06	337	0
Optimizer	0.11	0.37	120	0
Data allocation	0.01	0.01	1	0
Generate code list	0.08	0.27	98	0
Register allocation	0.01	0.01	8	0
Peephole optimization	0.04	0.17	29	0
Write object module	0.05	0.07	35	0
DST generation	0.05	0.06	17	0
Listing generation	0.07	0.43	28	7
Compilation library	0.33	2.21	202	69
Compiler totals	1.52	7.78	1547	129

COMPILATION STATISTICS

```
Weak warnings:              0
Warnings:                   0
Errors:                     0

Peak working set:        2418
Virtual pages used:      4797
Virtual pages free:     65203
CPU Time:               00:00:01.52    (2802 Lines/Minute)
Elapsed Time:           00:00:07.78
Compilation Complete
```

d. Source Code

The following sections provide the source code for the components that comprise the proof-of-concept implementation of the FSM Constructor.

(1) CATALOG.ART

This component contains the template for entries in the Ada missile software parts catalog. It is also intended to contain all of the catalog entries, however, for the proof-of-concept implementation, only one partial catalog entry is provided.

```
;#+VAX (in-peckege #L'art-user)

(def-viewpoint-levels)

; ********************************************************************************
;                          CATALOG SCHEMA
; ********************************************************************************


;-------------------------------------------------------------------------
; Template for catalog entries
;-------------------------------------------------------------------------
(defschema part-entry
   (pert-id)
   (revision-id)
   (name)
   (version)
   (security-classificetion-part)
   (security-classification-entry)
   (type)
   (level)
   (cleas)
   (inline)
   (cetegory)
   (keywords)
   (fixed-code-location)
   (operation
      (slot-how-meny multiple-velues))
   (verification-atetus)
   (dete-ceteloged)
   (developed-by)
   (developed-for)
   (requirements-documentetion)
   (design-documentetion)
   (aize-source)
   (aize-object)
   (eccurecy)
   (timing-charecterizetions)
   (herdwere-dependencies)
   (other-restrictions)
   (catalog-units-withed)
   (withing-cetelog-units)
   (abstrect)
   (remarks)
   (timestemp-last-revision))



;-------------------------------------------------------------------------
; Catalog entries for software parts
;-------------------------------------------------------------------------
(defschema z99
   (instance-of part-entry)
   (part-id e001)
   (revision-id 0)
   (type subprogram)
   (fixed-code-locetion "fsml.ada"))
```

(2) PROLOG.ART

This component contains ART code that handles 'front-end' processing that is required regardless of the facility selected by the user. It is from here that control is transferred to the selected facilities.

```
;#+VAX (in-package #L'art-user)

(def-viewpoint-levels)

; ***************************************************************************
;                        PROLOG for AMPEE System
; ***************************************************************************


;---------------------------------------------------------------------
; Relations
;---------------------------------------------------------------------
(defrelation build-response-link
   (?part-id ?revision-id ?response-scbema))

(defrelation state
   (?level-1 ?level-2))

(defrelation user-id
   (?uid))

(defrelation user-verified)

(defrelation part-available
   (?PI ?RI))



;---------------------------------------------------------------------
; Facts to link parts to schemata that will store user inputs
;---------------------------------------------------------------------
(deffacts initial-links
   (build-response-link a001 0 fsm-user-responses))



;---------------------------------------------------------------------
; Valid users
;---------------------------------------------------------------------
(defschema valid-user
   (has-instances  u200538  u201965  u203093  u207215)
   (access-privileges
      (slot-datatype sequence)))



;---------------------------------------------------------------------
; Schema to keep track of which part is to be built
;---------------------------------------------------------------------
(defschema part-to-build
   (part-id)
   (revision-id)
   (to-build-from)  ;sequence of files from which fixed portions of code
                    ;in component being built will be copied
   (to-build-at))
```

```
;---------------------------------------------------------------------
; Cleers the screen end prompts user to log in
;---------------------------------------------------------------------
(defrule initialize-system
=>
   (cell-out erese-page 1 1)
   (assert (user-id =(prompt-end-reed #L':expression
                        "Enter User Id: "))))



;---------------------------------------------------------------------
; Verifies that thet user code used es log-in is velid.
;---------------------------------------------------------------------
(defrule verify-user
   (user-id ?uid)

   (schema ?uid (instence-of velid-user))
=>
   (essert (user-verified)))



;---------------------------------------------------------------------
; This will eventually be replaced by e complete menu; it currently
; goes directly to the 'Component Construction' function.
;---------------------------------------------------------------------
(defrule which-function
   ?x <- (user-verified)
=>
   (retrect ?x)
   (cell-out erese-pege 1 1)
   (mein-menu))



;---------------------------------------------------------------------
; Prompts user for the identity of the pert thet he wents to construct.
; At a leter dete it is enticipeted thet the ueer will have been
; provided with e list of perts & will select one rether than being
; prompted in this fashion.
;  Two rules ere used to get the pert id end revision id beceuse with
;  Bete 3, the compiletion resulted in the order of the prompts being
;  chenged.
;---------------------------------------------------------------------
(defrule which-pert-1
   (stete component-construction component-generetion)
=>
   (terpri)
   (essert
      (scheme =(gensym)
         (instence-of pert-to-build)
         (pert-id =(prompt-end-reed #L':expression "Pert ID: "))))
```

```
(defrule which-part-2
   (schema ?x (instance-of part-to-build) (part-id ?PI))
=>
   (terpri)
   (assert
      (schema ?x
         (revision-id =(prompt-and-read #L':expression "Revision ID: "))))))



;-------------------------------------------------------------------
; Verifies that the part identified does indeed exist in the parts
; catalog.
;  If the part does not exist, the user should be taken back to the
;  prompt for part id/revision id or be allowed to exit.
;-------------------------------------------------------------------
(defrule verify-existance
   (declare (salience 100))

   (schema ?x
      (instance-of part-to-build)
      (part-id ?PI)
      (revision-id ?RI))

   (case
      ((schema ?y
         (instance-of part-entry)
         (part-id ?PI)
         (revision-id ?RI))
       =>
       (assert (part-available ?PI ?RI)))

       (otherwise
         =>
       (printout t t "Specified part does not exist in the parts catalog.")
       (retract ?x))))



(defrule get-fixed-code-and-place-to-build
  ?x <- (part-available ?PI ?RI)

    (schema ?Y
       (instance-of part-to-build)
       (part-id ?PI)
       (revision-id ?RI))

    (case
       ((schema ?
          (instance-of part-entry)
          (part-id ?PI)
          (revision-id ?RI)
          (fixed-code-location ?file-names))
       =>
       (assert (schema ?Y (to-build-from ?file-names)))))
```

```
=>
   (retract ?x)
   (assert (schema ?Y (to-build-at =(prompt-and-read #L':expression
                        "Enter file name (pathname) for component: "))))
```

(3) FSM.ART

This ART component is the central portion of the proof-of-concept implementation. It contains the code necessary to carry on a dialog with the user to elicit requirements and construct the required Ada implementation of a finite state machine.

```
;#+VAX (in-peckage #L'art-user)

;*****************************************************************************
;                  Finite Stete Mechine Pert Cnnstructor                      *
;                                                                             *
;  This pert cnnstructor solicts informetinn from the user cnncerning the     *
;  specific finite stete mechine that is tn be built.  The user's input is    *
;  stored in e 'response' scheme.  In the prototype implementetinn, the ueer  *
;  requirements knowledge bese will be updeted by this 'respnnse' schema in   *
;  order tn fecilitate cnmponent regeneretion.  Once ell input is received    *
;  frnm the user, the fsm cnmponent is cnnstructed.                           *
;*****************************************************************************


;-----------------------------------------------------------------------------
; Reletinns:  Fects esserted intn the knnwledge bese ere treated as relations.
;   Undeclered reletions generete warnings at compile time, thus they are
;   declered here.
;-----------------------------------------------------------------------------
(defreletinn creete-stetes-slot
   (?time))

(defrelation init-st-input
   (?time))

(defrelation prompt
   (?PI ?RI ?TIME ?SCH))

(defreletinn get-beginning-time)

(defreletion initiel-sequence)

(defrelation nd-check
   (?PI ?RI ?TIME))

(defreletion get-component-name
   (?PI ?RI ?TIME))

(defrelation get-initiel-state
   (?PI ?RI ?TIME))

(defrelation get-beginning-stete)

(defreletion beginning-stete
   (?BS))

(defrelation get-events)

(defreletinn events
   (?EVENTS))

(defrelation get-ending-state)

(defrelation ending-stete
   (?ES))

(defreletion get-actions)

(defreletion ection
   (?ACTION))
```

```
(defrelation error
   (?ERROR-TYPE))

(defrelation check-for-unreachable)

(defrelation extract-sea

   (?PI ?RI ?TIME))

(defrelation build-part
   (?PI ?RI ?TIME))

(defrelation action-packages
   (?ap))

(defrelation beginning-states
  (?all-baginning-states))
   .



;---------------------------------------------------------------------------
; This schema template is used to store user input for the construction of a
; particular fem component. The information provided is stored for future use
; (e.g., if the component constructed is not as the user wanted, he can make
; changes to his input without having to re-enter all of it.
;---------------------------------------------------------------------------
(defschema fem-user-responses
   (part-id)
   (revision-id)
   (user-id)
   (timestamp)
   (file-name)            ;name of file where component will be written
   (component-name)       ;name of component to be built
   (initial-state)
   (states)
   (state-transitions))



;---------------------------------------------------------------------------
; Establishes a schema for the user's requirements, and asserts a fact to
; initiate solicitation of those requirements.  It is initiated only after
; verification that a part constructor exists for the part requested by the
; user.
;---------------------------------------------------------------------------
(defrule solicit-input-fem
   (schema ?
      (instance-of part-to-build)
      (part-id ?pi)
      (revision-id ?ri)
      (to-build-at ?fo))

   (build-response-link ?pi ?ri fem-user-responses)

   (user-id ?UID)
=>
   (bind ?time (get-universal-time))
```

```
   (essert (schema =(gensym)
              (instence-of fsm-user-responses)
              (pert-id ?PI)
              (revision-id ?RI)
              (user-id ?UID)
              (timestemp ?time)
              (file-name ?fn)))

   (essert (get-component-neme ?PI ?RI ?TIME)))



(defrule get-component-name
   ?x <- (get-component-neme ?PI ?RI ?TIME)

   (scheme ?sch
      (instence-of fsm-user-responses)
      (pert-id ?PI)
      (revision-id ?RI)
      (timestemp ?TIME))
=>
   (retrect ?x)
   (terpri)
   (bind ?component-neme (prompt-end-reed #L':expression
      "Enter Component Name: "))
   (if (symbolp ?component-name) then
     (if (velid-ede-identifier ?component-name) then
        (essert
           (schema ?sch (component-name ?component-name))
           (get-initiel-stete ?PI ?RI ?TIME))
      else
        (printout t t "Invelid Ade identifier entered for component name.")
        (essert (get-component-name ?PI ?RI ?TIME)))
    else
      (printout t t "Component name must be e symbol.")
      (essert (get-component-neme ?PI ?RI ?TIME))))



(defrule get-initiel-stete
   ?x <- (get-initiel-stete ?PI ?RI ?TIME)

   (schema ?sch
      (instence-of fsm-user-responses)
      (pert-id ?PI)
      (revision-id ?RI)
      (timestemp ?TIME))
=>
   (retrect ?x)
   (terpri)
   (bind ?initiel-stete (prompt-end-reed #L':expression
      "Enter Initiel Stete: "))
   (if (symbolp ?initiel-stete) then
      (if (velid-ede-identifier ?initiel-stete) then
         (bind ?seq-stete (seq$ (list ?initiel-stete)))
         (essert
            (scheme ?sch (initiel-stete ?initial-stete)
```

```
                        (states ?seq-state))
              (init-st-input ?TIME))
         else
           (printout t t "Initial state must be a valid Ada identifier.")
           (assert (get-initial-state ?PI ?RI ?TIME)))
      else
        (printout t t "Initial state must be a symbol.")
        (assert (get-initial-state ?PI ?RI ?TIME))))



;---------------------------------------------------------------------------
; Initiates State-Transition input from the user.
;---------------------------------------------------------------------------
(defrule initiate-state-trans-input
   ?x <- (init-st-input ?time)

   (schema ?
      (instance-of part-to-build)
      (part-id ?PI)
      (revision-id ?RI))

   (schema ?sch
      (instance-of fsm-user-responses)
      (part-id ?PI)
      (revision-id ?RI)
      (timestamp ?time))
=>
   (retract ?x)
   (printout t t "Enter states and transitions as prompted below.")
   (printout t t "Events are to be entered in the following format:")
   (printout t t "   (event_1 event_2 ... event_n)")
   (terpri)
   (printout t t "Actions are to be in the following format:")
   (printout t t "   (<action_package> <action_procedure>)")
   (printout t t "If no actions are associated with the transition enter NIL")
   (terpri)
   (printout t t "States are to be entered as symbols, e.g., state_1")
   (terpri)
   (assert
      (prompt ?PI ?RI ?TIME ?SCH)
      (get-beginning-state)
      (initial-sequence)))



;---------------------------------------------------------------------------
; Gets the beginning state for a state-transition
; Data and error checking is performed on the input supplied by the user
;---------------------------------------------------------------------------
(defrule get-beginning-state
   ?z <- (prompt ?PI ?RI ?TIME ?SCH)
   ?x <- (get-beginning-state)
=>
   (retract ?x)
   (terpri)
   (printout t t "Beginning State: ")
```

```
   (bind ?BS (read))
   (if (equalp ?BS #L'quit) then
      (retract ?z)
      (assert (nd-check ?PI ?R1 ?TIME))
    else
      (if (and (symbolp ?BS) (not (null ?BS))) then
         (if (valid-Ada-identifier ?BS) then
            (assert (beginning-state ?BS)
                    (get-events))
          else
            (printout t t "Invalid Ada identifier entered for Beginning State")
            (assert (get-beginning-state)))
       else
          (printout t t "Beginning State must be a symbol - i.e., not a list")
          (assert (get-beginning-state)))))



;-------------------------------------------------------------------------------
; Gets the stimuli associated with a state-transition
;-------------------------------------------------------------------------------
(defrule get-events
   (declare (salience 100))
   ?x <- (get-events)
   (beginning-state ?)
=>
   (retract ?x)
   (printout t t  "Events: ")
   (bind ?EVENTS (read))
   (if #L(lisp:listp ?EVENTS) then
      (if (valid-list-of-Ada-identifiers ?EVENTS) then
         (bind ?seq-events (seq*$ ?EVENTS))
         (assert (events ?seq-events)
                 (get-ending-state))
       else
         (printout t t "Error - Invalid Ada identifier entered as an event.")
         (assert (get-events)))
    else
      (printout t t "Events must be entered as a list of symbols (e.g., (a b c))")
      (assert (get-events))))



;-------------------------------------------------------------------------------
; Obtains the ending state for a state-transition
;-------------------------------------------------------------------------------
(defrule get-ending-state
   (declare (salience 100))
   ?x <- (get-ending-state)
   (beginning-state ?BS)
   (prompt ?PI ?R1 ?TIME ?SCH)

   (schema ?SCH
      (instance-of fsm-user-responses)
      (initial-state ?IS))
=>
   (retract ?x)
   (printout t t "Ending State: ")
```

```
   (bind ?ES (read))
   (if (end (symbolp ?ES) (not (null ?ES))) theo
      (if (valid-Ada-identifier ?ES) then
         (assert (ending-stete ?ES)
                 (get-actioos))
       else
        (printout t t "Error - Invalid Ade ideotifier entered.")
        (assert (get-ending-state)))
    else
      (printout t t "Eoding etate must be e symbol.")
      (assert (get-ending-stete))))



;-------------------------------------------------------------------------------
; Obtains the actions associated with the trsnsition
; The user must specify the Ada peckage that contains a routine to perform the
; desired ection, aod the oame of the routine.  The specificetion ie in the
; form of e LISP list,
;   e.g., (prepere_for_leuoch ignite_eogines)
;-------------------------------------------------------------------------------
(defrule get-actions
   (declere (selieoce 100))
   ?z <- (get-actions)
   (ending-state ?)
=>
   (retrect ?z)
   (printout t t "Action: ")
   (bind ?ACTION (read))
   (if #L(lisp:listp ?ACTION) theo
      (if (valid-list-of-Ada-identifiers ?ACTION) theo
         (bind ?seq-action (seq*$ ?ection))
         (assert (ection ?seq-ection))
       else
         (priotout t t "Invelid Ada ideotifier entered for a compooent of ACTION.")
         (essert (get-actions)))
    else
      (printout t t "Actions must be entered es a list io the following form: ")
      (printout t t "   (<action_peckege> <ection_routine>) or NIL")
      (essert (get-actioos))))



;-------------------------------------------------------------------------------
; Updates the state-trsnsition information with the latest state-treositioo
; thet has been provided by tha user.
;-------------------------------------------------------------------------------
(defrule update-state-trensitions
   (prompt ?PI ?RI ?TIME ?sch)
   ?a <- (beginning-state ?BS)
   ?b <- (events ?EVENTS)
   ?c <- (ending-state ?ES)
   ?d <- (action ?ACTION)

   (=> (biod ?list-events (list*$ ?EVENTS))
       (bind ?list-action (liet*$ ?ACTION)))
```

```
   (case
      (?x <- (initial-sequence)
         =>
      (retract ?x)
      (bind ?SEQST (seq*$ (list (list ?BS ?Events ?ES ?ACTION))))
      (assert (schema ?sch (state-transitions ?SEQST))))

      ((schema ?sch (state-transitions ?R))
         =>
      (bind ?ST (list ?BS ?list-events ?ES ?list-action))
      (bind ?listr (list*$ ?R))
      (bind ?seqst (seq*$ (redundancy-elimination ?st ?listr)))
      (modify (schema ?sch (state-transitions ?seqst)))))
=>
   (retract ?a ?b ?c ?d)
   (assert (get-beginning-state)))
```

```
;-------------------------------------------------------------------------
; Verifies that the same stimuli applied to the same state does not result in
; 2 different transitions.
;-------------------------------------------------------------------------
(defrule check-for-nondeterminism
   ?x <- (nd-check ?PI ?RI ?TIME)

   (schema ?
      (instance-of fsm-user-responses)
      (part-id ?PI)
      (revision-id ?RI)
      (timestamp ?TIME)
      (state-transitions ?ST))
=>
   (retract ?x)
   (if (signal-nondeterminism-error (list*$ ?ST)) then
      (printout t t "***    Invalid State-Transition Data Entered    ***")
      (printout t t "*** Unable to continue processing for this part ***")
      (assert (error input-data))
    else (printout t t "Data passed nd check")
         (assert (extract-sea ?PI ?RI ?TIME))))
```

```
;-------------------------------------------------------------------------
; Extracts states, events, and actions from an embedded sequence and forms a
; sequence for each of the three.  This data is used when generating the Ada
; code for the component under construction.  It is simpler to prepare the
; information ahead of time.
;-------------------------------------------------------------------------
(defrule extract-states-events
   ?x <- (extract-sea ?PI ?RI ?TIME)

   (schema ?sch
      (instance-of fsm-user-responses)
      (part-id ?PI)
```

```
      (revision-id ?RI)
      (timestamp ?TIME)
      (state-transitions ?R)
      (states ?states))
=>
   (assert (check-for-unreachable ?PI ?RI ?TIME))
   (retract ?x)
   (bind ?listr (list*$ ?R))
   (bind ?seqstates (seq$ (make-states ?listr (list$ ?states))))
   (bind ?seqevents (seq$ (make-events ?listr nil)))
   (bind ?seqactions (seq$ (make-actions ?listr nil)))
   (bind ?seqbstates (seq$ (make-bstates ?listr nil)))
   (assert (action-package ?seqactions))
   (assert (beginning-states ?seqbstates))
   (assert (events ?seqevents))
   (modify
      (schema ?scb
         (states ?seqstates))))



;---------------------------------------------------------------------------
; Checks state-transitions for unreachable states
;---------------------------------------------------------------------------
(defrule check-unreachable-state
   ?x <- (check-for-unreachable ?PI ?RI ?TIME)
   (beginning-states ?BSTATES)

   (scheme ?
      (instance-of fsm-user-response)
      (part-id ?PI)
      (revision-id ?RI)
      (timestamp ?TIME)
      (initial-state ?IS)
      (state-transitions ?R))
=>
   (retract ?x)
   (bind ?list-bstates (list*$ ?bstates))
   (bind ?listr        (list*$ ?R))
   (bind ?no-unreachables (signal-unreachable-state ?IS ?list-bstates ?listr))
   (if ?no-unreachables then
      (printout t t "Data passed check for unreachable states.")
      (assert (build-part ?PI ?RI ?TIME))
    else
      (printout t t "***  Invalid State-Transition Data Entered   ***")
      (printout t t "Unreachable state detected in fsm: " ?no-unreachables)
      (assert (error input-data))))



;---------------------------------------------------------------------------
; Generates the FSM component specified by the user
;---------------------------------------------------------------------------
(defrule build-fsm
   (not (error ?))
   ?a <- (build-part ?PI ?RI ?TIME)
```

```
   ?x <- (action-packages ?actions)
   ?y <- (events ?events)
   ?z <- (beginning-states ?bstates)

   (schema ?sch
      (instance-of fsm-user-responses)
      (part-id ?PI)
      (revision-id ?RI)
      (timestamp ?TIME)
      (initial-state ?IS)
      (file-name ?file-name)
      (component-name ?CN)
      (states ?STATES)
      (state-transitions ?ST))

   (schema ?b
      (instance-of part-to-build)
      (part-id ?PI)
      (revision-id ?RI)
      (to-build-from ?from))
=>
   (printout t t "Constructing component " ?CN)

   (bind ?output #L(open ?file-name :direction :output
                                    :if-exists :new-version
                                    :if-does-not-exist :create))

   (write-fsm-header ?output (list$ ?actions) ?cn (list$ ?states) (list$ ?events) ?is)

   (bind ?input #L(open ?from :direction :input))
   (read-loop ?input ?output)

   (write-selection-code ?output (list$ ?bstates) (list*$ ?ST))

   (princ "    end Signal;" ?output)
   (terpri ?output)
   (terpri ?output)

   (princ "end " ?output)
   (princ ?cn    ?output)
   (princ ";"    ?output)
   (terpri ?output)

   (close ?output)

   (retract ?x ?y ?z ?s ?b))
```

(4) LFSM.LSP

This component contains the LISP utilities. Many of these routines are quite general, and can be used by other functions within the AMPEE system.

```
;-----------------------------------------------------------------------
; External routine to clear the screen - DEC extension to Common Lisp
;-----------------------------------------------------------------------
(define-external-routine
   (erase-page :image-name "scrshr"
               :entry-point "lib$erase_page"
               :check-status-return t)
   (line :lisp-type integer
         :vax-type :word)
   (col :lisp-type integer
        :vax-type :word))



;-----------------------------------------------------------------------
; Name: read-loop
; Alphe & beta must be bound to stream names
; Processing:  This routine reads from streem 'alphe' and writes to stream
;    'beta'.  The input stream is closed after end-of-file is reached, but it
;    is left to the calling routine to close the output streem.
;-----------------------------------------------------------------------
(defun read-loop (alpha beta)
   (cond ((write-line (read-line alpha nil) beta) (read-loop alpha beta))
         (t (close alpha))))



;-----------------------------------------------------------------------
; Name: write-event-or-list
; Writes disjunction of event elements in 'liste' to 'output-stream'
; Example: liste := (a b c)
;    .       output: (event a) or (event b) or (event c)
;-----------------------------------------------------------------------
(defun write-event-or-list (output-streem liste line-lengtb)
   (cond ((> (length liste) 0)
            (setq line-lengtb (+ line-length
                                 (+ 14 (length (string (car liste))))))
            (cond ((> line-length 119)
                     (terpri         output-stream)
                     (princ "      " output-stream)
                     (setq line-length 6))
                  (T nil)))

         (T nil))

   (cond ((> (length liste) 1) (princ "(event = " output-stream)
                               (princ (car liste) output-streem)
                               (princ ") or " output-stream)
                               (write-event-or-list output-stream
                                                    (cdr liste)
                                                    line-lengtb))
         (t (princ "(event = " output-stream)
            (princ (car liste) output-stream)
            (princ ")" output-stream))))



;-----------------------------------------------------------------------
```

```
; Name: write-alteration-list
; Writes disjunction of elements in 'liste' to 'output-stream' with
; the alteration symbol instead of 'or'
; Example: liste := (a b c)
;          output: a | b | c
;-------------------------------------------------------------------
(defun write-alteration-list (output-stream liste line-length)
   (cond ((> (length liste) 0)
          (setq line-length (+ line-length
                               (+ 3 (length (string (car liste))))))
          (cond ((> line-length 119)
                 (terpri          output-stream)
                 (princ "       " output-stream)
                 (setq line-length 6))
                (T nil)))

         (T nil))

   (cond ((> (length liste) 1) (princ (car liste) output-stream)
                               (princ " | " output-stream)
                               (write-alteration-list output-stream
                                                     (cdr liste)
                                                     line-length))
         (t (princ (car liste) output-stream))))



;-------------------------------------------------------------------
; Name: write-comma-list
; Writes 'comma list' of elements in LISTE to OUTPUT-STREAM
; Example: LISTE := (a b c)
;          OUTPUT: a, b, c
;-------------------------------------------------------------------
(defun write-comma-list (output-stream liste line-length)
;  first decide if output should be written on current line or if a
;  linefeed is needed.
   (cond ((> (length liste) 0)
          (setq line-length (+ line-length
                               (+ 2 (length (string (car liste))))))
          (cond ((> line-length 119)
                 (terpri          output-stream)
                 (princ "       " output-stream)
                 (setq line-length 6))
                (T nil)))
         (T nil))

   (cond ((> (length liste) 1) (princ (car liste) output-stream)
                               (princ ", " output-stream)
                               (write-comma-list output-stream
                                                 (cdr liste)
                                                 line-length))
         (t (princ (car liste) output-stream))))



;-------------------------------------------------------------------
; Name: make-states
; Extracts the states from the embedded lists of state-transitions (i.e., it
```

```
;   forms a list of all of the states found in the input data.  The list is
;   used to declare an enumeration data type in the Ada component being
;   generated.
; SEQ is in the form: ((BS (events) ES (action)) ...)
; STATES is in the form: (S1 S2 ... Sn)
;-------------------------------------------------------------------
(defun make-states (seq states)
   (cond ((> (length seq) 0)
              (setq states (union
                           (remove-duplicates (list (caar seq) (caddar seq)))
                           states))
              (make-states (cdr seq) states))
         (t states)))



;-------------------------------------------------------------------
; Name: make-bstates
; Extracts only the beginning states from the embedded lists of
; state-transitions; SEQ is of the same form as above
;-------------------------------------------------------------------
(defun make-bstates (seq bstates)
   (cond ((> (length seq) 0)
            (setq bstates (union (list (caar seq)) bstates))
            (make-bstates (cdr seq) bstates))
         (T bstates)))



;-------------------------------------------------------------------------
; Name: make-events
; Extracts all of the events from the embedded lists of state-transitions
;   (i.e., it forms a list of all events found in the input data provided by
;   the user).
; SEQ is of the same form as above
; EVENTS is in the form: (E1 E2 ... En)
;-------------------------------------------------------------------------
(defun make-events (seq events)
   (cond ((null events) (setq events (cadar seq))
                        (make-events (cdr seq) events))
         ((> (length seq) 0) (setq events (union (cadar seq) events))
                             (make-events (cdr seq) events))
         (t events)))



;-------------------------------------------------------------------------
; Name: make-actions
; Extracts all of the action packages from the embedded lists of state-transi-
;   tions; if the action is NIL, it is not added to list of actions.  The
;   actions are WITHed into the Ada component that is to be generated.
; SEQ is the same form as above.
; ACTIONS is of the form: (A1 A2 ... An)
;-------------------------------------------------------------------------
(defun make-actions (seq actions)
   (cond ((and (null (car (last (car seq)))) (> (length seq) 0))
            (make-actions (cdr seq) actions))
```

```
        ((oull ections) (setq actions (list (ceer (lest (car seq)))))
                        (maks-ections (cdr seq) actions))
        ((> (length seq) 0) (setq ections (unioo (list (ceer (lest (car eeq)))) attions))
                            (maks-ections (cdr eeq) ections))
        (T ections)))



;-----------------------------------------------------------------------
; Name: write-fsm-heeder
; Writes the ioitiel portion of Ade code for the fsm pert
; OUTPUT-STREAM:  Name of output stream
; WITH-ACTION:  List of peckeges to be WITHed ioto the Ade compooeot thet is
;   uoder coostruction.  It is of the form (A1 A2 ... An)
; CN:  The name of the compoosot under coostructioo.  This must be a valid Ade
;   identifier
; STATES:  List of stetes used in decleretion of eoumeretion dete type
;   representiog ell possible stetes.
; EVENTS:  List of events used io the decleretioo of enumeretion data type
;   represeoting ell possible events.
; IS:  Ioitiel stete
;-----------------------------------------------------------------------
(defuo write-fsm-heeder (output-stream with-ectioos co stetes eveote ie)
  (cond ((> (leogth WITH-ACTIONS) 1)
         (prioc "with " output-streem)
         (write-comme-list OUTPUT-STREAM WITH-ACTIONS 5)
         (princ ";" output-streem))
        ((eod (= (leogth WITH-ACTIONS) 1)
              (oot (equelp (cer WITH-ACTIONS) oil)))
         (prioc "with " OUTPUT-STREAM)
         (prioc (cer WITH-ACTIONS) OUTPUT-STREAM)
         (prioc ";" output-stream))
        (T oil))

  (terpri    output-etream)

  (prioc "peckege " OUTPUT-STREAM)
  (prioc CN         OUTPUT-STREAM)
  (prioc " is"      OUTPUT-STREAM)
  (terpri OUTPUT-STREAM)

  (prioc "  type Stetes is (" OUTPUT-STREAM)
  (write-comme-list OUTPUT-STREAM STATES 18)
  (prioc ");" OUTPUT-STREAM)
  (terpri OUTPUT-STREAM)

  (prioc "  type Stimuli is (" OUTPUT-STREAM)
  (write-comme-list OUTPUT-STREAM EVENTS 19)
  (prioc ");" OUTPUT-STREAM)
  (terpri OUTPUT-STREAM)

  (prioc "  function Curreot_Stete returo Stetes;" OUTPUT-STREAM)
  (terpri OUTPUT-STREAM)
  (princ "  procedure Sigoel (Eveot : io Stimuli);" OUTPUT-STREAM)
  (terpri OUTPUT-STREAM)

  (prioc "  lovelid_Stimuli : EXCEPTION;" OUTPUT-STREAM)
  (terpri OUTPUT-STREAM)
```

```
   (princ "end " OUTPUT-STREAM)
   (princ CN     OUTPUT-STREAM)
   (princ ";"    OUTPUT-STREAM)
   (terpri OUTPUT-STREAM)
   (terpri OUTPUT-STREAM)

   (princ "package body " OUTPUT-STREAM)
   (princ CN              OUTPUT-STREAM)
   (princ " is "          OUTPUT-STREAM)
   (terpri OUTPUT-STREAM)

   (princ "  Present_State : States := " OUTPUT-STREAM)
   (princ IS  OUTPUT-STREAM)
   (princ ";" OUTPUT-STREAM)
   (terpri    OUTPUT-STREAM))


   --

;-----------------------------------------------------------------
; Name: write-selection-code
; Determines whether the fsm component should be written with a
; 'case' statement or with an 'if-then-else';  the criteria is the
; number of states that were specified (i.e., the length of STATES).
;-----------------------------------------------------------------
(defun write-selection-code (output-stream states state-transitions)
   (cond ((> (length states) 2)
          (write-state-case output-stream states state-transitions))

         ((and (<= (length states) 2) (> (length states) 0))
            (write-state-if-elsif output-stream states state-transitions))

         (T 'ERROR)))



;-----------------------------------------------------------------------
; Name: write-state-case
; Processes the use of a 'case' statement for the major state selection
; criteria (based on total number of states)
; OUTPUT-STREAM:  the name of the output stream
; STATES:  A list of all 'beginning' states
; STATE-TRANSITIONS:  A list of all state-transitions entered by the user
;-----------------------------------------------------------------------
(defun write-state-case (output-stream states state-transitions)
   (princ "    case Present_State is " output-stream)
   (terpri output-stream)

   (loop
      (setq state (car states))
      (princ "      when " output-stream)
      (princ state     output-stream)
      (princ "     =>"     output-stream)
      (terpri output-stream)

      (setq state-i-transitions
         (make-state-transitions state '() state-transitions))
```

```
      (process-transitions output-stream state-i-transitions)

      (setq states (cdr states))

      (cond ((= (length states) 0) (return))
            (T nil)))

      (terpri output-stream)
      (terpri output-stream)

      (princ "      when others => raise invalid_stimuli;" output-stream)
      (terpri output-stream)
      (princ "    end case;" output-stream)
      (terpri output-stream))



;-----------------------------------------------------------------------
; Name: write-state-if-elsif
; Processes the use of an 'if-then-else' statement for the 'state'
; selection criteria (based on the total number of states)
;-----------------------------------------------------------------------
(defun write-state-if-elsif (output-stream states state-transitions)
   (princ "    if " output-stream)
;;; LOOP
   (loop
      (setq state (car states))
      (princ "present_state = " output-stream)
      (princ state output-stream)
      (princ " then" output-stream)
      (terpri output-stream)

      (setq state-i-transitions
         (make-state-transitions state '() state-transitions))

      (process-transitions output-stream state-i-transitions)

      (setq states (cdr states))

      (cond ((= (length states) 0) (return))
            (T (terpri output-stream)
               (princ "    elsif " output-stream))))
;;; END LOOP

   (princ "    else " output-stream)
   (terpri            output-stream)
   (princ "  raise Invalid_Stimuli;" output-stream)

   (terpri output-stream)
   (princ "    end if;" output-stream)
   (terpri output-stream))



;-----------------------------------------------------------------------
; Name: make-state-transitions
```

```
; Extracts one state-transition set from the entire set of state-transitions.
;   For a given state, this routine makes a list whose car is that state and
;   whose cdr is a list of stimuli, transitions, and associated actions that
;   originate at 'STATE'.
; STATE:  A single initial state; all tranisitions that begin with this state
;   will be found
; ONE-STATE:  The set of transitions associated with a particular state; it is
;   in the following form:
;   (STATE (((event_list1) ES1 (actions1)) ((event_list2) ES2 (actions2)) ...)
;   When passed in, this variable should be a null list; it is then initialized
;   to a list containing the value of 'STATE'.
; STATE-TRANSITIONS:  In the form shown below:
;  ((s0 (event_list0) es0 a0)) (s1 (event_list1) es1 a1) ...)
;-------------------------------------------------------------------
(defun make-state-transitions (state one-state state-transitions)
   (cond ((and (> (length state-transitions) 0)
               (equal (caar state-transitions) state))
          (setq one-state (append one-state (list (cdar state-transitions))))
          (make-state-transitions state one-state (cdr state-transitions)))

         ((> (length state-transitions) 0)
            (make-state-transitions state one-state (cdr state-transitions)))

         (T (append (list state) one-state))))



;-------------------------------------------------------------------
; Name: process-transitions
;-------------------------------------------------------------------
(defun process-transitions (output-stream state-i-transitions)
   (cond ((> (length (cdr state-i-transitions)) 2)
          (write-event-case output-stream state-i-transitions))

         ((> (length (cdr state-i-transitions)) 0)
             (write-event-if-elsif output-stream state-i-transitions))

         (T 'ERROR)))



;-------------------------------------------------------------------------
; Name: write-event-case
; Processes the use of a 'case' statement for the events; i.e., if there are
;   multiple conditions that cause transition, the total number will determine
;   whether they will be handled with a 'case' statement or an 'if-then-else'
; OUTPUT-STREAM:  The name of the output-stream
; STATE-I-TRANSITIONS:  The set of transitions associated with a particular
;  state; this is obtained via the 'MAKE-STATE-TRANSITIONS' function
; Internal varisbles:
;  STATE:  A beginning state (for a state transition)
;  TRANSITIONS:  The set of all transitions (stimuli, anding state, and
;    associated actions) associsted with 'STATE'
;  CASE-1:  One particular transition associated with 'STATE' (it consists of
;    the following: ((event_list) ending_state actions)
;-------------------------------------------------------------------------
```

```
(dafun writa-evant-case (nutput-streem steta-i-trensitinns)
   (princ "        case Event is " nutput-streem)
   (terpri nutput-streem)
   (satq stata (car stata-i-trensitinns))

   (setq trensitinns (cdr stete-i-transitinns))

   (lnnp
      (satq cese-i (cer transitinns))
      (princ "      " nutput-straam)
      (writa-case-standerd nutput-straam (car cesa-i))

      (fsm-action-updata nutput-stream stete cese-i)

      (setq transitions (cdr transitinns))
      (cnnd ((= (length transitinns) 0) (raturn))
            (T nil)))

   .. (princ "        when nthars => reise Invelid_Stimuli;" nutput-stream)
      (terpri nutput-straam)
      (princ "        end casa;" nutput-stream)
      (tarpri nutput-stream)).
```

```
;-----------------------------------------------------------------------
; Name: write-cese-stenderd
;-----------------------------------------------------------------------
(dafun writa-casa-standard (output-straem case-of)
   (princ "  when " output-streem)
   (cond ((> (langth cesa-nf) 1)
            (writa-alteration-list output-stream cese-of 7))

         (T (princ (cer cese-nf) nutput-streem)))

   (princ "  =>" nutput-stream)
   (terpri nutput-atream))
```

```
;-----------------------------------------------------------------------
; Nama: writa-avant-if-elsif
; Prncessea the usa nf en 'if-then-else' stetemant fnr handling
; multipla avents thet cause tha same trensition
;-----------------------------------------------------------------------
(defun write-evant-if-elsif (nutput-stream stete-i-trensitinns)
   (setq steta (car stete-i-trensitinns))
   (setq trensitinns (cdr stete-i-transitinns))

   (princ "        if " nutput-streem)

   (lnnp
      (setq cese-i (cer transitions))
      (cnnd ((> (length (car casa-i)) 1)
               (write-avant-nr-list nutput-stresm (cer cese-i) 11))
            (T (princ "(event = " nutput-streem)
               (princ (caar case-i) nutput-stresm)
               (princ ")" nutput-stream)))
```

```
      (princ " then" nutput-stream)
      (terpri nutput-stream)

      (fsm-actinn-update nutput-stream state case-i)

      (setq transitinns (cdr transitinns))
      (cond ((= (length transitions) 0)
             (princ "        else" nutput-stream)
             (terpri nutput-stream)
             (princ "          raise Invalid_Stimuli;" nutput-stream)
             (terpri nutput-stream)
             (return))
            (T (princ "        elsif " nutput-stream))))

   (princ "        end if;" output-stream)
   (terpri output-stream))



;-------------------------------------------------------------------------
; Name: fsm-actinn-update
; Processing:  Writes the Ada code to (1) call the routinss that perform the
;  actions associated with the transition, and (2) update the current state.
;  Nnte that the current state is nnt updated if the stimuli does not result in
;  an actual change in state.
;-------------------------------------------------------------------------
(defun fsm-actinn-update (output-stream state case-i)
   (setq action-i (caddr case-i))
   (cond ((not (null actinn-i))
             (princ "          " output-stream)
             (princ (string-append (car action-i) "." (cadr action-i)) output-stream)
             (princ ";" output-stream)
             (terpri output-stream))
         (T nil))

   (cond ((equalp state (cadr case-i)) nil)
         (T (princ "          Present_State := " output-stream)
            (princ (cadr case-i)        output-stream)
            (princ ";"                  nutput-stream)
            (terpri output-stream)))

   (cond ((and (null actinn-i) (equalp state (cadr case-i)))
             (princ "          NULL;" output-stream)
             (terpri                  output-stream))
         (T nil)))




;-------------------------------------------------------------------------
; Name: signal-nondeterminism-error
; Processing: This routine determines if there exists two sets of transitions
;   such that the beginning state for both are the same, the ending state for
;   each transition is different, but they have at least one stimuli in common;
;   i.e., it looks for situatinns where the same stimuli at a given state
;   results in transitinns to 2 different states.
; ST is the collectinn nf state-transitions input by the user
```

```
;-----------------------------------------------------------------
(defun signal-nondeterminism-error (ST)
   (setq error nil)
   (setq state (caar st))
   (setq ending-state (caddar st))
   (setq event-seq (cadar st))
   (setq action (car (cddar st)))
   (setq reuse-st (cdr st))
   (setq new-st '())

;LOOP
   (loop
      (cond
;        ------------------------------------------------------------------
;        if the beginning states and ending states are the same
;          than proceed by getting the next state-transition (don't need to
;            examine the stimuli for this condition)
;        ------------------------------------------------------------------
         ((and (> (length reuse-st) 0)
                  (equal state (caar reuse-st))
                  (equal ending-state (caddar reuse-st)))
            (cond
               ((list-equal action (car (cdddar reuse-st)))
                  (setq reuse-st (cdr reuse-st)))
               (T (setq error T)
                  (return))))

;        ------------------------------------------------------------------
;        if the beginning states are the same, but the ending states are
;          different, and the two transitions have stimuli in common
;            then signal an error
;        ------------------------------------------------------------------
         ((and (> (length reuse-st) 0)
               (equal state (caar reuse-st))
               (not (equal ending-state (caddar reuse-st)))
               (not (null (intersection avent-seq (cadar reuse-st)))))
            (setq error T)
            (return))

;        ------------------------------------------------------------------
;        if there are no more state-transitions to examine
;          than exit the loop
;        ------------------------------------------------------------------
         ((= (length reuse-st) 0)
            (return))

;        ------------------------------------------------------------------
;        otherwise
;          add the state transition just looked at to new-st
;          proceed with the examination of the next state-transition in the
;            list reuse-st
;        ------------------------------------------------------------------
         (T (setq new-st (append (list (car reuse-st)) new-st))
            (setq reuse-st (cdr reuse-st)))))
;END LOOP

   (cond
      ((equal error T) error)

      ((> (length st) 1)
```

```
        (setq st new-st)
        (signal-nondeterminism-error st))

      (T NIL)))



;-------------------------------------------------------------------------
; Name: redundancy-elimination
; Processing: As the user enters a state-transition, this routine eliminates
;   the following redundancy:
;     if (bs0 = bs1) and (es0 = es1) and (actions0 = actions1)
;     then event-seq = (event-seq0 + event-seq1)
;-------------------------------------------------------------------------
(defun redundancy-elimination (one-state-transition seq-of-state-transitions)
   (setq beginning-state (car one-state-transition))
   (setq event-seq (cadr one-state-transition))
   (setq ending-state (caddr one-state-transition))
   (setq action (car (cdddr one-state-transition)))
   (setq new-st '())

;LOOP
   (loop
      (cond
         ((and (> (length seq-of-state-transitions) 0)
               (equal beginning-state (caar seq-of-state-transitions))
               (equal ending-state (caddar seq-of-state-transitions)))
            (setq event-seq
               (remove-duplicates
                  (append event-seq (cadar seq-of-state-transitions))))
            (setq seq-of-state-transitions (cdr seq-of-state-transitions)))

         ((= (length seq-of-state-transitions) 0)
            (return (append
               (list (list beginning-state event-seq ending-state action))
               new-st)))

         (T (setq new-st (append (list (car seq-of-state-transitions)) new-st))
            (setq seq-of-state-transitions (cdr seq-of-state-transitions))))))
;END LOOP



;-------------------------------------------------------------------------
; Name: list-equal
; Processing: Given two lists, determines if they are equal.  Test-elements is
;   called to check the lists element by element.
;-------------------------------------------------------------------------
(defun list-equal (A B)
   (cond
      ((not (= (length A) (length B))) NIL)

      (T (test-elements A B))))
```

```
(defun test-elements (A B)
   (cond
      ((and (> (length A) 0)
            (equal (car A) (car B)))
         (test-elements (cdr A) (cdr B)))

      ((= (length A) 0) T)

      (T NIL)))




;-------------------------------------------------------------------------
; Name: signel-unreachable-stete
; Output: returns the unreechable state if one is found, otherwise returns T
;-------------------------------------------------------------------------
(defun signal-unreecheble-stete (initial-stete batetes stete-trensitions)
   (setq errnr nil)
   (setq bs (cer batstes))
   (setq reuse-st stete-transitinns).

; LOOP
   (lnop
      (setq single-st (cer reuse-st))
      (setq reuse-st (cdr reuse-st))
      (cnnd

;        ---------------------------------------------------------------
;        if beginning state is the initisl stete then dnn't look fnr
;        trensitions intn it
;        ---------------------------------------------------------------
         ((equslp bs initiel-stete)
            (return nil))

;        ---------------------------------------------------------------
;        if the beginning stete = tbe ending state of some nther stste
;        trensition, and the beginning stete of thet transition is not
;        seme as bs, then there is a trensitinn into bs and it is nnt
;        unreechable
;        ---------------------------------------------------------------
         ((end (equslp (ceddr single-st) bs)
               (nnt (equelp (cer single-st) bs)))
            (return nil))


;        ---------------------------------------------------------------
;        if all stete-trensitinns have been checked, but no trensitions
;        have been fnund intn the stete, thsn it is unreechable
;        ---------------------------------------------------------------
         ((equsl (length reuse-st) 0)
            (setq errnr T)
            (return error))

         (T nil)))
;END LOOP

   (cnnd
```

```
      (error bs)

      ((> (length bstates) 1)
         (signal-unreschable-state initial-state
                                   (cdr bstates)
                                   state-transitions))

      (T T)))



;-------------------------------------------------------------------------
; Name: valid-list-of-Ada-identifiers
; Inputs: list of identifiers to be checked for validity as Ada identifiers
; Processing:  Each element of the list is tested for validity as a valid
;   Ada identifier
; Outputs: T if each element of the list is a valid Ada identifier;
;          Nil otherwise
;-------------------------------------------------------------------------
(defun valid-list-of-Ada-identifiers (list-of-identifiers)
   (cood ((and (> (length list-of-idsntifiers) 0)
               (valid-Ada-identifier (car list-of-identifiers)))
          (valid-list-of-Ada-identifiers (cdr list-of-ideotifiers)))

         ((= (length list-of-identifiers) 0) T)

         (T nil)))



;-------------------------------------------------------------------------
; Name: valid-Ada-identifier
; Ioputs: An symbol thst is to be tested as a valid Ads ideotifier
; Processing: The symbol is first 'exploded' to form a list of each of the
;   constitueot elements of the symbol, e.g., compute_oorth_velocity becomes
;   ("c" "o" "m" "p" "u" "t" "e" "_" "o" "o" "r" "t" "h" "_" "v" "e" "l" "o"
;    "c" "i" "t" "y").
;   After explodiog the symbol, various tests are applied to determine if it
;   conforms to the requirements for a valid Ada identifisr.
;  Outputs: T   if the symbol represeots a valid Ada ideotifier
;           NIL otherwiss
;-------------------------------------------------------------------------
(dsfun valid-Ada-identifisr (idsntifisr)
   (sstq chsractsrs '("A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M" "N"
                      "O" "P" "Q" "R" "S" "T" "U" "V" "W" "X" "Y" "Z"))

   (setq numbers '("0" "1" "2" "3" "4" "5" "6" "7" "8" "9"))

   (setq list-identifier (explode identifier))

   (cond ((member (car list-identifier) characters :test #'equslp)
          (psrse-ideotifier (cdr list-ideotifier) characters numbers))

         (T NIL)))
```

```
;-------------------------------------------------------------------------
; Neme: perse-identifier
; Inputs:
;   list-identifier: e list made up of the constituent elements of the
;      identifier (ell elements must be in the same order es in the
;      original symbol
;   cherecters:  e list consisting of ell alphe cherecters (order is not
;      important)
;   numbers:  e list consisting of the numbers from 0 to 9 (order is not
;      importent)
;-------------------------------------------------------------------------
(defun parse-identifier (list-identifier cheracters numbers)
   (cond ((null list-identifier) T)

         ((member (cer list-identifier) (union cherecters numbers)
                  :test #'equelp)
            (perse-identifier (cdr list-identifier) cheracters numbers))

         ((equelp (car list-identifier) "_")
            (cond ((member (cadr list-identifier)
                           (union characters numbers) :test #'equelp)
                     (perse-identifier (cddr list-identifier)
                                       characters numbers))

                  (T nil)))

         ((equelp (cer list-identifier) ".")
            (cond ((member (cedr list-identifier) cherecters :test #'equalp)
                     (perse-identifier (cddr list-identifier)
                                       cberecters numbers))

                  (T nil)))

         (T nil)))




;-------------------------------------------------------------------------
; Name: explode
; Inputs:  A symbol thet is to be trensformed into e list of its consituent
;   elements
; Outputs:  A list of the elements (in the same order as they eppear in the
;   identifier) thet comprise the identifier.  Duplicetes are not removed as
;   they are significent to proper parsing.
;-------------------------------------------------------------------------
(defun explode (identifier)
   (cond ((> (length (string identifier)) 1)
            (make-identifier-list (string identifier)
                                  '()
                                  (length (string identifier))))

         ((= (length (string identifier)) 1)
            (list (string identifier)))

         (T NIL)))
```

```
;-------------------------------------------------------------------------
; Name: make-identifier-list
; Inputs:
;   string-identifier:  the string representation of the original identifier
;   list-identifier: the list representation of string-identifier
;   index: used to index elements in the string in order to break them out
;      separately in the list
;-------------------------------------------------------------------------
(defun make-identifier-list (string-identifier list-identifier index)
   (setq list-identifier
      (append (list (subseq string-identifier (- index 1) index))
              list-identifier))

   (cond ((> index 1)
          (setq index (- index 1))
          (make-identifier-list string-identifier
                                list-identifier
                                index))

         (T list-identifier)))



;-------------------------------------------------------------------------
; Name: main-menu
; Processing:  This routine clears the screen, established menu entries for
;  the AMPEE System main menu, and calls routines to display that menu.  The
;  users response is processed.  If an unimplemented feature is selected, the
;  routine is called again.
; Output:  The menu is displayed
;-------------------------------------------------------------------------
(defun main-menu ()
   (call-out erase-page 1 1)

   (setq path '(|Ada Missile Parts Engineering Expert System|))

   (setq menu_list (add_numbers '((|Parts Catalog|)
                                  (|Parts Identification|)
                                  (|Component Construction|))))

   (setq answer (menu_read path menu_list nil nil))

   (cond ((equalp (car answer) '|Parts Catalog|)
          (terpri)
          (pprint "The Parts Catalog facility is not yet available.")
          (pprint "Please hit 'return' to continue.")
          (read-line)
          (terpri)
          (main-menu))

         ((equalp (car answer) '|Parts Identification|)
          (terpri)
          (pprint "The Parts Identification facility is not yet available.")
          (pprint "Please hit 'return' to continue.")
          (read-line)
```

```
      (terpri)
      (main-menu))

    ((equalp (car answer) '|Component Construction|)
      (construction-menu))

    (T nil)))


;-------------------------------------------------------------------------
; Name:  construction-menu
; Processing:  This routine displays the menu for constructs and displays the
;  menu for the Component Construction faciltiy of the AMPEE System.
; Output:  The menu is displayed
;-------------------------------------------------------------------------
(defun construction-menu ()
  (call-out erase-page 1 1)

  (setq path '(|Component Construction|))

  (setq menu_list (add_numbers '((|Component Generation|)
                                 (|Component Regeneration|))))

  (setq answer (menu_read path menu_list nil nil))

  (cond ((equalp (car answer) '|Component Generation|)
         (part-constructor-menu))

        ((equalp (car answer) '|Component Regeneration|)
          (terpri)
          (pprint "The Component Regeneration facility is not yet available.")
          (pprint "Please hit 'return' to continue.")
          (read-line)
          (terpri)
          (construction-menu))

        (T nil)))


;-------------------------------------------------------------------------
; Name: part-constructor-menu ()
; Processing:  This routine constructs and displays a menu for the part
;  constructors that comprise the AMPEE System Component Generation facility.
; Output:  The menu is displayed.
;-------------------------------------------------------------------------
(defun part-constructor-menu ()
  (call-out erase-page 1 1)

  (setq path '(|Component Constructors|))

  (setq menu_list (add_numbers '((|Finite State Machine|))))

  (setq answer (menu_read path menu_list nil nil))

  (cond ((equalp (car answer) '|Finite State Machine|)
```

```
   (assert
      (state component-construction component-generation)))

(T nil)))
```

REFERENCES

[1] Lanergan, Robert G., and Denis K. Dugan, "A Successful Approach to Managing, Developing, and Maintaining Software", IEEE82 Trends and Applications: Advances in Information Technology, 1982.

[2] Lanergan, Robert G., and Denis K. Dugan, "Software Engineering with Reusable Designs and Code", IEEE 1981 Compcon Fall, pp 296-303.

[3] The Hartford Insurance Group, The Productivity Challenge, March 1982.

[4] The Hartford Insurance Group, The Productivity Challenge II, March 1983.

[5] National Bureau of Standards, Software Summary for Describing Computer Programs and Automated Data Systems, Federal Information Processing Standards (FIPS) Publication 30, June 30, 1974.

[6] American National Standards Institute, American National Standard for Computer Program Abstracts (ANSI X3.88-1981).

[7] Promotional material from DACS

[8] NTIS, A Directory of Computer Software, National Technical Information Service, Springfield, Virginia, 1984.

[9] NTIS, NTIS Subject Classification (A Guide for SRIM Users), National Technical Information Service, May, 1980.

[10] IMSL, IMSL Library Reference Manual, 1979, International Mathematical and Statistical Library, Houston, TX

[11] Promotional material from NAG, Inc.

[12] ACM, Collected Algorithms from CACM, ACM, New York, New York, 1980.

[13] Datapro Research Corporation, Datapro Directory of Micro Computer Software, 2 volumes, Datapro Research Corporation, Delran, New Jersey, 1984.

[14] International Computer Programs, ICP Software Directory, 52nd edition, 7 volumes, International Computer Programs, Indianapolis, Indiana, 1984.

[15] Parker, Robert A., Kathryn L. Heninger, David L. Parnas, and John E. Shore, Abstract Interface Specifications of the A-7E Interface Module, Information Systems Processing Branch, Communications Division, Naval Research Lab, Nov. 20, 1980.

[16] Clements, Paul C., R. Alan Parker, David L. Parnas, and John Shore, A Standard Organization for Specifying Abstract Interfaces, Naval Research Laboratory, Computer Sciences and Systems Branch, Information Technology Division, June 14, 1984.

[17] Neighbors, J.M., Software Construction Using Components, Ph.D. Dissertation, Department of Information and Computer Science, Univ. of California, Irvine, Technical Report 160, 1980.

[18] De Roze, Barry C., Defense System Software Management Plan, Defense Technical Information Center, Defense Logistics Agency, March 1976.

[19] Jones, Capers, "A Survey of Programming Design and Specification Techniques", Proceedings of Conference on Specifications of Reliable Software, 1979.

[20] Biermann, Alan W., "Approaches to Automatic Programming", Advances in Computer Science, vol. 15, Morris Rubinoff and Marshall C. Yovits, eds., Academic Press, 1976.

[21] Brown, John R., "Getting Better Software Cheaper and Quicker", Practical Strategies for Developing Large Software Systems, Ellis Horowitz, ed., Addison-Wesley, 1975.

[22] Prywes, Noah S., "Automatic Generation of Computer Programs", Advances in Computers, vol. 16, Morris Rubinoff and Marshall C. Yovits, eds., Academic Press, 1977.

[23] Barr, Avron and Edward A. Feigenbaum (eds.), The Handbook of Artificial Intelligence, vol. 2, William Kaufmann, Inc., Los Altos, CA 1982.

[24] Rawlings, Terry L., "A Technological Approach to Automating Software Maintenance", Proceedings of the 1st Software Maintenance Workshop, IEEE, 1983.

[25] Rawlings, Terry L., "A Discussion of Knowledge Representation within the DARTS Technology", Proceedings of the 17th Asimolar Conference on Circuits, Systems, and Computers, IEEE, 1984.

[26] McFarland, C. and Rawlings, T., "DARTS - A Software Manufacturing Technology", Proceedings of the AIAA 21st Aerospace Sciences Meeting, Jan. 10-13, 1983, Reno, Nevada.

[27] Promotional material from HOS

[28] Hamilton, M. and S. Zeldin, "The Functional Life Cycle Model and Its Automation: USE.IT", The Journal of Systems and Software, vol 3, 1983.

[29] Hendrix, Gary G. and Earl D. Sacerdoti, "Natural Language Processing - The Field in Perspective", Byte, Sept. 1981.

[30] Barr, Avron and Edward A. Feigenbaum (eds.), The Handbook of Artificial Intelligence, vol. 1, William Kaufmann, Inc., Los Altos, Ca., 1981.

[31] Stoegerer, J.K., "A Comprehensive Approach to Specification Languages", The Australian Computer Journal, vol. 16, no. 1, February 1984.

[32] McDonnell Douglas Astronautics Company, "SEP 2.205: Software Requirements Engineering Tools", Software Engineering Practices Manual, McDonnell Douglas Corporation, January, 1982.

[33] Prywes, Noah and Amir Pnueli, "Compilation of Nonprocedural Specifications into Computer Programs", IEEE Transactions on Software Engineering, vol. SE-9, no. 3, May, 1983.

[34] McDonnell Douglas Astronautics Company, Ada Design Language (ADL) Reference Manual, McDonnell Douglas Corporation, 1983.

[35] Using Ada (TM) as a Design Language, Draft Version 2.2, IEEE Working Group on "Ada as a Program Design Language", July 31, 1984.

[36] Biermann, Alan W., etal (eds.), Automatic Program Construction Techniques, Macmillian Publishing, 1984.

[37] MIT Laboratory for Computer Science, Laboratory for Computer Science Progress Report, Massachusetts Institute of Technology, July 1982-June 1983.

[38] Booch, Grady, Software Engineering with Ada, The Benjamin/Cummings Publishing Company, Inc., 1983.

[39] Buhr, Raymond, System Design with Ada, Prentice-Hall, 1984.

[40] Partsch, H., and R. Steinbruggen, "Program Transformation Systems", ACM Computing Surveys, vol. 15, no. 3, Sept., 1983.

[41] Rich, Charles, and Howard E. Shrobe, "Design of a Programmer's Apprentice", Artificial Intelligence: An MIT Perspective, vol. 1, Patrick Henry Winston and Richard Henry Brown, eds., The MIT Press, Cambridge, MA, 1979.

[42] Green, Cordell, et al, Report on a Knowledge-Based Software Assistant, Kestrel Institute, June, 1983.

[43] Osborn, S.L., in a review of "SOFSPEC: A Pragmatic Approach to Automated Specification Verification", Erika Nyari and Harry M. Sneed, Computing Reviews, vol. 25, no. 10, p. 465.

[44] McDonnell Douglas Astronautics Co., Ada Missile Parts Engineering Expert System Software Requirements Specification, Final, September, 1985.

[45] McDonnell Douglas Astronautics Co., Ada Missile Parts Engineering Expert System Software Top-Level Design Document, Final, September, 1985.

[46] Inference Corporation, ART™ Reference Manual, April, 1985.

[47] Clayton, Bruce D., ART™ Programming Primer, Inference Corp., April, 1985.

[48] Clayton, Bruce D., ART™ Programming Tutorial, Volume 1: Elementary ART Programming, Inference Corp., March, 1985.

[49] Clayton, Bruce D., ART™ Programming Tutorial, Volume 2: A First Look at Viewpoints, Inference Corp., March, 1985.

[50] Dym, Clive M., "Expert Systems: New Approaches to Computer-Aided Engineering", Xerox PARC, April, 1984.

[51] "NISO to Introduce Software Numbering System", Advanced Technology Libraries, vol.13, no. 9, Sept. 1984, Knowledge Industries Publications, Inc.

INITIAL DISTRIBUTION

DTIC-DDAC	2	NASA (CODE RC)	1
AUL/LSE	1	NSA	1
FTD/SDNF	1	NTSC/CODE 251	1
HQ USAFE/INATW	1	AD/XRB	1
AFWAL/FIES/SURVIAC	1	LOCKHEED (DR SURY)	1
AFATL/DOIL	2	HUGHES (MR BARDIN)	1
AFATL/CC	1	IBM (MS VESPER)	1
AFCSA/SAMI	1	RAYTHEON (WILLMAN)	1
AFATL/CCN	1	SOFTECH INC (MS BRAUN)	1
AFATL/FXG	10	MITRE CORP (MR SYLVESTER)	1
ASD/RWX	1	AEROSPACE CORP (MR HOGAN)	1
SPAWAR (814AB NC #1)	1	TEXAS INSTRUMENTS (MR FOREMAN)	1
SPAWAR (CODE 613)	1	RATIONAL (MR BOOCH)	1
NRL (CODE 5150)	1	ROCKWELL INTERNATIONAL (MR GRIFFIN)	1
ASD/XRX	1	MITRE CORP (MS CLAPP)	1
AFWAL/AAA-2	1	NOSC (CODE 423)	1
HQ AFSC/PLR	3	GENERAL DYNAMICS (MR MURRAY)	1
INST OF DEFENSE ANALYSES	5	HONEYWELL INC (MS GIDDINGS)	1
STARS JOINT PROGRAM OFFICE	1	HONEYWELL INC (DR FRANKOWSKI)	1
USA MATERIEL CMD/AMCDE-SB	1	HUGHES DEFENSE SYS DIV (S.Y. WONG)	1
NAVAL SEA SYS CMD (SEA 61R2)	1	NOSC (MR WASILAUSKI)	1
NAVAL AIR DEVELOPMENT CTR (CODE 50C)	1	NAC (N)	1
BMDATC	1	TASC (MR SERNA)	1
NSWC/CODE N20.	1	MARTIN MARIETTA (MR CUDDIE)	1
NAVAL UNDERSEA SYS CTR (CODE 3511)	1	TASC (DR CRAWFORD)	2
NOSC/CODE 423	1	SYSCON CORP (DR BRINTZINHOFF)	1
AFWAL/AAAF-2	1	ADA TRAINING SECTION	1
USA EPG/STEEP-MT-DA	1	COMPUTER SCI CORP (MR FRITZ)	1
AFSC/DLA	1	LINKABIT (MR SIMON)	1
USA MSL CMD/AMSMI-OAT	1	RAYTHEON (MR GINGERICH)	1
SPAWAR (CODE 06 NC #1)	1	UNIVERSITY OF COLORADO	1
ASD/EN	1	MCDONNELL AIRCRAFT (MR MCTIGUE)	1
AD/ENE	1	HUGHES AIRCRAFT (MR NOBLE)	1
SD/ALR	1	GENERAL DYNAMICS (MR PRZYBYLINSKI)	1
RADC/COEE	2	AJPO (MS CASTOR)	1
NRL/CODE 7590	1	NWC (CODE 3922)	1
AFSC/SDZD	1	NAVAIRSYSCOM HQS	1
HQ USAF/RDPV	1	ASD/ENASF	1
AD/ENSM	1	ASD/ENA	5
BMO/ENBE	1	UNIVERSITY OF TEXAS	1
ESD/ALS	1	MARTIN MARIETTA (MR SORONDO)	1
ESD/ALSE	2	MCDONNELL DOUGLAS (DR MCNICHOLL)	2
AJPO	1	GENERAL DYNAMICS (MR SCHNELKER)	1
AFATL/AS	1	TRW (MR SHUGERMAN)	1
AFATL/SA	1	RATIONAL (MR HAKE)	1
AD/XR	1	WESTINGHOUSE (MR GREGORY)	2
WIS JPMO/AOT	1	GENERAL DYNAMICS (DR TABER)	1
AFWAL/AAAF	1	BOEING COMMERCIAL AIRPLANE CO	1

INITIAL DISTRIBUTION (CONTINUED)

ROCKWELL INTERNATIONAL (MIKULSKI)	1	ROCKWELL (MS KIM)	1
BOEING AEROSPACE (MR HADLEY)	1	SCIENCE APPLICATIONS INTERNATIONAL (MR STUTZKE)	1
IBM (MR MCCAIN)	1	NSWC (U-33)	1
FSU (DR BAKER)	1	COMPUTER SOFTWARE & SYSTEMS	1
AEROSPACE CORP (MR LUBOFSKY)	1	GRUMMAN DATA SYSTEMS (MR MARKMAN)	1
DATA GENERAL (MR DAMASHEK)	1	AFWAL/FIGL	1
SDC (MR HERMANN)	1	AFWAL/FIGX	1
WINTEC (MR CONNEL)	1	AFWAL/AARI-1	1
NORTHROP (MR OHLSEN)	1	EMERSON ELECTRIC (MR BYRNES)	1
ARC (MR ROBERSON)	1	TASC (MR JAZMINSKI)	1
GRC (DR ALBRITTON)	1	E-SYSTEMS (MR SNODGRASS)	1
AFWL/NTSAC	1	NTSC (CODE 742)	1
USA MSL CMD/SCI INFO CTR	1	UNIVERSITY OF ALABAMA	1
NSWC/TECH LIB	1	MCDONNELL DOUGLAS (MR VILLACHICA)	1
NWC (CODE 343)	1	SANDERS ASSOCIATES (MR FRY)	1
OO-NAVAL RESEARCH (CODE 784DL)	1	AFSC/PLR	1
NAVAL POSTGRAD SCHOOL (CODE 1424)	1	AFSC/DLA	1
DEFENSE COMMUNICATIONS AGENCY	1	WESTINGHOUSE (MR SQUIRE)	1
NASA AMES RESEARCH CTR (CHAPMAN)	1	NAVAIR SYSCOM (AIR 54662)	1
NASA LANGLEY RESEARCH CTR (MOTLEY)	2	BOEING AEROSPACE (MR BOWEN)	1
RAND CORP/AFELM	1	GOULD INC (DR ARORA)	1
AVCO SYS DIV/RESEARCH LIB	1	LOCKHEED (MR PINCUS)	1
AVCO-EVERETT RES LAB/TECH LIB	1	INTERMETRICS (MR BROIDO)	1
FAIRCHILD IND/INFO CTR	1	GENERAL ELECTRIC (MS MICKEL)	1
GENERAL DYNAMICS, CONVAIR DIV	1	HUGHES AIRCRAFT (DR HUANG)	1
GENERAL DYNAMICS, FT WORTH DIV	1	SYSTEMS DEVELOPMENT (MR SIMOS)	1
HONEYWELL/TECH LIB	1	GENERAL DYNAMICS (MR WARNER)	1
HUGHES AIRCRAFT/TECH LIB	1	CARNEGIE-MELLON UNIVERSITY	1
HUGHES AIRCRAFT/MSL SYS GP/TECH LIB	1	COMPUTER TECH ASSOCIATES (HEYLIGER)	1
LOCKHEED/TECH LIB	1	NORTHROP CORP (MR SWAN)	1
LOCKHEED/TECH INFO CTR	1	CNI SOFTWARE (SINGER KEARFOTT DIV)	1
MARTIN MARIETTA/TECH LIB	1	LOCKHEED (MR COHEN)	1
MCDONNELL DOUGLAS/TECH LIB	1	MCDONNELL DOUGLAS (SANDY COHEN)	2
MCDONNELL DOUGLAS/LIB SVCS	1	DRAPER LABORATORY (MR DAVID)	1
NORTHROP CORP/AIRCRAFT DIV	1	DRAPER LABORATORY (DR DEWOLF)	1
RAYTHEON/TECH LIB	1	GOULD INC (MR WILLIS)	1
VOUGHT CORP/LIB	1	ADVANCED TECHNOLOGY (MR COOPER)	1
SDC (MR MILLAR)	1	HQ USAF/RDST	1
INTERMETRICS (MR ZIMMERMANN)	1	AFPRO/EN	1
ROCKWELL/TECH INFO	1	AFPRO/TRW/EPP	1
TRW (MR MUNGLE)	1	LOCKHEED (MR DORFMAN)	1
AT&T (MR MAY)	1	HONEYWELL (MR LANE)	1
AVCO SYS TEXTRON (MR SOHN)	1	SPERRY A&MG (MR ROSS)	1
BOEING ELECTRONICS (MR MEDAN)	1	UNITED TECHNOLOGIES (STOLZENTHALER)	1
USA CECOM/DRSEL-TCS-MCF	1	LTV AEROSPACE & DEFENSE	1
USA CECOM/DRSEL-TCS-ADA	1	EL66 B 4610 (MR FREEMAN)	1
UNIVERSITY OF WASHINGTON	1	SYSTEMS DEVELOPMENT CORP (ATCHLEY)	1
AFATL/GRC	1		
USA-ARDC	1		

INITIAL DISTRIBUTION (CONCLUDED)

ROCKWELL (DILLHUNT)	1
IBM FED SYSTEMS DIV (MR ANGIER)	1
JET PROPULSION LAB (MR KRASNER)	1
MARTIN MARIETTA ORLANDO AEROSPACE	1
GENERAL ELECTRIC (MR DELLEN)	1
AFWAL/PDFIC	1
LOCKHEED (READY)	1
BOEING AEROSPACE (MR DOE)	1
FORD AEROSPACE (DR FOX)	1
LOCKHEED SOFTWARE TECH CTR (LYONS)	1
IBM (MR BENESCH)	1
MITRE CORP (MR SHAPIRO)	1
MITRE CORP (MR MAGINNIS)	1
GRUMMAN AEROSPACE (MR POLOFSKY)	1
MCDONNELL DOUGLAS (KAREN L SAYLE)	1
MCDONNELL DOUGLAS (GLENN P LOVE)	1
RDME AIR DEVELOPMENT CTR (CHRUSCICKI)	1
RAYTHEON EQUIPMENT DIV (MS SILLERS)	1
AT&T BELL LABS (MS STETTER)	1

(The reverse of this page is blank)

THIS REPORT HAS BEEN DELIMITED
AND CLEARED FOR PUBLIC RELEASE
UNDER DOD DIRECTIVE 5200.20 AND
NO RESTRICTIONS ARE IMPOSED UPON
ITS USE AND DISCLOSURE.

DISTRIBUTION STATEMENT A

APPROVED FOR PUBLIC RELEASE;
DISTRIBUTION UNLIMITED.

B102655

ERRATA

AFATL-TR-85-93

VOLUMES I, II, and III

COMMON ADA® MISSILE PACKAGES (CAMP)

FINAL REPORT

AIR FORCE ARMAMENT LABORATORY

EGLIN AIR FORCE BASE, FLORIDA

32542-5434

1. DD Form 1473:

   In Block 16, add the following:

   "Methodology used in this report does not constitute computer software as defined in AFR 300-6."

2. This errata is unclassified.

Lynn S. Marrs

LYNN S. MARRS
Chief, Technical Reports Section

SUPPLEMENTARY

INFORMATION

DEPARTMENT OF THE AIR FORCE
WRIGHT LABORATORY (AFSC)
EGLIN AIR FORCE BASE, FLORIDA, 32542-5434

ERRATA
AD-B102655

REPLY TO ATTN OF: MNOI

13 Feb 92

SUBJECT: Removal of Distribution Statement and Export-Control Warning Notices

TO: Defense Technical Information Center
ATTN: DTIC/HAR (Mr William Bush)
Bldg 5, Cameron Station
Alexandria, VA 22304-6145

1. The following technical reports have been approved for public release by the local Public Affairs Office (copy attached).

	Technical Report Number	AD Number
1.	88-18-Vol-4	ADB 120 251
2.	88-18-Vol-5	ADB 120 252
3.	88-18-Vol-6	ADB 120 253
4.	88-25-Vol-1	ADB 120 309
5.	88-25-Vol-2	ADB 120 310
6.	88-62-Vol-1	ADB 129 568
7.	88-62-Vol-2	ADB 129 569
8.	88-62-Vol-3	ADB 129-570
9.	85-93-Vol-1	ADB 102-654
10.	85-93-Vol-2	ADB 102-655
11.	85-93-Vol-3	ADB 102-656
12.	88-18-Vol-1	ADB 120 248
13.	88-18-Vol-2	ADB 120 249
14.	88-18-Vol-7	ADB 120 254
15.	88-18-Vol-8	ADB 120 255
16.	88-18-Vol-9	ADB 120 256
17.	88-18-Vol-10	ADB 120 257
18.	88-18-Vol-11	ADB 120 258
19.	88-18-Vol-12	ADB 120 259

2. If you have any questions regarding this request call me at DSN 872-4620.

LYNN S. WARGO
Chief, Scientific and Technical
Information Branch

1 Atch
AFDTC/PA Ltr, dtd 30 Jan 92

ERRATA

DEPARTMENT OF THE AIR FORCE
HEADQUARTERS AIR FORCE DEVELOPMENT TEST CENTER (AFSC)
EGLIN AIR FORCE BASE, FLORIDA 32542-5000

REPLY TO ATTN OF: PA (Jim Swinson, 882-3931) 30 January 1992

SUBJECT: Clearance for Public Release

TO: WL/MNA

The following technical reports have been reviewed and are approved for public release: AFATL-TR-88-18 (Volumes 1 & 2), AFATL-TR-88-18 (Volumes 4 thru 12), AFATL-TR-88-25 (Volumes 1 & 2), AFATL-TR-88-62 (Volumes 1 thru 3) and AFATL-TR-85-93 (Volumes 1 thru 3).

Virginia N. Pribyla

VIRGINIA N. PRIBYLA, Lt Col, USAF
Chief of Public Affairs

AFDTC/PA 92-039