2

NPS52-88-021

NAVAL POSTGRADUATE SCHOOL

Monterey, California

THESIS

A LANGUAGE TRANSLATOR
FOR A
COMPUTER AIDED RAPID PROTOTYPING
SYSTEM

Charlie Robert Moffitt, II

Thesis Advisor September 1988 Luqi

Approved for public release; distribution is unlimited.

Prepared for:

Naval Postgraduate School
Monterey, CA 93943

NAVAL POSTGRADUATE SCHOOL
Monterey, California

Rear Admiral R. C. Austin
Superintendent

H. Shull
Provost

This report was prepared in conjunction with research conducted for the National Science Foundation and funded by the Naval Postgraduate School.

This report was prepared by:

Luqi

LUQI
Assistant Professor
of Computer Science

Reviewed by:

Robert B McGhee

ROBERT B. MCGHEE
Chairman
Department of Computer Science

Released by:

K.T. Marshall

KNEALE T. MARSHALL
Dean of Information
and Policy Science

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) NPS52-88-021	5. MONITORING ORGANIZATION REPORT NUMBER(S)

6a. NAME OF PERFORMING ORGANIZATION Naval Postgraduate School	6b. OFFICE SYMBOL *(If applicable)* 52	7a. NAME OF MONITORING ORGANIZATION National Science Foundation
6c. ADDRESS *(City, State, and ZIP Code)* Monterey, CA 93943		7b. ADDRESS *(City, State, and ZIP Code)* Washington, DC 20550
8a. NAME OF FUNDING/SPONSORING ORGANIZATION Naval Postgraduate School	8b. OFFICE SYMBOL *(If applicable)*	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER O&MN, Direct funding
8c. ADDRESS *(City, State, and ZIP Code)* Monterey, CA 93943		10. SOURCE OF FUNDING NUMBERS

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT ACCESSION NO.

11. TITLE *(Include Security Classification)*

A LANGUAGE TRANSLATOR FOR A COMPUTER AIDED RAPID PROTOTYPING SYSTEM (U)

12. PERSONAL AUTHOR(S)
MOFFITT, Charlie R., LUQI

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT *(Year, Month, Day)*	15. PAGE COUNT
Annual	FROM 87/09 TO 88/08	1988, Sept	116

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS *(Continue on reverse if necessary and identify by block number)*
FIELD	GROUP	SUB-GROUP	computer aided software engineering, rapid prototyping, specification, real-time software, embedded systems, software design, reusability

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

While the cost of computing hardware has decreased steadily, the cost of software design, development and, maintenance has increased. One approach to reduce the cost of software development is rapid prototyping. Further, it has been proposed to combine the design strategy of rapid prototyping with a computer aided software prototyping system. Such a system would allow the software designer to employ a software base of reusable program modules. It would assist in prototyping and would automate a large part of the development effort. An important component of the automation would be a language translator facility. This translator would allow the designer to develop a software prototype in a high level specification language which would be simple and convenient to use and would translate the specification statements into an executable software language.

This report demonstrates the feasibility of using a language translator by developing a prototype translator for computer aided software prototyping system. The translator

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT. □ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED

22a. NAME OF RESPONSIBLE INDIVIDUAL Luqi	22b. TELEPHONE *(Include Area Code)* (408) 646-2735	22c. OFFICE SYMBOL 52Lq

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

SECURITY CLASSIFICATION OF THIS PAGE

☆ U.S. Government Printing Office: 1986—606-243

19. is written in Attribute Grammar (AG) language and translates software specifications stated in the Prototype System Description Language (PSDL) into computer executable code in the Ada language.

A Language Translator
For A
Computer Aided Rapid Prototyping System

by

Charlie Robert Moffitt, II
Lieutenant, United States Navy
A.B., Central Wesleyan College, 1972

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN TELECOMMUNICATIONS SYSTEM
MANAGEMENT

from the

NAVAL POSTGRADUATE SCHOOL
March 1988

Author: ______________________________
Charlie Robert Moffitt, II

Approved by: Luqi
Luqi, Thesis Advisor

____
Major John B. Isett, Second Reader

____
David R. Whipple, Chairman,
Department of Administrative Science

____
James M. Fremgen,
Acting Dean of Information and Policy Sciences

ABSTRACT

While the cost of computing hardware has decreased steadily, the cost of software design, development and, maintenance has increased. One approach to reduce the cost of software development is rapid prototyping. Further, it has been proposed to combine the design strategy of rapid prototyping with a computer aided software prototyping system. Such a system would allow the software designer to employ a software base of reusable program modules. It would assist in prototyping and would automate a large part of the development effort. An important component of the automation would be a language translator facility. This translator would allow the designer to develop a software prototype in a high level specification language which would be simple and convenient to use and would translate the specification statements into an executable software language.

This thesis demonstrates the feasibility of using a language translator by developing a prototype translator for a computer aided software prototyping system. The translator is written in Attribute Grammar (AG) language and translates software specifications stated in the Prototype System Description Language (PSDL) into computer executable code in the Ada language.

Keywords: Real Time, embedding, reusability (KR)

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

I. INTRODUCTION

A. COMPUTER AIDED PROTOTYPING SYSTEM

A computer aided prototyping system (CAPS) has been proposed which would implement many ideas for improving software productivity [Ref. 1: p. 68]. Figure 1 on page 2 illustrates the proposed architecture of such a system. This architecture is designed to be implemented in an automated environment, the rapid prototyping schema. The automated environment will make it practical to develop, test, and quickly modify prototypes of a proposed system. It will make possible the demonstration of a working system (or perhaps several) to the customer in order to firm up requirements and functional specifications.

1. Major CAPS Components

The CAPS architecture consists of six major subsystems. The central objective of the system is to optimize the use of the programmer's time in prototype development. The objective of prototype development is to:

- provide a firm sct of requirements and functional specifications which will guide development of the production software.
- ensure agreement between customer and developer as to the requirements and expected performance characteristics of the system
- generate a modular, skeletal structure of the software system which will serve to guide further implementation
- shorten prototype development time and thus accelerate production system delivery
- assist in estimating the ultimate development costs of the finished system

The CAPS allows the designer to enter a specification-based description of the proposed system in a high level languag^e^ constructed cspecially for prototype development. These specifications are acted upon by a rewrite subsystem and an execution support subsystem. The rewrite subsystem converts the specification statements into a normalized form. The normalized statements are used to search a software database of reusable components which are then provided to the execution support subsystem for instantiation in the prototypc. The specifications are also actcd upon by the execution support subsystem to produce executable code into which the reusable software modules

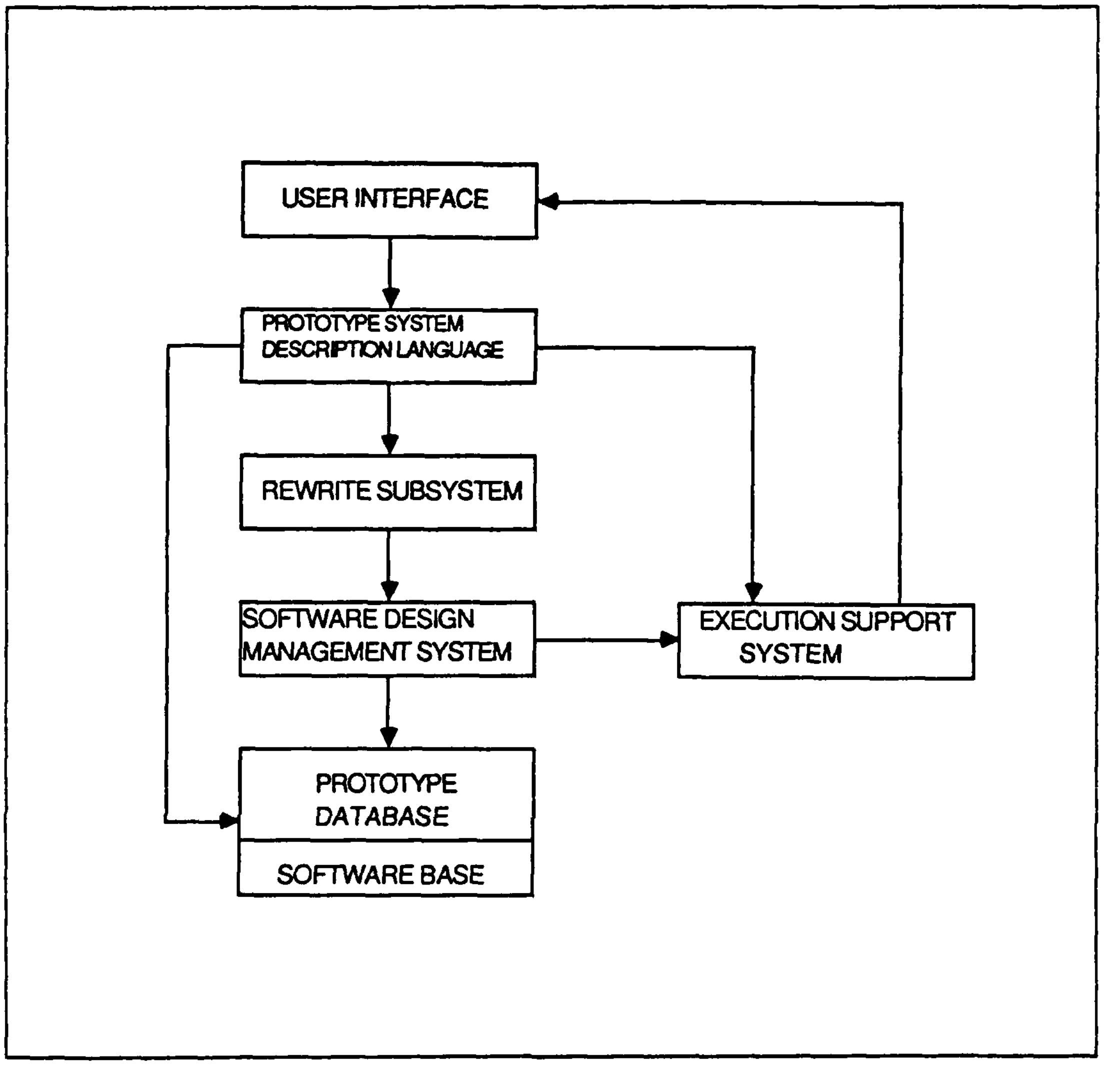


Figure 1. Computer Aided Prototyping System Architecture (CAPS)

are instantiated. The resulting prototype can then be tested for conformance to specifications and proper operation. New versions or redesigned versions can be quickly constructed and tested as the need arises.

2. A Prototype Language

The core of the CAPS is the Prototype System Description Language (PSDL). It is optimized for use at the specification and design level of programming. Special structures exist for describing real-time systems. A PSDL description represents a system as operators communicating via data streams. The structure of the language encourages modular design of thc prototype and by extension the eventual production version. A more detailed examination of PSDL will be undertaken in Chapter 3 of this paper.

3. Rewrite System

The rewrite system examines the PSDL file and produces a normalized version of the specifications which is used to search the software base for appropriate components. If no component is found, the designer may examine the module to see if it can be decomposed into more primitive modules. If it can be, then the new modules are specified in PSDL, the specifications are normalized, and the search is repeated. If no modules are found and the modules cannot be decomposed then they must be hand coded in the executable language. When modules are found in the software base, they are provided to the execution support subsystem for instantiation in the prototype program. The functions of managing the database, searching it for appropriate modules, and calling forth those that are found is the province of the Software Design Management System. Currently a special Object-Oriented DBMS is being developed to meet the special requirements of the SDMS [Ref. 2]. For present testing it may be necessary to employ a commercially available database, though none currently meets the special requirements of this system. [Ref. 1: p. 70]

4. Execution Support System

The Execution Support System (ESS) consists of three interrelated parts, one of which is the subject of this paper. Figure 2 on page 4 illustrates the relationship between the components of the ESS. Each element of the system and its function will be briefly described. The Translator design will be developed in Chapter 3 and 4.

a. Translator

The Translator (TL) converts PSDL source code into Ada®[1] source code. Output from the TL is provided to the Ada compiler/linker along with some additional information from :he Static Scheduler (SS) to produce Ada object code. The object code is then exported to the operating system and can be run for test and demonstration purposes. The TL passes real time constraints through without translation. The TL

[1] Ada is a registered trademark of the United States Government, Ada Joint Program Office.

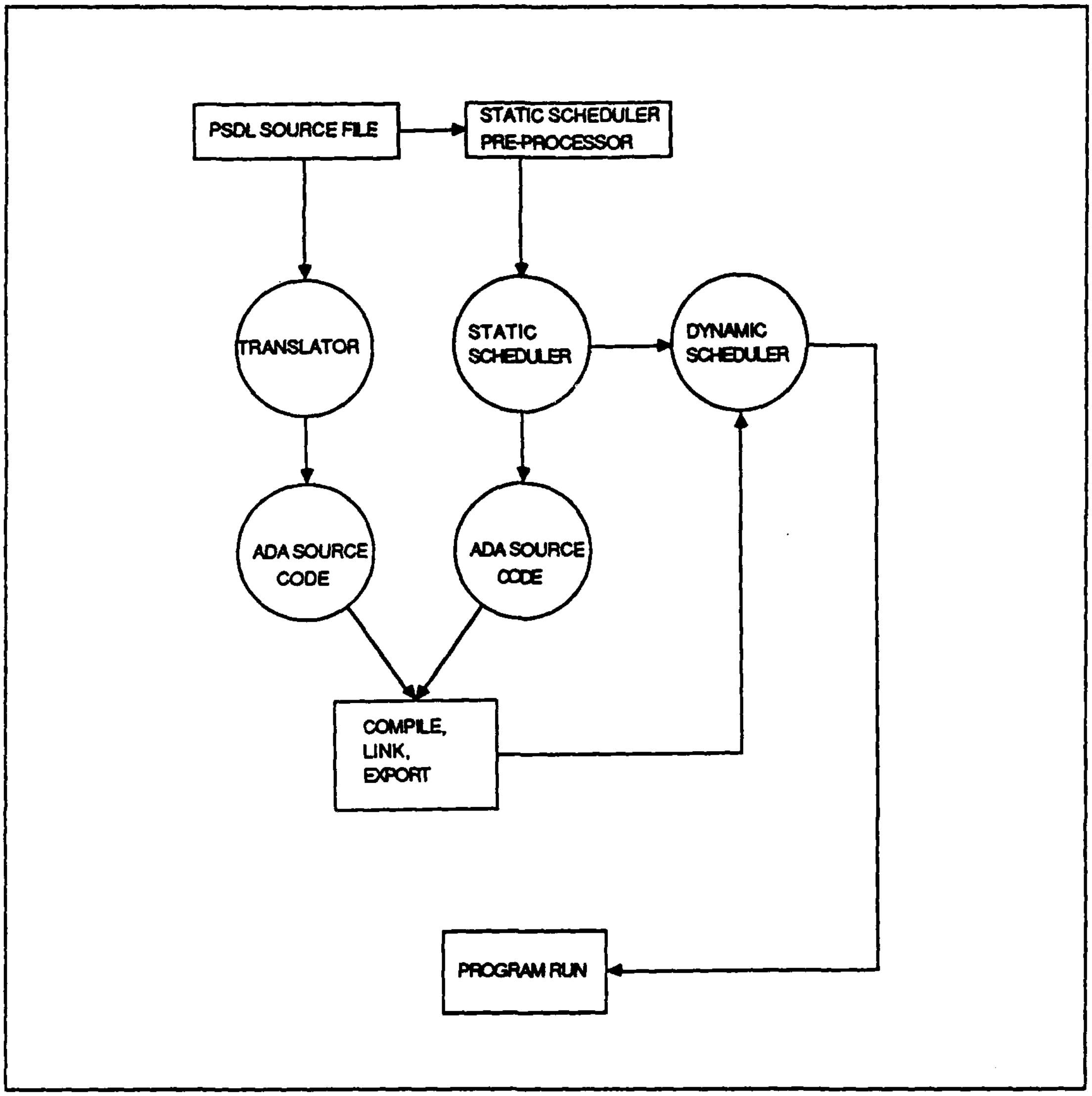


Figure 2. Execution Support System (ESS) structure

creates code to implement the operators as procedures which will be called by the main subprogram/schedule created by the SS. The TL is responsible for instantiating a generic package which models the data stream buffers between operators. The TL also ensures that all operator triggering conditions are encoded correctly, and that the Trigger data type and the Exception data type are properly encoded for the final model.

b. Static Scheduler

The SS examines the PSDL source file to locate all modules having real-time constraints, and to determine if any special precedence relations exist among the modules. The SS then generates the necessary Ada code to implement the timing constraints and the precedence relationships. The SS also generates the main subprogram or task. The SS finally generates a schedule of operation for the program which takes into account the worst case time schedule for all modules that have critical, real-time constraints such as maximum execution time, minimum calling period, and minimum response time. This information is encoded into the modules to enforce timing constraints at run time. Figure 3 illustrates the action of the SS. Janson [Ref. 3] and O'Hern [Ref. 4] have studied the conceptual and initial empirical investigations into the design and implementation of the SS.

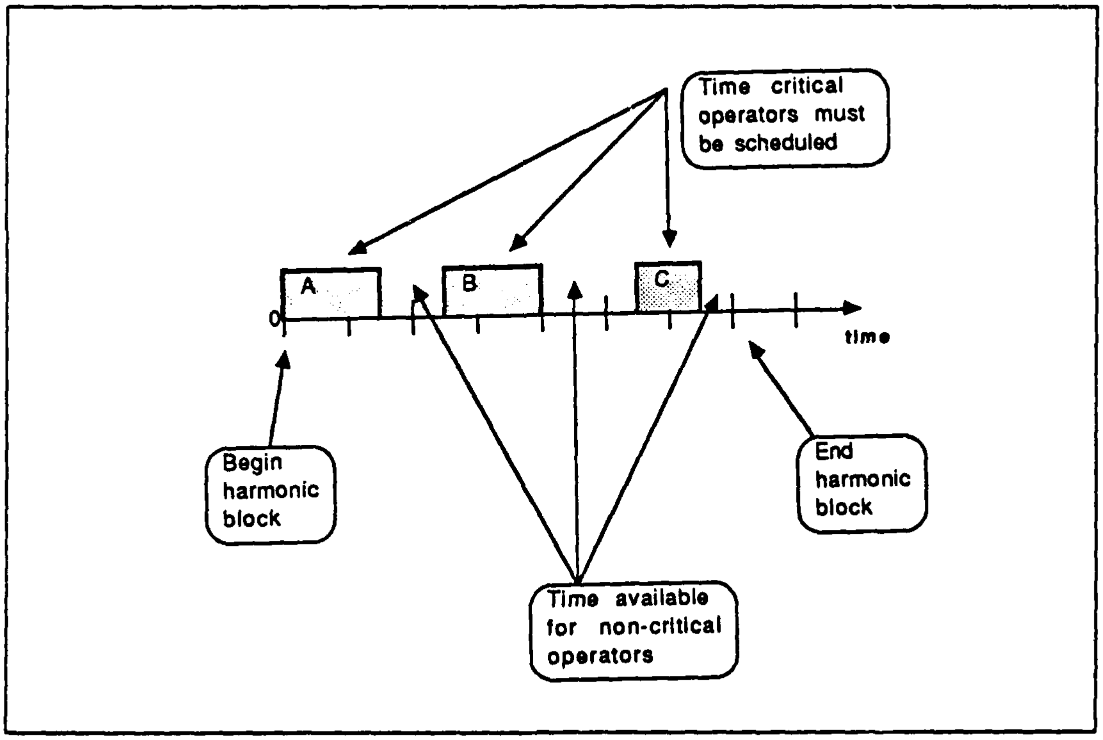

Time critical operators must be scheduled
A
B
C
0
time
Begin harmonic block
End harmonic block
Time available for non-critical operators

Figure 3. Static and Dynamic Schedule Schema

c. *Dynamic Scheduler*

The Dynamic Scheduler (DS) operates at runtime along with the prototype model. It is designed to control the execution of all non-critical operators within the program. A non-critical operator is one which is not subject to hard real-time constraints. The DS is invoked each time there is spare time within the static runtime schedule created by the SS. At that time DS commences execution of the next available module in its set of operators and continues to invoke non-critical modules until the available time is exhausted. At that point, operation of the DS is interrupted and control is returned to the SS to continue the time critical operations. Figure 3 on page 5 shows the relationship between the DS operation and the SS operation. Eaton [Ref. 5] has examined the conceptual and fundamental design issues for the DS.

B. CENTRAL AIM OF THIS PAPER

In order to approach the development of the proposed CAPS architecture on a sound basis, it is necessary to consider the important theoretical ideas on which the effort will be based. The key literature which made possible the effort to produce this prototype translator will be reviewed. The reader may also wish to consult additional references cited in the bibliography. Many of the materials therein provide insight into the difficult problems of improving productivity in software engineering through automated means, and of configuring software systems to address real-time constraints on system performance.

The purpose of this paper is to demonstrate the feasibility and functionality of an automated language translation facility which can be coupled into a larger, integrated system for automated software prototyping. This translator will receive as input a source file in PSDL which specifies the system to be prototyped. It will produce as output, source code in the Ada language which will be compiled and exported to the operating system. Discussion of the rationale for choosing PSDL and Ada for use in a prototyping environment will be presented in Chapter 3. Architecture and design of the translator will be developed in Chapter 3. This study will be limited to producing a translator capable of recognizing the full PSDL syntax and producing, at most, rudimentary Ada output. This limitation is imposed because a rigorous, formal definition of the relationship between Ada and PSDL has not yet been accomplished. Once such a definition is achieved, the results must be applied to the elementary translator created in the present effort. The resulting translator, combining a formally established relationship between the source and target languages with a translator which recognizes PSDL syntax, will meet the requirements of the CAPS architecture for a translator applicable to general cases.

The present work is arranged as follows:

Chapter 2 discusses the theoretical basis for the CAPS system and surveys previous research which lays the foundation for the present work.

Chapter 3 presents the basic implementation approach to the translator construction.

Chapter 4 presents some possible applications of CAPS research to the field of telecommunications.

Chapter 5 presents conclusions and possible future avenues for research.

II. THEORETICAL UNDERPINNINGS OF CAPS

A. HARDWARE AND SOFTWARE: A PROBLEM

Several trends have become apparent in the computing industry. These trends have a significant impact on the field of software engineering. The first of these trends is the expansion of computer usage into an ever widening arena of applications. Early digital computers were largely confined to military, governmental, and research applications. A relatively small population of users was affected by the computer. Today the computer is a significant feature of everyday life for almost the entire industrialized world. Few governments or businesses function without the aid of computer systems. Computer systems route our telephone calls and record our bank transactions. Military forces worldwide employ computers for handling record traffic and a variety of command and control functions, as well as many tactical applications.

One study estimated that forty percent of the U.S. labor force relied on computers in performance of their daily work during 1985. Another barometer of the growth in demand for computing is the percentage of the Gross National Product (GNP) that it represents. It has been estimated that the total amount spent on all aspects of computing in 1980 was approximately 5 percent of GNP or about $130 billion. It is expected that this will rise to as much as 12.5 percent of GNP by 1990 [Ref. 6: p. 124].

Another trend, is the increasing power of each new generation of computing machines and the corresponding decrease in relative cost for a machine of that power. The cause of this trend is found in improved engineering and production methods for transistors and integrated circuits. The advent of Large Scale and Very Large Scale Integration (LSI, VLSI) have made possible great improvements in computing hardware architecture and lower costs of production. Each new generation of computing machines has benefited from engineering and production knowledge gained in previous generations. Today's machines are more reliable and robust in performance than their predecessors.

The decrease in hardware costs and increasing demand for computing services has generated a third trend in the industry. There is an increasing cost of software development and maintenance as compared to the costs of hardware, and there is an increasing cost of software as a total fraction of computing costs. Figure 4 on page 9 shows the changing ratio of expenditures for hardware and software over time [Ref. 7]. The figure should not be interpreted as applying to any specific system. Instead, it represents the general trend within the industry, that software development and especially maintenance represents an increasingly large portion of the cost of computing. The shift in resources to software maintenance arises from several considerations. There is more and more software to be maintained so a correspondingly larger number of persons are required to perform maintenance functions.

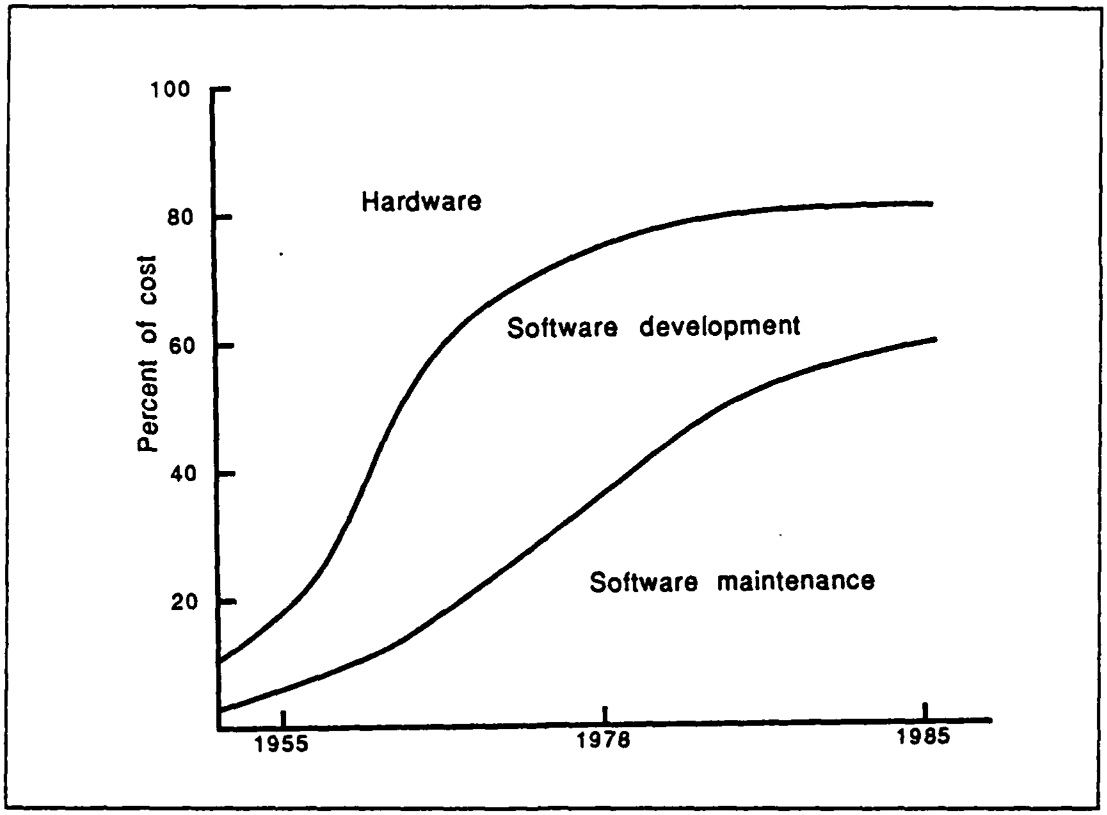


Figure 4. Changing hardware/software cost ratio

Mills [Ref. 8: p. 267] points out that in only 25 years of software development history, some 75 percent of data processing personnel are taken up with maintenance, not development. He states two reasons for this. One is logistic and the other a technical reason. The logistic reason is that systems are maintained indefinitely after a definite period of development. Each time a development is completed some fraction of the work force must be diverted to maintenance. Mills [Ref. 8: p. 267] demonstrates that, for a constant work force working for a long period of time, the 75 percent fraction devoted to maintenance can be predicted. He states that only the purging or replacement of older applications keeps the figure below 100 percent. The technical reason is that it has proven more difficult to develop correct and capable systems in the first place. The ability to integrate and debug systems has been consistently underestimated. Time after time software systems are late in delivery and do not do the things the users expected them to do. Also, there have consistently been underestimations of the uncertainty and change facing software applications. For both these reasons, a large work force is required to do both corrective and adaptive maintenance to keep the application software functioning [Ref. 8: p. 267].

Another aspect of maintenance is what we mean by that term in the software industry. Maintenance of software systems does not simply mean corrective maintenance in the strictest sense. Carrio [Ref. 9: p. 19] lists many other activities which are often encompassed by the term, including:

- Enhancing the system ("gold-plating") in ways that do not alter the core requirements of the system
- Adding new or substituting other requirements for performance relative to those implemented (often the result of a poorly defined requirements set at the beginning of development)
- Changing the baseline performance level to expand the performance envelope or due to expected changes in doctrine-optimization
- Changing baseline requirements due to a planned evolutionary development of the system

Mills [Ref. 8: p. 267] humorously describes the terms "debugging" and "maintenance" as euphemisms in the software engineering world. Debugging is the correction of errors in the program which were originally put there by the programmers.

Maintenance is the restoral of the program to a correct state of operation; but the program was never correct in the first place. The point he aims at is that proper software design and engineering techniques are required to achieve maximum productivity and quality software systems.

Beohm [Ref. 10] estimates that in 1980, the cost of software for computer manufacturers, user organizations, and software firms was $40.2 billion dollars. This amount represented 84 percent of the total budget spent on computing hardware and software. As seen in Figure 5, software may account for 90 percent of the amount spent on computing systems by the 1990's [Ref. 11: p. 49].

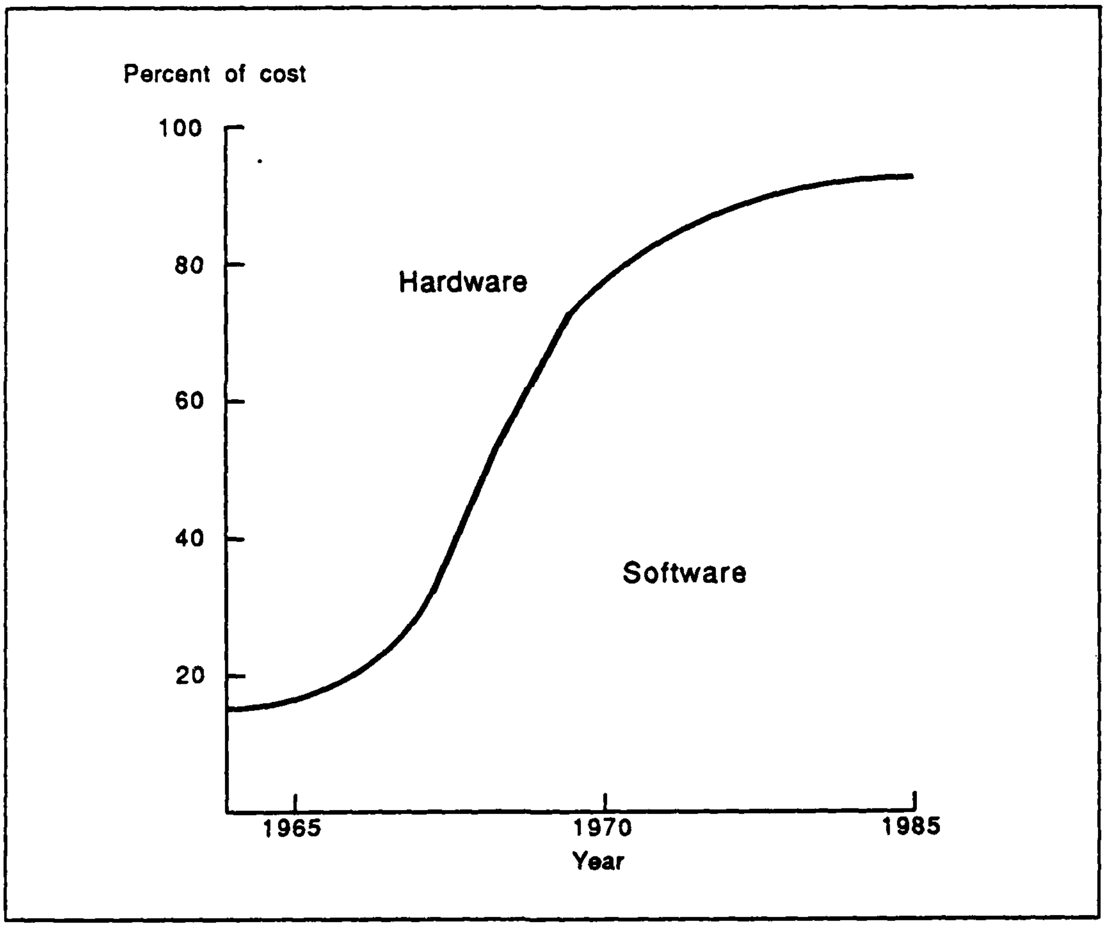


Figure 5. Harware/Software cost trends

The rising costs of software have been well documented in DOD. In 1973, software costs represented over 46 percent of the total DOD software budget [Ref. 12: p. 14]. It has been noted that DOD experienced a 51 percent increase in the direct costs of its computing systems, in spite of dramatic declines in the cost of hardware [Ref. 13: p. 3].

Unfortunately, productivity in software engineering has not kept pace with the growth in demand for computing systems and software applications. This is graphically illustrated in Figure 6 [Ref 14].

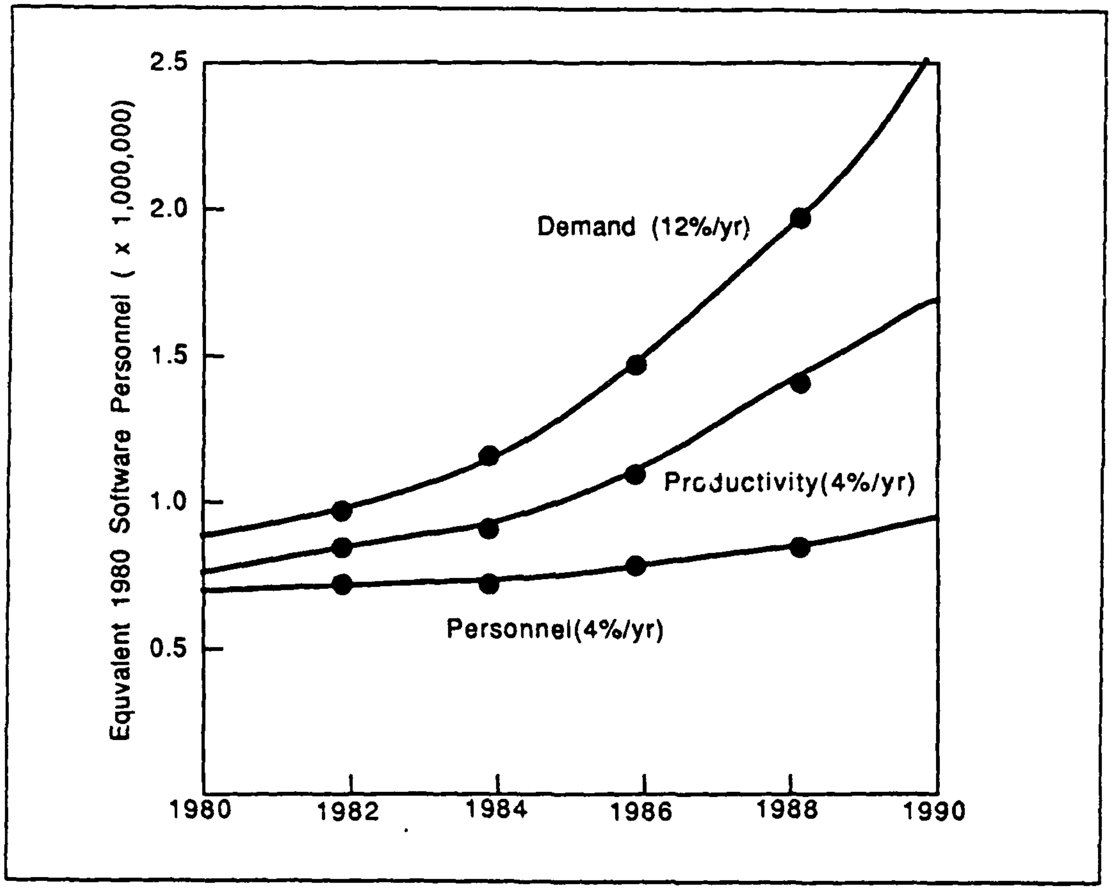


Figure 6. Software supply and demand trends

The figure shows that growth in demand for qualified software personnel is growing at a rate which outstrips their availability. Furthermore, the growth in productivity among software personnel also lags demand. It has been estimated that the average programmer, in the absence of modern programming methods, can produce six to ten lines of debugged code per day. This is influenced by a variety of factors ranging from programmer competence to the number of persons working on a project and the purpose for which the program is written. Fairley [Ref. 15: p. 17] states as a rule of thumb, that typical productivity levels for a programmer on a per day basis as a function of task complexity are:

- less than one line per day for systems programming
- 5 to 10 lines per day for utility programs
- 25 to 100 lines per day for application programs

It is a truism that, in general, a computing system is only as capable and reliable as the software employed in the system. In an age of incredible advances in hardware technology, the computing industry is hampered by slow gains in productivity in software engineering. Various sources of this situation have been cited. One element is the relative youth of the software engineering discipline in comparison to other engineering fields. Only three decades of experience and study support software engineering. These have been three decades of momentous change. The early leaders of the computing revolution were not native to the field. There has been a great deal of learning "on the job" for most software engineers. Until barely ten years ago there was a lack of rigor associated with program development and software engineering. As time has passed software engineers have recognized the need to develop a more rigorous approach to programming [Ref. 8: p. 268-269]. Even the relatively young field of electronics engineering is founded in the rigor and discipline of centuries of physical science and mathematics.

Another problem has been the failure to recognize the importance of human communication to the discipline of software engineering. Computing is a human endeavor, in support of human needs. Humans must be able to communicate those needs to the system developer, who in turn must express an answer to those needs in the computing system. If there is any failure of communication by either party the result will be a system that fails in one degree or another to meet the requirements of the human user.

These trends lead us to conclude that some effort must be made to achieve greater productivity and effectiveness in software engineering.

B. THE TRADITIONAL "WATERFALL LIFE CYCLE"

1. Characteristics

The traditional method of software engineering is the "waterfall life cycle." Figure 7 on page 15 shows a graphic representation of this approach. Under this schema, the customer perceives a need for a computing application for his operation or organization. He approaches a software developer and describes his problem. After some negotiation, the software developer determines what he believes the user's needs are and an agreement is reached to produce a computing package to meet the need. Contracts are let and the developer converts the customer's statements of need into precise (hopefully) functional specifications which can be implemented by the programmers. An architectural design is established based on some method of data flow or control flow. The system is then parceled out to programmers in manageable modules which each programmer is free to implement. As modules are developed they are assembled. When the system is complete then full scale testing and debugging of the system begins. If the system tests satisfactorily, the job is done and the system delivered to the customer for acceptance. Then begins the cycle of system maintenance. If the system fails or has numerous bugs (as is invariably the case with large systems) or if the system does not meet the functional specifications, or, worse, does not function as the customer expected, then the system must be restructured in various ways to correct the problem. This can be very costly, especially since tremendous amounts of manpower will have been already been invested at this point.

2. Difficulties With The Traditional Approach

Carrio [Ref. 9: p. 17] describes this life cycle as a three phase event consisting of:

- conceptual and definition phase (the requirements analysis phase)
- development phase (from functional specifications through test system)
- deployment and operational phase (maintenance and support)

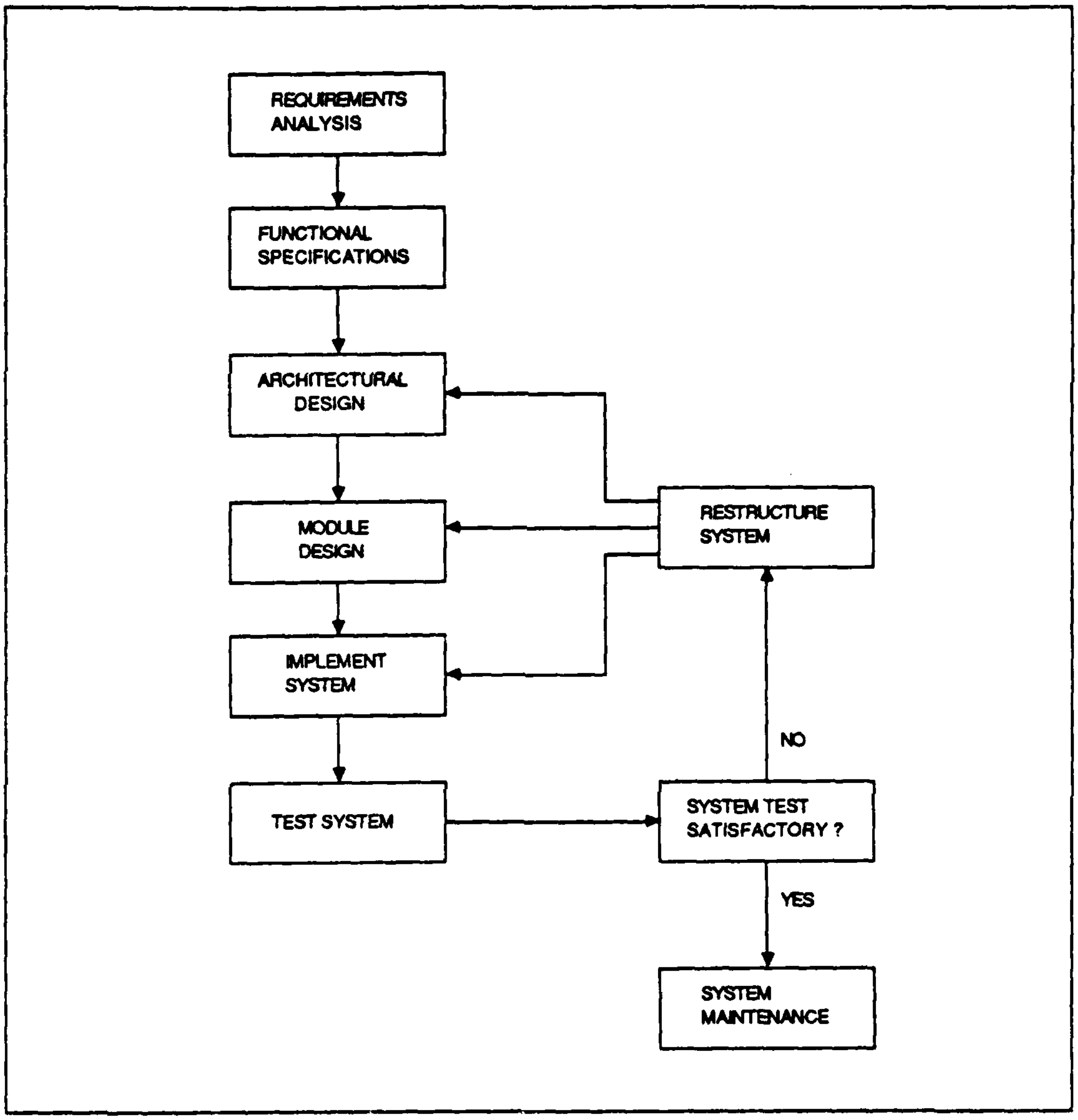


Figure 7. Traditional "waterfall" approach to the software lifecycle

He points out that the problem with this approach is the lack of interaction between the keepers of doctrine (the customers) and the developers in the early stages of the life cycle. Phase one is the province of the users. Phase two belongs to the developers and their supporting programmers and subcontractors. Then in phase three the two groups begin to interact in earnest. The key difficulty with this life cycle is communications -- the ability of the user and developer to communicate, understand and insure the integrity of the initial set or requirements. The question of whether the "as specified," the "as designed," the "as tested," and the "as built" systems are all the same must be asked again and again. Under this life cycle the answer is no [Ref. 9: p. 18]. Frequently this life cycle approach has led to cost overrun, late product delivery, and failure of the "as delivered" system to meet the needs of the customer. It may be concluded that the traditional life cycle is one source of difficulty in the struggle to achieve greater effectiveness and productivity in software engineering.

Several techniques have been proposed to improve upon the traditional life cycle. First of all, a rigorous design phase, in which customer requirements are exhaustively examined to produce a firm set of functional specifications which accurately reflect what the customer wants. These are used throughout the remaining life cycle as the standard for system development. Second, the use of prototyping in an automated environment to provide guideline models for the entire life cycle. Use of automated tools, AI/knowledge based systems, and various application support environments to aid the software engineer in developing, documenting, and maintaining the system [Ref. 9: p. 20]. This would be coupled with top down development and a structured approach to design to enhance system maintainability and reliability [Ref. 8: pp. 269-271].

C. RAPID PROTOTYPING

1. Description of Rapid Prototyping

An alternative to the traditional approach is rapid prototyping. Under the rapid prototyping paradigm, an effort is made to ensure that the customer and the developer both understand what the customer's requirements for a software system are. This schema is graphically illustrated in Figure 8 on page 17. In this approach, there is again a period of discussion with the customer to determine his requirements. The requirements are used to generate functional specifications. With the functional specifications, a prototype of the intended system is constructed and demonstrated for the customer. At this point the customer can decide if the prototype reflects the type system he had in mind; and the developer can see whether his perception of the customer's

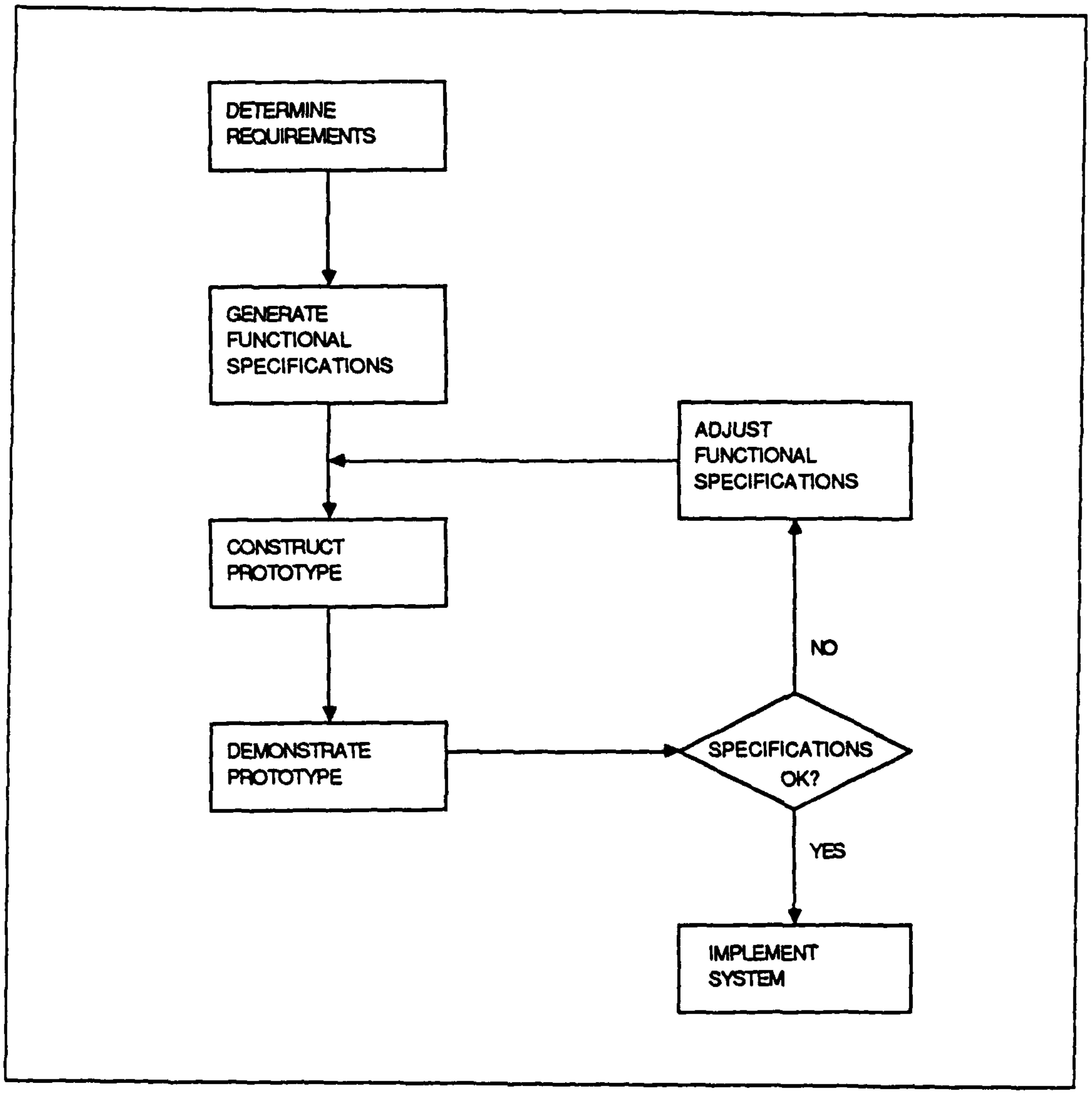


Figure 8. Rapid prototyping approach to software engineering

requirements was correct. Any adjustment needed in the functional specifications are made, the prototype system is recoded to reflect the adjustments, and the system is once again demonstrated. This process is repeated until the prototype behaves as the customer and the developer expect. Full scale development of the system is commenced once prototyping is completed. [Ref. 16]

2. Objectives of Rapid Prototyping

The iterative, rapid prototyping approach accomplishes several goals. First, it insures accurate communication between the customer and the developer. Due recognition to the difficulties of human interaction is given. The customer certainly knows his profession and has a clear mental picture of what he wants to accomplish with a computing system, but may not understand computing systems themselves. The software engineer understands computing systems but may not understand the world of the customer. They are both speaking English but may have no idea what each other is saying. Rapid prototyping seeks to cut through the communication difficulty by providing an executable model of the intended system which the customer can see. The customer will usually be able to recognize whether a working software system performs as he expects. This will ensure a stable set of requirements is achieved early in system development. [Ref. 1: p. 71]

Prototype construction aims to make efficient use of the designer's time. As such it differs from production software in which the goal may be driven by the need to optimize speed, or memory usage, or accuracy and ease of use. Production software is designed to be fault tolerant and capable of handling a wide range of error conditions. The prototype may not be fault tolerant at all. In all probability, it will not be optimized in performance. Prototyping the system generates a skeletal design framework which may serve as the initial design structure of the production version [Ref. 1: p. 71]. The early prototypes provide a traceable link between requirements, design, implementation and maintenance [Ref. 9: p. 20]. The use of prototypes aids in feasibility studies. Various methods of implementing portions of the system can be tested and the more promising methods can then be selected for implementation in the production system. Finally the prototyping approach aids in cost estimation. The cost of the final system will often be proportional to the final cost of the production version. [Ref. 1: p. 71]

D. IDEAS FOR INTEGRATED, AUTOMATED PROGRAMMING ENVIRONMENTS

In his ACM award winning dissertation, *Generating Language Based Environments*, Thomas W. Reps [Ref. 17: pp. 1-2] raises many salient issues regarding software engineering and software productivity. He observes that much of software development requires exhaustive attention to organizational detail. By this he means many things.

Among them are:

- the need to constantly be concerned with details of language syntax and semantics
- the accurracy of program entry
- the details of operating a series of software tools such as editors, compilers, linkers, debuggers, and library managers (all in the proper order)
- maintaining an audit trail of documentation for the system under development
- the necessity to communicate with others in the development process

All the while the system developer or programmer also hopes to perform creative intellectual work, yet it comprises a small part of his daily effort. The remainder of his time is eroded away by the mundane details of the job. A similar observation has been made by Fairley [Ref. 15: p. 12-13] and Brooks [Ref. 18: p. 16-18]. Reps goes on to point out that much effort has been expended to make the programmer's life easier; to shield him from the details and allow him to do creative work. The form of this help has characteristically been a series of automated tools such as editors, debuggers, parser generators and the like. These tools have provided some relief, and have aided productivity. However, they have generated problems of their own such as:

- learning to operate each of these independent tools
- employing the tools in the correct sequence when needed

Worse, the individual tools are not normally integrated with each other to take full advantage of computing power now available, and to automate away the maximum amount of detail, leaving the programmer completely free to pursue productive creative endeavor. Reps argues that to make true breakthroughs in this area it will be necessary to create an automated design environment incorporating all necessary tools under one coherent interface. He contends that such a system would be optimized to the particular language for which it is designed. This would be achieved by designing an integrated environment which "understands" semantics of the programming language being used in it.

Reps then presents the development of a *Synthesizer Generator* whose purpose is to generate language-based edtiors for different programming languages. The tool uses a specification of the display format, syntax, and static semantics of the language to be edited. The objective is to create an editing environment which will prevent entry of incorrect syntax while the programmer is editing the program. The primary concern of the Reps dissertation is developing a framework for the semantic component of the language based editor. He discusses various methods to generate a programming environment from an attribute-grammar description of a language. Reps also discusses what attribute grammars are and discusses several algorithms for attribute evaluation. He then shows how the semantic component of a language-based editor can be developed from an attribute grammar description and discusses some of the problems created by using attribute grammar based development systems, chief of which is the extravagant use of storage resources. [Ref. 17: p. 4]

Several ideas in Reps work have impact on the design features envisioned for the CAPS. These include:

- incorporation of an "intelligent" editor environment which will aid the program designer in entering the prototype description correctly
- integration of all the tools necessary for program prototyping under one coherent interface.
- use of attribute grammar based approaches to language description.

There are similarities and differences in what Reps does and in what is aimed for in the CAPS generally and in the Translator in particular. Reps is specifically concerned with development of editing environments based on attribute-grammar descriptions of a language. CAPS is concerned with incorporating an intelligent editor along with numerous other tools in order to remove a great deal of the mundane drudgery from software development. Reps uses attribute-grammar approaches to develop editing environments. In this thesis, an attribute grammar based tool is used to develop a translator which can convert PSDL into Ada.

Reps' work sets a direction for future programming development environments. It helps reveal a promising application for the concept of attribute-grammars. It demonstrates the practical application of important theoretical concepts to the problems of productivity in software engineering.

E. DESCRIPTIONS OF A COMPUTER AIDED PROTOTYPING SYSTEM

A general description of a CAPS is provided in two papers. First is the technical report, *A Computer Aided Prototyping System,* by Luqi and Ketabchi [Ref. 1]. Second is the technical report, *Research aspects of Rapid Prototyping,* by Luqi [Ref. 16]. These papers describe the overall concept of a CAPS. They lay out an architectural design for such a system and provide a starting point for the research in this thesis.

The CAPS would provide an integrated environment for the development and testing of prototypes of software systems. It would be specifically designed to address systems which were large, embedded, and had hard, real-time constraints. It would make use of the Ada language, and would employ a database system to store and recall both reuseable software components in the Ada language, and previously designed prototypes in the PSDL language. A system to automatically translate the PSDL descriptions of a system into Ada code and compile them so that they could be executed to demonstrate the prototype would be provided. The CAPS would be based on two ideas which would establish the fundamental character of the system. One is the methodology of rapid prototyping, the other is a language (PSDL) specifically designed for writing prototype designs of systems with hard, real-time constraints. PSDL would give expression to the methodology of rapid prototyping and form the core of the CAPS.

F. THE PSDL LANGUAGE AND RAPID PROTOTYPING

The central paper on the PSDL language and the application of the rapid prototyping methodology is Luqi's Ph.D. dissertation, *Rapid Prototyping For Large Software System Design* [Ref. 19]. Four related papers have been published which provide similar detail on the nature of PSDL and rapid prototyping. These are:

- *A Prototyping Language for Real Time Software* [Ref. 20]

- *Rapid Prototyping of Real-time Systems* [Ref. 21]
- *Languages for Specification, Design, and Prototyping* [Ref. 22]
- *Execution of Real-Time Prototypes* [Ref. 23]

The *Execution of Real-Time Prototypes* paper is a short technical report prepared for the Naval Postgraduate School. It very briefly summarizes the concept of CAPS and the rapid prototyping methodology. The remaining papers are closely related in content and purpose to one another, and are separated by the depth to which they examine the subject from the technical report.

Th(seminal paper among the remaining papers is the Luqi Ph.D. dissertation. The paper begins by introducing the PSDL language. An extensive discussion of the CAPS system is set forth. The various components of the PSDL language are presented. The application of rapid prototyping to a system developed using PSDL is discussed in some detail. There is a brief discussion of the implementation of various PSDL language components within the ESS, and a discussion of the functions of the SS, DS, and TL. An example of a PSDL prototype is presented. Finally, a summary of PSDL syntax in BNF form is provided.

The BNF summary of PSDL syntax is included as Appendix A of this thesis. From the standpoint of translator design, the most important sections of the dissertation, are section 2, on PSDL language elements and the discussion, in section 4, on how certain PSDL elements might be implemented by the Translator. Since the objective of this paper is to develop a Translator, section 4 of the Luqi dissertation provides the foundation for chapter 3 and 4 of this thesis.

Two of the papers are available in published journals. The paper, *A Protoyping Language for Real-Time Software* [Ref. 20], is essentially a reprise of the information presented in the Luqi thesis, without the BNF diagrams for PSDL. The paper presents a detailed description of PSDL and its employment under a rapid prototyping paradigm.

Rapid Prototyping of Real-Time Systems [Ref. 21] presents an abbreviated discussion of PSDL and its use in a rapid prototyping setting. Less emphasis is placed on the specifics of PSDL syntax and language elements, and more on the general model and concepts involved in employing PSDL under the rapid prototyping methodology. The paper serves as an excellent introduction to the fundamentals of PSDL and rapid prototyping in the CAPS environment.

Languages for Specification, Design, and Prototyping [Ref. 22], is an extensive presentation of the current state of language development in the three separate areas of specification, design, and prototyping. The authors distinguish between the three goals and discuss the characteristics of a languages aimed at satisfying the demands of each of the particular areas. Discussions and illustrations of various currently available languages are presented. The paper is an excellent general discussion of issues involved in selecting a language for a particular purpose. The paper points up the different problems associated with each approach to software production and demonstrates possible solutions. PSDL is presented as a good general purpose language for specification, design, and prototyping. PSDL has many features which make it convenient for use with Ada including:

- is an executable language construction unlike many specification or design languages which are not
- supports a modular approach to program design.
- supports data, control, and operator abstraction
- supports exception handling, separate compilation of generic units, and use of reuseable components.

G. ATTRIBUTE GRAMMARS AND TOOLS

The objective of this thesis is to generate a translator which will read a PSDL source file and produce and Ada source file. This might prove a daunting task were it not for the availability of an automated translator generator tool called Kodiyak [Ref. 24]. The Kodiyak system requires as input, an attribute grammar (AG) description of the source language. It is proper to consider some literature which addresses AG's in general, and the Kodiyak in particular.

1. Attribute Grammars: What Are They?

The classic work on AG's, is *Semantics of Context-Free Languages* [Ref. 25: pp. 127-145]. The paper sets forth " . . . a technique for specifying the "meaning" of languages defined by context-free grammars . . . ." [Ref. 25: p. 127] It is assumed that the language is "context-free". That is, the "meaning" of any string or element in the language is independent of the context in which it is used. This is usually not the case for natural languages (e.g., English, et al.), but often is the case for programming languages. It is asserted that the "meaning" of any string in a context-free language can be determined ". . . by defining "attributes" of the symbols in a derivation tree for that string." [Ref. 25: p. 127] If the production rules for a given language are known, it is possible to assign functions to each of the production rules which define the "attributes" of a given symbol or combination of symbols. The attributes may be developed in one or both of two ways. They may be "synthesized", defined in terms of their descendants; or they may be "inherited", defined in terms of their ancestors [Ref. 25: p. 128]. Colloquially, synthesized attributes are developed from the bottom up in the derivation tree, while inherited attributes are developed from the top down. Once all the attributes of all the symbols in the string are known, the "meaning" of the string is known. These simple but powerful concepts form the foundation of AG approaches. Knuth presents an applicative example of these principles as the first part of his paper [Ref. 25 pp. 128-130]. The remainder of the paper is devoted to the mathematic and formal properties of the technique, and another example of how the method can be applied to programming languages. Finally, Knuth compares his method with other known methods of semantic definition.

For the purposes of this paper it is possible to summarize Knuth's work. First, suppose there is a language for which there are a set of production rules. PSDL is such a language, with a context-free grammar and a set of production rules in the form of BNF diagrams for the language. Then to determine the "meaning" of any string constructed according to those rules, it is necessary to:

1. parse the string into its component parts and create a derivation tree of the string
2. create a set of functions (equations) which assign meaning to each of the components of the string
3. reduce (determine the meaning of) the string based on the BNF rules and the meaning of each of the components

The Kodiyak system allows the application of the technique in a practical and convenient fashion to real problems. Detailed discussion of the AG approach will be deferred to chapter four of this thesis. Suffice it to say, that AG's have been used for a variety of purposes, among them, the construction of compilers, pretty-printers, and translators. Knuth's short paper is at once the cornerstone and keystone of a whole area of software engineering research.

2. An AG Based Tool For Translator Generation

The effort required to produce a translator of the type desired for the CAPS is considerable. Fortunately, a tool has been developed which makes possible the automatic generation of translators. That tool is the Kodiyak system. Kodiyak is an AG based tool developed by Robert M. Herndon as a Ph.D. dissertation at the University of Minnesota [Ref. 24]. The Ph.D. dissertation provides exhaustive details on the technical aspects of translator generation, the operation of AG based systems, and the design and construction of Kodiyak. Another work on the Kodiyak is *AG: A Useful Attribute Grammar Translator Generator* [Ref. 26]. Although it refers to an earlier version of the Kodiyak (then known as AG), it provides a useful description of the Kodiyak system. The most useful work is *The Kodiyak Reference Manual,* which is an appendix to the dissertation [Ref. 24: app. 1]. This is a detailed reference manual describing how to employ the Kodiyak to generate a translator.

Kodiyak itself is ". . .a language designed for constructing translators [Ref. 24: p. 1, app 1]" It is AG based. "The Kodiyak translator accepts a context-free grammar along with such attribute declarations and equations, a scanner specification, and output declarations, and generates the described translator [Ref. 24: p. 1, app 1]." Kodiyak works on many Unix®[2] based systems. It requires the use of various resident utilities. A C library and compiler, the LEX (lexical analyzer) [Ref. 27] and the Yacc (yet another compiler compiler) [Ref. 28] must be present in order to use Kodiyak. The system is very effective and is presently in use at this institution to develop a pretty printer, as well as the translator presented in this thesis. It is presently in operation on a Vax®[3] 11/785 and a Sun®[4] 3/50 diskless workstation. The present translator is being developed on the Sun station.

[2] Unix is a registered trademark of Bell Laboratories.

[3] VAX is a registered trademark of the Digital Equipment Corporation.

[4] SUN is a registered trademark of Sun Microsystems Incorporated

There are only a few significant difficulties with the present Kodiyak. First, the system requires a great deal of storage, and a great deal of cpu time. The translator listing for the CAPS, presented in appendix C, requires about five minutes to compile on the Sun station. This is a station dedicated to the translator work and is otherwise idle. On the Vax 11/785, with normal user loads, the same listing requires about 10 minutes to compile. The five minute figure on the Sun station represents actual cpu time. Second, the error messages and error handling in the system is not always as helpful as it could be. Error messages often refer to temporary files created by LEX or Yacc and not to the original source file. Also, when Kodiyak scans the input file, it may allow certain error conditions to pass through which will later be fatal during Lex or Yacc scans. Typical of this type error is a mispelled variable name. So long as Kodiyak finds correct syntax in the input file it will allow the file to be presented to Lex and Yacc for processing. A mispelled variable name will result in a fatal crash of the Yacc scan and may be fatal to the Lex scan. Ideally Kodiyak should trap any errors of this type and exit immediately so that the user can correct the problem before the time consuming LEX and Yacc scans begin. Nevertheless, Kodiyak is powerful and significantly eases the effort required to construct the translator.

The Kodiyak operates by taking an input file which is an AG description of the input language and the attribute equations which relate the input language to the output language. After scanning the file to insure it is in correct Kodiyak syntax, the file is passed to Lex and Yacc for processing. The end result is an executable translator, compiled in the C language. This translator can accept textfile input and will produce textfile output.

III. IMPLEMENTATION AND DESIGN CHOICES

A. CAPS

Prototype System Description Language (PSDL) provides the backbone of the CAPS for design and specification, while Ada was chosen as the language for implementation. The basis for this choice is found in the characteristics of the languages chosen. Each offers advantages and disadvantages for the design, specification, and implementation of hard real-time, large, and embedded systems. Alone, each presents difficulties in use. Used together in CAPS, the two languages experience a symbiosis, which results in flexibility, power, and ease of use for the system developer. The same power, convenience, and ease of use are available for the development of CAPS itself.

1. Implementation Questions for CAPS

CAPS is under development and not yet fully implemented. This paper aims to demonstrate a working prototype for the CAPS translator. Several other papers are in progress which specifically address other aspects of the system. The capabilities envisioned for CAPS are extensive.

- How can it achieve them?
- What is the foundation of the system?
- Why is that choice of foundations better than others?
- Why is Ada not sufficient in itself to achieve hard, real-time system design and implementation?
- What are the general properties of real-time systems that demand a tool like CAPS?

These questions and others form the basis of this chapter.

B. FOUNDATIONS FOR CAPS

1. Prototype System Description Language (PSDL)

PSDL is the foundation on which CAPS is being built. It is a language designed to support construction of large and embedded systems and those with hard, real-time constraints.

a. Embedded and Real-Time System Properties

Embedded and hard real-time systems have several general properties which place special demands on the designer and his language tools. These properties are:

1. Often large, running to millions of lines of code and thousands of modules
2. Often operated in a multiprocessor environment
3. Under the DOD concept, their primary function is often not computing but controlling or monitoring the operation of complex or safety critical systems
4. Generally have requirements for high reliability, and penalize the user severely upon failure (loss of aircraft and crew, loss of control of critical manufacturing or industrial process, etc.)
5. Expect to be employed over an extended lifetime, with periodic updates and modification to maintain currency
6. Are too large for a single individual to understand or program alone but require the efforts of teams of programmers and maintenance personnel
7. Often require hard, real-time constraints in operation (i.e., operational schedules and deadlines within the program in response to real world conditions) [Ref. 12: p. 15-16]

These characteristics demand several features of a prototyping language which are summarized as follows:

1. Should have a simple computational model which limits and exposes the interactions between modules and is consistent with the prototyping methodology
2. Should produce executable prototypes
3. Should be simple and easy to use
4. Should support hierarchical design to simplify construction of large, complex systems
5. Should apply at both specification and design phase, thereby providing a unified notation to the user
6. Should provide specifications suitable for retrieval of reuseable modules from a software base
7. Should support data abstraction, control abstraction, and function abstraction
8. Should contain abstractions which can be used to construct real-time systems [Ref. 19: p. 10]

b. Why Use PSDL?

PSDL and Ada both approach the design of software in the same manner. There are several advantages to employing PSDL in the CAPS over using Ada directly. First, PSDL is a much simpler language. Its grammar (see Appendix A) is very small, compared to the Ada grammar which is very large. The compactness of PSDL allows its use as a tool with which to search a software base by automated means for previously written modules which will implement the designer's objectives. The designer does not need to know what units are available. The CAPS will search for Ada components in the software base for him, and will incorporate them into the prototype as long as they match the PSDL description. Second, CAPS will use the PSDL description to produce a graphic representation of the prototype program's hierarchical structure. PSDL is a distillation of the Ada language's constructs. Third, the CAPS translator will automatically generate interconnections for Ada procedures to implement PSDL operators.

c. PSDL Computational Model

PSDL supports the specification and design of hard, real-time and embedded systems with a simple and executable computational model. PSDL models software systems as a set of OPERATORS communicating via DATA STREAMS. The formal computational model is an augmented graph:

$$G = (V,E,T(v),C(v))$$

where:

- V is the set of vertices
- E is the set of edges
- T(v) is the maximum execution time for each vertex
- C(v) is the set of control constraints for each vertex v

Each vertex represents an operator while each edge represents a data stream. Components V, E, and T(v) are called the ENHANCED DATA FLOW DIAGRAM. [Ref. 19: p. 11]

2. Major PSDL Language Structures

a. Operators

In PSDL, Operators may be either atomic or composite. Composite operators can be decomposed into two or more operators, each of which may be composite or atomic. Atomic operators cannot be decomposed into simpler components. This is a colloquial rather than formal distinction. It envisions a hierarchical breakdown of the system into logical components which are as simple as possible without becoming trivial. No special rules for decomposition are imposed. This distinction allows the modeling of hierarchically structured programs as sets of operators. Operators at higher levels in the program structure are composite while those at the lowest level of the program structure become the atomic operators. PSDL can therefore be used to support top down design strategies.

A second classification considers that operators may be data driven or periodic. Under this schema, the firing of a data driven operator is accomplished due to the presence of data in its input data stream(s), while the firing of a periodic operator is dependent upon timing constraints which must be met during program operation. The data driven operator allows the modeling of systems which utilize data flow as a means of control instead of the more traditional timing control in real-time systems. In either case, when an operator fires, it reads one data object from each of its input streams and writes, at most, on object to each of its output streams.

A third classification of operators is allowed. An operator may be either a function or a state machine. This description relates to the values output from the operator. The output value of the function type operator is dependent solely on the current set of values present on the input streams to the operator. The output of the state machine type depends, not only upon the current set of input values, but also on the values of a finite number of state variables internal to the operator. Figure 9 on page 31 illustrates several aspects of the PSDL operator concept.

Each of the preceding operator classifications can be directly related to existing concepts in Ada. Ada supports both top down and bottom up design strategies in a hierarchical, modular program structure. PSDL allows the description of each module as an operator. In Figure 9 on page 31 A is an operator with one input stream,

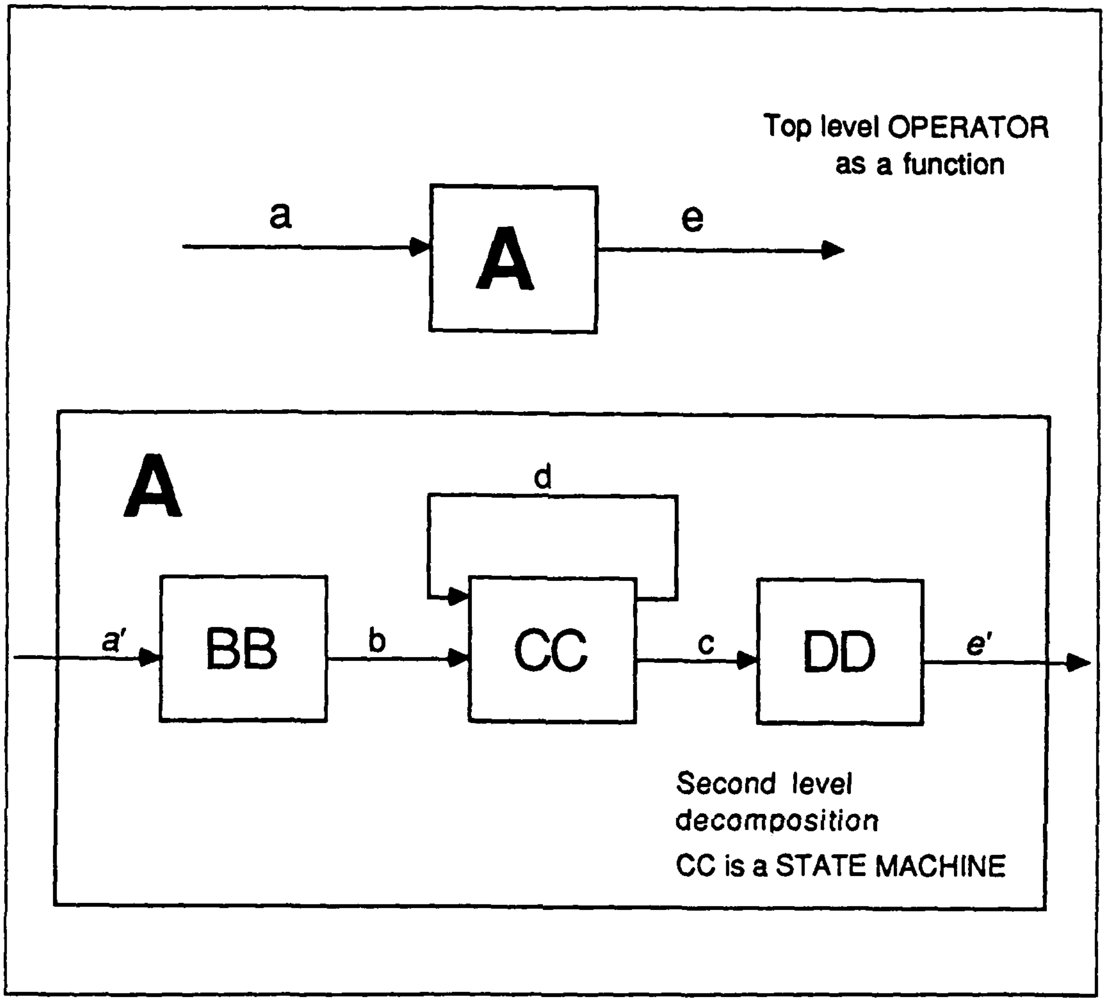


Figure 9. Various types of PSDL operators

a, and one output stream, e. In this case A is a function since no state variables are seen. A is also a composite operator which can be decomposed into three operators, BB, CC, and DD which are atomic operators (they are not or cannot be decomposed further). In this representation, CC is a state machine, since it has state variable, found on data stream d, which is combined with the value on its input stream, b, to generate the output value on data stream c.

At the lower level of decomposition, A still exists, but is represented in greater detail by the three atomic operators and their associated data streams. The input data stream to **BB** is **a'**. The data type and value found on **a'** will be the same data type and value found on **a**, and similarly for **e** and **e'**. This structure is analogous to an Ada program being composed of one or more subprograms. For example, we might use an Ada procedure to represent A. This procedure might contain three Ada subprograms (functions or procedures) which are called within it to implement A. Procedure **DD** would produce value which would be passed to A on an output parameter of **DD**. This would be passed out of A as a value on an output parameter of A. In Ada, each of the operators could be separately compiled. **BB**, **CC**, and **DD** could be written first, then A written and compiled (bottom up), or the specification of A could be written and compiled, then the specifications of **BB**, **CC**, and **DD**, and finally the implementation code for each of the operators could be written (a combination of top down and bottom up).

In the model shown in Figure 9 on page 31, the arrows represent data streams. Each of these is labeled with a lower case letter. The label is a name for the data stream. PSDL data streams can carry two types of data values. The first type may considered the normal type. Normal type data can be any abstract data type. It is characterized by being immutable and no global representations are allowed. This feature prevents coupling problems within the prototype where operators communicate via shared data. State variables for an operator are specifically local to the operator and can only be changed internal to their own operator. This also prevents coupling problems in the prototype design. PSDL uses the immutable subset of built in Ada types, plus user defined types, and the special types TIMER and EXCEPTION.

The second type of data which can be transmitted are tokens representing exception conditions. This is the PSDL type EXCEPTION and corresponds to the Ada exception construct. Thus, PSDL uses the Ada approach of representation hiding and data abstraction in program design. It is much simpler to use PSDL than to use Ada directly. For the translator, all variables, including user defined types, will be placed into an Ada package. The resulting Ada program will employ the with/use construct from Ada to make these variables available to the program.

b. Data Streams

In PSDL, data streams represent a communication link between exactly two operators. One operator is the producer of the data while the other is the consumer of the data. There are two types of PSDL data streams. One is the DATA FLOW STREAM the other is the SAMPLED STREAM. The DATA FLOW STREAM can be thought of as a first in first out (FIFO) queue capable of holding, at most, one data value. This data value may be used one time by the consumer operator. It may not be overwritten by the producer. In effect, this stream guarantees deliver of the data value, and guarantees that each individual data value will be read once and only once. The second type queue can also be thought of as a queue of length one. In this case, (the sampled stream), delivery of an individual data value is not guaranteed. The data value may be overwritten by the producer before the consumer reads it, or may be read multiple times by the consumer, or not at all. The choice of data stream is dependent upon the control conditions specified for the operator.

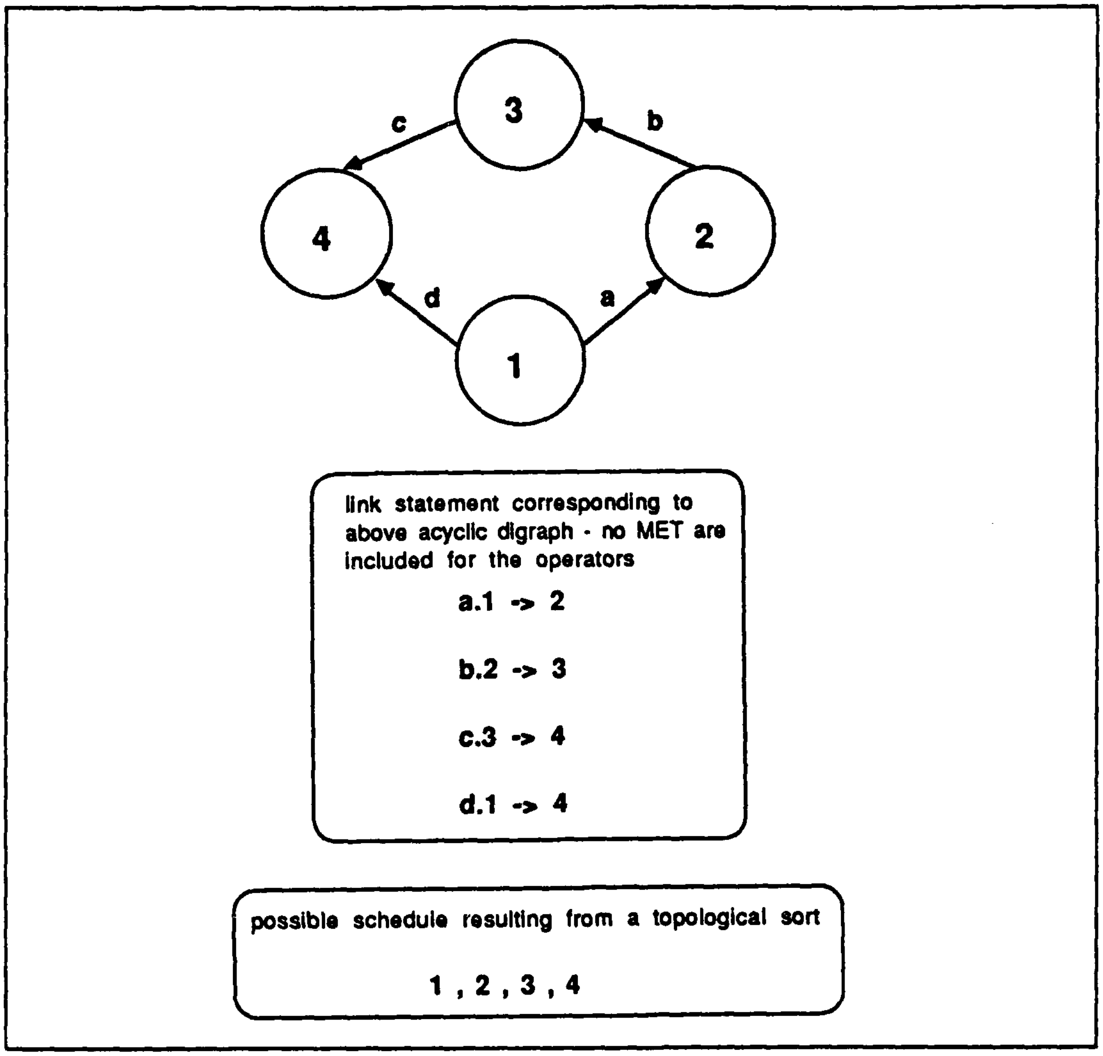
c. Operator Control Techniques

Two types of control are allowed in PSDL. The first is periodic. This is a common form of operator control in which operators are fired by some regular schedule. This form of control is supported in PSDL by several constructs. The primary construct is PERIOD followed by a time value. The SS in the ESS will recognize the PERIOD token and will utilize the time value supplied to generate an Ada schedule program which will invoke the Ada procedure representing the PSDL operator. The periodic operator must fire sometime between the beginning of the period and some deadline which defaults to the end of the period [Ref. 19: p. 17]. Thus, PERIOD is an upper bound on the length of time allowed between any two firings of a given operator. This is an explicit period.

It is possible to arrive at an implicit period. Such an implicit period would be known as an equivalent firing period. An operator for which an equivalent firing period would be calculated by the SS would not contain the PERIOD token. It might inherit a period from a higher level of decomposition in a hierarchical prototype or it might contain PSDL tokens for MAXIMUM EXECUTION TIME (MET), MAXIMUM RESPONSE TIME (MRT), or MINIMUM CALLING PERIOD (MCP) which would result in the SS calculating an equivalent firing period for the operator. MET is an upper bound on the length of time which may elapse from the beginning of execution of a module to the end of the execution of that module [Ref. 19: p. 20]. MET may be applied to all operator types.

MRT has two different interpretations. The first applies to periodic operators. In this case, MRT is an upper bound on the time from the beginning of a period and the time when the last data has been output onto the output stream of the operator [Ref. 19: p. 20]. The second case for MRT applies to a class of operators known as Sporadic operators. Sporadic operators lack an explicit PERIOD. Sporadic operators are triggered by the arrival of data on the input streams of an operator (or set of data streams for the NATURAL DATA FLOW (NDF)) [Ref. 19: p. 20]. NDF is a form of control dependent on the flow of data through the prototype to cause the firing of operators. For the Sporadic operator, MRT is an upper bound on the elapsed time from the arrival of new data on the input streams to the operator and the time when the last data value is placed on the output stream of the operator in response to the arrival of the new data values. MCP is a lower bound on the elapsed time allowed between the arrival of one set of values on the input streams of an operator and the arrival of the next set of values on the input streams. For SPORADIC operators, if MRT is used, then MCP must also be used [Ref. 19: p. 20].

For sporadic operator control PERIOD is not specified. The SS calculates an equivalent firing period if the operators have the MET token. It uses the information calculated to generate a calling schedule for program operation just as SS would if the program used the PERIOD token and were therefore periodically controlled. If the operator is sporadic and does not contain MET then the SS will conduct a topological sort of the operators to determine a calling schedule In Figure 10 on page 35 we see the application of the topological sort to a set of operators. The information required for the sort is found in the link construct of PSDL which is part of the GRAPH token.
The acyclic digraph is generated from the link information. In the case of Figure 10 on page 35 no MET information is supplied in the link construct. In Figure 11 on page 36 MET information is supplied within the lirk construct. The resulting schedule for each set of operators is the same.



link statement corresponding to above acyclic digraph - no MET are included for the operators

a.1 -> 2

b.2 -> 3

c.3 -> 4

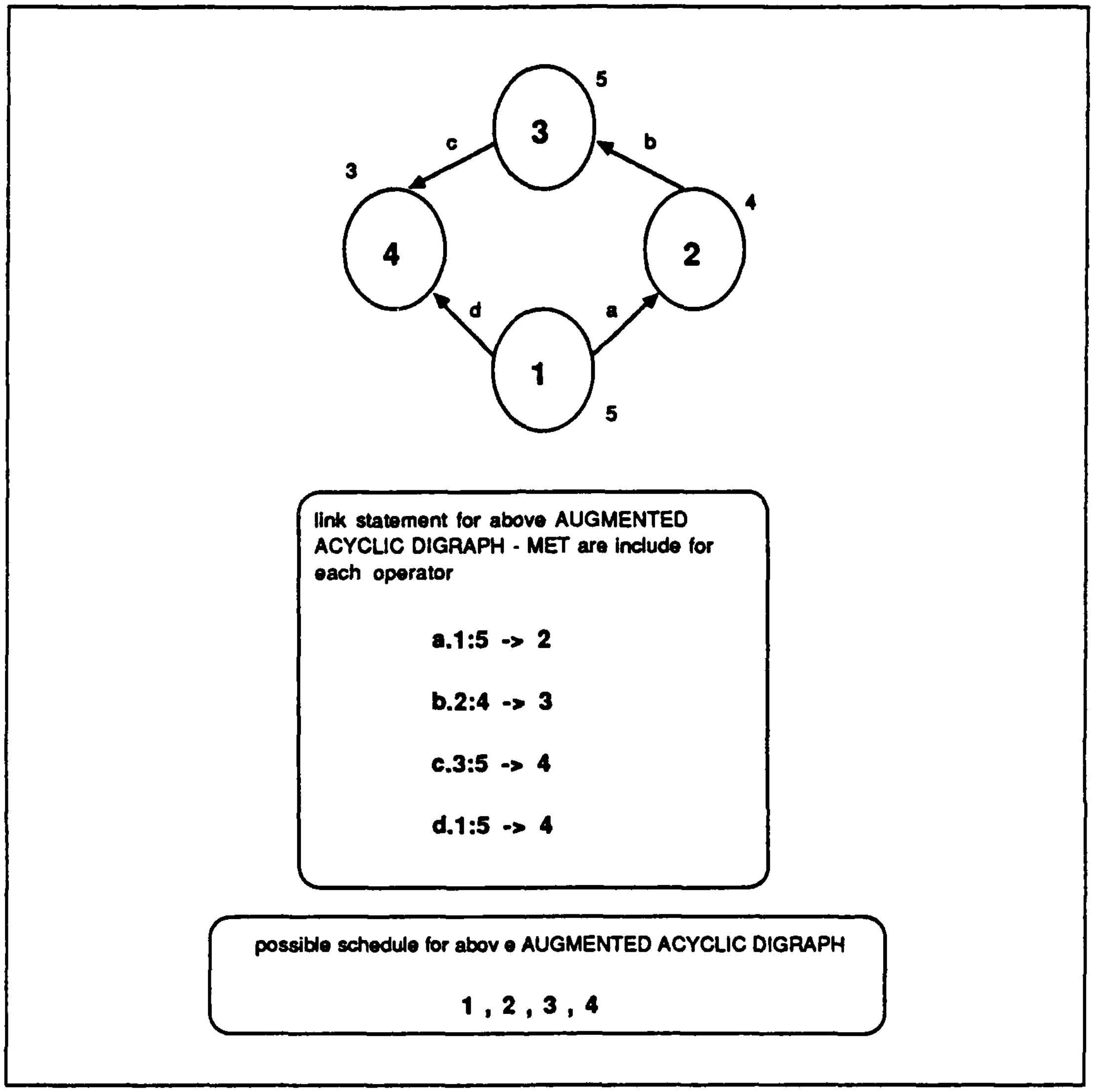
d.1 -> 4

possible schedule resulting from a topological sort

1 , 2 , 3 , 4

Figure 10. Acyclic Digraph

NDF control of sporadic operators is signified by the PSDL token TRIGGERED BY. This token will be qualified by either the additional token ALL or SOME. TRIGGERED BY ALL indicates that an operator is to be fired when new data values have arrived on all the input streams to the operator. TRIGGERED BY SOME implies that the operator will be fired by the arrival of a new data value on any one of the input streams to the operator. Figure 12 on page 37 illustrates these two different constructions. Note that the designer must specify which input streams the TRIGGERED



link statement for above AUGMENTED ACYCLIC DIGRAPH - MET are include for each operator

a.1:5 -> 2

b.2:4 -> 3

c.3:5 -> 4

d.1:5 -> 4

possible schedule for abov e AUGMENTED ACYCLIC DIGRAPH

1 , 2 , 3 , 4

Figure 11. Augmented Acyclic Digraph

BY ALL/SOME construction refers to. He may specify a proper subset of the input streams in either case. In this way, if an operator has multiple input streams, but only a few of them are critical to the firing of the operator, the designer may so specify. NDF is not normally combined with periodic control. The application of timing control to a model using NDF is allowed. The MRT and the MCP tokens may be used with the NDF form of control among SPORADIC operators.

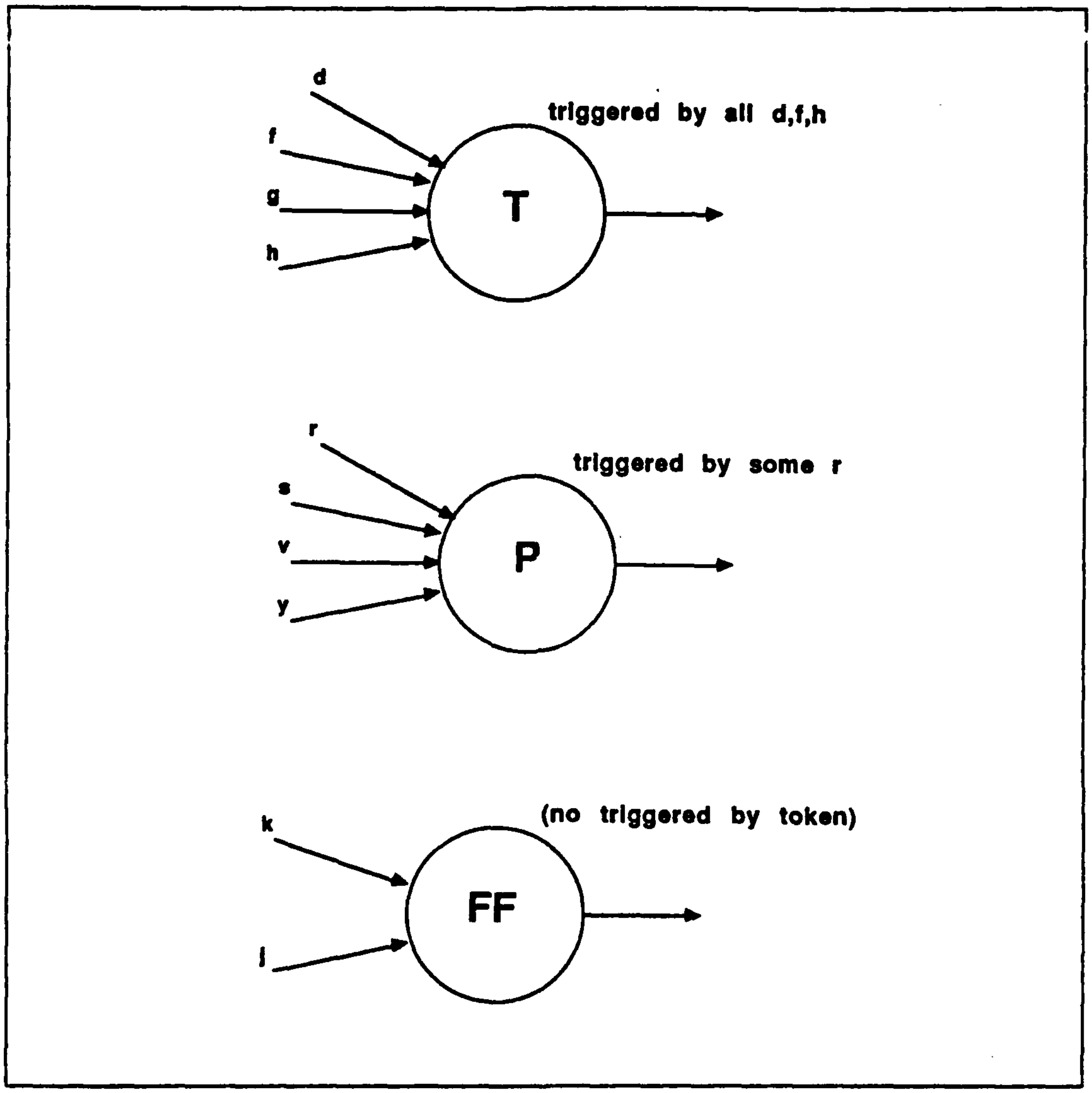


Figure 12. "Triggered By" construction in PSDL

Figure 13 on page 38 illustrates the combination of Sporadic and Periodic control. In this case, a conflict develops between the two schedules developed on the basis of:

1. Topological sort
2. Periodicity constraints

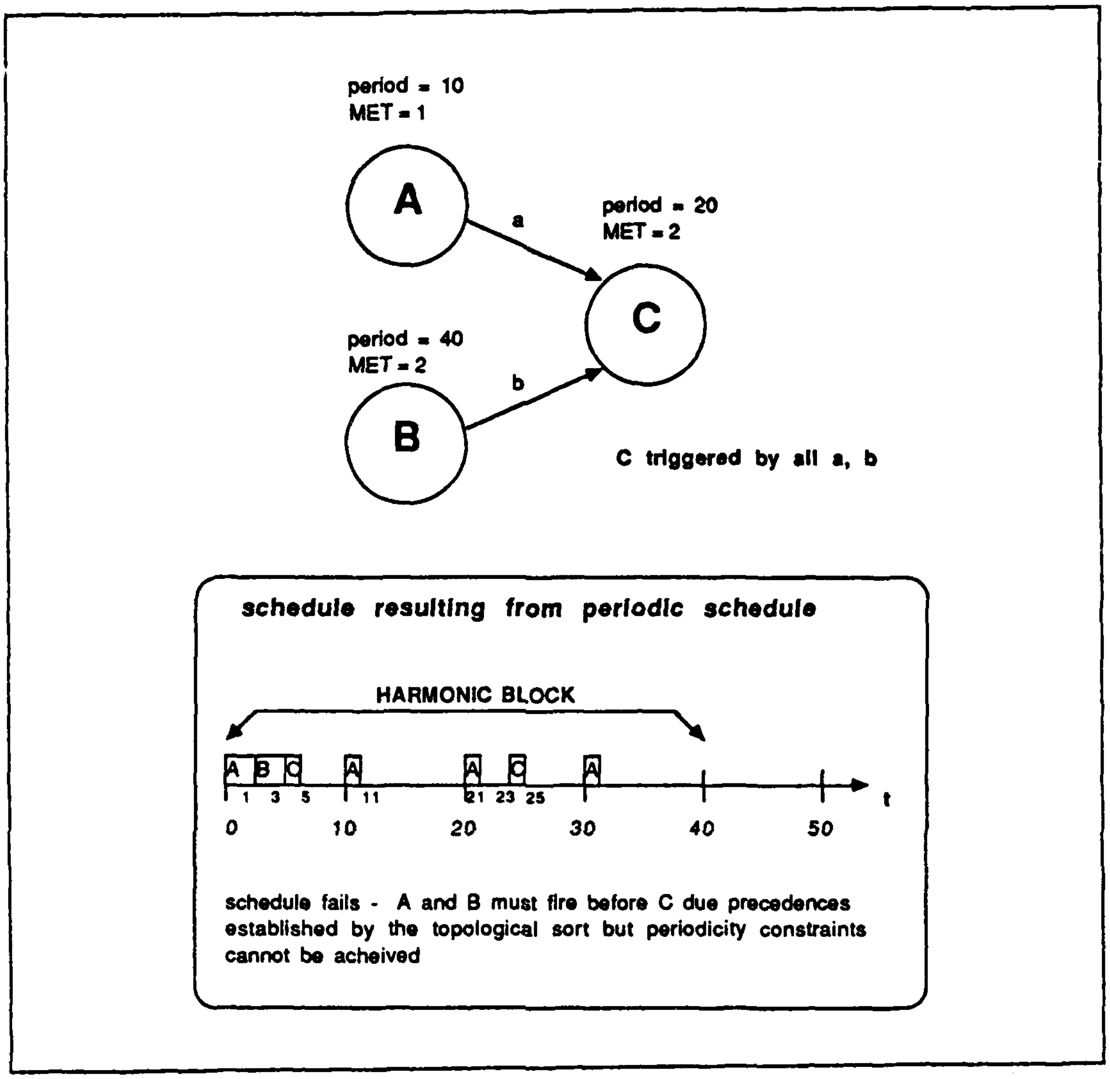

period = 10
MET = 1
A
a
period = 20
MET = 2
C
period = 40
MET = 2
b
B
C triggered by all a, b
schedule resulting from periodic schedule
HARMONIC BLOCK
A
B
C
A
A
C
A
1
3
5
11
21
23
25
0
10
20
30
40
50
t
schedule fails - A and B must fire before C due precedences established by the topological sort but periodicity constraints cannot be acheived

Figure 13. Combination of Periodic and Sporadic Operators

The SS would develop a schedule based on the periods specified. It would also develop a topological sort. It would compare the two schedules and would recognize that they do not match and might fail. It would nevertheless allow the program to be compiled and run on the basis of the periodic schedule which would fail when C attempts to fire a second time before B has fired a second time. This indicates a flaw in the design of the prototype and would require the designer to intervene to correct the problem.

It is not the purpose of this paper to discuss in detail the development of schedules from the PSDL specification. The aim is to demonstrate that PSDL has a powerful set of language constructions to deal with real-time constraints in software systems. PSDL offers a variety of means to control the operation of a real-time systems. It is necessary to discuss the forms of control available so that certain implementation aspects for the translator can be introduced. It is also important to recognize that Ada is not nearly so flexible in describing real-time constraints as is PSDL.

Conditional firing of operators can be accomplished by the addition of input or output predicates in the PSDL specification. Referring to Figure 9 on page 31, the designer might specify one of the following:

- OPERATOR A TRIGGERED BY ALL a IF a:critical
- OPERATOR CC TRIGGERED BY ALL b IF b:NORMAL AND d:critical

This illustrates the use of an input predicate. The triggering condition acts as a guard for the operator. The conditional can be applied to both Sporadic and Periodic operators. A Periodic operator would fire only if the input predicate were true. If it were not true, the Periodic operator would read the inputs without firing. The input conditional can depend only on the input values to the operator and any TIMER values.

An example of an output control would be:

OPERATOR DD OUTPUT x IF x > 100

This functions as if we had an explicit, conditionally executed filter operator following it [Ref. 19: p. 19]. The output guard provides a convenience to the designer but could be simulated by adding another operator to the prototype with an input condition on its firing.

d. Timer

TIMER is a PSDL construct which is useful in the development of real-time systems. A timer is an abstract state machine. In PSDL it is somewhat like a stopwatch. It has the primitive operations of START, STOP, RUN, and RESET. It is used for such things as measuring the length of time between two events, or the length of time the system or an operator has remained in a particular state. TIMER does not function in the same way as a clock construct for an operating system. It does not provide direct control of operator firing, but can be used as a value for a PSDL input or output conditional to act as a guard to the firing of an operator. It is primarily provided to collect statistics about the prototype system.

e. Exception

It was noted above that PSDL supports both normal and EXCEPTION data types. The PSDL EXCEPTION is a built in type. It can be transmitted on any data stream as a data value. It can be suppressed by the use of input or output conditionals. It can be handled in PSDL or in Ada. Some possible operations for the PSDL EXCEPTION are

- to create an exception with a given name
- to detect if a value on a data stream is
  an exception with a given name
  normal (not an exception) [Ref. 19: p. 14]

Although the PSDL exception is a data type and the Ada EXCEPTION is not, the Ada EXCEPTION can be used to implement and handle PSDL EXCEPTION types very conveniently. The major benefit from treating EXCEPTION as a data type in PSDL is abstraction. By this abstract construction, a unified means of handling all exceptions throughout the prototyping process is created [Ref. 19: p. 14]. Since all exceptions are handled the same way, there is no need for special constructions to handle each specific case. Thus construction of prototypes is simplified, and another step is taken toward automation of the prototyping process. This also simplifies translation of the exception condition into Ada. A generic exception handler can be created in Ada and instantiated by the translator as needed during translation. The abstraction eases the job of the prototype designer, which is the whole point of a computered aided prototyping system.

C. ADA AND PSDL

1. Ada and Real-Time Systems Constraints

a. Difficult Direct Implementation of Real-Time Constraints in Ada

The Ada implementation of such aspects of real-time systems as PERIOD, MET, MCP, MRT, and TIMER is not trivial. Ada DELAY by itself has no upper bound but is a lower bound on the delay implied. The Ada DELAY and SELECT constructs cannot be used to implement these performance constraints directly for a system of operators. The use of the type DURATION allows the approximation of an interval in a loop construct but it is not as flexible as needed. The use of TASKS in Ada provides more capability through the use of conditional entry calls. The problem with these constructs is that they require a good deal of effort on the part of the programmer to implement, and the program is operating at the mercy of the Ada run-time system. The degree of effort required to implement these constructs directly in Ada is out of proportion with the aims of the rapid prototyping methodology. A more abstract and direct syntax is required to specify hard, real-time constraints which will make construction and demonstration of prototypes possible. If the designer is required to invest nearly as much effort into the creation of the prototype as the development of the system itself, there is no advantage to prototyping. Furthermore, the Ada run-time system will not guarantee that the prototype design behaves in exactly the same manner as specified. The purpose of the SS and the DS in CAPS, is to ensure that the prototype functions within the real-time constraints applied to the design. Barring errors in design, the feasibility of such aspects of the system as control flow, order of firing of program modules, time behavior, and I/O formats can be demonstrated with CAPS. The ESS, frees the designer from the implementation effort required in Ada by automatically generating executable code in Ada, and by automatically generating control code in the form of Static and Dynamic schedules which enforce control and timing behavior. Therefore, PSDL supports develpopment of large and embedded Ada programs directly and easily.

b. Ada in Support of The CAPS Environment

Ada is most suitable for the development of CAPS for several reasons:

- Ada is the language mandated for development of embedded and real-time systems for DOD
- Ada provides constructs which can be used to implement more abstract timing behavior.
- Ada constructs can be used in a multiprocessor environment
- Ada provides simple exception handling facilities
- the GENERIC feature in Ada provides a simple means to implement automated prototype construction

2. Implementation of The PSDL Model in Ada

At this point several design implementation aspects of the Translator (TL) portion of ESS will be presented.

a. Operator

The OPERATOR construction of PSDL can be implemented by producing an Ada procedure. This procedure would contain code to implement any PSDL input or output conditional statements. It would also contain code to check the validity and availability of data for NDF control. Before presenting an example of this construction it will be necessary to develop the implementation of the PSDL data streams.

b. Data Streams

A PSDL data stream may be thought of as a simple queue of length one. Appendix C, part A, illustrates the construction of a simple queue in Ada. It is a procedure. With some minor modification, the queue can be made generic. This is accomplished by enclosing the procedure in a package and adding the Ada GENERIC part. An Ada private type is declared in the generic part. This private type allows the passing of any data type into the queue simply by declaring the type description at the point of generic instantiation. Thus, a generic queue is created which can be used at any point where a data stream is needed, by the simple use of the Ada generic instantiation. This technique is illustrated in Appendix C, part A.

(1) Generic Buffer Task. Recall that there are two different type data streams in the PSDL schema. One is a FIFO queue while the other is the sampled stream. Therefore, two different generic queue models are required. One of these

receives and transmits data without condition. This is the sampled stream, and will be referred to as a simple queue. Each data value in the simple queue may either be read many times or not at all. The second queue model will have a Boolean flag indicating whether or not it has been written since the last read operation or whether it has been read since the last write operation. This is the FIFO queue used for NDF control. The Boolean flag is necessary since delivery at least once, but only once, of each data value sent through the queue is required in natural data flow. If there is a violation of the FIFO rule, then the Boolean flag will result in the queue raising an exception. There are two possible exceptions. One will be identified as Underflow, and the other as Overflow. Underflow will be raised if the consumer operator attempts to read the queue before it has been updated by the producer operator. Overflow will be raised when the producer attempts to write to the queue before the consumer has read the previous data value.

The translator must have some basis to select the appropriate queue for a given data stream. If an operator contains the TRIGGERED BY ALL tokens then FIFO queues will be selected for the streams listed following the ALL token. If the operator contains the TRIGGERED BY SOME tokens then simple queues will be selected for the data streams. A third condition is if the operator contains no TRIGGERED BY tokens. In this case simple queues will be selected. For example, in Figure 12 on page 37, operator **T** has four input streams. The specification for **T** is, **TRIGGERED BY ALL d,f,h**. The translator will select FIFO queues for streams **d,f,** and **h**. Stream **g** will be a simple queue. In the same figure, operator **P** has four input streams. The specification for **P** is, **TRIGGERED BY SOME r**. In this case all data streams will be simple. Again in Figure 12 on page 37, operator **FF** has two input streams. The specification for **FF** lacks a TRIGGERED BY token. Therefore, all the streams are simple streams. Thus, if the operator specification lacks the TRIGGERED BY token, or contains the SOME token, the streams will be simple. If a stream is not listed in the ALL specification it will be simple. Only when the operator contains the ALL token will a FIFO queue be selected. Note that it is the triggering conditions for the consumer operator that determine the type data stream(s) that exist between any two operators.

Thus far, the data streams are modeled as a generic package containing a queue procedure in Ada. This construction is not sufficient. The SS and DS have generated a schedule for the time critical operators and this schedule is enforced to ensure real-time constraints are met. Some operators do not have time critical constraints. These operators are called into the empty or excess time in the worst case schedule for th time critical operators. It is possible that a time critical operator is the consumer of data from a non-time critical operator. The time critical operator has priority and is scheduled to run by the SS on some repetitive cycle. The non-time critical operator is fired. as convenient for the DS, in the excess time in the main schedule. Suppose a non-time critical operator is called and is attempting to write to the data stream, when it is interrupted by the DS in order to run a time critical operator. Also suppose that the time critical operator is the consumer for the data from the non-time critical operator. When the consumer attempts to read the queue, the results will be uncertain.

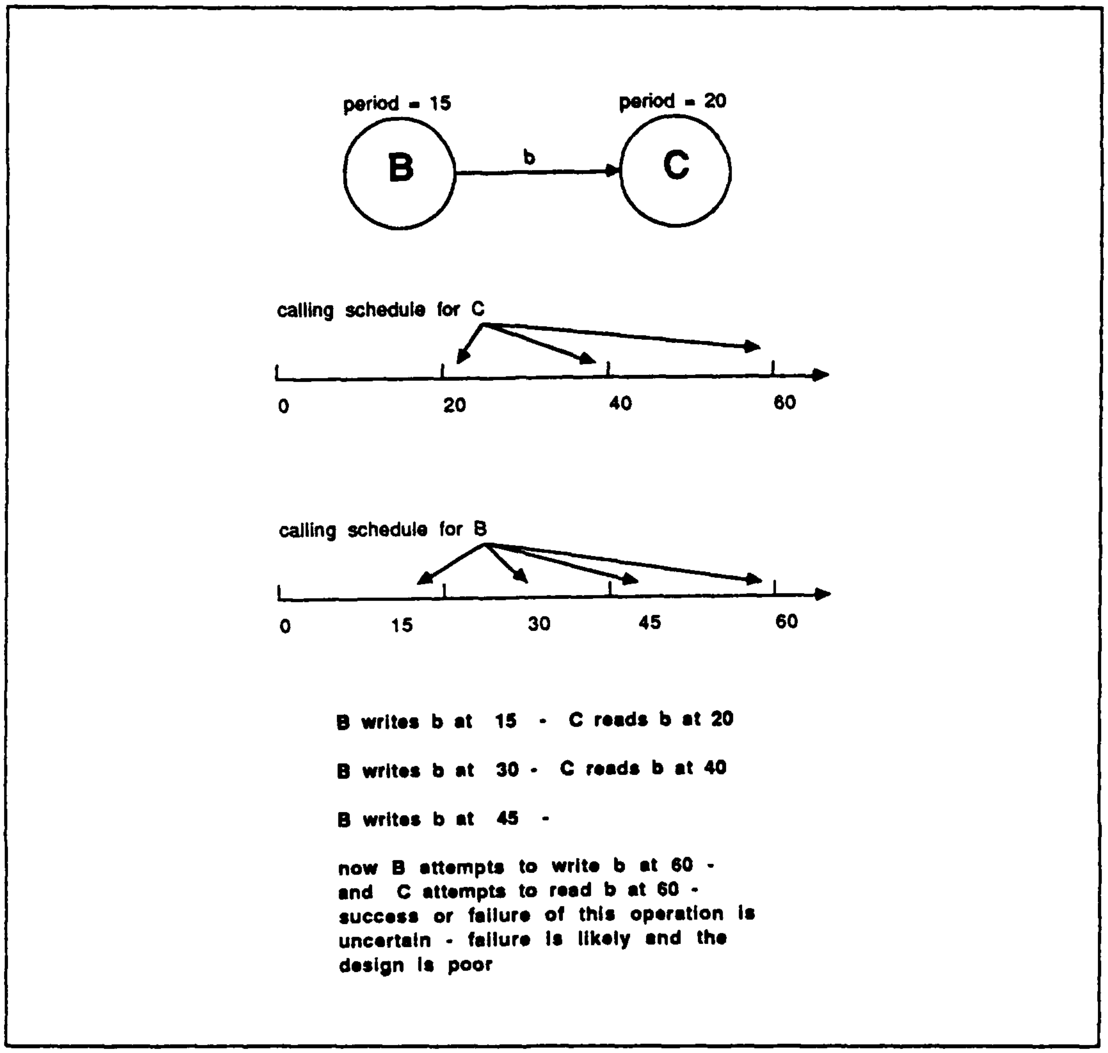
This difficulty can be overcome by making the generic queue into an Ada task. This task will be called a buffer task. The task is then enclosed as a generic package which can be generically instantiated as before. The difference is that the producer and consumer operators will use entry calls to write to or read from the buffer. In this way, once the buffer task is called, whatever operation is taking place on the buffer must be allowed to complete before an interrupt can take place. The operation time for any buffer task should be very short, so there should be little time penalty to the scheduled operation of the program. On the other hand, buffer operation is protected from interruption and the operators are unlikely to get uncertain results from reading them. Appendix C, part B, contains a listing for the Ada code to implement the two types of buffer tasks, SAMPLED STREAM and FIFO.

(2) Buffer Task Selection. How does a data driven operator know that the data stream (buffer) has new data, and that it should therefore fire? The buffer already contains a Boolean flag to indicate that it has been updated (either written to or read from). However, now that it is a task, an entry call must be made to access the Boolean flag. After finding the state of the flag, the consumer operator would then need to execute a task entry to access the actual data in the buffer. This would be

inconvenient. A simpler method would be to apply a similar Boolean flag to each producer operator of a NDF data stream. This would be an Ada in/out parameter to the producer procedure. The consumer procedure would incorporate a conditional guard to test the state of the Boolean in/out parameter of the producer. If the condition of the flag indicated that the producer has executed a write operation to the buffer since last read, the consumer would reset the variable to the state indicating that the data has not been updated and would then execute an entry call to the buffer(s) in order to fire itself.

(3) Buffer Length Selection. It may be asked why a buffer length of size one has been chosen to implement all buffers. The choice of buffer length is arbitrary in any case. Figure 13 on page 38 illustrates the source of the problem. Suppose a designer builds a system which contains both periodicity constraints and data flow control as in the figure. As previously discussed, the SS will generate a schedule based on periodicity and will also conduct a topological sort for control based on NDF. If the two schedules happen to match then the system will operate. If they do not, then the system is likely to fail. The SS will still allow the compilation and operation of the program based on the periodicity constraints. This will allow the designer to see the failure and decide on necessary changes and design alterations to make the program work. Figure 13 on page 38 shows the failure of the program will occur on the second time C attempts to fire. In this case buffer length has no effect on the operation or failure of the program. However, it is possible that a combination of various buffer lengths, periodicity constraints, and NDF constraints might operate correctly for some length of time before failing.

Figure 14 on page 46 shows a case where operation of the buffer is uncertain in the presence of both periodicity and NDF constraints. In this case, the fact that we have chosen a buffer of length one ensures that very little runtime will be required to reveal the instability of this design. Since one objective of the CAPS architecture is to save development time, it is important to reveal errors in design quickly in testing. By selecting buffers of length one throughout the prototype, we ensure that flawed designs, such as the one in Figure 13 on page 38 and Figure 14 on page 46 are revealed after a very short amount of run time. In general, a flawed design will fail eventually no matter what length buffer is chosen. Since the buffer length is an arbitrary



B writes b at 15 - C reads b at 20

B writes b at 30 - C reads b at 40

B writes b at 45 -

now B attempts to write b at 60 -
and C attempts to read b at 60 -
success or failure of this operation is
uncertain - failure is likely and the
design is poor

Figure 14. Uncertain buffer operation

choice, it is better in the CAPS to ensure rapid failure of poor designs. A buffer length of one will ensure this selection.

(4) Buffer Selection Conflicts. Another problem which arises in buffer selection is illustrated in Figure 9 on page 31. In this case we have the decomposition of an operator into three lower level operators. The designer will enter a specification for both the top level operator A and for the lower level operators **BB**, **CC**, and **DD**. Suppose operator **A** includes the tokens **TRIGGERED BY ALL a.** Also suppose that operator **BB** does not contain the TRIGGERED BY ALL tokens. When the TL selects

a buffer task for A, it will instantiate a FIFO buffer task to implement **a**. For **BB**, it would select a sampled stream task to implement **a'**. Although, **a** and **a'** carry the same data, and they have not been implemented with the same type buffer. The TL does not check inheritance rules. In operation data would be placed onto **a** and would then be passed to **a'** and into **BB**. The results of this translation will be uncertain. It may present no difficulty or may behave erratically. The user must prevent this type error by ensuring that operators which result from the decomposition of higher level operators have the same triggering conditions at the input in order to prevent the buffer mismatch just demonstrated. This difficulty only arises for lower level buffers which mirror the input buffers of the highest level operator of which they are a part. This is true because the type buffer required at any point in the system is determined by the triggering conditions of a consumer operator. Therefore, decomposition rules do not affect the specification requirements of operators **CC** and **DD** in Figure 9 on page 31. However, if **A** is **TRIGGERED BY ALL a**, then **BB** must be **TRIGGERED BY ALL a'**. It is a rule which the designer must enforce at this point. A utility similar to the C language lint, could be developed to check for this type inconsistency and incorporated into the ESS as an automatic part of the prototype translation, compilation, and export facility.

(5) The State Buffer. A final difficulty in data stream implementation is that of PSDL state variables, designated by the token STATES INITIALLY. Each state variable will have its own buffer task. An example is seen in Figure 9 on page 31. Operator **CC** is a state machine. It has a state variable which is transmitted along buffer task **d**. The value of the data type traveling along **d** must have some initial value. That value is found in the STATES INITIALLY statement in PSDL. To insure the correct initial value for the state variables in the program. buffer task **d** must be loaded with the correct value prior running the prototype. An Ada procedure called PRELOAD will be produced by the TL for all PSDL prototypes. It will contain a series of statements to put the correct initial values into the appropriate buffer tasks. If there are no state variables in the program, the procedure will simply be empty. The SS will always call PRELOAD before the execution of any schedule it creates for the prototype. The preloading procedure will not be part of the schedule proper.

It will run one time only to initialize the state buffers and will not be run again unless the prototype program is restarted from the beginning.

c. User Declared Data Abstractions

Already mentioned is the fact that all user declared types will be placed in an Ada package which will be used throughout the program. The listing for such a package is found in Appendix B, part C. This method allows the use of private types in the generic buffer task. At instantiation, the particular type variable to be sent through the buffer is declared. The actual description of the type is in the variable package. This package requires only an Ada specification part since it does not implement anything itself. In addition to user declared types, all other variables which would appear in the specification part of the Ada implementation will be placed in this same package. This technique is a useful Ada design tactic. It is especially useful in programs where ranges, intervals, delta values, or constants need to be assigned to variables, types, or subtypes. It insures that when variables need to be changed in a program, they can be found quickly and changed. There is no need to worry that a particular instance of the variable was overlooked somewhere in the program. In real-time systems such assignments of ranges, delta values, and constants may be seen to be quite common. For example, in an engineering plant control system, fixed point numbers might be employed to describe temperature measurements. These would have a particular delta value, perhaps .1 degree centigrade. The accuracy required might result from engineering considerations such as available sensor accuracy or the criticality of the system. If the program were written to accept data from a sensor of .1 degree centigrade and a sensor was needed and eventually developed which was accurate to .01 degree centigrade, the program would have to be modified to reflect the new delta value of .01. If the package technique had been used in program development, the effort required for the change would be minimal. A single point in the program would be adjusted and the modification would be complete. Lacking the package technique, the entire program listing would have to be examined to ensure a correct change. CAPS is thus developing Ada code which is easily maintained and modified.

d. Timer

The TIMER module must be implemented. The purpose of TIMER is to measure elapsed time between two events, the length of time an operator has been in a particular state, or to act as a conditional guard for operator firing. The four primitive operations for the timer are START, STOP, RESET, and READ. It will use the Ada standard package CALENDAR to access the system clock. The timer will have a Boolean run switch.

At START, the Boolean run switch will be set to true, the system clock read and the value of the reading stored as the initial starting point. At some time later a READ is performed. The system clock will be read and the value of the initial reading subtracted from it tu calculate the elapsed time. The initial value will not be changed. Actual clock time is not output. Elapsed time is output. At STOP, the system clock is read and the value stored in a simple array. The initial actual clock time is not output. Elapsed time is output. At STOP, the system clock is read and the initial reading is subtracted from the reading at STOP and the value output as the TIMER value. The reading thus obtained is stored as the grand total time elapsed. At a subsequent START, the system clock will be read and written over the old value. The grand total will not be disturbed. At another STOP, the new elapsed time will be added to the grand total and the will be output as the elapsed time. The RESET operation will stop the timer and return all timer values to the zero state. TIMER will be an Ada generic package. It can be instantiated wherever needed in the prototype very easily. An example of an Ada package to implement TIMER is found in Appendix C, part C.

3. Advantages of The Ada Implementation of PSDL Constructs

The CAPS utilizes the relatively simple PSDL design and specification language to describe prototypes. It creates Ada source code for an operational prototype which can be compiled and run tested. It utilizes an automated translation facility to produce this code. It takes advantage of the powerful generic construct in Ada to simplify translation. The resulting code uses packaging of data types to simplify translation and program maintenance. Use of private types supports representation hiding. Since PSDL data types are immutable, it is necessary to utilize a strictly typed language to implement them. Otherwise the protection against unpredictable side effecting afforded by the immutable PSDL data types might be lost in translation. Ada provides the strong type checking required. A similar observation can be made regarding the PSDL prohibition against global variables. CAPS combines the powerful features of Ada and PSDL to provide an effective tool to support the rapid prototyping methodology.

D. TRANSLATOR DESIGN AND CONSTRUCTION

1. The KODIYAK System

A few words should be said regarding the design and construction of the translator itself. The translator is created using an automated translator generator called KODIYAK. KODIYAK was developed by Robert Herndon at the University of Minnesota as a doctoral dissertation. [Ref. 24] It is available as a research tool and is quite effective. The system is based on Knuth's work in attribute grammars. It utilizes a version of Jalili's algorithm to evaluate the semantic tree it creates when generating the translator. The tool incorporates a file called K as a pre-processor to the LEX [Ref. 27] and Yacc [Ref. 28] tools in the UNIX operating system.

The process of translator production and usage is illustrated in Figure 15 on page 51. To produce a translator with KODIYAK, the user must create a source file. This file contains a listing of the terminal and non-terminal tokens of the source language to be translated. It also contains a listing of the valid attributes which each token may take on, as well as any precedence relationships which may be required to properly evaluate ambiguous cases in the grammar. Finally, the file contains a listing of attribute equations. These equations describe the relationship between the source language (in this case PSDL) and the target language (in this case Ada). The translator generator system, KODIYAK, utilizes these equations to produce a translator in executable C code. The translator thus created is an executable program. By running this program with a text file in the source language as input, an output file is created which contains the equivalent code in the target language. A complete listing of the translator generator source file for the PSDL to Ada translator is found in Appendix D.

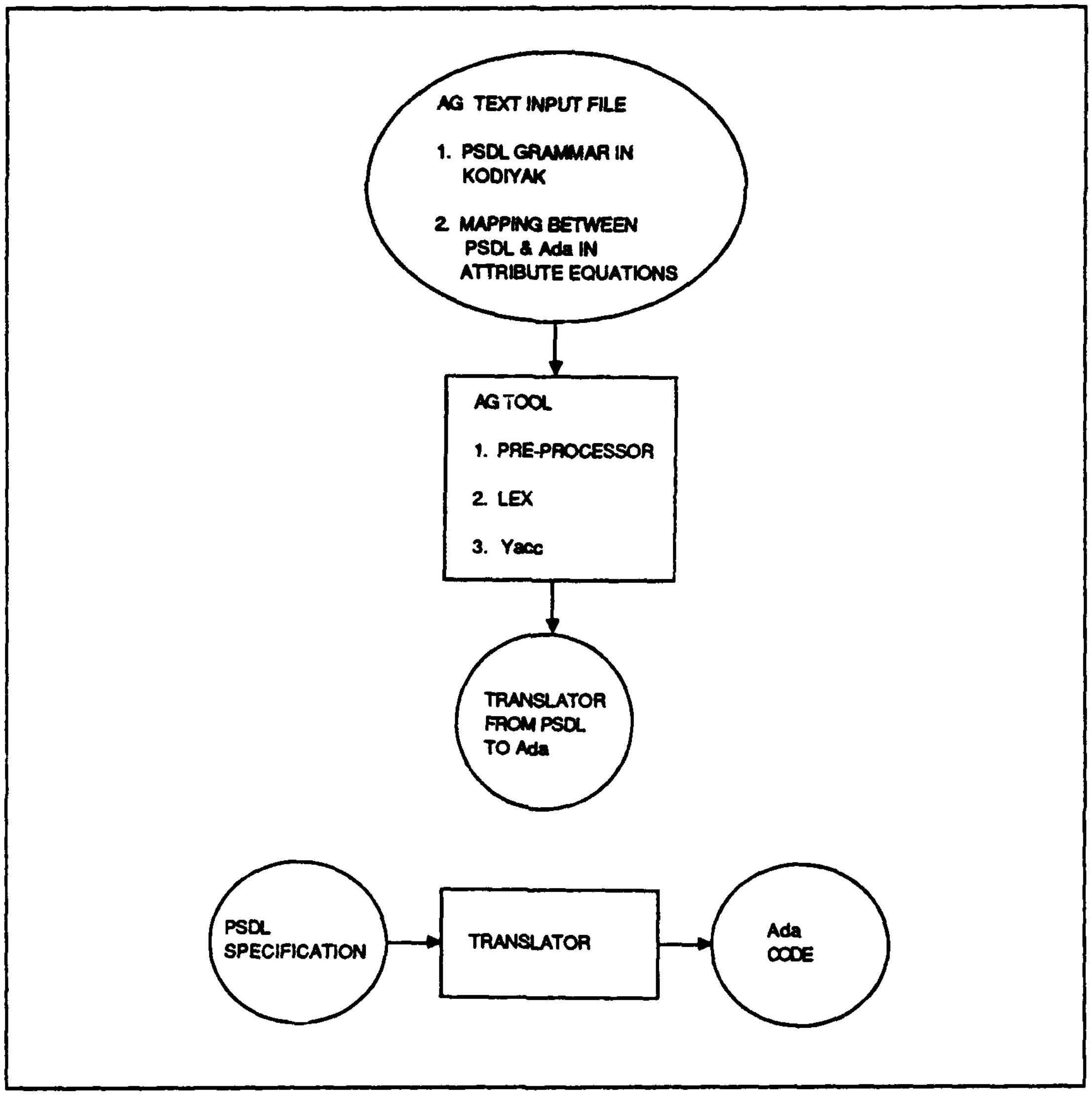

AG TEXT INPUT FILE
1. PSDL GRAMMAR IN KODIYAK
2. MAPPING BETWEEN PSDL & Ada IN ATTRIBUTE EQUATIONS
AG TOOL
1. PRE-PROCESSOR
2. LEX
3. Yacc
TRANSLATOR FROM PSDL TO Ada
PSDL SPECIFICATION
TRANSLATOR
Ada CODE

Figure 15. Translator construction and usage

IV. GENERAL APPLICABILITY TO TELECOMMUNICATIONS SOFTWARE SYSTEMS

What is the relationship of this research to Naval telecommunications systems and software? Current DOD policy indicates that software for embedded computing systems will be written in Ada or converted to Ada, although the application of this policy is left to the individual services [Ref. 29 p. 71-72; Ref. 30]. Embedded systems are those computers which form an integral part of a larger system, such as a communications switching processor, a missile guidance system, or a manufacturing process control computer [Ref. 12 p. 3]. Naval telecommunications systems are embedded systems and therefore are subject to this policy. No current Naval telecommunication system is written in Ada. Naval Data Automation Command (NAVDAC) has expressed an interest in the development of software tools and techniques to improve productivity in the maintenance and production of Navy software systems [Ref. 31]. This thesis addresses the creation of a software tool designed to improve the productivity level and efficiency with which Ada software can be produced. It also demonstrates, coincidentally, the application of several existing software engineering tools and techniques which can be used to address the conversion to Ada or the development of software components for future systems.

A. SOME CURRENT NAVAL TELECOMMUNICATIONS SYSTEMS

Table 1 on page 53 [Ref. 32] summarizes some information regarding several current Navy telecommunications systems. These are the Common User Digital Information Exchange System (CUDIXS) and the Naval Modular Automated Communications System (NAVMACS). The annual maintenance cost figure cited is for the software system in each case. No hardware maintenance costs are included. The maintenance costs for NAVMACS V5 and V5a is unknown as the systems are still undergoing development. Not listed in the table, is the development costs for the systems. Numerous government and private laboratories and corporations participated in the development of these systems over an extended period so that the development costs are not easily

Table 1. A SUMMARY OF SOME CHARACTERISTICS OF CURRENT NAVY TELECOMMUNICATIONS SYSTEMS AND THEIR SOFTWARE

		NAVMACS Family			
	CUDIXS	**V1**	**V2**	**V3**	**V5/5a**
Annual Maintenance Cost	$500K	$200K	$250K	$500K	unknown
IOC	1975	NOV 79	APR 80	DEC 76	(1)
Required Memory	64K	64K	64K	128K	(2)
Code Size (Lines)	120K	49K	54K	90K	(3)
Language	ULTRA-16, the 16 bit assembler code for the AN/UYK-20 computer which is the basic hardware unit for these systems				
Operating System	none	none	none	MOS (4)	(5)
Constraints	CUDIXS must maintain precise timing to properly operate within the receive/transmit windows required by link protocols. NAVMACS family, due to heavy loading of the system, concentrates on efficient use of system resources such as central processor unit and I/O capacity.				

(1) NAVMACS V5 is being developed in two phases IOC for Phase A was JUL 83. IOC for Phase B was JUL 86. IOC for V5a is expected to be OCT 88.

(2) NAVMACS V5 is a three computer system. Main computer memory is 256K. It can operate in degraded mode in 192K. The remaining computers require 64K. One will normally have 256K for fallback purposes.

(3) Code size by lines does not accurately reflect the presence of comments and the extensive use of macro instruction statements. Current size is 309,000 (decimal) 16-bit words.

(4) MOS = Modular Operating System

(5) NAVMACS Operating System (IOC). This is a highly modified and enhanced version of the MOS used in the V3.

determined. An examination of the initial operational capability (IOC) dates for the systems makes it clear that Ada was not a feasible choice for the development of the software for these systems, since Ada was not standardized until 1983 [Ref. 33]. It is also clear that there are hardware limitations on the size of code which can be tolerated, due to the small memory capacities available in the AN/UYK-20 computer which is the central processor for all the systems listed. Note that the code is very large in terms of number of instructions (or line count) albeit very compact, owing to the use of assembler language. NAVMACS V5 and V5a use up to three AN/UYK-20 computers, while CUDIXS and NAVMACS V1, V2, and V3 are single processor units. The various versions of the NAVMACS family differ in the variety and quantity of capabilities and services provided to users by the system. The V1 and V2 are typically found on frigate and destroyer size ships, while the V3 is reserved for cruisers, large amphibious ships, large supply ships, and flag configured ships. NAVMACS V5 is found only on carriers and !arge command and control ships.

The software for all systems is written in assembler language (ULTRA-16, the assembler language native to the AN/UYK-20 computer). As many common elements of the developed assembler code as possible have been used among all the systems [Ref. 32 encl. 3]. The software for the V5 has also been developed to operate on the AN/UYK-44 computer [Ref. 32: encl. 3].

B. SOME PROPOSED NAVMACS FOLLOW ON SYSTEMS

There is currently no formally accepted follow on to these systems. Initiatives to enhance and improve NAVMACS exist. Two approaches to proposals for follow on to NAVMACS will be briefly presented which will serve to illustrate possible applications for CAPS. Some possible opportunities for the application of tools and techniques on which CAPS is built will also be suggested.

1. NAVMACS Model II

There is an idea for a follow on system called NAVMACS Model II (afterward referred to as Model II) [Ref. 34]. Table 2 on page 56 is a listing of typical services found in current NAVMACS systems and the proposed additional services for NAVMACS Model II [Ref. 34: pp. 15-16]. The Model II proposal envisions a single type of computer and software package which could be used in many different applications by changing the peripheral suites attached to the central processor [Ref. 34: pp. 28-34]. The Model II envisions the use of some "smart" peripherals. These would include:

- Programmable Front End processors to interface:
    1. circuit cryptos
    2. system computers
    3. offline storage devices
    4. operator interface devices
- remote terminals for message preparation and distribution [Ref. 34 : pp. 17-22]

The Model II would use data display units at operator terminals vice control teletypes. This would speed message entry, screening, and distribution. The terminal would provide some means to ensure correct format and entry of information during message preparation [Ref. 34: p. 12]. This might take the form of message templates or canned message formats. Remote terminals in non-mission critical areas might use non-development items (NDI) [Ref. 34 p. 21]. "NDI is usually off-the-shelf or commercial-type products, but may also include equipment already developed by or for the Department of the Navy, or other military services or foreign military services [Ref. 35].." IOC for a follow on system might be the mid 1990's [Ref. 32].

2. Unified Network Technology

Unified Network Technology (UNT) and Communication Support System (CSS) are current initiatives to improve and advance the state of the art in Naval communications systems. UNT anticipates the creation of communication packet networks which will have flexible topology. These networks would provide most efficient use of existing and future communications systems by allowing routing of communications through any available communication media in an automated packet network. Present systems involve the use of dedicated links and dedicated hardware systems. This can result in inefficient use of communications resources as some media remain idle while other media is heavily loaded. UNT would use automated means to select the available media and use it transmit communications traffic. The CSS comprises the shipboard or shorebased network controllers and interface units to establish connectivity between

Table 2. TYPICAL FUNCTIONS FOR NAVMACS AND PROPOSED NAVMACSMODEL II

Current Function Description	**V1**	**V2**	**V3**	**V4**	**V5**
Up to 4 Fleet Broadcast Circuits	x	x	x	x	x
Up to 4 Full Period Termination Circuits			x	x	x
IXS Subscriber Capability	x	x	x	x	x
Flexible Circuit Definitions				x	x
System Control by Displays				x	x
On-line Message Composition			x	x	x
Long Term Msg Storage/Retrieval			x	x	x
Data Base Storage/Retrieval				x	x
Remote Terminal Sites				x	x
Data Base For Onboard Organization				x	x
Automatic Onboard Delivery				x	x
Duplicate Message Processing				x	x
Automatic Circuit Selection and Relay				x	x
Additional Model II Functions	**V2**	**V3**	**V4**	**V5**	
Tactical CUDIXS (Ship-Ship OTO's)	x	x	x	x	
Add System Control by Displays	x				
On-line Composition With Formats	x	x	x	x	
Flexible Circuit Definitions	x	x	x	x	
(including CUDIXS broadcast, LAB, NATO circuits fleet broadcast, FPT, automated nets)					
Remote Sites	x	x			
Add More Remote Sites			x	x	
Flexible Configuration of Remote Sites and Circuits	x	x	x	x	
Increased On-line Message Storage	x	x	x	x	
Automated HF Net Subscriber	x	x	x	x	
Automated HF Net Control			x	x	
Semi-Automatic Distribution	x	x			
Improved Long Term Message Storage and Retrieval	x	x	x	x	
Improved Duplicate Search			x	x	
Canned Message Composition	x	x			
Decrease Msg Transmission Delays		x			

users by employing the various hardware resources available. These systems approaches to communications will be software intensive. Distributed network control, flexible network topology, and adaption to changing communications loads will require software control. CAPS could be utilized in the development of such software.

C. POSSIBLE CONTRIBUTIONS TO TELECOMMUNICATIONS FROM CAPS RESEARCH

Current budgetary uncertainties, changing threat and mission requirements, changing technology, and long developmental lead times will certainly impact any future systems. As uncertainty is inherent in any discussion of future technology applications, it is only possible to suggest several possible research avenues arising from CAPS research, which might be applied to telecommunications software systems problems.

1. Rapid Prototyping and CAPS

It is likely that future systems will seek to provide more and faster service to users by automating many more functions. Automated functions implies the use of computing systems and software. Software requires development and the first step in software development is definition of the functional specifications. Rapid prototyping methodology directly addresses the early, precise definition of functional specifications so that full scale development of the system can proceed. CAPS offers a tool to implement Ada program prototyping and design in a rapid prototyping environment. Once fully implemented CAPS can be applied directly to the development of new telecommunications software systems.

New guidance under Secretary of the Navy Instruction 5200.37 [Ref. 36] defines acquisition policy for software intensive command and control information systems. This policy applies to those research and development programs in which software cost represent a substantial fraction of the total system development costs (more than 60 percent) [Ref. 36]. Specifically addressed are the use of software prototypes to simulate important interfaces and to perform the main functions of an intended system without strict adherence to the final standards in hardware, speed, size, or cost constraints required of the finished system [Ref. 36]. The CAPS system as currently planned will provide system simulations of precisely that type. The CAPS system, however, aims to provide simulations which do conform closely to any real-time constraints required of the proposed software system. Furthermore, CAPS implements the rapid prototyping paradigm, offering demonstrations for the customer. This meets the requirement to promote ". . . effective interaction between the user and the developer [Ref. 36]." The policy to promote early delivery of command and control information systems software products through rapid prototyping can be met through the application of CAPS. CAPS could also meet the need to reuse as much existing software as possible, and the prototypes produced will be written in a high order language (Ada) [Ref. 36].

2. Reuseable Assembler Code

A generally available feature in Ada compilers is the ability to import assembler code to implement sub-program bodies where speed of execution, or compactness of code is a concern. CAPS will use retrieval of reuseable software modules to speed prototyping. These reuseable modules are expected to be Ada code, but could be sections of assembler code where necessary. So long as Ada compilers are available for the target machine, the assembler code already written for that machine could be reused. Therefore, the question of conversion to Ada is not only, "Should the systems be converted to Ada?; "but also, "How much of the existing code needs to be replaced?" Functional specifications for existing systems are understood (presumably) empirically since the systems exist and are operational. Given the functional specifications, they could be expressed conveniently in FSDL and input into CAPS to generate an Ada prototype, which could be proofed, then finished out using Ada or assembler to implement the Ada subprograms. Several additional questions also arise including:

- Can the assembler code be appropriately decomposed into modules?
- Can the assembler code modules be described by normalized specifications within the software base?
- Can the functions of the assembler code be decomposed so that part of the system can be implemented in Ada and the current code reused?
- Does there exist an Ada compiler for the AN/UYK-44 computer and for that matter, what will be the next generation communications computer?
- What costs are associated with such an approach as opposed to implementing the system entirely in Ada?

3. Subordinate Tools And Techniques

a. Translators

Subordinate to the overall CAPS is the technique of developing and utilizing automated translator generators to produce automated translators. In principle, this approach could be applied to the conversion of existing programs in any language into any desired implementation language. Thus it may be possible to translate current assembler code software directly into Ada. It would be necessary to examine the issues of cost and feasibility of such an approach. It would also be neccesary to empirically demonstrate the concept and to produce a formal definition of the relationship between the two languages to ensure correctness in the final product.

b. Editor Generators

The Model II envisions the use of templates or preformatted message blanks for preparation of messages for transmission on electronic terminals. This facility currently exists in some instalations of NAVMACS and CUDIXS. In CAPS, a similar capability is envisioned. It takes the form of a syntax directed editor for PSDL. This editor would understand the correct syntax and usage for PSDL and would assist the operator to enter a syntactically correct PSDL prototype into the system. There exist several automated application generator facilities to create such "smart" editors [Ref. 17: pp. 12-14]. The approach in CAPS will be to utilize such a generator to create the syntax directed editor for CAPS. It may well be feasible to apply such an editor generator to generate editor facilities which "understand" the correct format for various types of Naval messages. Generation of custom editors for general message or structured messages (JINTACS, et.al.) might be possible. These techniques are incidental to the central thrust of CAPS and this thesis, which is to create an integrated system of tools for the generation of Ada applications.

c. Network Simulations

CAPS models software systems as systems of operators communicating via data streams. Each data stream in the CAPS could be a FIFO queue or a sampled stream. Each operator may have time constraints and conditional input or output. Thus, a CAPS model closely resembles a petri net, a system of nodes connected by communication paths. In principle, the basic elements of CAPS could be utilized to model and study the behaviour of networks. The data streams which now have queue length one, could be easily modified to provide generic queues with length *n*. Thus it may well be possible to use CAPS as a tool to model various network architectures, to provide operations research simulations of any network problem. Statistics collected from the run time profiler could provide insight into questions of network stability, throughput, and possible choke points. The graphic user interface would provide a pictoral representation of the network. The syntax directed editor and the software base management system would simplify construction of network models.

V. CONCLUSIONS AND FUTURE RESEARCH POSSIBILITIES FOR CAPS

It is feasible to describe a prototype in PSDL and to use an automated facility to translate the prototype into Ada. The present translator lays a sound foundation for further developmen:. It implements and recognizes the full syntax of PSDL as published by Luqi in her Ph.D. dissertation [Ref. 19]. The fundamental conceptual design implementation of the major PSDL syntactical constructs has been completed and documented. The translator produces rudimentary Ada code for interconnection of reuseable software program modules. Several additional research possibilities exist. First, the current translator is an empirical demonstration of the capability. Therefore, it should not be expected to function properly in all cases. Work must be undertaken to establish a rigorous, formal definition of the relationship between the syntax/semantics of PSDL and the syntax/semantics of Ada. Once such a rigorous definition has been produced, it must be applied to the translator to produce a facility which works for general cases.

Second, Ada is a robust language with a large syntax. PSDL is also a robust language, but has a very small syntax. Can PSDL effectively describe all (or most) of the constructions possible with Ada? This is similar to the formal definition problem. It may be necessary to define certain PSDL constructions and specify the Ada construction used to implement it in much the same way as Timer, Operator, and Data Stream have been specified in this thesis. It may also be necessary to specify that certain Ada constructs cannot be adequately represented in PSDL. This is unlikely; however, implementation of some Ada constructs may require highly sophisticated versions of the translator.

Third is the issue of code optimization. Some programs may require optimization for speed of execution, while others require optimization for code size. Can the translator be made to generate Ada implementations based on optimization criteria?

Fourth, the Static Scheduler (SS) uses a pre-processor written in Kodiyak to extract information about real-time constraints for various operators. This information is used to generate the static schedule for program operation. Kodiyak provides the facility to define separate sets of lexical definitions and attribute equations which apply in specified cases. Thus the pre-processor should be integrated into the Translator. This would eliminate the pre-processor as a single entity in the Execution Support System and simplify the integration of the Translator, Static Scheduler, and Dynamic Scheduler. Finally, the Translator, Static Scheduler, and Dynamic Scheduler must be integrated into a single tool, the Execution Support System, which can be integrated into CAPS.

APPENDIX A. PSDL GRAMMAR SUMMARY

Several conventions are used for symbology in the grammar. [Square Braces] indicate optional items. { Curly Braces } indicate items which may appear zero or more times. Bold face type indicates a named terminal symbol which must appear in the program listing the programmer writes. "Double quotes" indicate character literals which must appear in the program listing. The "|" vertical bar indicates an exclusive-or selection. In this case the programmer selects one and only one of the items separated by the vertical bar.

As an example, the token timing_info is one of six mutually exclusive possibilities which may define the attribute token. The attribute token may appear zero or more times to define the interface token, which is a required attribute of the operator_spec token. Timing_info, *if* selected for attribute, may be empty, or it may contain one or more of the optional tokens allowed to define timing_info. Each of these tokens may appear no more than one time for a given instance of timing_info.

psdl = { component }

component = | data_type
| operator

data_type = **type** id type_spec type_impl

operator = **operator** id operator_spec operator_impl

type_spec = **specification** [type_decl] {op_spec_list} [functionality] **end**

op_spec_list = **operator** id operator_spec

operator_spec = **specification** interface [functionality] **end**

interface = {attribute [reqmts_trace]}

attribute = | generic_param
| input
| output
| states
| exceptions
| timing_info

generic_param = **generic** type_decl

input = **input** type_decl

output = **output** type_decl

states = **states** type_decl **initially** expression_list

exceptions = **exception** id_list

id_list = id { "," id }

timing_info = [**maximum execution time** time]
[**minimum calling period** time]
[**maximum response time** time]

time = **number** [unit]

unit = | **microsec** | **ms** | **sec** | **min** | **hours**

reqmts_trace = **by requirements** id_list

functionality = [keywords] [informal_desc] [formal_desc]

keywords = **keywords** id_list

informal_desc = **description** "{" text "}"

formal_desc = **axioms** "{" text "}"

type_impl = | **implementation Ada** id
| **implementation** type_name { op_impl_list } **end**

op_impl_list = **operator** id *operator_impl*

operator_impl = | **implementation Ada** id
| **implementation** psdl_impl

psdl_impl = data_flow_diagram
[streams]
[timers]
[control_constraints]
[informal_desc]
end

data_flow_diagram = **graph** { link }

link = id "." opid "->" id

opid = id [":" time]

streams = **data_stream** type_decl

type_decl = id_list ":" type_name { "," id_list ":" type_name }

type_name = | id
| id "[" type_decl "]"

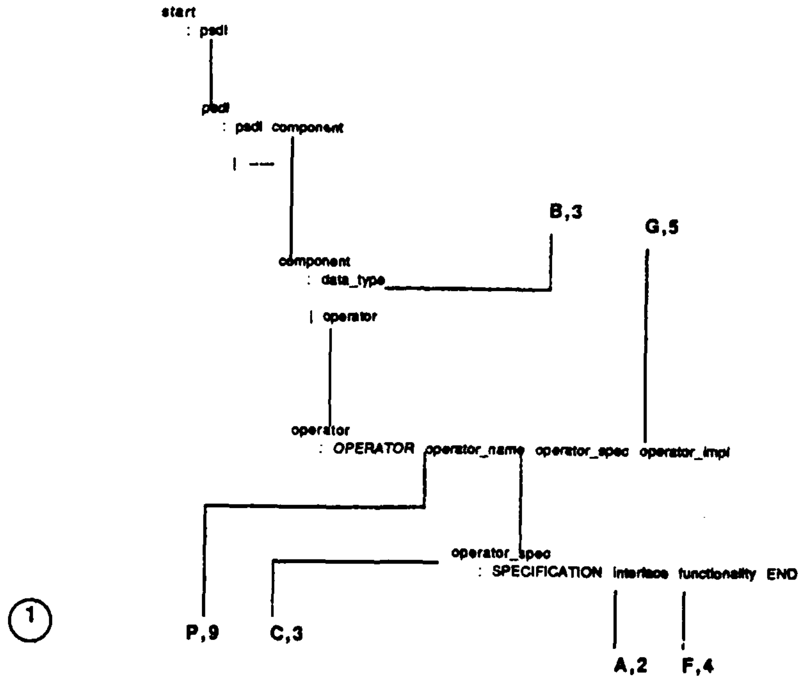
timers = **timer** id_list

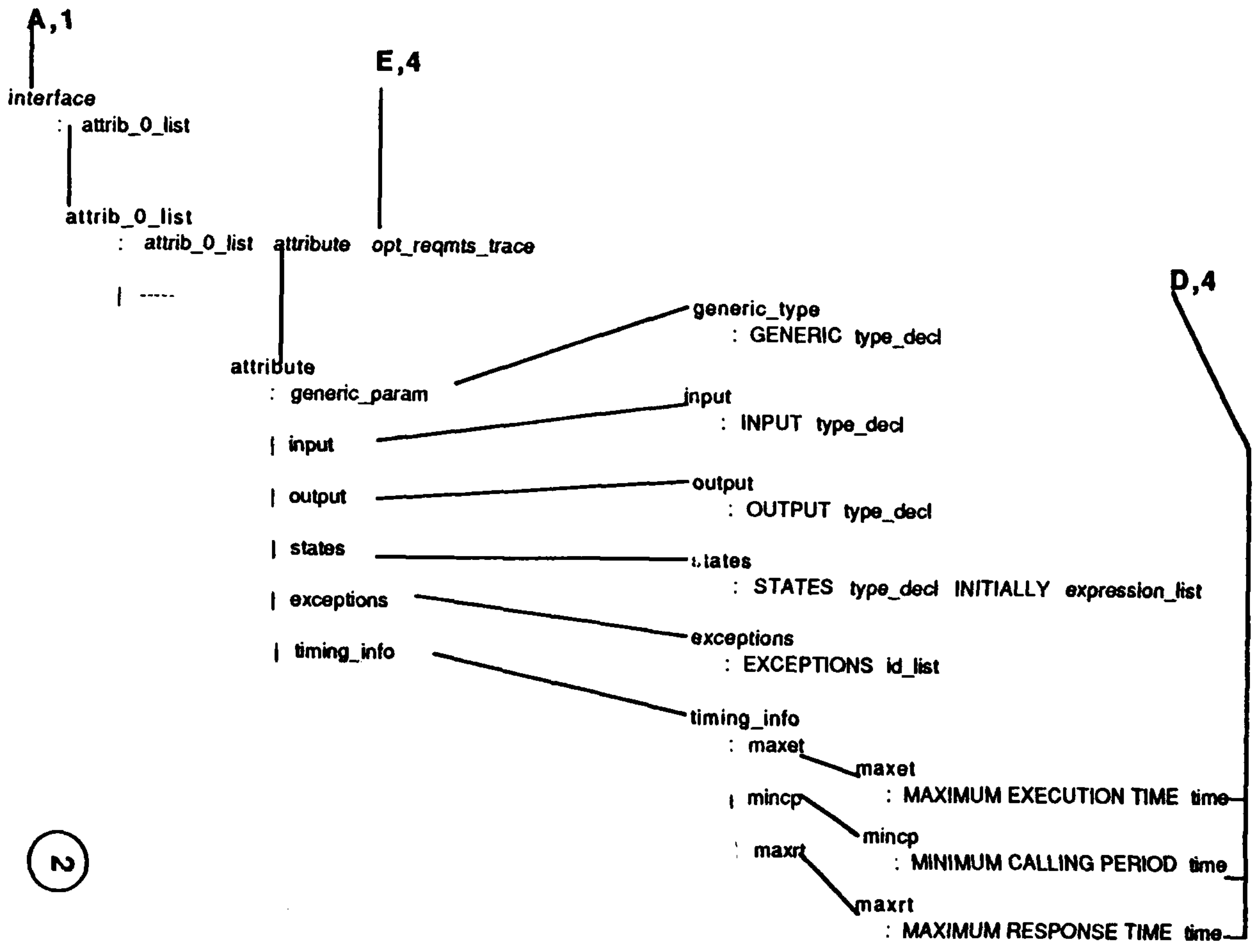
control_constraints = **control constraints** { constraint }

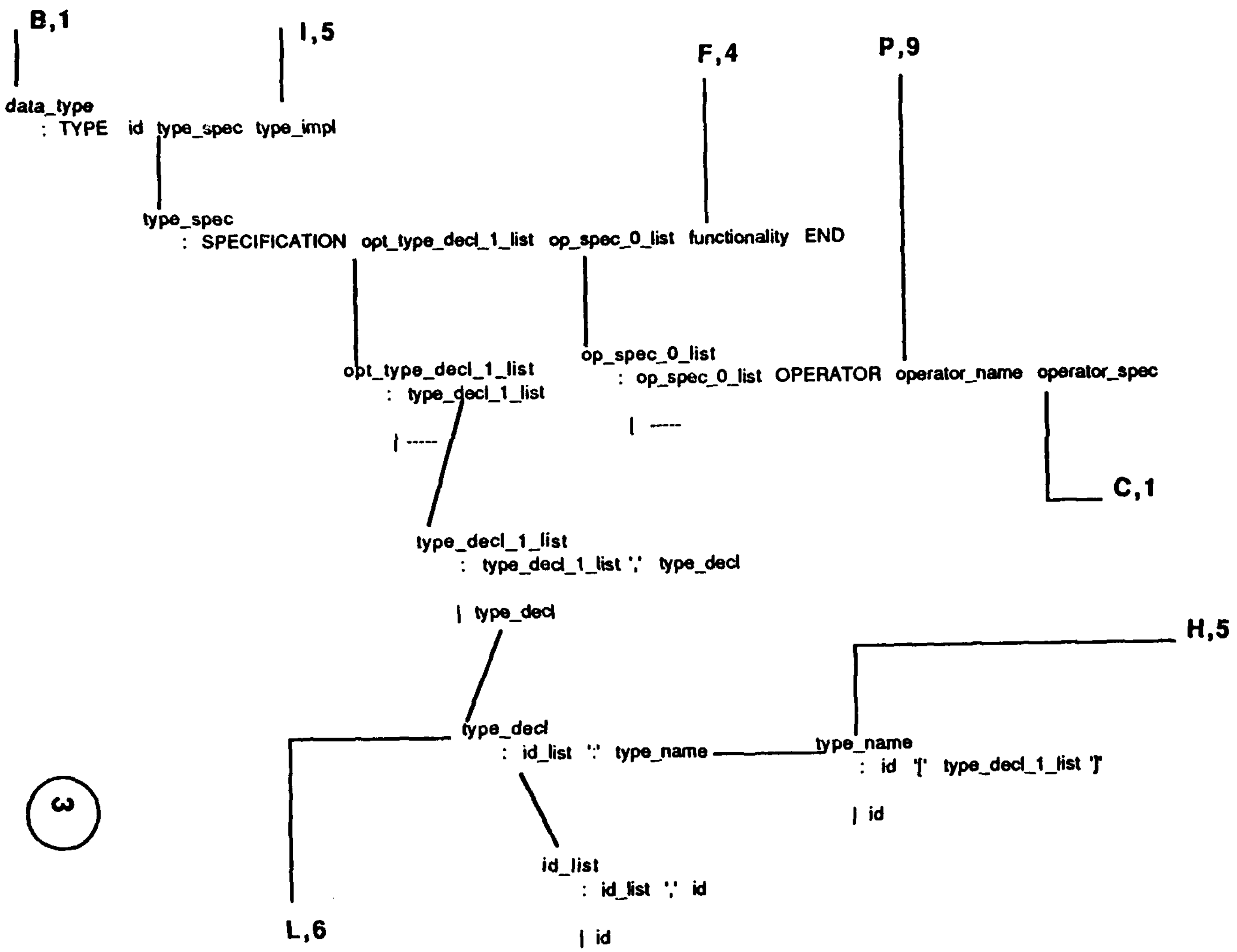
```
constraint =    operator id
                [triggered [trigger] [ "if" predicate] [reqmts_trace] ]
                [period time [reqmts_trace] ]
                [finish within time [reqmts_trace] ]
                {output id_list if predicate [reqmts_trace] }
                {exception id [if predicate] [reqmts_trace] }
                {timer_op id [if predicate] [reqmts_trace] }

timer_op = | start | stop | read | reset

trigger =    | by all id_list
             | by some id_list

predicate = | not predicate
            | predicate and predicate
            | predicate or predicate
            | expression_list
            | id ":" id_list

expression_list = expression { "," expression}

expression =    | number
                | constant
                | id
                | type_name "." id "(" expression_list ")"
```

APPENDIX B. DIAGRAMATIC REPRESENTATION OF PSDL

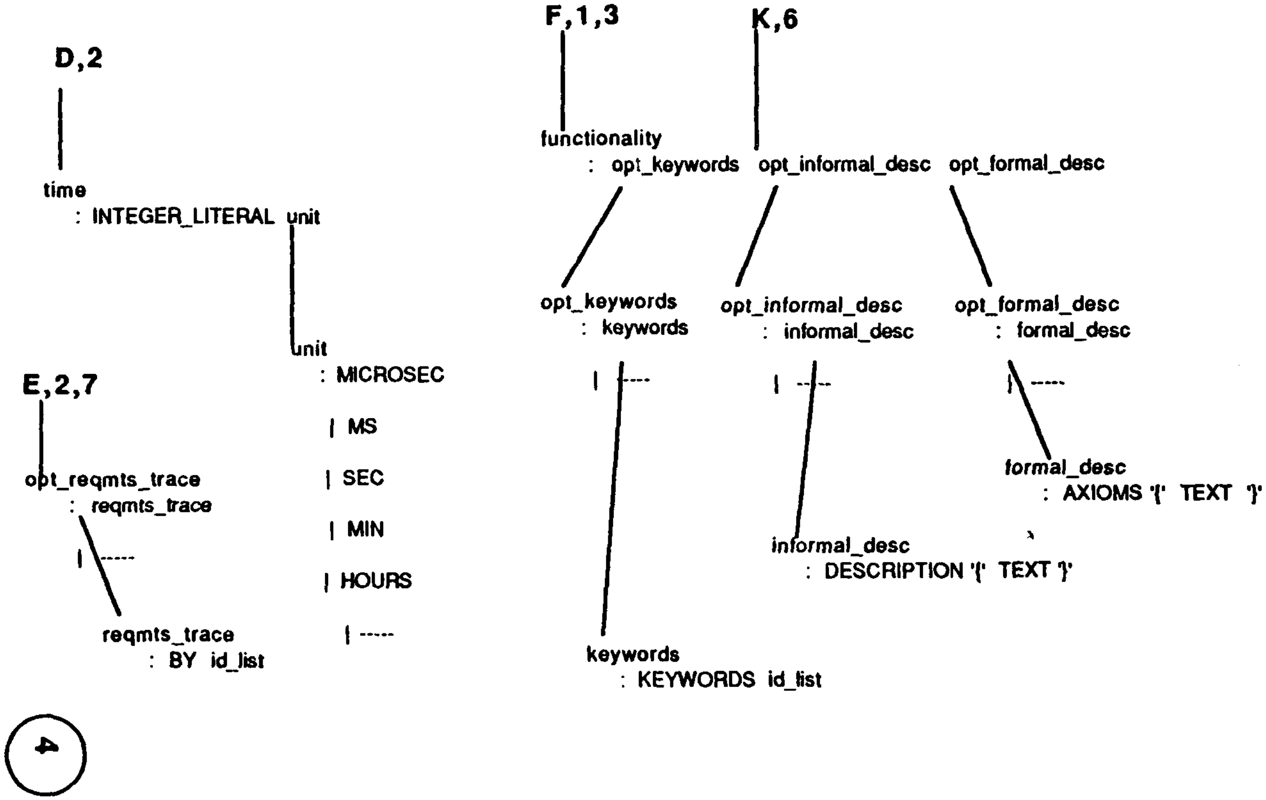
The folowing diagrams present a tree structured breakdown of the PSDLlanguage as applied in the translator. Each section is numbered with a large arabic numeral inside a circle in the lower left corner. This is a "key" number. Transitions between "key" sections are marked as lines terminated with a capital letter and one or more "key" numbers. For example, the non-terminal symbol, data_type, is found under "key" section 1, as a possible representation of the non-terminal symbol, component. The transition to a section with more detail on data_type is marked as **B,3**. This means go to the line marked **B** under "key" section 3. Moving to that section leads to the tree structured breakdown of the non-terminal symbol, data_type, into the terminal symbol, TYPE, followed by the non-terminal symbols, id, type_spec, and type_impl.

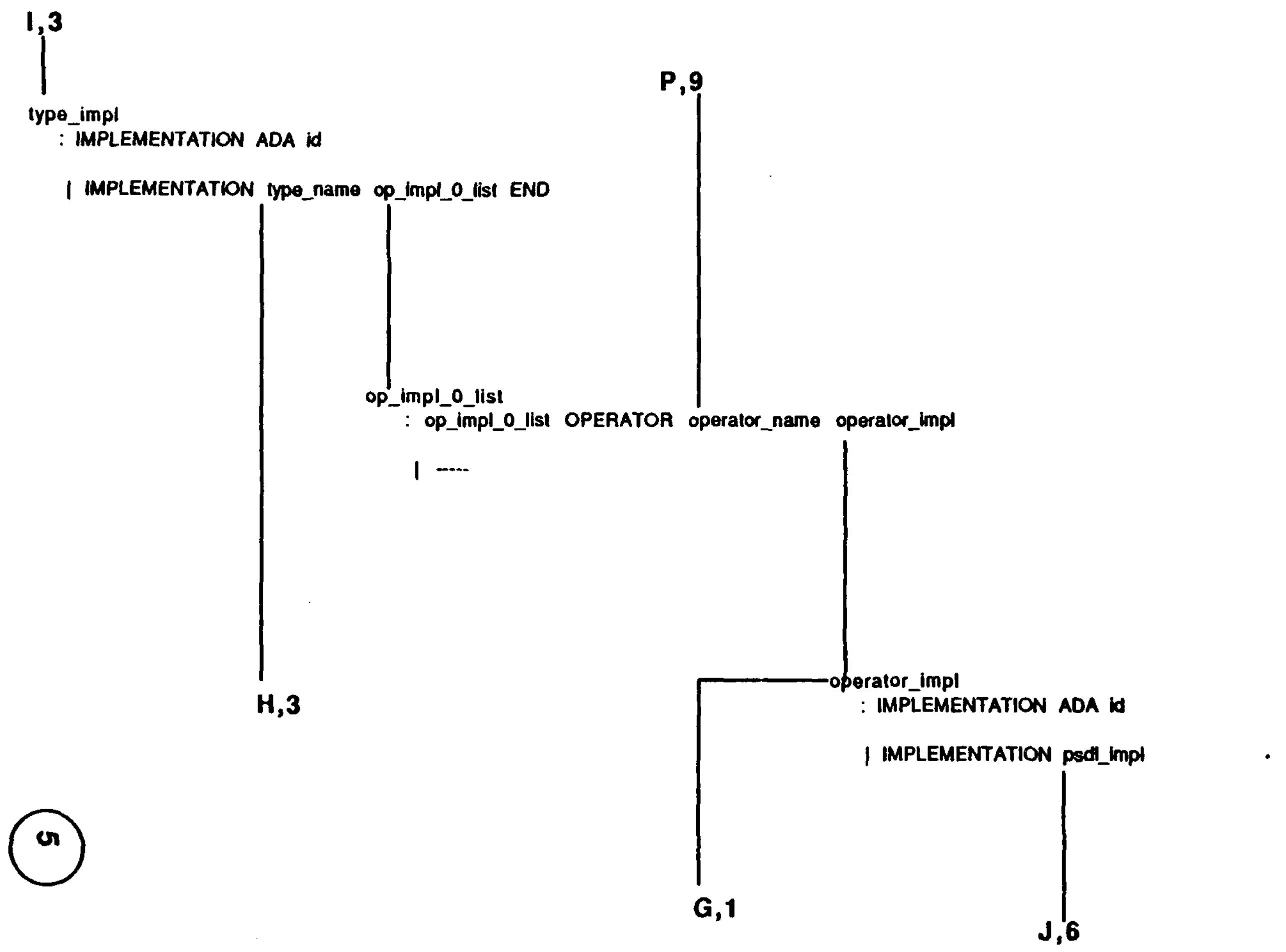



A,1
E,4
D,4
interface
: attrib_0_list
attrib_0_list
: attrib_0_list attribute opt_reqmts_trace
| -----
attribute
: generic_param
| input
| output
| states
| exceptions
| timing_info
generic_type
: GENERIC type_decl
input
: INPUT type_decl
output
: OUTPUT type_decl
states
: STATES type_decl INITIALLY expression_list
exceptions
: EXCEPTIONS id_list
timing_info
: maxet
| mincp
| maxrt
maxet
: MAXIMUM EXECUTION TIME time
mincp
: MINIMUM CALLING PERIOD time
maxrt
: MAXIMUM RESPONSE TIME time
2

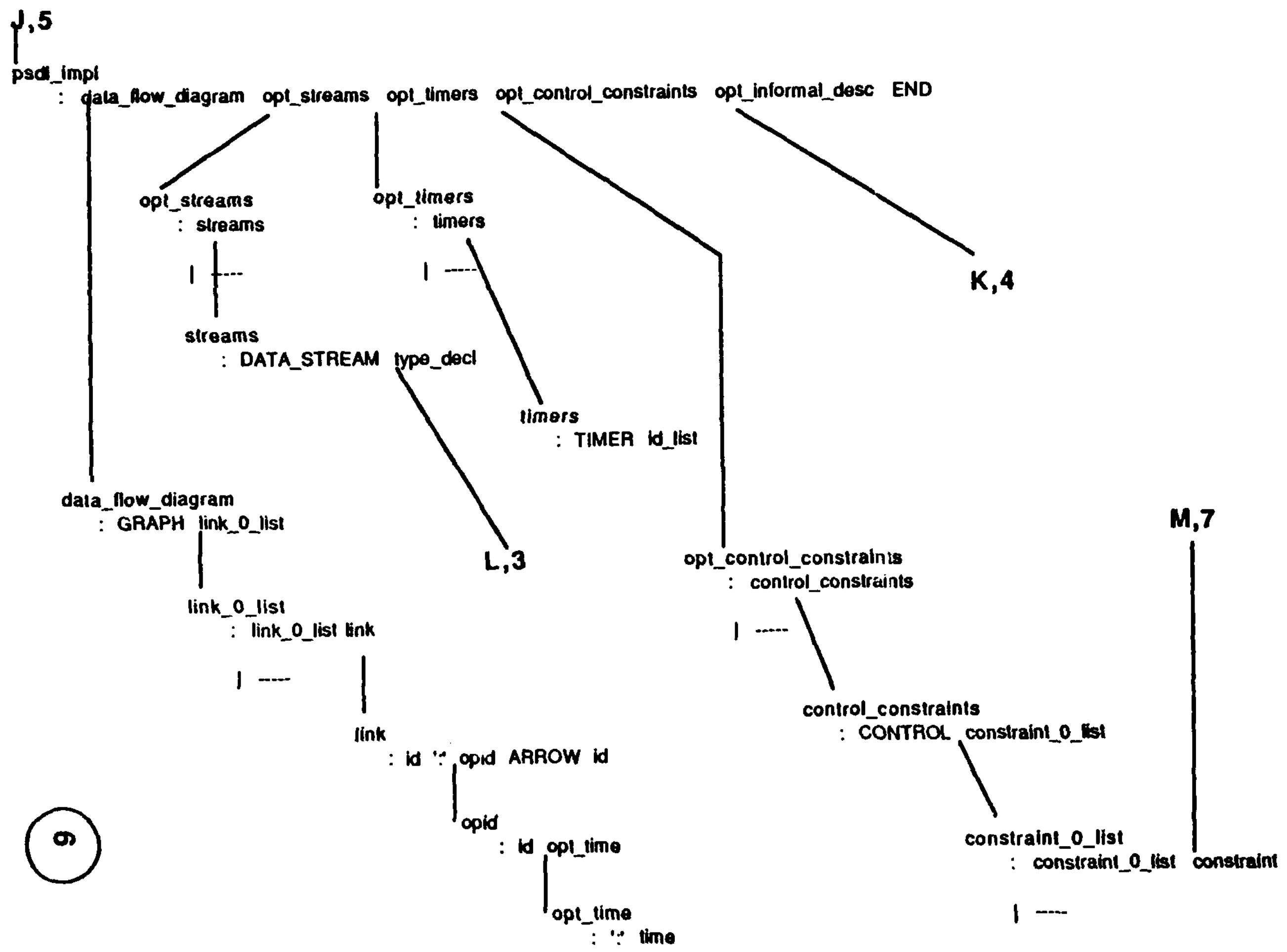


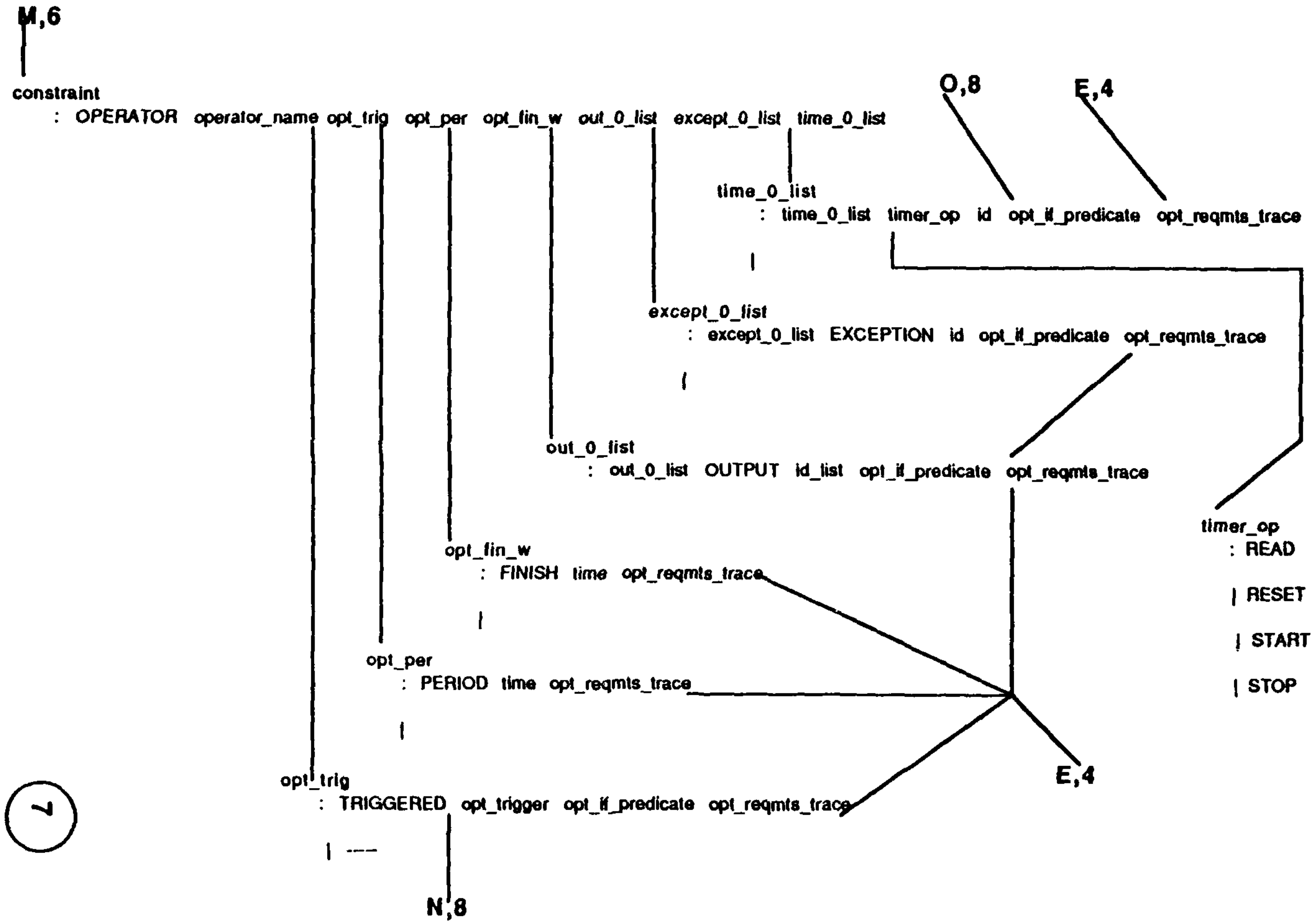
```
data_type
    : TYPE  id  type_spec  type_impl

type_spec
    : SPECIFICATION  opt_type_decl_1_list  op_spec_0_list  functionality  END

opt_type_decl_1_list
    : type_decl_1_list
    | -----

op_spec_0_list
    : op_spec_0_list  OPERATOR  operator_name  operator_spec
    | -----

type_decl_1_list
    : type_decl_1_list ','  type_decl
    | type_decl

type_decl
    : id_list  ':'  type_name

type_name
    : id  '['  type_decl_1_list  ']'
    | id

id_list
    : id_list  ','  id
    | id
```


D,2
time
: INTEGER_LITERAL unit
unit
: MICROSEC
| MS
| SEC
| MIN
| HOURS
| -----
E,2,7
opt_reqmts_trace
: reqmts_trace
| -----
reqmts_trace
: BY id_list
4
F,1,3
K,6
functionality
: opt_keywords opt_informal_desc opt_formal_desc
opt_keywords
: keywords
| -----
keywords
: KEYWORDS id_list
opt_informal_desc
: informal_desc
| -----
informal_desc
: DESCRIPTION '{' TEXT '}'
opt_formal_desc
: formal_desc
| -----
formal_desc
: AXIOMS '{' TEXT '}'

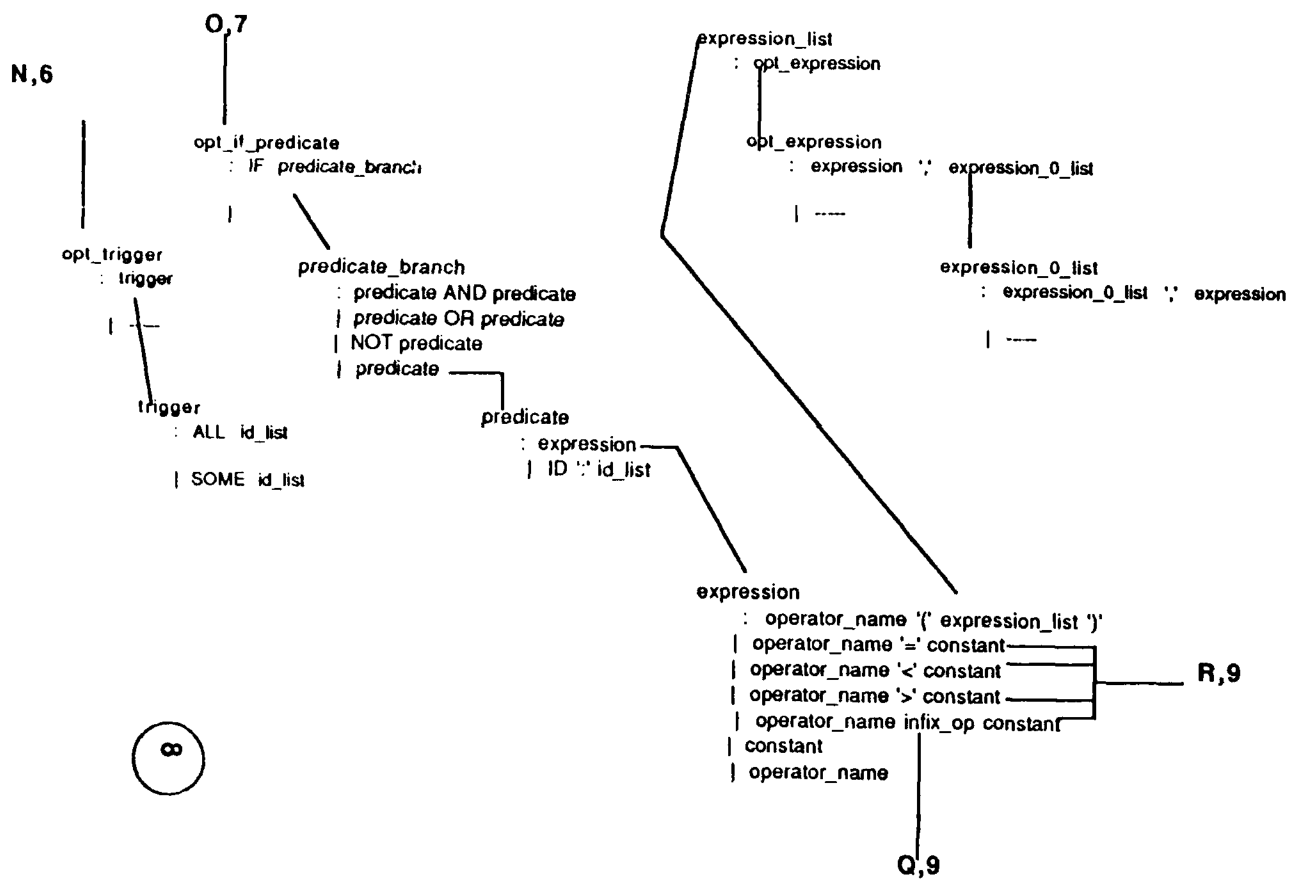


```
type_impl
    : IMPLEMENTATION ADA id

    | IMPLEMENTATION type_name op_impl_0_list END

op_impl_0_list
    : op_impl_0_list OPERATOR operator_name operator_impl

    | ----

operator_impl
    : IMPLEMENTATION ADA id

    | IMPLEMENTATION psdl_impl
```


J,5
psdl_impl
: data_flow_diagram opt_streams opt_timers opt_control_constraints opt_informal_desc END
opt_streams
: streams
| ----
streams
: DATA_STREAM type_decl
opt_timers
: timers
| ----
timers
: TIMER id_list
K,4
L,3
data_flow_diagram
: GRAPH link_0_list
link_0_list
: link_0_list link
| ----
link
: id ':' opid ARROW id
opid
: id opt_time
opt_time
: ':' time
opt_control_constraints
: control_constraints
| ----
control_constraints
: CONTROL constraint_0_list
constraint_0_list
: constraint_0_list constraint
| ----
M,7
6



```
constraint
    : OPERATOR operator_name opt_trig opt_per opt_fin_w out_0_list except_0_list time_0_list

time_0_list
    : time_0_list timer_op id opt_if_predicate opt_reqmts_trace
    |

except_0_list
    : except_0_list EXCEPTION id opt_if_predicate opt_reqmts_trace
    |

out_0_list
    : out_0_list OUTPUT id_list opt_if_predicate opt_reqmts_trace

opt_fin_w
    : FINISH time opt_reqmts_trace
    |

opt_per
    : PERIOD time opt_reqmts_trace
    |

opt_trig
    : TRIGGERED opt_trigger opt_if_predicate opt_reqmts_trace
    | ---

timer_op
    : READ
    | RESET
    | START
    | STOP
```



```
opt_trigger
    : trigger
    | ---

trigger
    : ALL id_list
    | SOME id_list

opt_if_predicate
    : IF predicate_branch
    |

predicate_branch
    : predicate AND predicate
    | predicate OR predicate
    | NOT predicate
    | predicate

predicate
    : expression
    | ID ':' id_list

expression_list
    : opt_expression

opt_expression
    : expression ',' expression_0_list
    | ----

expression_0_list
    : expression_0_list ',' expression
    | ----

expression
    : operator_name '(' expression_list ')'
    | operator_name '=' constant
    | operator_name '<' constant
    | operator_name '>' constant
    | operator_name infix_op constant
    | constant
    | operator_name
```

P,1,3,5

```
operator_name
    : type_name '.' ID
    | ID
```

Q,8

```
infix_op
    : GTE
    | LTE
    | NEQV
```

R,9

```
constant
    : TRUE

    | FALSE

    | INTEGER_LITERAL

    | REAL_LITERAL

    | STRING_LITERAL
```

9

APPENDIX C. ADA SOURCE CODE IMPLEMENTATION OF VARIOUS PSDL CONSTRUCTS

A. GENERIC QUEUE MODEL

```
generic
  type ITEM is private

package QUEUES is
  type QUEUE (Size : POSITIVE) is limited private;
  procedure CLEAR (TheQueue : in out QUEUE);
  procedure ADD (TheItem : in Item;
                 ToTheQueue : in out QUEUE);
  procedure REMOVE (TheItem : out Item;
                    FromTheQueue : in out QUEUE);
  OVERFLOW;
  UNDERFLOW : exception;
private
  type LIST is array (INTEGER range <>) of ITEM;
  type QUEUE (Size : POSITIVE) is
    record
      TheItems : LIST (0..Size);
      TheBack : NATURAL := 0;
    end record;
end QUEUES;

package body QUEUES is

  procedure CLEAR (TheQueue
 out QUEUE) is
  begin
    TheQueue.TheBack := 0;
  end CLEAR;

  procedure ADD (TheItem : in ITEM;
                 ToTheQueue : in out QUEUE) is
  begin
    ToTheQueue.TheItems(ToTheQueue.TheBack + 1) := TheItem,
    ToTheQueue.TheBack := ToTheQueue.TheBack + 1;
  exception
    when Constraint_Error =>
      raise OVERFLOW;
  end ADD;

  procedure REMOVE (TheItem : out ITEM;
                    FromTheQueue : in out INTEGER) is
  begin
    if FromTheQueue.TheBack = 0 then
      raise UNDERFLOW;
    else
```

```
      TheItem := FromTheQueue.TheItems(1);
      FromTheQueue.TheBack := FromTheQueue.TheBack - 1;
    end if;
  end REMOVE;
```

B. GENERIC PACKAGES CONTAINING FIFO AND SAMPLED STREAM BUFFER TASKS

1. FIFO Queue

```
generic type ELEMENT_TYPE is private;
package FIFO is
  task FIFO_BUFFER is
    entry CHECK (NEW_DATA : out BOOLEAN);
    entry PUT (VALUE : in ELEMENT_TYPE);
    entry GET (VALUE : out ELEMENT_TYPE);
  end FIFO_BUFFER:
  BUFFER_READ_ERROR,
  BUFFER_WRITE_ERROR : exception;
end FIFO;

package body FIFO is
  task body FIFO_BUFFER is
    BUFFER : ELEMENT_TYPE;
    VALUE : ELEMENT_TYPE;
    NEW_DATA_VALUE : BOOLEAN := false;
  begin
    loop
      select
        accept CHECK (NEW_DATA_VALUE : out BOOLEAN) do
          NEW_DATA := NEW_DATA_VALUE;
        end CHECK;
      or
        accept GET (VALUE : out ELEMENT_TYPE) do
          if NEW_DATA_VALUE then
            VALUE := BUFFER;
            NEW_DATA_VALUE := false;
          else raise BUFFER_WRITE_ERROR;
          end if;
        end GET;
      or
        accept PUT (VALUE : in ELEMENT_TYPE) do
          if not NEW_DATA_VALUE then
            BUFFER := VALUE;
            NEW_DATA_VALUE := true;
          else raise BUFFER_READ_ERROR;
          end if;
        end PUT;
      end select;
    end loop;
  end FIFO_BUFFER;
end FIFO;
```

2. Sampled Stream Queue

```
generic type ELEMENT_TYPE is private
package SAMPLED is
  task SAMPLED_BUFFER is
    entry CHECK (NEW_DATA : out BOOLEAN);
    entry PUT (VALUE : in ELEMENT_TYPE);
    entry GET (VALUE : out ELEMENT_TYPE);
  end SAMPLED_BUFFER:
end SAMPLED;

package body SAMPLED is
  task body SAMPLED is
    BUFFER : ELEMENT_TYPE;
    VALUE : ELEMENT_TYPE;
    NEW_DATA_VALUE : BOOLEAN := false;
  begin
    loop
      select
        accept CHECK (NEW_DATA : out BOOLEAN) do
          NEW_DATA := NEW_DATA_VALUE:
        end CHECK:
      or
        accept GET (VALUE : out ELEMENT_TYPE) do
          VALUE := BUFFER;
          NEW_DATA_VALUE := false;
        end GET;
      or
        accept PUT (VALUE : in ELEMENT_TYPE) do
          BUFFER := VALUE;
          NEW_DATA_VALUE := true;
        end PUT;
      end select;
    end loop;
  end SAMPLED_BUFFER:
end SAMPLED:
```

C. GENERIC PACKAGE IMPLEMENTING TIMER

```
generic

with CALENDAR;
 use CALENDAR;
package TIMER is
  StartTime : TIME;
  ReadTime : TIME;
  ElapsedTime : DURATION;
  TotalElapsedTime : DURATION;
  Run : BOOLEAN;
end TIMER;

with CALENDAR;
 use CALENDAR;
package body TIMER is;
```

```
procedure START (StartTime: out TIME;
                 Run : BOOLEAN);
begin
  if not Run =>
    StartTime := CLOCK;
    Run := True;
  end if;
end START;

procedure STOP (StartTime : in TIME;
                ReadTime : out TIME;
                ElapsedTime : out DURATION;
                TotalElapsedTime : out DURATION;
                Run : in out BOOLEAN);
begin
  if Run =>
    ReadTime := CLOCK;
    ElapsedTime := ReadTime "-" StartTime;
    TotalElapsedTime := TotalElapsedTime "+" ElapsedTime;
    Run := False;
  end if;
end STOP;

procedure READ (StartTime : in TIME;
                ReadTime : out TIME;
                ElapsedTime : out DURATION);
                TotalElapsedTime : out DURATION);
begin
  ReadTime := CLOCK;
  ElapsedTime := ReadTime "-" StartTime;
  TotalElapsedTime := TotalElapsedTime "+" ElapsedTime;
end READ;

procedure RESET (StartTime : out TIME;
                 ReadTime : out TIME;
                 ElapsedTime : out DURATION;
                 TotalElapsedTime : out DURATION;
                 Run : out BOOLEAN);
begin
  StartTime := CLOCK;
  ReadTime := CLOCK;
  ElapsedTime := 0.0;
  TotalElapsedTime := 0.0;
  Run := False;
end RESET;
```

APPENDIX D. PROGRAM LISTING FOR THE TRANSLATOR

The following is a listing of the Kodiyak input file which is compiled create the translator. It is composed of three sections delimited by the **%%** marker. Comments are indicated by the **!** mark and extend to the end of the line. Backslash followed by **t** or **n** follows the UNIX convention and stands for "tab" and "newline" respectively.

The first part of the file is the lexical definition section. The various lexical tokens in PSDL are identified. In order to assist this definition, classes of lexical characters can be defined. Such definitions are identified by the **%define** statement. Standard "Kleene" closures are used throughout (i.e., {}+ indicates one or more, {}* indicates zero or more). The solid vertical bar (|) indicates an "or" selection. The circumflex (shifted 6) in the definition for Char (character) indicates "all symbols except those immediately following" (i.e., all except left and right curly braces). Left and right brackets between two words indicates they are to be evaluated together as a lexical token.

The %% marker begins the second section. Here, the attributes for non-terminal and some terminal symbols of the language are defined. Kodiyak allows either string or integer type attributes. In this case all attributes are string type. Each non-terminal (e.g., start) has one attribute, **trn** (shorthand for translation), of type string. All Kodiyak translators have a start symbol which is used to indicate that the input file has been completely reduced and output can begin. Terminal symbols can also have attributes. In this case five terminal symbols have been assigned the special attribute **%text**. This attribute returns the value of the input text which the terminal symbol matched.

Section three of the Kodiyak file begins with the second %% marker. It is a representation of the grammatical structure of PSDL. It begins with the start symbol. The start symbol cannot appear on the right side of any production rule. If it did, output would commence even though the parsing tree of the input file would not have been completely reduced. Each producton rule in the grammar is represented and attached to each rule is an "attribute equation" surrounded by curly braces. The "attribute equation" specifies what output is to be created when the corresponding PSDL production rule is reduced. Within the "attribute equation," square brackets surrounding a series of items indicates the concatenation of the items. The solid vertical bar is used to indicate alternate possibilities for a given production rule. This is an exclusive or selection. It is also precedence ordered (i.e., top to bottom, the first rule which matches is the rule evaluated). Care must be exercised here as some states are implied and not explicit. For example, functionality has but one attribute equation. However, it has an implied empty state, since all three of the non-terminal symbols which are part of the production rule for functionality can have an empty state. Recursion and optional cases are supported. The naming convention used in this translator is as follows:

- opt_name means the item is optional
- name_1_list means one or more of the item
- name_0_list means zero or more of the item

When compiled, a program of about 230 kilobytes in size is created. The compiled program is **C** object code. Certain features are incorporated in all products created with Kodiyak. The executable code recognizes the standard UNIX **-h**, help, switch and responds with the correct usage syntax and a listing of optional switches. The three most useful are:

- -o outfile_name, allows the naming of a file to receive the output of the translator
- -l, causes the translator to display each PSDL token as it is recognized
- -y, causes the translator to display each PSDL production rule as it is resolved

The last two switches are especially helpful in debugging an input program.

```
!definitions of lexical classes

%define Digit   :[0-9]
%define Int     :{Digit}+
%define Letter  :[a-zA-Z_]
%define Alpha   :({Letter}|{Digit})
%define Blank   :[ \t\n]
%define Char    :[¬{}]
%define Quote   :["]

! definitions of white space

                :{Blank}+
! definitions of compound symbols and keywords

GTE               : ">="
LTE               : "<="
NEQV              : "/="
ARROW             : "->"
TYPE              : type|TYPE
OPERATOR          : operator|OPERATOR
SPECIFICATION     : specification|SPECIFICATION
END               : end|END
GENERIC           : generic|GENERIC
INPUT             : input|INPUT
OUTPUT            : output|OUTPUT
STATES            : states|STATES
INITIALLY         : initially|INITIALLY
EXCEPTIONS        : exceptions|EXCEPTIONS
MAX_EXEC_TIME     : maximum[ ]execution[ ]time|MAXIMUM[ ]EXECUTION[ ]TIME
MAX_RESP_TIME     : maximum[ ]response[ ]time|MAXIMUM[ ]RESPONSE[ ]TIME
MIN_CALL_PERIOD   : minimum[ ]calling[ ]period|MINIMUM[ ]CALLING[ ]PERIOD
MICROSEC          : microsec|MICROSEC
MS                : ms|MS
SEC               : sec|SEC
MIN               : min|MIN
HOURS             : hours|HOURS
BY                : by[ ]requirements|BY[ ]REQUIREMENTS
KEYWORDS          : keywords|KEYWORDS
DESCRIPTION       : description|DESCRIPTION
AXIOMS            : axioms|AXIOMS
IMPLEMENTATION    : implementation|IMPLEMENTATION
ADA               : ada|Ada|ADA
GRAPH             : graph|GRAPH
DATA_STREAM       : data[ ]stream|DATA[ ]STREAM
TIMER             : timer|TIMER
CONTROL           : control[ ]constraints|CONTROL[ ]CONSTRAINTS
TRIGGERED         : triggered|TRIGGERED
ALL               : by[ ]all|BY[ ]ALL
SOME              : by[ ]some|BY[ ]SOME
PERIOD            : period|PERIOD
FINISH            : finish[ ]within|FINISH[ ]WITHIN
EXCEPTION         : exception|EXCEPTION
READ              : read|READ
RESET             : reset|RESET
START             : start|START
STOP              : stop|STOP
IF                : if|IF
NOT               : "?"|"not"|"NOT"
AND               : "&"|"and"|"AND"
OR                : "|"|"or"|"OR"
TRUE              : true|TRUE
FALSE             : false|FALSE
ID                : {Letter}{Alpha}*
STRING_LITERAL    : {Quote}{Char}*{Quote}
INTEGER_LITERAL   : {Int}
REAL_LITERAL      : {Int}"."{Int}
TEXT              : "{"{Char}*"}"
```

```
! operator precedences
! %left means group and evaluate from the left

%left   OR;
%left   AND;
%left   NOT;
%left   '<', '>', '=', GTE, LTE, NEQV;

%%
! attribute declarations for nonterminal symbols

start { trn: string; };
psdl { trn: string; };
component { trn: string; };
data_type { trn: string; };
operator { trn: string; };
type_spec { trn: string; };
opt_type_decl_1_list { trn: string; };
type_decl_1_list { trn: string; };
type_decl { trn: string; };
op_spec_0_list { trn: string; };
operator_spec { trn: string; };
interface { trn: string; };
attrib_0_list { trn: string; };
attribute { trn: string; };
generic_param { trn: string; };
input { trn: string; };
output { trn: string; };
states { trn: string; };
exceptions { trn: string; };
timing_info { trn: string; };
maxet { trn: string; };
maxrt { trn:string; };
mincp { trn: string; };
time { trn: string; };
unit { trn: string; };
id_list { trn: string; };
opt_reqmts_trace { trn:string; };
reqmts_trace { trn: string; };
functionality { trn: string; };
opt_keywords { trn: string; };
opt_informal_desc { trn: string; };
opt_formal_desc { trn: string; };
keywords { trn: string; };
informal_desc { trn: string; };
formal_desc { trn: string; };
type_impl { trn: string; };
op_impl_0_list { trn: string; };
operator_impl { trn: string; };
psdl_impl { trn: string; };
data_flow_diagram { trn: string; };
link_0_list { trn: string; };
link { trn: string; };
opid { trn: string; };
opt_time { trn: string; };
```

```
opt_streams { trn: string; };
opt_timers { trn: string; };
opt_control_constraints { trn: string; };
streems { trn: string; };
type_name { trn: string; };
timers { trn: string; };
control_constraints { trn: string; };
constraint_0_list { trn: string; };
constraint { trn: string; };
operator_name { trn: string; };
opt_trig { trn: string; };
opt_trigger { trn: string; };
trigger { trn: string; };
opt_per { trn: string; };
opt_fin_w { trn: string; };
out_0_list { trn: string; };
except_0_list { trn: string; };
time_0_list { trn: string; };
timer_op { trn: string; };
opt_if_predicate { trn: string; };
predicate_branch { trn: string; };
predicate { trn: string; };
expression_list { trn: string; };
opt_expression { trn: string; };
expression_0_list { trn: string; };
expression { trn: string; };
infix_op { trn: string; };
constant { trn: string; };

!attrbute declarations for terminal symbols

ID{ %text: string; };
TEXT{ %text: string; };
STRING_LITERAL{ %text: string; };
INTEGER_LITERAL{ %text: string; };
REAL_LITERAL{ %text: string; };


%%
!psdl grammar

start
        : psdl
         { %output(psdl.trn); }
        ;

psdl
        : psdl component
         { psdl[1].trn = [psdl[2].trn,component.trn]; }
        |
         { psdl[1].trn = ""; }
        ;
```

```
component
        : data_type
          { component.trn = data_type.trn; }
        | operator
          { component.trn = operator.trn; }
        ;

data_type
        : TYPE ID type_spec type_impl
          { data_type.trn = ["type ",ID.%text,"\n",type_spec.trn,
                                  "\n",type_impl.trn,"\n"] ; }
        ;

operator
        : OPERATOR operator_name operator_spec operator_impl
          { operator.trn = ["procedure ",operator_name.trn,"is;\n",
                          operator_spec.trn,"\n",operator_impl.trn,"\n"]; }
        ;

type_spec
        : SPECIFICATION opt_type_decl_1_list op_spec_0_list functionality END
          { type_spec.trn =  [opt_type_decl_1_list.trn,"\n",op_spec_0_list.trn,
                                  "\n",functionality.trn," end;\n"]; }
        ;

opt_type_decl_1_list
        : type_decl_1_list
          { opt_type_decl_1_list.trn = type_decl_1_list.trn; }
        |
          { opt_type_decl_1_list.trn = ""; }
        ;

type_decl_1_list
        : type_decl_1_list ',' type_decl
          { type_decl_1_list[1].trn = [type_decl_1_list[2].trn,
                                           "\n",type_decl.trn]; }
        | type_decl
          { type_decl_1_list.trn = type_decl.trn; }
        ;

type_decl
        : id_list ':' type_name
          { type_decl.trn = [id_list.trn,":",type_name.trn]; }
        ;

op_spec_0_list
        : op_spec_0_list OPERATOR operator_name operator_spec
          { op_spec_0_list[1].trn = [op_spec_0_list[2].trn,"\n procedure ",
                                      operator_name.trn," is \n",
                                          operator_spec.trn]; }
        |
          { op_spec_0_list.trn = ""; }
        ;
```

```
operator_spec
        : SPECIFICATION interface functionality END
          { operator_spec.trn = [interface.trn,"\n",
                               functionality.trn," end;\n"]; }
        ;

interface
        : attrib_0_list
          { interface.trn = attrib_0_list.trn; }
        ;

attrib_0_list
        : attrib_0_list attribute opt_reqmts_trace
          { attrib_0_list[1].trn = [attrib_0_list[2].trn,opt_reqmts_trace.trn]; }
        |
          { attrib_0_list.trn = ""; }
        ;

attribute
        : generic_param
          { attribute.trn = generic_param.trn; }
        | input
          { attribute.trn = input.trn; }
        | output
          { attribute.trn = output.trn; }
        | states
         { attribute.trn = states.trn; }
        | exceptions
          { attribute.trn = exceptions.trn; }
        | timing_info
          { attribute.trn = timing_info.trn; }
        ;

generic_param
        : GENERIC type_decl
          { generic_param.trn = [" generic \n",type_decl.trn]; }
        ;

input
        : INPUT type_decl
          { input.trn = [" : in ",type_decl.trn]; }
        ;

output
        : OUTPUT type_decl
          { output.trn = [" : out ",type_decl.trn]; }
        ;

states
        : STATES type_decl INITIALLY expression_list
          { states.trn = ["procedure PRELOAD is;\n  PUT (",type_decl.trn,
                                     ");\n",expression_list.trn]; }
        ;
```

```
exceptions
        : EXCEPTIONS id_list
          { exceptions.trn = ["raise exception ",id_list.trn,";\n"]; }
        ;

id_list
        : id_list ',' ID
          { id_list[1].trn =  [id_list[2].trn," ",ID.%text] ; }
        | ID
          { id_list[1].trn =  ID.%text; }
        ;

timing_info
        : maxet
          { timing_info.trn = maxet.trn; }
        | mincp
          { timing_info.trn = mincp.trn; }
        | maxrt
          { timing_info.trn = maxrt.trn; }
        ;

maxet
        : MAX_EXEC_TIME time
          { maxet.trn =  time.trn; }
        ;

mincp
        : MIN_CALL_PERIOD time
          { mincp.trn = time.trn; }
        ;

maxrt
        : MAX_RESP_TIME time
          { maxrt.trn =  time.trn; }
        ;

time
        : INTEGER_LITERAL unit
          { time.trn = [INTEGER_LITERAL.%text,unit.trn]; }
        ;

unit
        : MICROSEC
          { unit.trn = "\n"; }
        | MS
          { unit.trn = "\n"; }
        | SEC
          { unit.trn = "\n"; }
        | MIN
          { unit.trn = "\n"; }
        | HOURS
          { unit.trn = "\n"; }
        |
          { unit.trn = ""; }
        ;
```

```
opt_reqmts_trace
          : reqmts_trace
            { opt_reqmts_trace.trn = reqmts_trace.trn; }
          |
            { opt_reqmts_trace.trn = ""; }
          ;

reqmts_trace
          : BY id_list
            { reqmts_trace.trn = ""; }
          ;

functionality
          : opt_keywords opt_informal_desc opt_formal_desc
            { functionality.trn = [opt_keywords.trn,opt_informal_desc.trn,
                                   opt_formal_desc.trn] ; }
          ;

opt_keywords
          : keywords
            { opt_keywords.trn = keywords.trn; }
          |
            { opt_keywords.trn = ""; }
          ;

opt_informal_desc
          : informal_desc
            { opt_informal_desc.trn = informal_desc.trn; }
          |
            { opt_informal_desc.trn = ""; }
          ;

opt_formal_desc
          : formal_desc
            { opt_formal_desc.trn = formal_desc.trn; }
          |
            { opt_formal_desc.trn = ""; }
          ;

keywords
          : KEYWORDS id_list
            { keywords.trn = "\n"; }
          ;

informal_desc
          : DESCRIPTION TEXT
            { informal_desc.trn = "\n"; }
          ;

formal_desc
          : AXIOMS TEXT
            { formal_desc.trn = "\n"; }
          ;
```

```
type_impl
        : IMPLEMENTATION ADA ID
          { type_impl.trn = ["procedure ",ID.%text," is;\n"]; }
        | IMPLEMENTATION type_name op_impl_0_list END
          { type_impl.trn = ["\n package DATA_TYPES is \n",type_name.trn,"\n",
                              op_impl_0_list.trn,"\n",
                              "end;\n"]; }
        ;

op_impl_0_list
        : op_impl_0_list OPERATOR operator_name operator_impl
          { op_impl_0_list[1].trn = ""; }
        |
          { op_impl_0_list[1].trn = ""; }
        ;

operator_impl
        : IMPLEMENTATION ADA ID
          { operator_impl.trn = ["procedure ",ID.%text," is \n"]; }
        | IMPLEMENTATION psdl_impl
          { operator_impl.trn = [psdl_impl.trn]; }
        ;

psdl_impl
        : data_flow_diagram opt_streams opt_timers opt_control_constraints
          opt_informal_desc END
          { psdl_impl.trn = [data_flow_diagram.trn,"\n",opt_streams.trn,"\n",
                             opt_timers.trn,"\n",opt_control_constraints.trn,
                             "\n",opt_informal_desc.trn," end;\n"]; }
        ;

data_flow_diagram
        : GRAPH link_0_list
          { data_flow_diagram.trn = ["\n-- Graphic representation: \n\t",
                                     link_0_list.trn,"\n"] ; }
        ;

link_0_list
        : link_0_list link
          { link_0_list[1].trn = [link_0_list[2].trn," ",link.trn]; }
        |
          { link_0_list.trn = ""; }
        ;

link
        : ID '.' opid ARROW ID
          { link.trn = [opid.trn,"_",ID[2].%text,"_",ID[1].%text,"\n"] ; }
        ;

opid
        : ID opt_time
          { opid.trn = [ID.%text,opt_time.trn]; }
        ;
```

```
opt_time
        : ':' time
          { opt_time.trn = [":",time.trn,"\n"]; }
        |
          { opt_time.trn = "\n"; }
        ;

opt_streams
        : streams
          { opt_streams.trn = streams.trn; }
        |
          { opt_streams.trn = ""; }
        ;

opt_timers
        : timers
          { opt_timers.trn = timers.trn; }
        |
          { opt_timers.trn = ""; }
        ;

opt_control_constraints
        : control_constraints
          { opt_control_constraints.trn = control_constraints.trn; }
        |
          { opt_control_constraints.trn = ""; }
        ;

streams
        : DATA_STREAM type_decl
          { streams.trn = [" task STREAM is new FIFO \n",
                               type_decl.trn,";\n"]; }
        ;

type_name
        : ID '[' type_decl_1_list ']'
          { type_name.trn = [ID.%text,"[",type_decl_1_list.trn,"]\n"]; }
        | ID
          { type_name.trn = ID.%text; }
        ;

timers
        : TIMER id_list
          { timers.trn = ["package ",id_list.trn," is new TIMER;\n"] ; }
        ;

control_constraints
        : CONTROL constraint_0_list
          { control_constraints.trn = constraint_0_list.trn; }
        ;
```

```
constraint_0_list
        : constraint_0_list constraint
          { constraint_0_list[1].trn = [constraint_0_list[2].trn," ",
                                         constraint.trn]; }
        |
          { constraint_0_list.trn = ""; }
        ;

constraint
        : OPERATOR operator_name opt_trig opt_per opt_fin_w out_0_list
          except_0_list time_0_list
          { constraint.trn = [" procedure ",operator_name.trn,"\n",opt_trig.trn,
                     "\n",opt_per.trn,"\n",opt_fin_w.trn,"\n",out_0_list.trn,
                     "\n",except_0_list.trn,"\n",time_0_list.trn,"\n"]; }
        ;

operator_name
        : type_name '.' ID
          { operator_name.trn = [type_name.trn,".",ID.%text]; }
        | ID
          { operator_name.trn = ID.%text; }
        ;

opt_trig
        : TRIGGERED opt_trigger opt_if_predicate opt_reqmts_trace
          { opt_trig.trn = [opt_trigger.trn,"\n",opt_if_predicate.trn,"\n",
                            opt_reqmts_trace.trn,"\n"]; }
        |
          {opt_trig.trn = ""; }
        ;

opt_trigger
        : trigger
          { opt_trigger.trn = trigger.trn; }
        |
          { opt_trigger.trn = ""; }
        ;

trigger
        : ALL id_list
          { trigger.trn = [" if ",id_list.trn," and "]; }
        | SOME id_list
          { trigger.trn = ["if ",id_list.trn," or "]; }
        ;

opt_per
        : PERIOD time opt_reqmts_trace
         { opt_per.trn = "\n"; }
        |
         { opt_per.trn = ""; }
        ;
```

```
opt_fin_w
        : FINISH time opt_reqmts_trace
         { opt_fin_w.trn = "\n"; }
        |
         { opt_fin_w.trn = ""; }
        ;

out_0_list
        : out_0_list OUTPUT id_list opt_if_predicate opt_reqmts_trace
         { out_0_list[1].trn = [out_0_list[2].trn," PUT ",id_list.trn," ",
                         opt_if_predicate.trn," ",opt_reqmts_trace.trn]; }
        |
         { out_0_list.trn = ""; }
        ;

except_0_list
        : except_0_list EXCEPTION ID opt_if_predicate opt_reqmts_trace
         { except_0_list[1].trn = [except_0_list[2].trn," RAISE ",ID.%text," ",
                                  opt_if_predicate.trn," ",opt_reqmts_trace.trn]; }
        |
         { except_0_list.trn = ""; }
        ;


time_0_list
        : time_0_list timer_op ID opt_if_predicate opt_reqmts_trace
         { time_0_list[1].trn = [time_0_list[2].trn," ",timer_op.trn," ",
                                  ID.%text,")\n  ",opt_if_predicate.trn,"\n  ",
                                  opt_reqmts_trace.trn]; }
        |
         { time_0_list.trn = ""; }
        ;

timer_op
        : READ
          { timer_op.trn = ["READ ("]; }
        | RESET
          { timer_op.trn = ["RESET ("]; }
        | START
          { timer_op.trn = ["START ("]; }
        | STOP
          { timer_op.trn = ["STOP ("]; }
        ;

opt_if_predicate
        : IF predicate_branch
          { opt_if_predicate.trn = ["if ",predicate_branch.trn]; }
        |
          { opt_if_predicate.trn = ""; }
        ;
```

```
predicate_branch
          : predicate AND predicate          %prec AND
            { predicate_branch.trn = [predicate[1].trn," AND ",
                                      predicate[2].trn] ; }
          | predicate OR predicate           %prec OR
            { predicate_branch.trn = [predicate[1].trn," OR ",
                                        predicate[2].trn] ; }
          | NOT predicate                    %prec NOT
            { predicate_branch.trn = ["NOT",predicate.trn] ; }
          | predicate
            { predicate_branch.trn = predicate.trn; }
          ;

predicate
          : expression
            { predicate.trn = expression.trn ; }
          | ID ':' id_list
            { predicate.trn = [ID.%text,":",id_list.trn] ; }
          ;

expression_list
          : opt_expression
            { expression_list.trn = opt_expression.trn; }
          ;

opt_expression
          : expression ',' expression_0_list
            { opt_expression.trn = [expression.trn," , ",expression_0_list.trn]; }
          |
            { opt_expression.trn = ""; }
          ;

expression_0_list
          : expression_0_list ',' expression
            { expression_0_list[1].trn = [expression_0_list[2].trn," , ",
                                          expression.trn]; }
          |
            {expression_0_list.trn = ""; }
          ;
```

```
expression
        : operator_name '(' expression_list ')'
          { expression.trn = [" ",operator_name.trn,
                                  " (",expression_list.trn,")\n"]; }
        | operator_name '=' constant      %prec LTE
          { expression.trn = [operator_name.trn," = ",constant.trn,"\n"]; }
        | operator_name '<' constant      %prec LTE
          { expression.trn = [operator_name.trn," < ",constant.trn,"\n"]; }
        | operator_name '>' constant      %prec LTE
          { expression.trn = [operator_name.trn," > ",constant.trn,"\n"]; }
        | operator_name infix_op constant
          { expression.trn = [operator_name.trn," ",infix_op.trn," ",
                                  constant.trn,"\n"]; }
        | constant

          { expression.trn = constant.trn; }
        | operator_name
          { expression.trn = [" = ",operator_name.trn,"; \n"]; }
        ;

infix_op
        : GTE                             %prec GTE
         { infix_op.trn = ">="; }
        | LTE                             %prec LTE
         { infix_op.trn = ">="; }
        | NEQV                            %prec NEQV
         { infix_op.trn = "/="; }
        ;

constant
        : TRUE
          { constant.trn = "true"; }
        | FALSE
          { constant.trn = "false"; }
        | INTEGER_LITERAL
          { constant.trn = INTEGER_LITERAL.%text; }
        | REAL_LITERAL
          { constant.trn = REAL_LITERAL.%text; }
        | STRING_LITERAL
          { constant.trn = STRING_LITERAL.%text; }
        ;
```

APPENDIX E. PROGRAM LISTING FOR TEST PROGRAM IN PSDL

The following test program is taken from the Ph.D. dissertation by Luqi which first described PSDL [Ref. 19]. It is representative of most features in the PSDL language. It contains descriptions at several levels of decomposition of the proposed system. The system envisioned is an embedded computer system for a medical treatment instrument known as a hyperthermia system. It implements real-time control constraints (required for safety of the patient as well as ensuring correct application of the theraputic technique). The system described would monitor and control the operation of a microwave generator. The microwave generator would be used to generate a hyperthermia condition for the treatment of tumors in the brain. There is a critical temperature range which would provide proper theraputic effect and yet remain safe for the patient. The system has stringent shutdown time limits when either treatment is completed or the temperature of the target tissues exceeds a limiting value. Obviously, there could be severe penalties should the system fail to function correctly. The time limits on startup and shutdown and the precise timing of the treatment period are critical. Maintenance of microwave power levels is critical to ensure correct temperature is maintained within a narrow range. As such, this program illustrates many of the features of an embedded system with real-time constraints. Since the program utilizes most of the features of PSDL and is a real-time system, it is a convenient one to utilize to test the translator. The Ada code produced thus far is elementary at best. As noted in the conclusion for this paper, the formal relationship between PSDL and Ada must be established and applied to the translator to ensure generality and correctness. Further, there is no library of reuseable Ada software modules from which to draw implementation code for the various parts of the hyperthermia system. The implementation code for this sytem would require development. The translator provides (as intended) interconnection code for the software.

```
OPERATOR brain_tumor_treatment_system
SPECIFICATION
 INPUT patient_chart:  medical_history,
    treatment_switch:  boolean
 OUTPUT treatment_finished:  boolean
 STATES temperature:  real
  INITIALLY 37.0
 DESCRIPTION
 { The brain tumor treatment system kills tumor cells
   by means of hyperthermia induced by microwaves.
 }
END

IMPLEMENTATION
 GRAPH

 DATA STREAM treatment_power:  real
 CONTROL CONSTRAINTS
  OPERATOR hyperthermia_system
   PERIOD 200 BY REQUIREMENTS shutdown
  OPERATOR simulated_patient
   PERIOD 200
 DESCRIPTION { paraphrased output }
END


TYPE medical_history
SPECIFICATION
 OPERATOR get_tumor_diameter
  SPECIFICATION
   INPUT patient_chart·  medical_history,
         tumor_location:  string
   OUTPUT diameter:  real
   EXCEPTIONS no_tumor
   MAXIMUM EXECUTION TIME 5 ms
   DESCRIPTION
   { Returns the diameter of the tumor at a given location,
     produces an exception if no tumor at that location.
   }
 END


 KEYWORDS patient_charts, medical_records, treatment_records,
    lab records
 DESCRIPTION
 { The medical history contains all of the disease and
   treatment information for one patient.  The operations
   for adding and retrieving information not needed by
   the hyperthermia system are not shown here.
 }
END


IMPLEMENTATION
 tuple [tumor_desc:  map[from:  string, to:  real], ... ]

 OPERATOR get_tumor_diameter
  IMPLEMENTATION
   GRAPH


    DATA STREAM td:  tumor_descr
    CONTROL CONSTRAINTS
     OPERATOR map.fetch
```

```
      EXCEPTIONS no_tumor IF not(map.has(tumor_location, td))
   END


END


OPERATOR hyperthermia_system
SPECIFICATION
 INPUT temperature:  real, patient_chart:  medical_history,
    treatment_switch:  boolean
 OUTPUT treatment_power:  real, treatment_finished:  boolean
 MAXIMUM EXECUTION TIME 100 ms
  BY REQUIREMENTS temperature_tolerance
 MAXIMUM RESPONSE TIME 300 ms
  BY REQUIREMENTS shutdown
 KEYWORDS medical_equipment, temperature_control,
    hyperthermia, brain_tumors
 DESCRIPTION
  { After the doctor turns on the treatment switch, the
    hyperthermia system reads the patient's medical record
    and turns on the microwave generator to heat the tumor
    in the patient's brain.  The system controls the power
    level to maintain the hyperthermia temperature of
    42.5 degrees C. for 45 minutes to kill the tumor cells.
    When the treatment is over, the system turns off the
    power and notifies the doctor.
  }
 END


IMPLEMENTATION
 GRAPH


 DATA STREAM estimated_power:  real
 TIMER treatment_time
  CONTROL CONSTRAINTS
   OPERATOR start_up
    TRIGGERED IF temperature < 42.4
     BY REQUIREMENTS maximum_temperature
    STOP TIMER treatment_time
    RESET TIMER treatment_time IF temperature <= 37.0

   OPERATOR maintain
    TRIGGERED IF temperature >= 42.4
     BY REQUIREMENTS maximum_temperature
    START TIMER treatment_time
     BY REQUIREMENTS treatment_time, temperature_tolerance
    OUTPUT treatment_finished IF treatment_time >= 45 min
     BY REQUIREMENTS treatment_time
END
```

```
OPERATOR start_up
SPECIFICATION
 INPUT patient_chart:  medical_history, temperature:  real
 OUTPUT estimated_power:  real, treatment_finished:  boolean
  BY REQUIREMENTS startup_time
 MAXIMUM EXECUTION TIME 90 ms
  BY REQUIREMENTS temperature_tolerance
 DESCRIPTION
 { Extracts the tumor diameter from the medical history and
   uses it to calculate the maximum safe treatment power.
   Estimated power is zero if no tumor is present.  The
   treatment finished is true only if no tumor is present.
 }
END


IMPLEMENTATION Ada start_up
END


OPERATOR maintain
SPECIFICATION
 INPUT temperature:  real
 OUTPUT estimated_power:  real, treatment_finished:  boolean
 MAXIMUM EXECUTION TIME 90 ms
  BY REQUIREMENTS temperature_tolerance
 DESCRIPTION
 { The power is controlled to keep the power between 42.4
   and 42.6 degrees C.
 }
END


IMPLEMENTATION Ada maintain
END


OPERATOR safety_control
SPECIFICATION
 INPUT treatment_switch, treatment_finished:  boolean
    estimated_power:  real
 OUTPUT treatment_power:  real
  BY REQUIREMENTS shutdown
 MAXIMUM EXECUTION TIME 10 ms
  BY REQUIREMENTS temperature_tolerance
 DESCRIPTION
 { The treatment power is equal to the estimated power
   if the treatment switch is true and treatment finished
   is false.  Otherwise the treatment power is zero.
 }
END


IMPLEMENTATION Ada start_up
END
```

LIST OF REFERENCES

1. Luqi, and Ketabchi, M. *"A Computer Aided Prototyping System."* IEEE Software, (March, 1988): 67-72.

2. Ketabchi, M., Berzins, V., and March, S. *"ODM: An Object Oriented Model for Design Databases"*. Proceedings, ACM Computer Science Conference, (February, 1986).

3. Janson, D. *"A Static Scheduler for Hard Real-Time Constraints in The Computer Aided Prototyping System."* M.S. Thesis, Naval Postgraduate School, March, 1988.

4. O'hern, J.T. *"A Conceptual Design of a Static Scheduler for Hard Real-Time Systems."* M.S. Thesis, Naval Postgraduate School, March, 1988.

5. Eaton, S.L. *"An Implementation Design of a Dynamic Scheduler for a Computer Aided Prototyping System."* M.S. Thesis, Naval Postgraduate School, March, 1988.

6. *"News in Perspective."* Datamation, (September, 1980): 124.

7. Boehm, B.W. *"Software Engineering."* IEEE Transactions on Computers, C-25 (December, 1976).

8. Mills, H.D. *"Software Development."* IEEE Transactions on Software Engineering, (December 1976): 265-273.

9. Carrio, M.A., Jr. *"Life Cycle and Ada."* DS&E, (July 1986): 17-21.

10. Boehm, B.W. *"The Hardware/Software Cost Ratio: Is It a Myth?"* IEEE Computer, 16,3 (March, 1983).

11. Boehm, B.W. *"Software and Its Impact: A Qualitative Assessment."* Datamation, (May, 1973): 48-59.

12. Booch, G.
.us Software Engineering With Ada. Menlo Park: The Benjamin/Cummings Publishing Company, 1986.

13. Fisher, D.A. *"A Common Programming Language For The Department of Defense -- Background And Technical Requirements."* Institute For Defense Analysis Report P-1191 (June, 1976).

14. *"A Strategy For A Software Initiative."* Department of Defense, Ada Joint Program Office, (1985).

15. Fairley, R. Software Engineering Concepts. New York: McGraw-Hill Book Company, 1985.

16. Luqi. *Research Aspects of Rapid Prototyping.* Monterey, California: Naval Postgraduate School Technical Report NPS52-87-006, [1987].

17. Reps, T.W. *"Generating Language Based Environments."* (ACM doctoral disseration award; 1983). Cambridge: The MIT Press, [1984].

18. Brooks, F. The Mythical Man-Month. Reading, MA: Addison-Wesley, 1975.

19. Luqi. *"Rapid Prototyping for Large Software System Design."* Ph.D. dissertation, University of Minnesota, May, 1986.

20. Luqi; Berzins, V.; and Yeh, R. *"A Prototyping Language for Real Time Software."* IEEE Transactions on Software Engineering, (April, 1988).

21. Luqi, and Berzins, V. *"Rapid Prototyping of Real Time Systems."* Monterey, California: Naval Postgraduate School Technical Report NPS52-87-005, [1987].

22. Luqi, and Berzins, V. *"Languages for Specification, Design, and Prototyping".* Handbook of Computer Aided Software Engineering. Van Nostrand Reinhold, 1988.

23. Luqi. *Execution of Real-Time Prototypes.* Monterey, California: Naval Postgraduate School Technical Report NPS52-87-012, [1987].

24. Herndon, R.M. *"Automatic Construction of Language Translators."* Ph.D. dissertation, University of Minnesota, May, 1987.

25. Knuth, D.E. *"Semantics of Context-Free Languages"* Mathematical Systems Theory, (November 1967): 127-145.

26. Herndon, R.M., and Berzins, V. *AG: A Useful Attribute Grammar Translator Generator.* Minneapolis, Minnesota: University of Minnesota Technical Report 85-25, [1985].

27. Lesk, M.E. *Lex - A Lexical Analyzer Generator.* Murray Hill, New Jersey: Bell Labs Computing Science Technical Report No. 39, [1975].

28. Johnson, S.C. *Yacc: Yet Another Compiler Compiler* Murray Hill, New Jersey: Bell Labs Computing Science Technical Report No. 32, [1975].

29. Wallace, R.H. Practitioner's Guide To Ada®. New York: McGraw-Hill Book Company, 1986.

30. Department of Defense Directive 5000.31. 10 June 1983.

31. *"Naval Data Automation Command Advisory Bulletin Number 40."* 3 January, 1983.

32. *"Response to Questions Regarding Thesis Research for Development of Computer Aided Software Prototyping Environment."* Director, Naval Telecommunications System Integration Center letter, 2023, Ser. 43/4008, 26 February, 1988.

33. U.S. Department of Defense. *Reference Manual For The Ada Programming Language, Proposed Standard Document July 1980.*

34. *"NAVMACS MODEL II Concept Paper."* Contract N00039-85-C-0156, Task 13, Item 3.4.1, 4 September, 1986.

35. Secretary of the Navy Instruction 4210.7A. 16 January, 1987.

36. Secretary of the Navy Instruction 5200.37. 5 January, 1988.

BIBLIOGRAPHY

Belinchon, R. *"Real Time System Development Based on Ada."* Signal, The International Journal of C^3 I. 42 (October 1987) 67-70.

Berzins, V. *Cache Management In Software Engineering Databases.* Monterey, California: Naval Postgraduate School Technical Report NPS52-87-010, [1987].

Berzins, V.; Gray, M.; and Naumann, D. *"Abstraction Based Software Development."* Communications of the ACM, vol 29 no 5 (May 1986): 402-415.

Berzins, V., and Smith, K. *A Bi-Directional Translator For Interfacing Citation Databases.* Monterey, California: Naval Postgraduate School Technical Report NPS52-87-013, [1987].

Berzins, V., and Simmel, S. *A Software Management System.* Monterey, California: Naval Postgraduate School Technical Report NPS52-87-014, [1987].

Boehm, B.W. *"Software and Its Impact: A Qualitative Assessment"* Datamation, (May, 1973): 48-59.

________. *"Software Engineering".* IEEE Transactions on Computers, C-25 (December, 1976).

________. *"Software Engineering Education: Some Industry Needs,"* in Software Engineering Education: Needs and Objectives. Freeman, P. and Wasserman, A. eds. Berlin: Springer-Verlag, 1976.

________. Software Engineering Economics. New Jersey: Prentice-Hall, Inc., 1981.

________. *"The Hardware/Software Cost Ratio: Is It a Myth?"* IEEE Computer, 16,3 (March, 1983).

Booch, G. Software Engineering With Ada. Menlo Park: The Benjamin/Cummings Publishing Company, 1987.

Bray, G., and Pokrass D. Understanding Ada, A Software Engineering Approach. New York: John Wiley & Sons, 1985.

Brooks, F. The Mythical Man-Month. Reading, MA: Addison-Wesley, 1975.

Carrio, M.A., Jr. *"Life Cycle and Ada."* DS&E, (July 1986): 17-21.

Dasarthy, B. *"Timing Constraints of Real-Time Systems: Constructs for Expressing Them, Methods of Validating Them."* IEEE Transactions On Software Engineering, Vol. SE-11, no. 1 (January, 1985): 80-86.

Eaton, S.L. *"An Implementation Design of a Dynamic Scheduler for a Computer Aided Prototyping System."* M.S. Thesis, Naval Postgraduate School, March [1988].

Fairley, R. Software Engineering Concepts. New York: McGraw-Hill Book Company, 1985.

Fisher, D.A. *"A Common Programming Language For The Department of Defense -- Background And Technical Requirements."* Institute For Defense Analysis Report P-1191 (June, 1976).

Herndon, R.M. *The Incomplete AG User's Guide and Reference Manual.* Minneapolis, Minnesota: University of Minnesota Technial Report 85-37, [1985].

________. *"Automatic Construction of Language Translators."* Ph.D. dissertation, University of Minnesota, May, 1987.

Herndon, R.M., and Berzins, V.A. *AG: A Useful Attribute Grammar Translator Generator.* Minneapolis, Minnesota: University of Minnesota Technical Report 85-25, [1985].

Jahanian, F. and Mok, A.K. *"Safety Analysis of Timing Properties in Real-Time Systems."* IEEE Transactions on Software Engineering, SE-12 (September, 1986): 890-904.

Janson, D. *"A Static Scheduler for Hard Real-Time Constraints in The Computer Aided Prototyping System."* M.S. Thesis, Naval Postgraduate School, March, [1988].

Johnson, S.C. *Yacc: Yet Another Compiler Compiler* Murray Hill, New Jersey: Bell Labs Computing Science Technical Report No. 32, [1975].

Ketabchi, M., Berzins, V., and March, S. *"ODM: An Object Oriented Model for Design Databases"*. Proceedings, ACM Computer Science Conference, (February, 1986).

Knuth, D.E. *"Semantics of Context-Free Languages"* Mathematical Systems Theory, (November 1967): 127-145.

Lesk, M.E. *Lex - A Lexical Analyzer Generator.* Murray Hill, New Jersey: Bell Labs Computing Science Technical Report No. 39, [1975].

Leveson, N.G. *Building Safe Software.* Irvine: University of California, [1986].

Luqi. *"Rapid Prototyping for Large Software System Design."* Ph.D. dissertation, University of Minnesota, May, 1986.

________. *Execution of Real-Time Prototypes.* Monterey, California: Naval Postgraduate School Technical Report NPS52-87-012, [1987].

________. *Research Aspects of Rapid Prototyping.* Monterey, California: Naval Postgraduate School Technical Report NPS52-87-006, [1987].

________. *Normalized Specifications For Identifying Reusable Software.*1 Monterey, California: Naval Postgraduate School Technical Report NPS52-87-007, [1987].

Luqi, and Berzins, V. *"Rapid Prototyping of Real Time Systems."* Monterey, California: Naval Postgraduate School Technical Report NPS52-87-005, [1987].

Luqi, and Berzins, V. *"Languages for Specification, Design, and Prototyping."* Handbook of Computer Aided Software Engineering. Van Nostrand Reinhold, 1988.

Luqi, and Ketabchi, M. *"A Computer Aided Prototyping System."* IEEE Software, (March, 1988): 67-72.

Luqi, Berzins, V.; and Yeh, R. *"A Prototyping Language for Real Time Software."* IEEE Transactions on Software Engineering, (April, 1988).

Mills, H.D. *"Software Development."* IEEE Transactions on Software Engineering, (December 1976): 265-273.

Mok, A.K. *"The Decomposition of Real-Time System Requirements Into Process Models."* Proceedings of the Real-Time Systems Symposium. Austin, Texas: (December, 1984): 125-134.

________. *"The Design of Real-Time Programming Systems Based on Process Models."* Proceedings of the Real-Time Systems Symposium. Austin, Texas: (December, 1984): 5-17.

Mok, A.K., and Supoj S. *"Modeling and Scheduling of Dataflow Real-Time Systems."* Proccedings of the Real-Time Systems Symposium. San Diego, California: (December, 1985): 178-187.

O'hern, J.T. *"A Conceptual Design of a Static Scheduler for Hard Real-Time Systems."* M.S. Thesis, Naval Postgraduate School, March, [1988].

Parnas, D.L. *"On The Criteria To Be Used in Decomposing Systems Into Modules."* Communications of the ACM, (December 1972): 1053-1058.

Plattner, B. *"Real-Time Execution Monitoring."* IEEE Transactions on Software Engineering, SE-10 (November, 1984): 756-764.

Reps, T.W. *"Generating Language Based Environments."* (ACM doctoral disseration award; 1983). Cambridge: The MIT Press, [1984].

Wallace, R.H. Practitioner's Guide To Ada®. New York: McGraw-Hill Book Company, 1986.

"News in Perspective." Datamation, (September, 1980): 124.

"A Strategy For A Software Initiative." Department of Defense, Ada Joint Program Office, (1985).

"Response to Questions Regarding Thesis Research for Development of Computer Aided Software Prototyping Environment." Director, Naval Telecommunications System Integration Center letter, 2023, Ser. 43/4008, 26 February, 1988.

"NAVMACS MODEL II Concept Paper." Contract N00039-85-C- 0156, Task 13, Item 3.4.1, 4 September, 1986.

Naval Data Automation Command Advisory Bulletin Number 40." 3 January, 1983.

Secretary of the Navy Instruction 4210.7A. 16 January, 1987.

Secretary of the Navy Instruction 5200.37. 5 January, 1988.

Department of Defense Directive 5000.31 10 June, 1983.

U.S. Department of Defense. *Reference Manual For The Ada Programming Language, Proposed Standard Document July 1980.*

INITIAL DISTRIBUTION LIST

No. Copies

1. Defense Technical Information Center 2
   Cameron Station
   Alexandria, VA 22304-6145

2. Library, Code 0142 2
   Naval Postgraduate School
   Monterey, CA 93943-5002

3. LT Charlie R. Moffitt, II, USN 2
   Department Head Class #104
   SWOSCOLCOM, Bldg. 446
   Newport, RI 02841-5012

4. Office of the Chief of Naval Operations 2
   Code OP-941
   Washington, D.C. 20350

5. Office of the Chief of Naval Operations 2
   Code OP-945
   Washington, D.C. 20350

6. Commander Naval Telecommunications Command 2
   Naval Telecommunications Command Headquarters
   4401 Massachusetts Avenue N.W.
   Washington, D.C. 20390-5290

7. Commander Naval Data Automation Command 1
   Washington Navy Yard
   Washington, D.C. 20374-1662

8. Office of Naval Research 1
   Office of the Chief of Naval Research
   Attn. CDR Michael Gehl, Code 1224
   Arlington, VA 22217-5000

9. Director, Naval Telecommunications System Integration Center 1
   NAVCOMMUNIT Washington
   Washington, D.C. 20397-5340

10. Space and Naval Warfare Systems Command 1
    Attn. Dr. Knudsen, Code PD50
    Washington, D.C. 20363-5100

11. Ada Joint Program Office 1
OUSDRE(R&AT)
The Pentagon
Washington, D.C. 20301

12. Professor L.Luqi 1
Code 52LQ
Naval Postgraduate School
Monterey, CA 93943

13. Maj. John B. Isett, USAF 1
Code 54IS
Naval Postgraduate School
Monterey, CA 93943

14. Professor D.C. Boger 1
Code 54BO
Naval Postgraduate School
Monterey, CA 93943

15. Defense Communications Agency 1
Attn: LT S.L. Eaton, USN, Code B531
Washington, D.C. 20350

16. Commander Naval Security Group Command 1
Attn: LT Joanne T. O'Hern, USN, Code G30
3801 Nebraska Avenue, N.W.
Washington, D.C. 20390

17. LT Dorothy M. Janson, USN 1
USCINCEUR Headquarters
General Delivery
APO New York, NY 09128-4209

18. MAJ Mike Dolezal 1
Director, Development Center
MCDEC
Quantico, VA 22134-5080

19. Naval Sea Systems Command 1
Attn. CAPT. Joel Crandall
National Center #2, Suite 7N06
Washington, D.C. 22202

20. Office of the Secretary of Defense 1
Attn. CDR Barber
The Star Program
Washington, D.C. 20301

21. Naval Ocean System Center 1
Attn. Linwood Sutton, Code 423
San Diego, CA 92152-5000

22. RADC/COES 1
Attn. LT Kevin Benner
Griffis Air Force Base
New York, NY 13441-5700

Initial Distribution List

Defense Technical Information Center 2
Cameron Station
Alexandria, VA 22314

Dudley Knox Library 2
Code 0142
Naval Postgraduate School
Monterey, CA 93943

Center for Naval Analysis 1
4401 Ford Avenue
Alexandria, VA 22302-0268

Office of the Chief of Naval Operations 2
Code OP-941
Washington, D.C. 20350

Office of the Chief of Naval Operations 2
Code OP-945
Washington, D.C. 20340

Commander Naval Telecommunications Command 2
Naval Telecommunications Command Headquarters
4401 Massachusetts Avenue NW
Washington, D.C. 20390-5290

Commander Naval Data Automation Command 1
Washington Navy Yard
Washington, D.C. 20374-1662

Office of Naval Research 1
Office of the Chief of Naval Research
Attn. CDR Michael Gehl, Code 1224
Arlington, VA 22217-5000

Director, Naval Telecommunications System Integration Center 1
NAVCOMMUNIT Washington
Washington, D.C. 20363-5100

Space and Naval Warfare Systems Command 1
Attn: Dr. Knudsen, Code PD50
Washington, D.C. 20363-5100

Ada Joint Program Office 1
OUSDRE(R&AT)
The Pentagon
Washington, D.C. 230301

Naval Sea Systems Command 1
Attn: CAPT Joel Crandall
National Center #2, Suite 7N06
Washington, D.C. 22202

Office of the Secretary of Defense 1
Attn: CDR Barber
The Star Program
Washington, D.C. 20301

Naval Ocean Systems Center 1
Attn: Linwood Sutton, Code 423
San Diego, CA 92152-5000

National Science Foundation 1
Division of Computer and Computation Research
Washington, D.C. 20550

Director of Research Administration 1
Code 012
Naval Postgraduate School
Monterey, CA 93943

Chairman, Code 52 1
Computer Science Department
Naval Postgraduate School
Monterey, CA 93943-5100

LuQi 150
Code 52Lq
Computer Science Department
Naval Postgraduate School
Monterey, CA 93943-5100