Calhoun: The NPS Institutional Archive

DSpace Repository

Faculty and Researchers | Faculty and Researchers' Publications

1988

A Static Scheduler for the Computer Aided Prototyping System, An Implementation Guide

Janson, D.; Luqi

Naval Postgraduate School

D. Janson and Luqi, "A Static Scheduler for the Computer Aided Prototyping System, An Implementation Guide", Technical Report NPS 52-88-006, Computer Science Department Naval Postgraduate School, 1988.

https://hdl.handle.net/10945/65244

Downloaded from NPS Archive: Calhoun

http://www.nps.edu/library

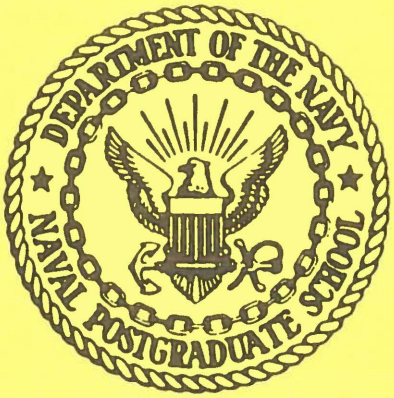
Calhoun is the Naval Postgraduate School's public access digital repository for research materials and institutional publications created by the NPS community. Calhoun is named for Professor of Mathematics Guy K. Calhoun, NPS's first appointed -- and published -- scholarly author.

Dudley Knox Library / Naval Postgraduate School
411 Dyer Road / 1 University Circle
Monterey, California USA 93943

T14

NPS52-88-006

NAVAL POSTGRADUATE SCHOOL
Monterey, California



A STATIC SCHEDULER FOR
THE COMPUTER AIDED PROTOTYPING SYSTEM:
AN IMPLEMENTATION GUIDE

Dorothy M. Janson

LuQi

March 1988

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

K. T. Marshall
Acting Provost

The work reported herein was supported in part by the Foundation Research Program of the Naval Postgraduate School with funds provided by the Chief of Naval Research.

Reproduction of all or part of this report is authorized.

This report was prepared by:

[redacted]
LUQI
Assistant Professor
of Computer Science

Reviewed by:

[redacted]
VINCENT Y. LUM
Chairman
Department of Computer Science

Released by:

[redacted]
JAMES M. FREMGEN
Acting Dean of Information
and Policy Science

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) NPS52-88-006	5. MONITORING ORGANIZATION REPORT NUMBER(S) NPS52-88-006
6a. NAME OF PERFORMING ORGANIZATION Naval Postgraduate School 6b. OFFICE SYMBOL (*If applicable*) 52	7a. NAME OF MONITORING ORGANIZATION Naval Postgraduate School
6c. ADDRESS (*City, State, and ZIP Code*) Monterey, CA 93943	7b. ADDRESS (*City, State, and ZIP Code*) Monterey, CA 93943
8a. NAME OF FUNDING / SPONSORING ORGANIZATION Naval Postgraduate School 8b. OFFICE SYMBOL (*If applicable*)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
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 STATIC SCHEDULER FOR THE COMPUTER AIDED PROTOTYPING SYSTEM: AN IMPLEMENTATION GUIDE (U)

12. PERSONAL AUTHOR(S) JANSON, DOROTHY M., LUQI

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (*Year, Month, Day*)	15. PAGE COUNT
Final	FROM Oct 87 TO Mar 88	March 1988	32

16. SUPPLEMENTARY NOTATION

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

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

Computer-aided rapid prototyping assists the software desi ner during the design and specification stages for hard real-time or embedded systems. Automated prototyping of these systems benefits from an Execution Support System (ESS) which validates software design before development of production software. This paper describes the pioneering efforts to implement the Static Scheduler component of the ESS which creates a static schedule, using worst case timing information, guaranteein that all critical timing constraints are met at run time.

20. DISTRIBUTION / AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION
☒ UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT ☐ DTIC USERS	UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL LuQi	22b. TELEPHONE (*Include Area Code*) (408)646-2735 22c. OFFICE SYMBOL 52Lq

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

A STATIC SCHEDULER FOR THE COMPUTER AIDED PROTOTYPING SYSTEM: AN IMPLEMENTATION GUIDE

Dorothy M. Janson

Luqi

Computer Science Department
Naval Postgraduate School
Monterey, California 93943

ABSTRACT

Computer-aided rapid prototyping assists the software designer during the design and specification stages for hard real-time or embedded systems. Automated prototyping of these systems benefits from an Execution Support System (ESS) which validates software design before development of production software. This paper describes the pioneering efforts to implement the Static Scheduler component of the ESS which creates a static schedule, using worst case timing information, guaranteeing that all critical timing constraints are met at run time.

KEYWORDS

rapid prototyping, hard real-time systems, static scheduler, Ada, specification language, computer-aided design

1. Introduction

The Department of Defense (DOD) and the Department of the Navy (DON) allocate billions of dollars each year for initial development or maintenance of progressively

more complex weapons and communications systems. These advanced systems increasingly rely on requirements for hard real-time processing of information, utilizing embedded computer systems to monitor or control system performance. These embedded systems currently perform time-critical functions that are primarily computational in nature, such as missile guidance or communications network control.

As they approach the 21st century, the DOD and the DON will be faced with ever increasing demands for complex, hard real-time or embedded systems. This growth of and dependency on embedded systems is readily apparent when compared with the growing civilian reliance on similar, small-scale systems to prepare their food and "drive" their automobiles. Considering the growth of software development and maintenance costs versus computer hardware costs [Ref. 1: p. 14], users must insist that delivered systems are received on schedule and are responsive to stated needs; that they are reliable, efficient and within cost estimates; and, that they are modifiable and transportable to other applications. Fulfilling these user demands requires a systematic approach to software development and an ability to deal with complex solutions.

1.1. Conventional Rapid Prototyping

Current research suggests a methodology for the software development life cycle, especially when designing hard real-time systems, which consists of two phases, rapid prototyping and automatic program generation [Ref. 2: p. 2]. Although current capabilities preclude completely automatic program generation, the required software tools and capabilities do exist for rapid prototyping. Rapid prototyping provides the user and designer with a fast, efficient and easy-to-use stepwise process. When utilized during the early stages of the development life cycle, rapid prototyping allows validation of the requirements, specifications and initial design before valuable time and effort are expended on implementation software.

Figure 1 on page 3 graphically describes this methodology as a typical feedback loop [Ref. 3: p. 3]. Rapid prototyping initially establishes an iterative process between the user and the designer to concurrently define specifications and requirements for the time critical aspects of the envisioned system. The designer then constructs a model or prototype of the system in a high-level, prototype description language. This prototype is a partial representation of the system, including only those critical attributes necessary for meeting user requirements, and is used as an aid in analysis and design rather than as production software. [Ref. 3: pp. 2-5] During demonstrations of the prototype, the

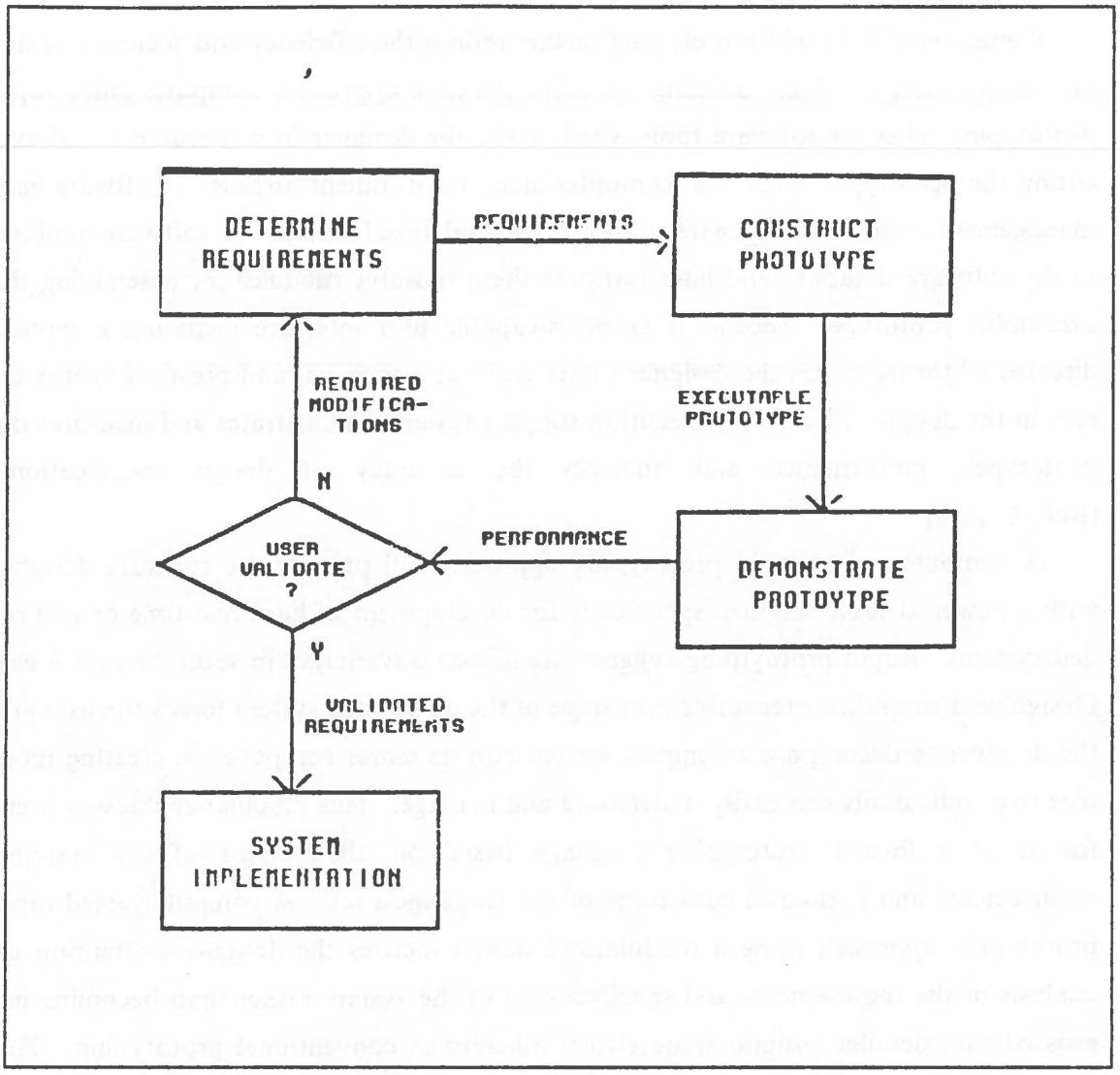


Figure 1. Rapid Prototyping Method

user validates the prototype's actual performance against its expected performance. If the prototype fails to execute properly or to meet any critical timing constraints, the user identifies required modifications and redefines the critical specifications and requirements. This process continues until the user determines that the prototype successfully meets the time critical aspects of the envisioned system. Following this final validation, the designer uses the prototype as a basis for the design and eventual hand coding of the production software.

1.2. Computer-Aided Rapid Prototyping

Computer-aided rapid prototyping further refines the efficiency and accuracy of this new methodology. While utilizing the same iterative approach, computer-aided rapid prototyping relies on software tools which assist the designer in constructing and executing the prototype. First, the computer-aided environment includes a software base management system which creates uniform retrieval specifications for software modules in the software database and later retrieves these reusable modules for assembling the executable prototype. Second, a graphics-capable user interface including a syntax-directed editor expedites the designer's data entry at a terminal and prevents syntax errors in the design. Finally, an execution support system demonstrates and measures the prototype's performance and analyzes the accuracy of design specifications. [Ref. 4: p. 4]

A computer-aided rapid prototyping approach will provide the software designer with a powerful tool, designed specifically for development of hard real-time or embedded systems. Rapid protoytping suggests significant advantages in several major areas. Designing a simplified executable prototype of the envisioned system forces the user and the designer to decompose a complex system into its major components, creating modules that individuals can easily understand and manage. This modularized design is enforced by a formal prototyping language based on abstractions of the system's requirements and high-level constructs of the language itself. A computer-aided rapid prototyping approach using a modularized design focuses the designer's attention on analysis of the requirements and specifications of the system rather than becoming engrossed with detailed programming efforts inherent in conventional prototyping. This approach allows the user to verify requirements and to identify problem areas early in the development cycle. This verification process eliminates expensive redesign efforts and increases the user's confidence that the system, as envisioned, is feasible. [Ref. 2: pp. 2-3]

Rapid prototyping offers promising advantages in improved software engineering productivity, increased reliability of the finished product, more realistic cost estimates based on identified system complexity, and a reduction in the total conception to operational timeframe [Ref. 3: pp. 11-12]. Ongoing research and pioneering efforts must now fully elevate computer-aided rapid prototyping from its conceptualized design into a functioning reality.

2. Computer Aided Prototyping System (CAPS)

The computer-aided rapid prototyping tool addressed in this paper which incorporates a Static Scheduler is the CAPS. Recognizing that available prototyping methodologies require extensive amounts of individual time and effort, CAPS is designed specifically as a development tool for hard real-time systems. Its primary objective is development of a specification method for identifying and later retrieving reusable software components from an online database while utilizing a formal prototyping language. Together with an iterative process similar in concept to Figure 1 on page 3, CAPS will provide an effective and efficient tool for constructing and validating a prototype. [Ref. 3: pp. 1-2] Rapid construction of this prototype relies on the applicable prototyping method and on a support environment which automates the steps involved. The following sections and Figure 2 on page 6 describe the components of the CAPS architecture and how they work together to make computer-aided rapid prototyping possible.

CAPS is initialized through the User Interface (UI) as the user and designer work together in defining the critical attributes of the envisioned system. The UI consists of a syntax-directed editor for the formal prototyping language and a graphics tool for displaying data flow diagrams. The editor eliminates syntax errors by prompting the designers with appropriate alternatives at each step of the design process. The graphics tool provides a picture of the data flow diagrams which reinforces the verbal description of the system specifications. [Ref. 5: pp. 6-7]

The PSDL was designed as the primary connection between the designer and the remaining components of CAPS. By definition, PSDL is a high-level prototyping language with special features appropriate for defining critical real-time constraints and is applied at the specification or design stage. [Ref. 6: pp. 3, 23] In order to rapidly construct a prototype, PSDL also includes its own associated prototyping method and automated support environment. Using a top-down design approach, the PSDL method aids the designer in systematically refining and decomposing each critical component into its lower level components. Uniform PSDL specifications associated with each lower level description act as templates for retrieving reusable software components having similar specifications from the CAPS Software Database. Thus, the PSDL method produces a computational model consisting of the basic building blocks needed to describe the abstractions and concepts of the hierarchically structured prototype. The PSDL execution support environment then verifies the design and the validity of the prototype's real-time requirements. The actual execution of the prototype demonstrates

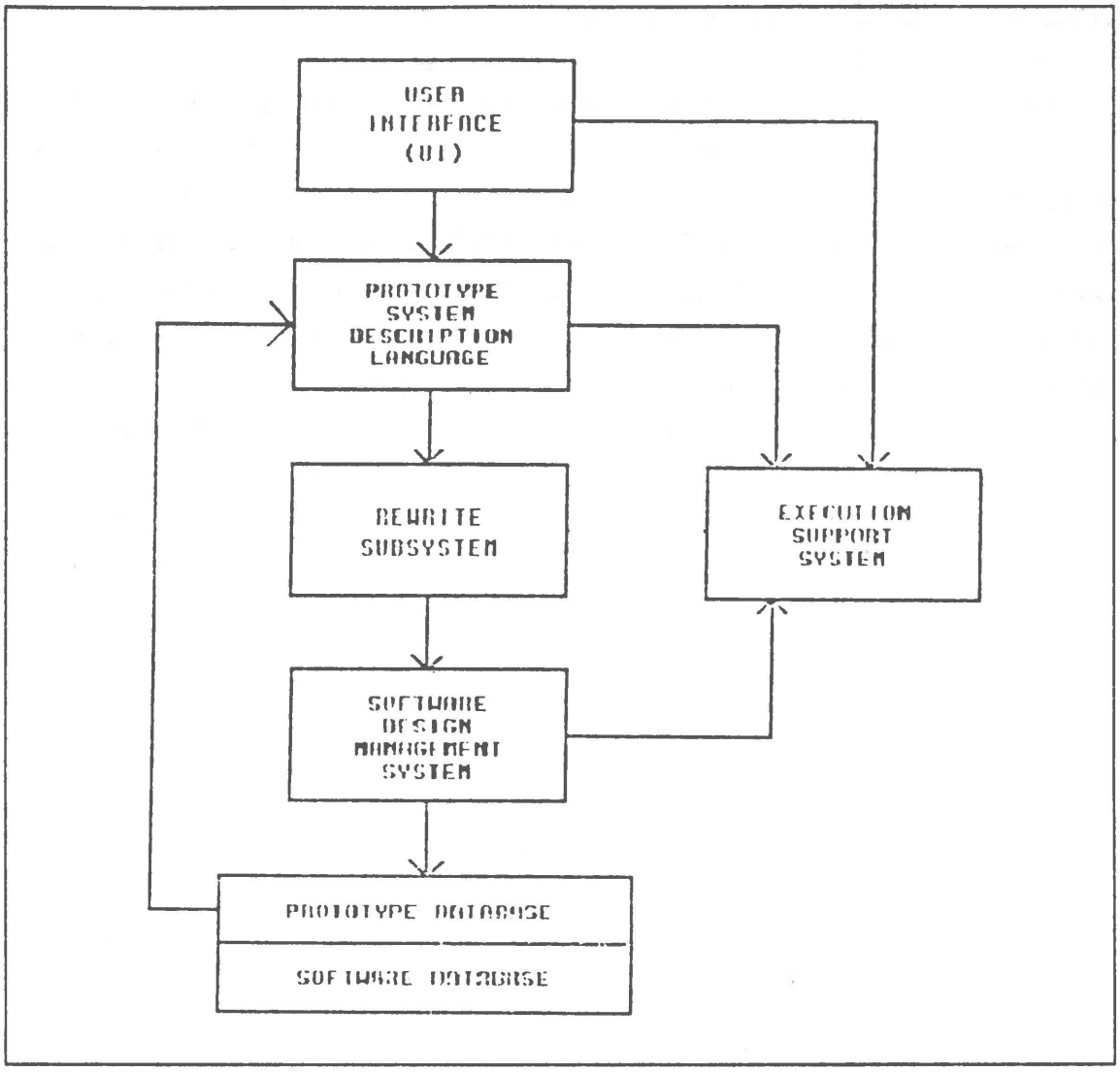


Figure 2. CAPS Architecture

whether these critical timing constraints will perform in an acceptable manner that meets the timing constraints of the system as a whole [Ref. 5: pp. 2-7].

The CAPS Rewrite Subsystem provides a means for automatically generating uniform specifications for each reusable software module in the CAPS Software Database and for each PSDL lower level component. Existing methods for retrieving reusable modules are based on keywords. The Rewrite Subsystem, however, uses an approach which allows the designer to give more precise PSDL specifications. The Rewrite Subsystem then transforms each specification into a uniform or normal form. This trans-

formation is achieved by mapping a semantic alias to its normalized term as shown in Figure 3 on page 8. [Ref. 2: pp. 7-8] These normalized specifications are free from ambiguity and create a template used by the Software Design Management System (SDMS) to retrieve software modules from the database with corresponding normalized specifications [Ref. 7: pp. 3-10].

The SDMS is similar to a database management system with additional features required for computer-aided design applications. The SDMS is responsible for organizing, retrieving and instantiating the reusable software modules from the CAPS Database. Retrieval of reusable modules is supported by the normalized specification templates described above. The SDMS instantiates these modules as specified by the designer for execution of the current PSDL prototype. Overall, the SDMS supports efficient selection and retrieval of the relevant software modules. [Ref. 2: p. 9] This minimizes instances where a non-match requires manual creation of a new software module before execution of the prototype occurs.

Two distinct subdivisions of the CAPS Database are the Prototype Database and the Software Database. The first maintains and manages the PSDL prototypes along with their sets of requirements. It also records successive refinements of the prototype alternatives. This process includes facilities for backtracking to previous versions or for combining successive design decisions from the different alternatives. [Ref. 4: p. 10] The Software Database contains the reusable software modules together with their PSDL normalized specifications. This database allows future growth as new modules are identified for inclusion into the database.

The Execution Support System (ESS), although a component of the CAPS architecture, actually provides the execution support environment for the PSDL prototyping method. After assembling a prototype of the envisioned system, the three ESS subsystems provide the capability to demonstrate the prototype's actual performance. One subsystem, the Static Scheduler, reads the PSDL prototype source program to identify and extract the PSDL operators with their timing constraints and precedence relationships. For operators with critical timing constraints, the Static Scheduler creates a static schedule, if a feasible one exists, guaranteeing their accurate execution using worst case time scenarios. A second subsystem, the Translator, also reads and translates the PSDL prototype source program to augment implementation of atomic operators and data types with software code. The Translator accomplishes this by communicating the PSDL prototype's specifications to the SDMS and assembling the executable prototype

TERM	ALIASES
READ	GET, FETCH, OBTAIN, INPUT, RETRIEVE
UPDATE	CHANGE, MODIFY, REFRESH, REPLACE

Replace all occurences of an alias with its term.

Example: "RETRIEVE temperature from thermometer and
REFRESH the temperature_reading"

is normalized to

"READ temperature from thermometer and
UPDATE temperature_reading"

Figure 3. Normalizing a Specification

from the reusable software modules. The final subsystem, the Dynamic Scheduler, maintains run-time execution control of the prototype using the completed static schedule. The Dynamic Scheduler must also schedule operators without critical timing constraints during any excess time slots that remain after each time critical operator completes execution. [Ref. 3: pp. 6-7]

3. Execution Support System

CAPS as an effective and efficient tool for creating an executable prototype of a hard real-time system. Rapid construction and validation of a prototype require an execution support environment that automates the many time-consuming tasks involved with developing the design up through analyzing the performance of the prototype. Although the ESS represents only one component of the CAPS prototyping tool, it is the primary factor that differentiates CAPS from manual prototyping tools. The ESS provides automated control and interface capabilities that allow the system to save cur-

rent state information when run-time discrepancies are noted and to execute modified versions of the prototype without repeating the initial steps of the execution [Ref. 5: p. 7]. These capabilities are essential to realize time and cost savings and to increase user confidence that the system's design is feasible.

The ESS contains a Dynamic Scheduler, a Static Scheduler, and a Translator. As conceptualized in Reference 8, Figure 4 on page 10 illustrates the ESS subsystems' external interfaces to other components of CAPS and the interactions within the ESS itself. The ESS utilizes four external interfaces from outside the ESS. Initially, a command from the UI to the Dynamic Scheduler activates the ESS when the designer requests execution of the PSDL prototype. Second, the Translator and the Static Scheduler require access to the PSDL prototype source program created jointly by the designer and the user. Third, the Combiner Linker Exporter (CLE) receives, compiles and links the Ada source code from the Translator and the Static Scheduler. The executable code generated by the CLE becomes input to the Dynamic Scheduler for run-time execution. The final interface from the Dynamic Scheduler to the UI provides prototype execution statistics, analysis and error message information direct to the designer.

3.1. Dynamic Scheduler

The conceptualized design for the Dynamic Scheduler utilized in this paper derives directly from Reference 9 and as initially conceived in Reference 3. The Dynamic Scheduler fulfills two major roles for the CAPS. First, the Dynamic Scheduler exercises the prototype which is a fundamental requirement of a rapid computer-aided prototyping system. Second, prompt feedback from the Dynamic Scheduler to the UI allows the designer and the user to assess the prototype's execution performance. In order to perform these roles, the Dynamic Scheduler coordinates the functions of the entire ESS.

In its first role, the Dynamic Scheduler acts as the primary link between the CAPS UI and the ESS. When the designer completes the PSDL prototype source program, a request from the UI (or a potential subsystem of the UI) to execute the PSDL prototype is received by the Dynamic Scheduler. The Dynamic Scheduler in turn invokes the Translator and the Static Scheduler, and initiates procedures which pre-load data stream buffers for the Translator. The Translator transforms the PSDL prototype program into an executable Ada program while the Static Scheduler provides a linear static schedule for time critical operators. An Ada-compiled implementation of the prototype then becomes an input to the Dynamic Scheduler. After successful completion of these pre-

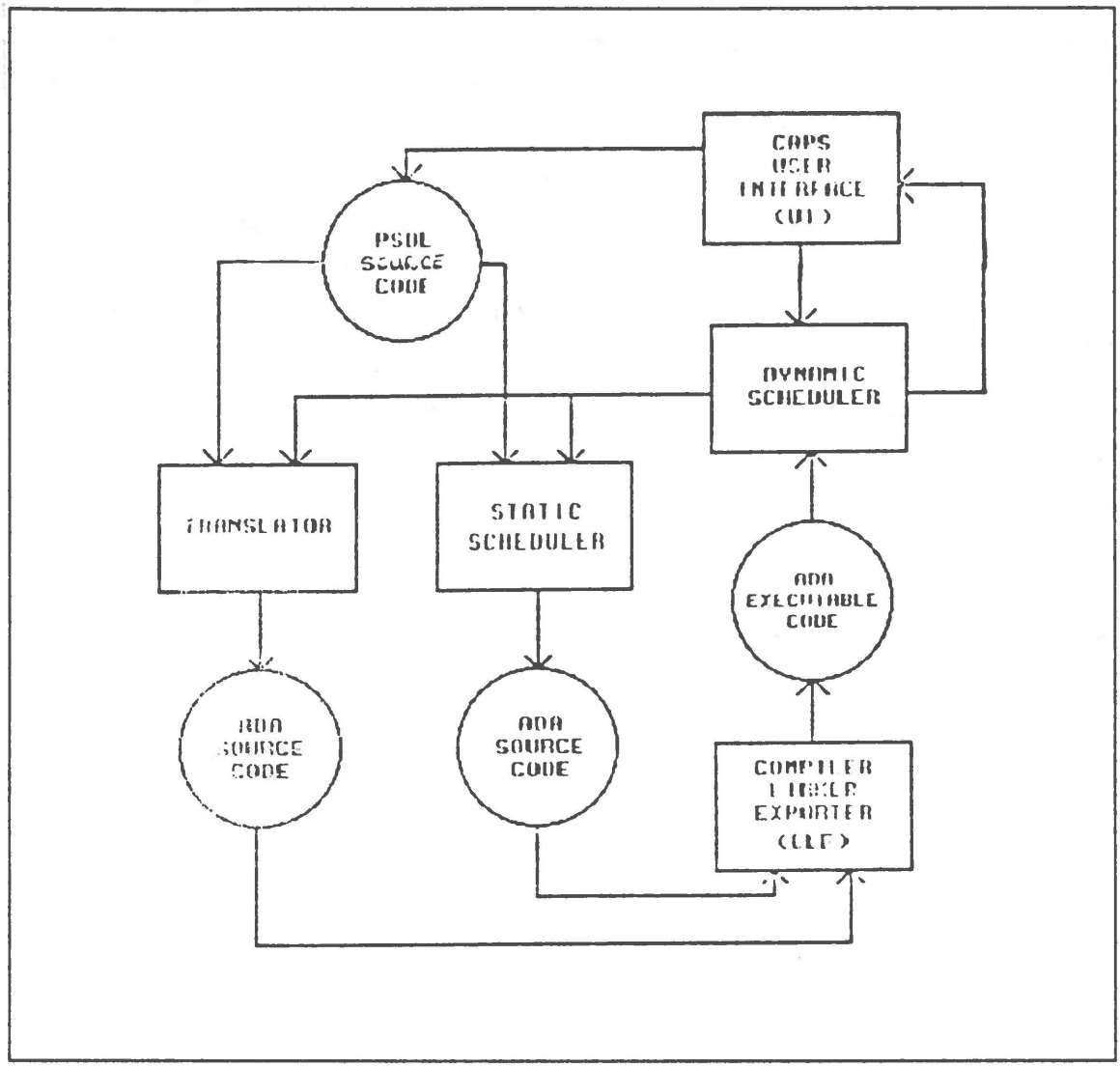


Figure 4. Execution Support System Interfaces

execution functions, the Dynamic Scheduler provides all run-time executive activities while exercising the prototype.

As the ESS driver program, the Dynamic Scheduler has two main responsibilities. First, it schedules operators without timing constraints that were not included in the static schedule. The Dynamic Scheduler receives an input file from the Static Scheduler containing these unscheduled operators. Since the Static Scheduler uses worst case times (METs) for scheduling the critical operators, on the average these operators will not utilize the entire allotted time slots. The Dynamic Scheduler identifies this spare capac-

ity as it occurs and schedules operators without timing constraints during the time remaining. In some cases the operators so scheduled may not complete execution during this time slot. The Dynamic Scheduler must then preempt the operator's execution and abandon the entire operation before the next pre-scheduled critical operator must begin execution.

The second ESS responsibility of the Dynamic Scheduler provides run-time exception handling and hardware/operator interrupts. EXCEPTIONs are raised from the Translator as either dataflow overloads when an operation attempts to write to a buffer that is full or dataflow underloads when an operation attempts to read from an empty buffer. EXCEPTIONs are raised from the Static Scheduler when a critical operator exceeds its worst case (MET) time slot or validity checks on constraint values indicate a static schedule is not feasible. In all of these cases, execution of the prototype will stop and an applicable error message is displayed at the UI. The Dynamic Scheduler must also handle conventional interrupts, such as a <control> C from the user or an equipment failure.

Future enhancements identified in addition to the current Dynamic Scheduler design would provide debugging capabilities and statistical information. During execution of the prototype, the debugging capabilities would trace relevant information concerning operator execution. Computed values and their associated input and output times would display a record of events that occur during execution. Statistical information collected during execution would include frequency of operator firing, quantity of EXCEPTIONs occurring, and statistical data on timing parameters for critical operators. [Ref. 3: pp. 10-11] When combined, these two enhancements would provide the designer and the user with precise information for measuring, analyzing and validating the prototype's performance.

3.2. Translator

The conceptualized design for the Translator utilized in this paper derives directly from Reference 10 and as initially conceived in Reference 3. The Translator's primary responsibility is the translation of the PSDL prototype source program into an executable Ada program that simulates the behavior of the prototype. The Translator accomplishes this translation by utilizing a version of the Kodiyak translator generator specifically tailored for this application. The Translator invokes the AG translator program which semantically parses the PSDL program statements into their Ada program

representations. The translator contains a list of PSDL grammar statements with their associated attribute definition equations that represent the corresponding Ada grammar. These equations define the semantics of the translation using a structured grammar tree approach.

Augmentation code for PSDL atomic operators is embedded within the attribute definition equations. These augmentations implement the PSDL data streams, PSDL operator conditional constraints and PSDL TIMER functions. Both sampled and data flow stream augmentations are implemented with individual buffers containing one computed value for each stream. PSDL operator triggering conditions and output guards are implemented by the equivalent Ada semantics. A PSDL TIMER is implemented using the CLOCK function from the standard Ada library package CALENDAR.

During the early prototype design phase, any PSDL composite operators are decomposed into atomic operator networks. Reusable modules from the CAPS Database, considered as Ada program units for this thesis, are inserted as the implementation code for these operators. The augmentation code described above, combined with these Ada implementations, produces the prototype's Ada source code. The CLE compiles this source code and links it with the compiled static schedule to generate the executable Ada program. This executable program then becomes the input to the Dynamic Scheduler for execution of the prototype.

4. The Static Scheduler

The conceptualized design for the Static Scheduler utilized in this paper derives from Reference 8 and 11 and as initially conceived in Reference 3. The primary development emphasis for CAPS was computer-aided rapid prototyping for hard real-time systems. By automating many of the time-consuming tasks of conventional rapid prototyping tools, the ESS and the SDMS differentiate the CAPS from its manual or semi-automated counterparts. But the Static Scheduler subsystem of the ESS alone represents the single most important component of CAPS as the basic requirement for computer-aided rapid prototyping of hard real-time systems. The Static Scheduler specifically addresses only those operators with critical timing constraints whose precise performance determine whether the system, as designed, will meet the required timing specifications.

As conceptualized, the primary purpose of the Static Scheduler is creation of a static schedule which gives the precise execution order and timing of operators with hard

real-time constraints in such a manner that all timing constraints are guaranteed to be met [Ref. 3: p. 7]. Assuming that such a schedule is feasible given the system specifications, the static schedule contains the pre-allocated starting time and execution time for each critical operator. This structure implicitly denotes the precedence relationships between the operators. Without the benefit of a Static Scheduler, execution of the prototype would rely on basic control flow and processor scheduling as currently utilized in the majority of software systems. Rapid prototyping in general would benefit from CAPS without a static schedule. However, the Static Scheduler provides CAPS with the unique capability required to realize increased gains in designer productivity and system reliability during development of hard real-time systems.

The remainder of this section provides implementation guidelines describing how the Static Scheduler functions as conceptualized in Reference 8 and 11. The implementation design in this paper addresses a single processor application only. The impact on or modification to this design when multi-processors and concurrent processing are utilized will not be addressed explicitly. Data Flow Diagrams (DFDs) will illustrate the conceptualized design for implementation of the Static Scheduler. The 1st level DFD presents a general description of the Static Scheduler while the lower level DFDs contain more specific implementation guidelines.

4.1. Static Scheduler 1st Level DFD

The 1st Level DFD, Figure 5 on page 14, outlines the five major functions of the conceptualized Static Scheduler [Ref. 11]. The initial input to the Static Scheduler is a text file containing the PSDL prototype program created jointly by the designer and the user. An intermediate output to the Dynamic Scheduler is a text file containing the non-time-critical operators that were extracted from the PSDL program together with the time critical operators. The final output from the Static Scheduler to the CLE is an Ada source file containing the static schedule. The CLE compiles and links this program to the Translator's compiled Ada program. This combined program is the executable Ada program used by the Dynamic Scheduler to demonstrate the prototype's performance. The following sections describe the functions performed by each lower level DFD.

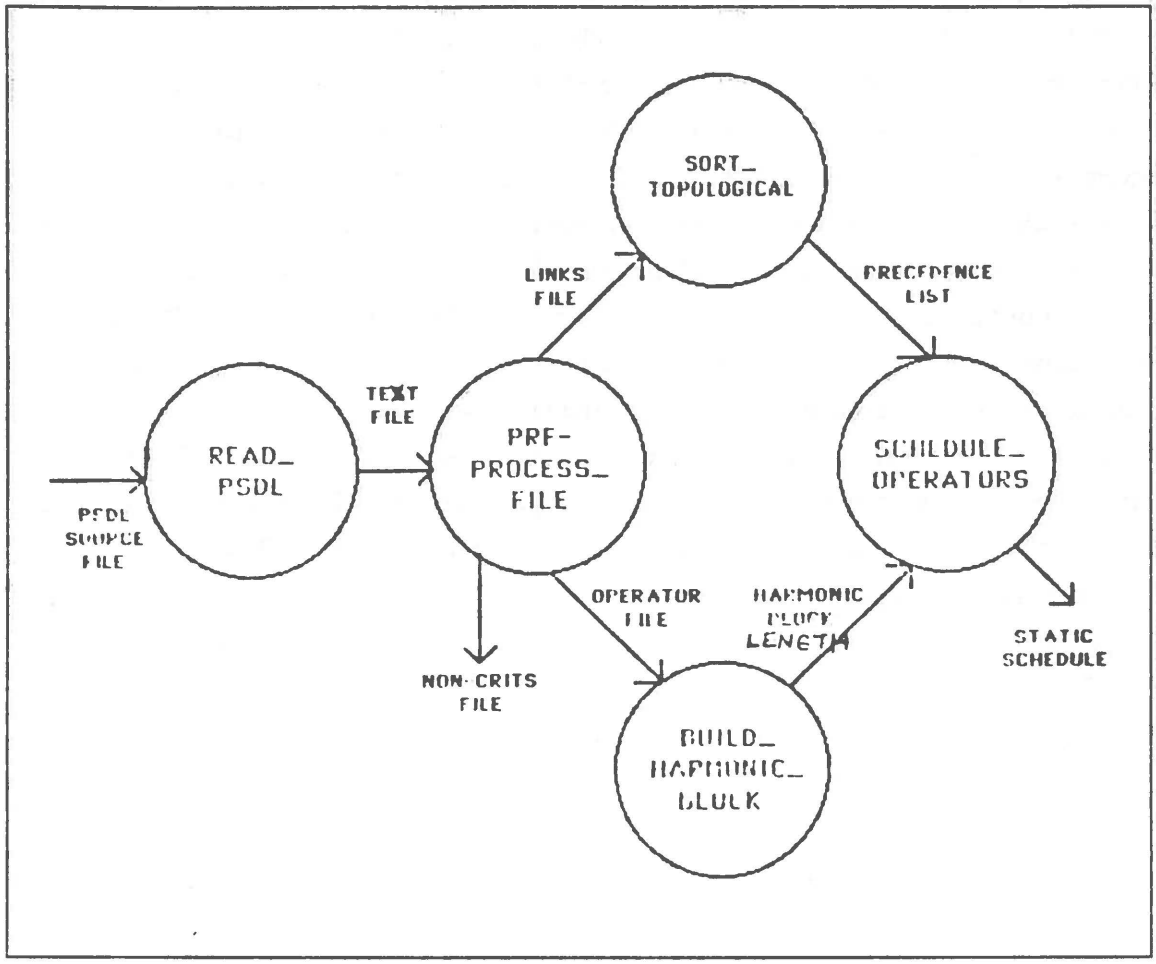


Figure 5. 1st Level DFD

(a) "Read_PSDL"

Following initiation by the Dynamic Scheduler, the Static Scheduler's first major function is reading and processing the PSDL prototype program. Although the Translator performs a similar but extensive process for the entire PSDL program, the Static Scheduler requires only that information which identifies critical operators along with their associated timing constraints and the link statements which syntactically describe the PSDL graphs. A specifically tailored version of the AG-based Kodiyak tool for the Static Scheduler identifies and extracts this information only from the PSDL source program. This process creates a sequential text file containing operator identifiers, timing information and link statements.

As conceptualized, implementation of the current design is based on two assumptions. First, this design assumes that the PSDL prototype program is syntactically correct. This implies that each line begins with a PSDL keyword or reserved word. Second, this design assumes that the designer structured the PSDL prototype program using a top-down design. This implies that the program begins with the highest level and then decomposes all composite operators, with the last (or lowest) level being the Ada implementation modules. These assumptions are realistic in that PSDL encourages the designer's use of a structured, modular architecture and also that the UI will include a syntax-directed editor.

(b) "Pre-Process_File"

After the AG processor creates the output text file, the Static Scheduler's next major function is sorting the contents of this file and performing basic validity checks on pertinent information. The input text file is a sequentially ordered file containing all required information as it was extracted from the PSDL program. This information must be divided into three separate files based on its destination or additional processing required. The Non-Crits file contains a sequential list of all non-time-critical operator identifiers which becomes an intermediate input to the Dynamic Scheduler. The Dynamic Scheduler schedules these operators during execution of the prototype as excess time becomes available. The Operator file contains all critical operator identifiers and their associated timing constraints. This file is organized as an array of records. Each record contains fields for the operator identifier and the numeric values of its MET, MRT, MCP, period and FINISH_WITHIN. The Links file contains the link statements which syntactically describe the PSDL implementation graphs. This file is also organized as an array of records. Each record contains fields for the data stream identifier which communicates between the two operators, the producer operator identifier and the numeric value of its MET, and the consumer (user) operator identifier. The derivation of these link statements appears in the section "Sort_Topological".

During this phase, the Static Scheduler also performs basic validity checks on the timing constraints contained in the critical operator file. At a minimum, three validity checks performed at this stage will increase the probability of creating a feasible schedule during the later stages. The first check verifies that an operator record containing an MCP value also contains an MRT value. If the MRT is missing, it is calculated as either (MRT equals FINISH_WITHIN) or (MRT equals MET). A second check verifies

that an operator's MET value does not equal its period value, if a period is present in the record. A third check verifies that an operator's MET value is less than its MRT value. [Ref. 12: p. 6] In all three cases, if any one of the checks fails an EXCEPTION will be raised and an appropriate error message submitted to the UI. The significance of these validity checks will become apparent in the section for "Build_Harmonic_Blocks".

As conceptualized, implementation of the current design is based on two assumptions. First, this design assumes that critical operators will always include an MET value. If this value is not present, the operator is assumed non-time-critical and is delivered to the Dynamic Scheduler. Second, this design assumes that all timing constraints are non-negative integer values. These assumptions are realistic in that "critical" here implies maximum or minimum timing constraints. In addition, a negative time value would be meaningless and the time units available in PSDL (i.e. mille- or micro-seconds) provide sufficient time divisions.

(c) "Sort_Topological"

After the "Pre_Process_File" function creates its output files, the Static Scheduler's next major function is performing a topological sort of the link statements contained in the Links file. In order to appreciate the complexity of this topological sort, Figure 6 on page 17 illustrates a PSDL linear augmented graph and its corresponding sorted link statements. Lower case characters identify data streams which provide data value communication paths between two operators. The upper case characters identify the critical operators. An operator identifier appearing on the left side of the arrow represents a producer of data. An operator identifier appearing on the right side of the arrow represents a consumer (user) of data values. The special word EXTERNAL identifies a situation where the data input/output arrives/exits the current level of the system under consideration depending on whether EXTERNAL is located on the left/right of the link statement. The numerical value on the right side of the colon records the MET value and unit of measurement for the operator identifier on the left side of the colon.

All link statements conform to this same format regardless of the level of decomposition under consideration. However, as these lower levels are encountered, each single statement could be replaced by or affect two or more statements. As an example, Figure 7 on page 18 illustrates the decomposition of operator B from Figure 6 on page 17. A comparison of the two figures indicates that two new statements were added:

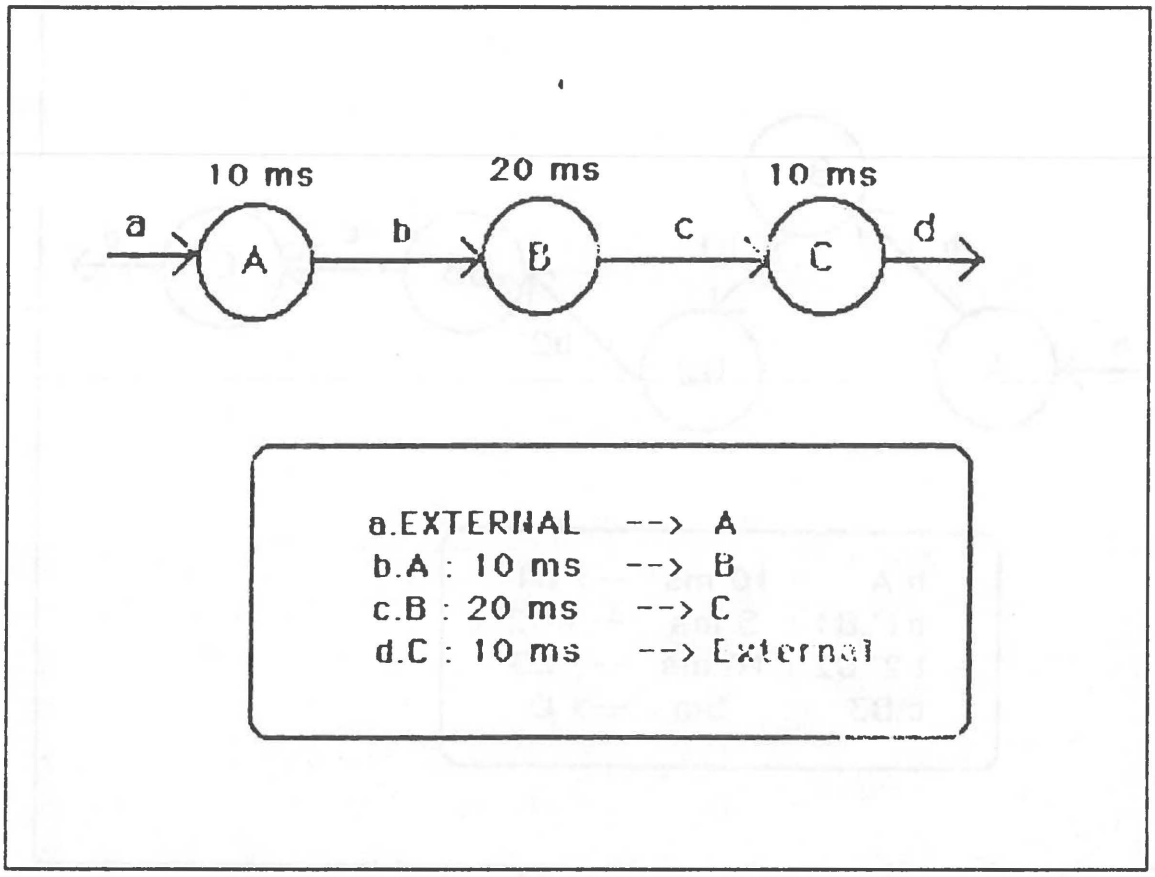


Figure 6. PSDL Graph and Link Statements

- b1' . B1 : 5 ms --> B2
- b2' . B2 : 10 ms --> B3

and two statements were modified:

- b . A : 10 ms --> B is now b . A : 10 ms --> B1
- c . B : 20 ms --> C is now c . B3 : 5 ms --> C

All of the link statements from each level appear sequentially in the Links file as they were extracted from the PSDL prototype program.

The requirement for a topological sort implies that the statements being sorted have a natural continuity and connectedness. These properties define the execution precedence of the time critical operators regardless of whether the graphs are linear or acyclic. With a linear graph, the sort establishes a start point by locating the statement containing the EXTERNAL keyword in the left-hand operator position. Conversely, the

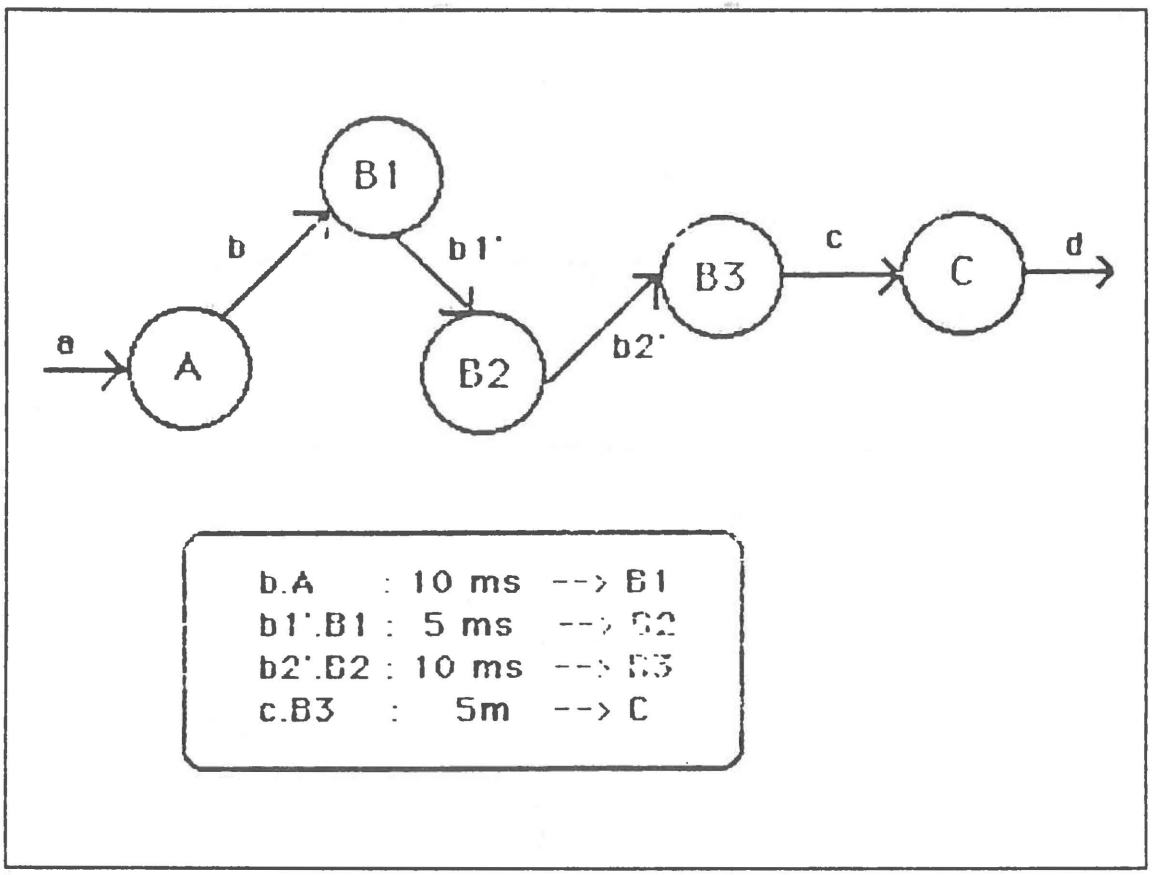


Figure 7. Decomposition of Operator B

end point is established by locating a statement containing the EXTERNAL keyword in the right-hand operator position. The remaining operators are ordered by locating matches between the right and left-hand operators. Sorting an acyclic digraph differs only in how the start and end points are established. The sort establishes a start point by locating the statement(s) having a left-hand operator with no matching right-hand operator. The end point is established by locating the statement having a right-hand operator with no matching left-hand operator. An augmented acyclic digraph is illustrated in Figure 8 on page 19. In this type of digraph, a decision to choose the "a.A" link first and the "d.A" link last is arbitrary. In either linear or acyclic, the output is a precedence list of critical operators stipulating the exact order in which they must be executed.

As conceptualized, implementation of the current design is based on two assumptions. First, this design assumes that the link statements are formatted correctly. This

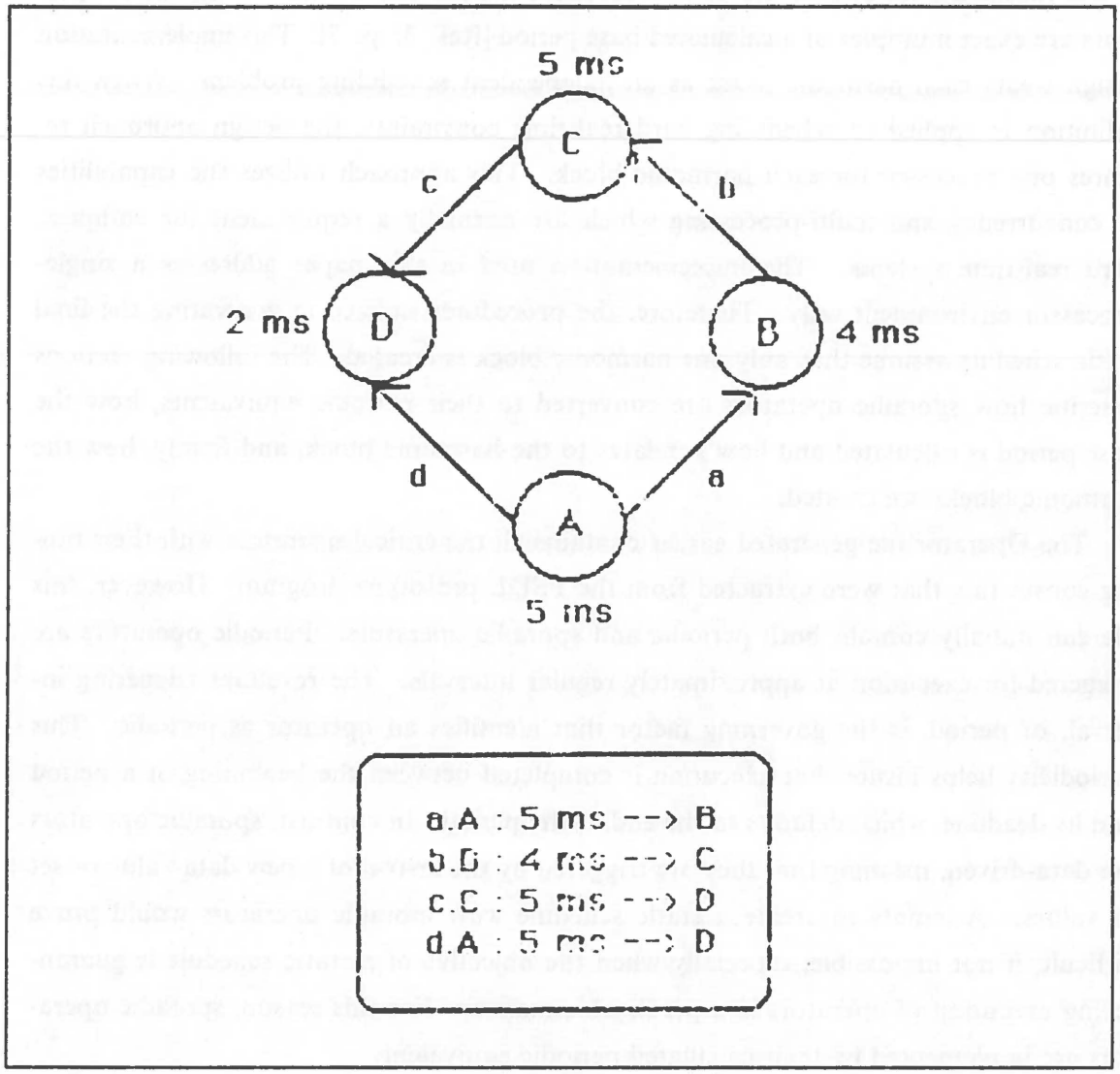


Figure 8. PSDL Augmented Acyclic Digraph

assumption is realistic here and in future designs when the UI contains a syntax-directed editor. Last, although this design assumes a linear graph, the sort procedure will accommodate both linear graphs and acyclic digraphs. The linear sort produces one precedence list while the acyclic sort produces two or more lists.

(d) "Build_Harmonic_Blocks"

A second output of the "Pre-Process_File" function, the Operator file, is the input to the next major function of building harmonic blocks. An harmonic block as defined

in this thesis is a set of periodic operators where the periods of all its component operators are exact multiples of a calculated base period [Ref. 3: p. 7]. This implementation design treats each harmonic block as an independent scheduling problem. When this definition is applied to scheduling hard real-time constraints, the design approach requires one processor for each harmonic block. This approach utilizes the capabilities of concurrency and multi-processing which are normally a requirement for complex, hard real-time systems. The implementation used in this paper addresses a single-processor environment only. Therefore, the procedures utilized in generating the final static schedule assume that only one harmonic block is created. The following sections describe how sporadic operators are converted to their periodic equivalents, how the base period is calculated and how it relates to the harmonic block, and finally, how the harmonic blocks are created.

The Operator file generated earlier contains all the critical operators with their timing constraints that were extracted from the PSDL prototype program. However, this file can initially contain both periodic and sporadic operators. Periodic operators are triggered for execution at approximately regular intervals. The resultant triggering interval, or period, is the governing factor that identifies an operator as periodic. This periodicity helps insure that execution is completed between the beginning of a period and its deadline, which defaults to the end of the period. In contrast, sporadic operators are data-driven, meaning that they are triggered by the arrival of a new data value or set of values. Attempts to create a static schedule with sporadic operators would prove difficult, if not impossible, especially when the objective of a static schedule is guaranteeing execution of operators in a predictable manner. For this reason, sporadic operators are implemented by their calculated periodic equivalent.

The first preliminary step in creating a static schedule uses an algorithm [Ref. 3: p. 8] which calculates the periodic equivalent for all sporadic operators. Use of this algorithm requires that all sporadic operators (those without periods) have values for MCP, MRT, and MET. If any of these values are missing they must be calculated from the available information. The MRT value was computed during the "Validate_Data" function. This implementation assumes that MET is given for all time critical operators and that either a period, an MCP, or both are also given for all critical operators. This author interprets the latter as indicating that an operator with both values defaults to a periodic operator. The following relationships between these values must exit to calculate a valid operator:

1. MET < MRT
2. MCP < MRT
3. MET < MCP.

The first condition insures that (MRT - MET) produces a positive value. The second condition is necessary, but it is not sufficient to insure a valid period. This condition guarantees that an operator can fire at least once before a response is expected. The last condition insures that the period calculated will conform to a single-processor environment. [Ref. 12: p. 6] The periodic equivalent is then calculated as:

P = minimum (MCP, MRT - MET).

The value of P must be greater than MET in order for the operator to complete execution within the calculated period. If this test fails, a last resort is setting P equal to MCP as a worst case, or tightest, scheduling constraint.

After all operators are in periodic form, they are sorted in ascending order based on the period values. This sort assumes that all units of time measurement were previously converted to microseconds. A second preliminary step to creating the static schedule uses an algorithm which calculates the base block and its period for the sorted sequence of operators. Within this paper, the base period is defined as the greatest common denominator (GCD) of all operators in one sequence (or block) that will be scheduled together. Two algorithms can be used for determining the GCD. One addresses a single-processor environment only. This algorithm divides each period value in the sequence by the smallest period value. Whenever a remainder occurs, the denominator is decreased by one and the process repeats until all remainders equal zero. This algorithm results in one sequence of periods (the base block) with one base period (the GCD).

The second algorithm, applicable to multi-processor environments, is similar in design but results in one or more base blocks, each having a unique GCD and a unique sequence of operators. An initial pass through all of the periods results in two sequences, only one of which is a final base block with a GCD. When division results in a zero remainder, the period is placed in a primary sequence. When division results in a non-zero remainder, the period is placed in an alternate sequence. Subsequent passes only use the most recent alternate sequence. This process is continued until the alternate sequence equals the null set. This implementation uses the second algorithm for two reasons. First, although the basic designs are similar, the implementation is more straightforward. Second, for a single-processor environment, the second pass verifies

that all periods were assigned correctly to the first sequence if the alternate sequence equals the null set.

The last preliminary step to create the static schedule uses an algorithm which calculates the length of time for the harmonic block. In a single-processor environment, the operators and their periods used to create the base block are the same as those in the harmonic block. The actual harmonic block length is the least common multiple (LCM) of all the operators' periods contained in the block. The algorithm first calculates the GCD as above for the first pair of periods in the block. The LCM is then calculated by dividing the product of this pair by the GCD. The calculated LCM is paired with the next period in the block, after which the GCD and LCM are again calculated. The LCM calculated using the last such pair is the LCM for the harmonic block. Mathematically, for a block of four periods the algorithm corresponds to

LENGTH = LCM [{ LCM (LCM, Period_3), Period_4 }].

The harmonic block and its length are an integral part of creating the static schedule. This block represents an empty timeframe within which the operators will be allocated time slots for execution.

(e) "Schedule_Operators"

The "Sort_Topological" and "Build_Harmonic_Blocks" functions generated output files for Precedence_Lists and Harmonic_Blocks, respectively. Both of these files are necessary to create a static schedule for time critical operators. The Precedence_Lists file contains the required sequential execution order for all time critical operators. The Harmonic_Blocks file contains the basic timeframe within which the critical operators are allocated non-overlapping time slots. The resulting static schedule is a linear table giving the exact execution start time for each critical operator and the reserved maximum execution time (MET) within which each operator completes it execution.

The algorithm used in this implementation is a two-step process, both of which use the operators' periods and METs. The first iterative process performs two distinct functions. Initially, it allocates an execution time interval for each operator [Ref. 13: p. 126] based on

INTERVAL = (current_time, current_time + MET).

Next the process creates a firing interval for each operator during which the second iterative process must schedule the operator. The firing interval stipulates the lower and

upper bound for the next possible start time for an operator based on its period. As an example, OP_2 in Figure 9 on page 24 is scheduled to begin execution at time 2 and to complete execution by time 3 based on its MET of 1. With a period of 10, OP_2 can not fire again before time 12, the lower bound. But OP_2 must fire at or before time 21, the upper bound, in order to guarantee that execution is completed on or before time 22.

The second process has three distinct functions. Initially, it uses the lower bound of each firing interval when it schedules operators during subsequent iterations. The sequence of operators is allocated time slots according to the earliest, lower bound first. For the example in the previous figure, the operators are scheduled in the order { OP_1, OP_2, OP_3 } during the first iteration in this process. Since OP_4 has a period of 20 units and the harmonic block length is also 20 units, OP_4 is scheduled only once in each harmonic block. Before an operator is allocated a time slot, this process verifies that either:

1. (current_time + MET) < harmonic block length
2. (current_time + MET) = < harmonic block length.

The second condition is applicable to the last operator scheduled in that harmonic block only. Failure to meet either condition results in an infeasible schedule. This situation raises an EXCEPTION which halts execution since the timing constraints of that operator, or of future operators in the next iteration, will not be met.

This process also calculates new firing intervals for each operator scheduled. As an example, Figure 10 on page 25 shows the static schedule and two harmonic blocks after three iterations of this process. This example illustrates the importance of calculating an accurate harmonic block. Once all operators are correctly scheduled within an entire harmonic block, all subsequent harmonic blocks are mirror-images of the first -- a static schedule.

5. Design Environment

During this research effort, a software translator generator tool and a dynamic programming language provided the environment required to design a feasible Static Scheduler. The Kodiyak AG translator generator provided a software tool which extracted the required information from the PSDL prototype source program. The Ada programming language provided a set of constructs specifically designed for develop-

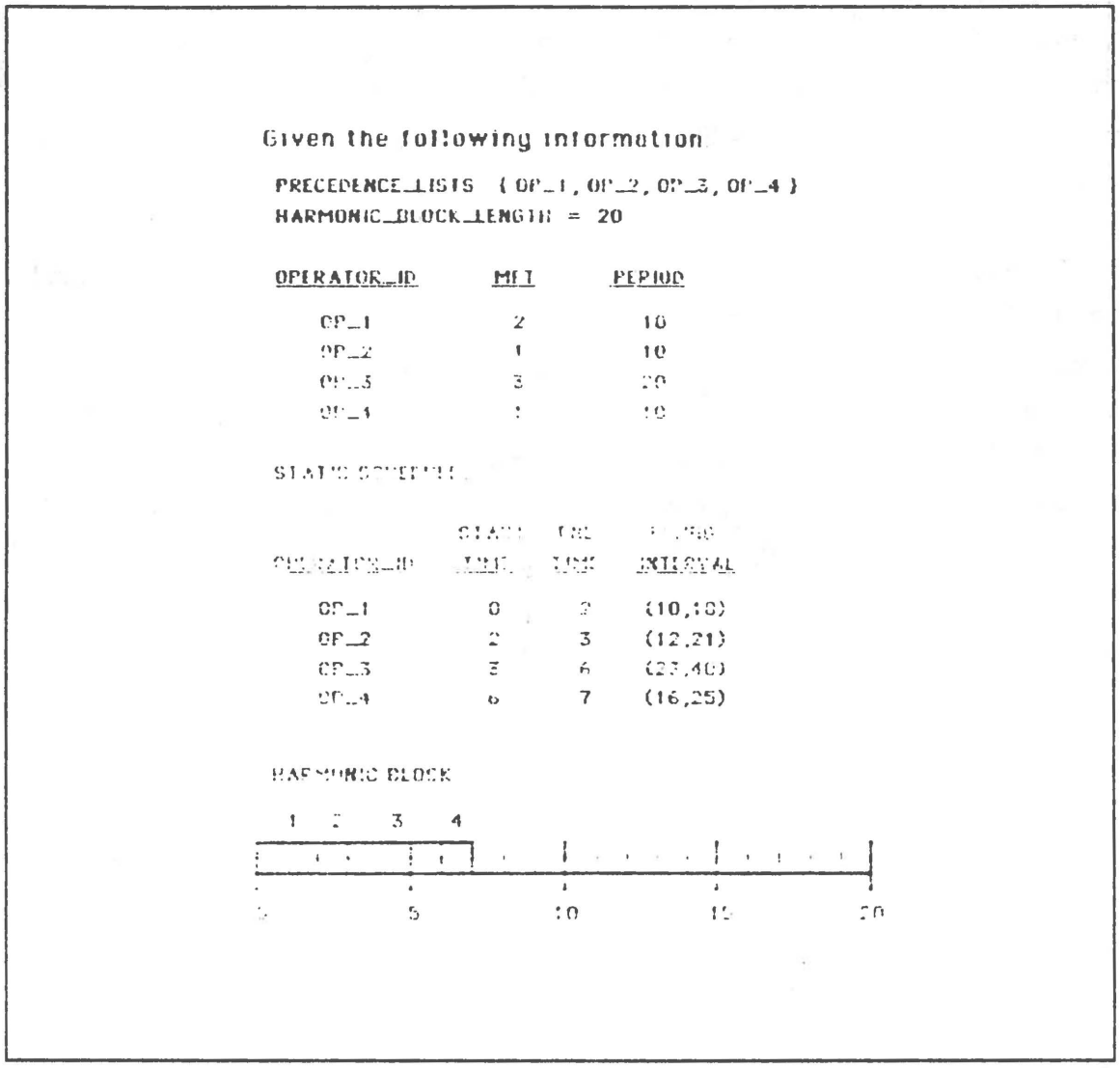


Figure 9. Static Schedule after First Process

ment of hard real-time or embedded systems. The following sections briefly describe each of these in terms of designing and implementing the Static Scheduler.

5.1. Kodiyak Translator Generator

Utilization of CAPS during rapid prototyping of hard real-time systems produces a PSDL prototype source program of the envisioned software system. The ESS's Static Scheduler must identify and extract the critical operators and their associated timing constraints from this PSDL source program before creating the static schedule. The

STATIC SCHEDULE:

OPERATOR_ID	START TIME	END TIME	FIRING INTERVAL
OP_1	0	2	(10,18)
OP_2	2	3	(12,21)
OP_3	3	6	(23,40)
OP_4	6	7	(16,25)
OP_1	10	12	(20,28)
OP_2	12	13	(22,31)
OP_4	16	17	(26,35)
OP_1	20	22	(30,38)
OP_2	22	23	(32,41)
OP_3	23	26	(43,60)
OP_4	26	27	(36,45)
OP_1	30	32	(40,48)
OP_2	32	33	(42,51)
OP_4	36	37	(46,55)

HARMONIC BLOCK:

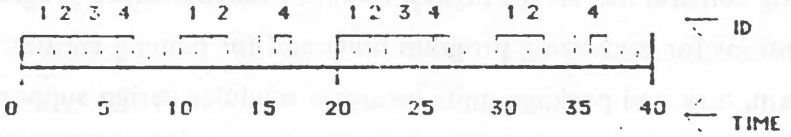


Figure 10. Static Schedule for 2 Harmonic Blocks

Kodiyak automatic translator generator is the tool which provides the Static Scheduler with the capability to process the PSDL source program. This section begins with a general description of the Kodiyak tool and concludes with its specific application for the Static Scheduler.

Kodiyak is an attribute grammar (AG) based tool which automatically generates a translator. The AG approach provides a way for the designer to assign meanings to each input string in a context-free manner. Thus, the AG-based source code contains application specific grammar and attribute equations written in Kodiyak. The compiled

output is a translator in C language. The translator parses each input string and places it's translation in a derivation tree whose structure is based on the specified equations. The structure of the tree, therefore, provides the meaning for each string. Reference 14 and 10 contain detailed information on the Kodiyak translator generator.

5.2. Ada Programming Language

With the increased demand for complex real-time systems, the DOD recognized that the design and management of these systems required a new development approach. DOD initiated development of a programming language incorporating new software engineering principles including capabilities designed specifically for complex systems. As a result, implementation of CAPS in Ada will provide a computer-aided rapid prototyping tool compatible with and directly related to the development of future complex software systems. [Ref. 1: pp. 3-4]

Early Ada designers recognized that complex software systems required parallel processing, real-time control, exception handling, and unique input/output (I/O) control [Ref. 1: p. 16]. The designers identified a programming style and its associated language constructs that are fundamental to the development of complex systems containing critical timing constraints. At the highest level, the recommended programming style includes conventions for organizing program units and for naming entities. Proper use of Ada's program, task and package units insures a modular design supported by separate compilation of individual units and by data abstractions. The preferred Ada naming conventions for all user-defined entities create program code that is easily understood and self-documenting. The Ada programming language includes a pre-defined language environment that contains extensive data types, calendar/timing functions, system exceptions, and several levels of I/O operations. However, Ada programming principles also stress user-defined data types, exceptions, and I/O operations to insure precise implementations and consistent, self-documenting code.

At a lower level, Ada constructs for task program units containing rendezvous operations are integral concepts for prototyping real-time systems. The rendezvous constructs ENTRY and ACCEPT provide explicit synchronization between two parallel tasks, supporting both concurrency of operation and precedence relationships between time critical operators.

6. Conclusion

The goals of this pioneering effort were to demonstrate the feasibility of implementing a Static Scheduler for CAPS and to provide guidelines for implementation. This paper outlined the tools and algorithms required, at a minimum, to implement the Static Scheduler and to integrate it within the Execution Support System. This empirical study accomplished these goals while identifying specific areas of concern for future research.

The Kodiyak AG translator generator is an effective and efficient tool for processing a PSDL prototype source program. An AG processor designed precisely for the Static Scheduler is fundamental to successful scheduling of critical operators. Misrepresentation of or failure to identify operators and their timing constraints negates the benefits of the best designed scheduling algorithms. Lack of detailed error messages and basic manuals causes an extensive learning period using trial and end or verbal "pass down".

The Ada programming language provides the constructs and enforces a modularized, self-documenting design, which enhance the feasibility of implementing the Static Scheduler. User-defined file and data types allow precise definition of critical operators' timing constraints. Rendezvous operations using ENTRY and ACCEPT statements provide a means to establish and enforce execution precedence among critical operators. Rendezvous operations provide the backbone of the runtime static schedule created by the Static Scheduler. Formal demonstration of the Static Scheduler will determine whether Ada constructs are sufficient and effective in meeting the critical timing constraints of hard real-time or embedded systems.

Concurrent research projects to conceptualize components of the CAPS Execution Support System are complete. These individual efforts empirically demonstrate that the initial goal of providing an automated execution environment for software design or specification prototypes is feasible. The CAPS will provide software designers with an automated tool allowing validation of prototypes for hard real-time or embedded systems before extensive time and money are invested in production software. Additional research projects are currently underway to conceptualize, implement and integrate the various components or subsystems of CAPS. Time and effort expended today toward formal demonstration of CAPS, together with increased usage of the Ada programming language, promise a future rapid prototyping environment which meets the demanding needs of the DOD and DON software procurement and development process.

LIST OF REFERENCES

1. Booch, G. *Software Engineering with Ada*, 2d ed., Benjamin/Cummings Publishing Co. Inc., Menlo Park, CA, 1986.

2. Luqi and Ketabchi, M. *A Computer Aided Prototyping System*. Technical Report NPS52-87-011, Naval Postgraduate School, 1987 and in *IEEE Software*, March 1988, pp. 66-72.

3. Luqi. *Execution of Real-Time Prototypes*. Technical Report NPS52-87-012, Naval Postgraduate School, 1987 and in *ACM First International Workshop on Computer-Aided Software Engineering*. Cambridge, Massachusetts, May 1987, Vol. 2, pp. 870-884.

4. Luqi. *Research Aspects of Rapid Prototyping*. Technical Report NPS52-87-006, Naval Postgraduate School, 1987.

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

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

7. Luqi. *Normalized Specifications for Identifying Reusable Software*. Technical Report NPS52-87-007, Naval Postgraduate School, 1987 and in *Proceedings of the ACM-IEEE 1987 Fall Joint Computer Conference* in Dallas, TX, October 1987, pp. 46-49.

8. Janson, D. *A Static Scheduler for the Computer Aided Prototyping System: An Implementation Guide*. M.S. thesis, Naval Postgraduate School, March 1988.

9. Eaton, S. *A Dynamic Scheduler for the Computer Aided Prototyping System (CAPS)*. M.S. thesis, Naval Postgraduate School, March 1988.

10. Moffitt, C. *A Language Translator for a Computer Aided Rapid Prototyping System.* M.S. thesis, Naval Postgraduate School, March 1988.

11. O'Hern, J. *A Conceptualized Design of a Static Scheduler for Hard Real-Time Systems.* M.S. thesis, Naval Postgraduate School, March 1988.

12. Mok, A. "The Design of Real-Time Programming Systems Based on Process Models." in the IEEE *Proceedings of the Real-Time Systems Symposium* in Austin, TX, December 4-6, 1984, pp. 5-17.

13. Mok, A. "The Decomposition of Real-Time System Requirements into Process Models". in the IEEE *Proceedings of the Real-Time Systems Symposium* in Austin, Texas, December 4-6, 1984, pp. 125-134.

14. Herndon, R. *The Incompleat AG User's Guide and Reference Manual.* Technical Report 85-37, University of Minnesota, October 1985.

Initial Distribution List

Defense Technical Information Center Cameron Station Alexandria, VA 22314	2
Dudley Knox Library Code 0142 Naval Postgraduate School Monterey, CA 93943	2
Center for Naval Analysis 4401 Ford Avenue Alexandria, VA 22302-0268	1
Director of Research Administration Code 012 Naval Postgraduate School Monterey, CA 93943	1
Chairman, Code 52 Computer Science Department Naval Postgraduate School Monterey, CA 93943-5100	1
LuQi Code 52Lq Computer Science Department Naval Postgraduate School Monterey, CA 93943-5100	150