2

AD-A207 368

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

RUN-TIME SUPPORT FOR RAPID PROTOTYPING

by

MaryLou Barrett Wood

December 1988

Thesis Advisor: Luqi

Approved for public release; distribution is unlimited.

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
2b Declassification/Downgrading Schedule	Approved for public release; distribution is unlimited.
4 Performing Organization Report Number(s)	5 Monitoring Organization Report Number(s)
6a Name of Performing Organization Naval Postgraduate School; 6b Office Symbol *(If Applicable)* 37	7a Name of Monitoring Organization Naval Postgraduate School
6c Address *(city, state, and ZIP code)* Monterey, CA 93943-5000	7b Address *(city, state, and ZIP code)* Monterey, CA 93943-5000
8a Name of Funding/Sponsoring Organization; 8b Office Symbol *(If Applicable)*	9 Procurement Instrument Identification Number
8c Address *(city, state, and ZIP code)*	10 Source of Funding Numbers

Program Element Number	Project No	Task No	Work Unit Accession No

11 Title *(Include Security Classification)* Run-Time Support for Rapid Prototyping

12 Personal Author(s) MaryLou Barrett Wood

13a Type of Report	13b Time Covered	14 Date of Report *(year, month, day)*	15 Page Count
Master's Thesis	From To	December 1988	82

16 Supplementary Notation The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

17 Cosati Codes		
Field	Group	Subgroup

18 Subject Terms *(continue on reverse if necessary and identify by block number)*
CAPS, prototyping, debugging, Dynamic Scheduler · military — [illegible]

19 Abstract *(continue on reverse if necessary and identify by block number)*

The Computer Aided Prototyping System (CAPS) uses rapid prototyping to quickly build an executable model of the proposed system. This thesis discusses two aspects of the run-time support system for CAPS. In particular, it addresses the implementation of the error reporting functions in the CAPS debugging system and of the Dynamic Scheduler. [illegible] Static schedule; computer programs. -

20 Distribution/Availability of Abstract [✓ unclassified/unlimited] ☐ same as report ☐ 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

DD FORM 1473, 84 MAR 83 APR edition may be used until exhausted security classification of this page

All other editions are obsolete Unclassified

Run-Time Support For Rapid Prototyping

by

MaryLou Barrett Wood
Lieutenant, United States Navy
B.S., Lock Haven State College, 1975

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
December 1988

Author: [illegible]
MaryLou Barrett Wood

Approved by: [illegible]
Luqi, Thesis Advisor

[illegible]
John Yurchak, Second Reader

[illegible]
Robert McGhee, Chairman,
Department of Computer Science

[illegible]
Kneale T. Marshall, Dean of
Information and Policy Sciences

ABSTRACT

The Computer Aided Prototyping System (CAPS) uses rapid prototyping to quickly build an executable model of the proposed system. This thesis discusses two aspects of the run-time support system for CAPS. In particular, it addresses the implementation of the error reporting functions in the CAPS debugging system and of the Dynamic Scheduler.

TABLE OF CONTENTS

LIST OF FIGURES

THESIS DISCLAIMER

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

I. INTRODUCTION

A. BACKGROUND

In 1980, the Department of Defense incurred software costs of $4.10 billion and hardware costs of $1.28 billion [Ref. 1:p. 425]. In 1990, the Department of Defense (DOD) expects to incur software costs of $37.99 billion and hardware costs of $5.89 billion [Ref. 1:p. 425]. Over a decade, software costs for DOD will have increased approximately 900 percent, while hardware costs will have increased only by approximately 460 percent. Using this calculation, software costs for DOD are rising twice as fast as hardware costs. Furthermore, these software costs are mainly concerned with hard real-time embedded computer systems. As the requirements for embedded computers in DOD increase, the cost of software will increase even more dramatically.

Software costs will continue to rise for two reasons. First, as advances occur in hardware technology, machines will become less expensive. Second, more and more complex applications are becoming possible candidates for automation. But as the complexity of applications increases, the complexity of the associated software development will become exponentially larger. Similarly, as the complexity of software development increases, the cost for that development will also dramatically increase.

Software today suffers from not only high development costs but also poor quality. Symptoms of this poor quality are the lack of:

1. Responsiveness. Computer-based systems often do not meet user needs.
2. Reliability. Software often fails.
3. Modifiability. Software maintenance is complex, costly, and error prone.
4. Timeliness. Software is often late and frequently delivered with less-than-promised capability.
5. Transportability. Software from one system is seldom used in another, even when similar functions are required.
6. Efficiency. Software development efforts do not make optimal use of the resources involved. [Ref. 1:p. 8]

A means must be found to reduce the prohibitive cost of software and simultaneously to increase the quality of software. Established techniques for developing software have proven incapable of accomplishing these improvements. Prototyping might prove to be one such means.

Prototyping is an engineering-inspired design approach in which an analyst quickly builds an executable system that may not be complete. This prototype can be demonstrated to users and then easily modified. Although prototypes can be more rapidly developed by using nonprocedural languages, generally, the prototype must be rewritten in a procedural language to improve efficiency or add features. [Ref. 2:p. 12]

There are several facts which indicate the advantages of prototyping. Users are not always aware of their requirements for a system.

If a user does know a requirement, he might not know the best method to realize that requirement. Development of the prototype can be started after only general objectives have been identified. After the prototype is developed, it is shown to the user. The user can thus observe the execution of the prototype and make recommendations for modifications. The prototype is modified to reflect the recommendations and then again shown to the user. This process of user observation and of modification is repeated until the system meets the user's requirements. But since the user is now more actively involved and is involved earlier in the design process, the development of the prototype occurs more rapidly and efficiently.

When prototyping is combined with automated software tools in order to develop the system even more quickly, that process is known as *rapid prototyping*. One such system is the Computer-Aided Prototyping System (CAPS). It combines rapid prototyping with automatic program generation.

Some of the software tools included in CAPS are an execution support system, a rewrite system, a syntax-directed editor with graphics capabilities, a software base, a design database, and a design management system [Ref. 3:p. 66].

By using both rapid prototyping and automatic program generation, CAPS will be able to develop a system that is responsive, reliable, modifiable, timely, transportable, and efficient. CAPS is discussed in greater detail in Chapter II.

B. THE SCOPE OF THE THESIS

The scope of this thesis is the debugging system and the Dynamic Scheduler in CAPS. This thesis implements the debugging systems for the Static Scheduler and the Dynamic Scheduler. This thesis also discusses an implementation of the Dynamic Scheduler.

Chapter II of this thesis discusses CAPS and the work accomplished on it to date as well as current work on several different debugging systems.

Chapter III discusses the Dynamic Scheduler with particular attention to how it coordinates execution of the critical and noncritical operators. Chapter IV discusses the design of the debugging systems. Chapter V addresses the implementation of the debugging systems and of the Dynamic Scheduler. Chapter VI contains the conclusions and recommendations for future work.

II. SURVEY OF PREVIOUS WORK

A. COMPUTER-AIDED PROTOTYPING SYSTEM

CAPS is an environment designed to automate prototyping of large software systems with real-time constraints. The environment consists of a group of tools whose collective purpose is to provide a means to write the specifications of a software system, to implement those specifications, and to execute the resulting prototype. Most prototyping systems perform these functions. CAPS is different in that it combines rapid prototyping with a variant of automatic program generation. This combination makes the process of developing the prototype more timely, efficient, and reliable.

CAPS includes a database of reusable software components. When a specification for the system has been *identified*, the database is searched for a matching specification. If one is found, the component for that matching specification is retrieved. If no match is found, the specification must be decomposed. The database is again searched for matching elements. If the specification cannot be decomposed any further, its implementation must be coded manually. The significance of being able to retrieve components is that the prototype is developed more rapidly. Time need not be spent repeatedly implementing the same specifications in different systems. Figure 2-1 [Ref. 3:p. 67] illustrates this process of decomposition and retrieval.

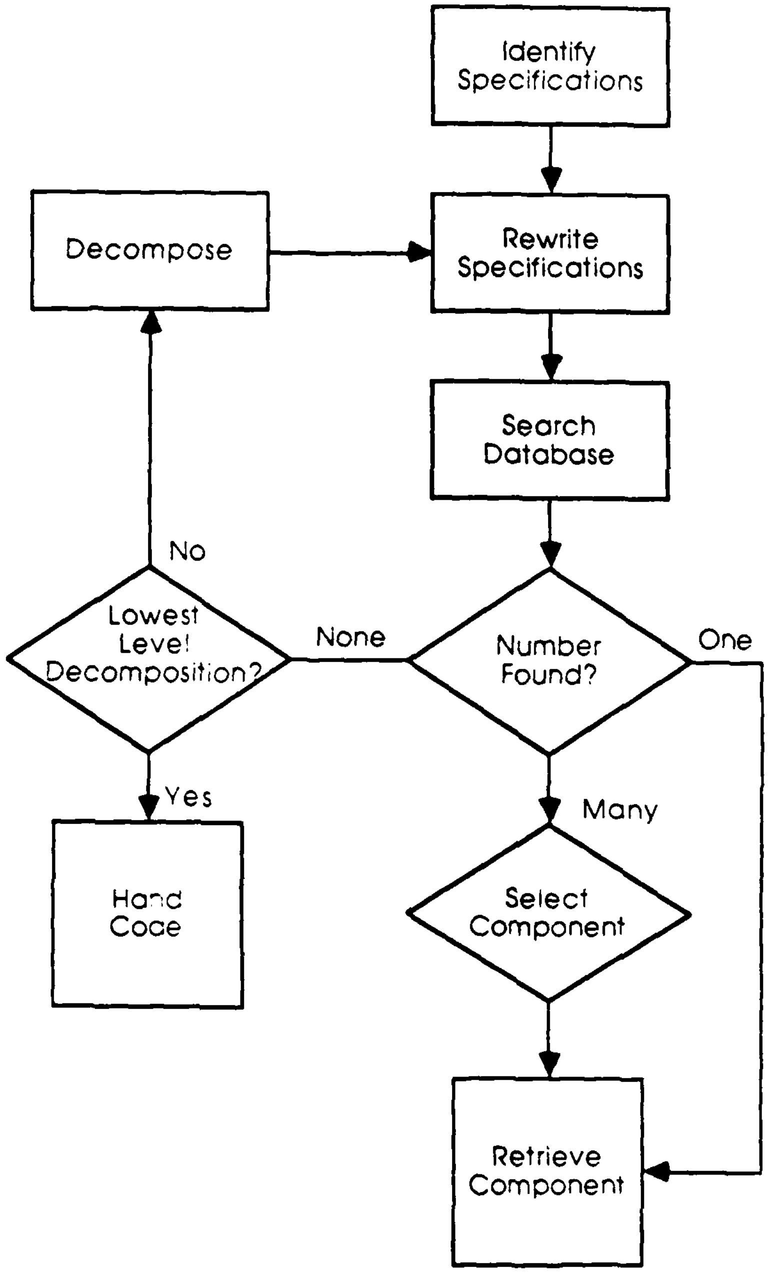

Figure 2-1. **The Prototype-Building Process**

The major tools in the environment are: a User Interface, a specification language called the Prototyping System Description Language (PSDL), an Execution Support System, a Design Management Base, a

Software Base, and a Design Data Base. The relationships among these tools are shown in Figure 2-2 [Ref. 4]. A general description of each of these tools follows. For more information regarding a specific tool, or CAPS in general, please refer to Reference 5.

The User Interface serves two functions. First, it is the means by which the user identifies the specifications for the system. The User Interface includes a syntax-directed editor and a graphics editor to enable the user to enter these specifications. Secondly, it serves as the control unit for the development. When the user requests an action, it is the User Interface which initiates the action. For example, when the user requests the execution of the prototype, the User Interface calls the Execution Support System, which is the process to accomplish that request. Reference 6 contains a description and implementation for the User Interface.

PSDL is the language used to develop specifications for the prototype and was designed specifically for CAPS. PSDL supports the development of prototypes for large systems by providing a computational model that reflects the designer's view of real-time systems. The language supports operators, data streams, and abstractions . Reference 5 contains a detailed description of PSDL. [Ref. 5:p. 11]

PSDL also supports hierarchical decomposition of operators through data-flow diagrams. These diagrams show the connectivity of the operators. Operators are connected through data streams which contain the data values required by the operators. The User Interface

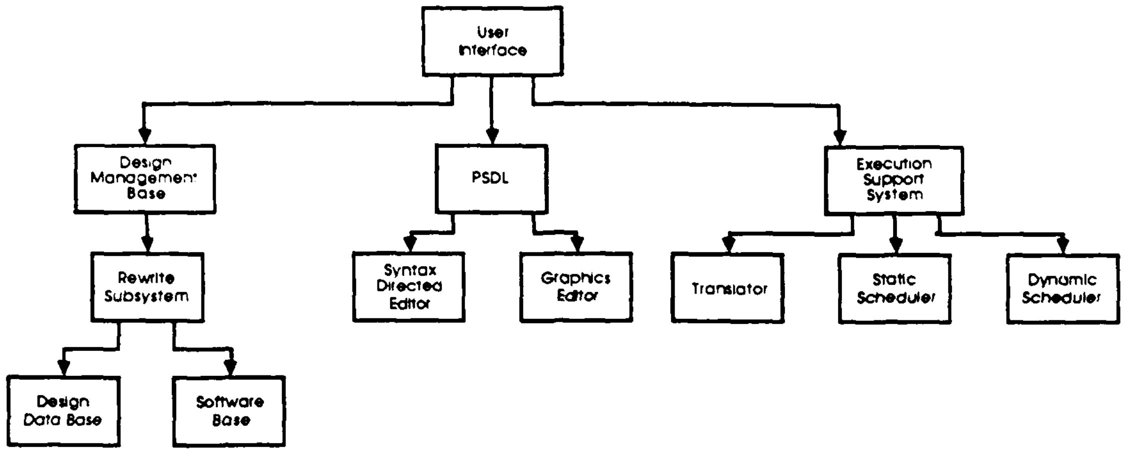

Figure 2-2. **CAPS**

contains a graphics editor to draw the data flow diagrams. Reference 7 discusses and implements the graphics editor.

The Software Base contains the reusable software components. On the other hand, the Design Data Base contains information on the design of a prototype and keeps copies of the different versions of a prototype. Reference 8 discusses the design of the Software Base.

The Design Management Base is the tool which organizes and retrieves components from the Software Base and which manages the different versions of prototypes in the Design Data Base. Before the management base stores or retrieves components or prototypes, the specifications for the components or prototypes must be normalized by the Rewrite Subsystem. The normalization of specifications is accomplished by transforming specific words in the specification to standard ones. For example, if the word *output* appears in the specification, it would be converted to the word *write*. Normalization is necessary because it reduces the time to search the databases. For

CAPS to be practical, the time to retrieve a component must be significantly less than it would be to code that same component.

The Execution Support System contains three processes: the Translator, the Static Scheduler, and the Dynamic Scheduler. The Translator transforms the PSDL source code into executable code for both the time-critical and the non-time-critical operators. The Static Scheduler creates a schedule by which the time-critical operators will be executed. The Dynamic Scheduler then coordinates the execution of the non-time-critical and the time-critical operators. References 9 and 10 discuss the design and implementation of the Translator. References 11, 12, and 13 discuss the design and implementation of the Static Scheduler.

As envisioned in Reference 11, the Dynamic Scheduler is invoked when the user asks to exercise the prototype. The Dynamic Scheduler then invokes the Translator and the Static Scheduler. The Translator produces executable Ada code, and the Static Scheduler produces a static schedule which is also executable. These two outputs must be compiled and linked together. The resulting executable code is an input to the Dynamic Scheduler. Repeated from Reference 11, Figure 2-3 illustrates the procedure described above.

B. DEBUGGING SYSTEMS

Debugging is one of the most time-consuming activities associated with programming. When an error is encountered during testing, it must be identified, located, and corrected. After that, the program must be recompiled and tested again. If only one error is detected at a time, the debugging process can rapidly mire in this cycle of identification and correction. For this reason, much research is being done to improve the capability of a debugging system not only to identify errors but also to locate and correct them.

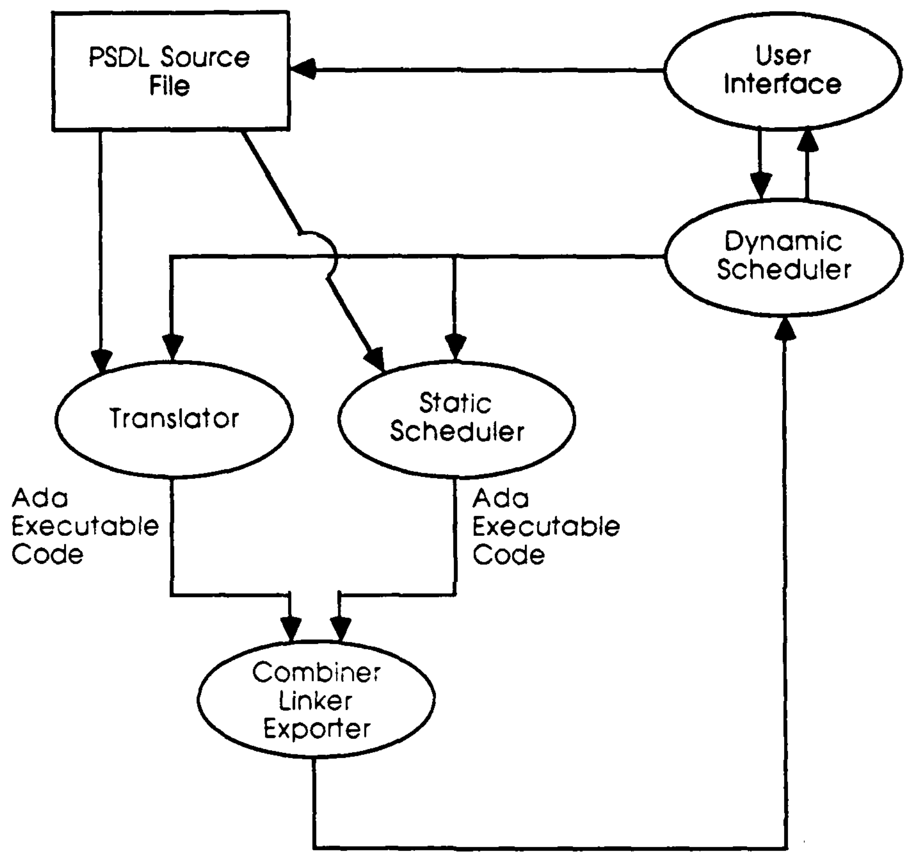

Figure 2-3. **Execution Support System**

Isoda, Shimomura, and Ono [Ref. 14] have developed a debugger for Ada called the Visual and Interactive Programming Support (VIPS). VIPS uses graphics to show the static and dynamic behavior of a program during execution. VIPS presents different views of the execution in windows. These windows are: data, program text, block structure, acceleration, figure definition, interaction, and editor. The data window is used to show which procedures or functions have been invoked and to show the calling relationship among these units. The program text window contains the source code and, during execution of the program, highlights the active line. The block structure window illustrates the nesting relationship of subprograms and internal packages. The acceleration window is used to display and to change the execution speed of the program. The figure definition window displays a list of variables that may or may not be stipulated by the user. The interaction window is used when the user must respond to prompts from the program. Lastly, the editor window is used to edit the source program. The quantity and quality of information that can be displayed in these windows greatly increases the ability of a programmer to locate and correct errors.

Seviora [Ref. 15] states that debugging involves two main phases: identification and repair. Knowledge-based debugging systems can use three approaches to debugging a program: the program-analysis, I/O-based, and internal-trace-based. The program-analysis approach compares the content of the program to its specification to determine whether they are consistent. In order to do this, the system performs a detailed analysis of the program. This detailed analysis takes a great deal of time and, hence, these knowledge-based debugging programs are only practical for small programs.

The I/O-based approach examines only those portions of code in which a bug might occur. To determine what the problem is, systems implementing this approach compare the actual output with what was expected. The system attempts to localize the bug in the section being examined. I/O-based systems are not successful in locating the error if the code has several errors in it, especially if the errors interact.

The internal-trace-based system compares the program code to the output. Certain characteristics of the program are tagged and traced through execution. This approach serves only to localize the error within the program code.

Knudsen [Ref. 16] uses the sequel concept to declare an exception and its handler together. He defines the sequel as an abstraction of the *goto* statement. A sequel defines the name and handler of an exception and its termination level. There are three types of sequels: the prefixed sequel, which permits the specification of smooth termination; the virtual sequel, which augments a handler in inner blocks; and the default sequel, which makes default exception handling possible and, hence, increases the possibility of smooth termination.

C. A PROTOTYPE IN SMALLTALK

Diederich and Milton [Ref. 17] state that Smalltalk is more than a programming language. It is, in a sense, a tool that encourages prototyping. Smalltalk encourages experimentation with prototyping because the designer is not caught in the midst of detail and because the designer can make vast changes to a system with a good chance of recovery. In Smalltalk, messages form modules which are simpler and easier to understand. Interfaces between these modules are not necessary because objects are passed as arguments to messages. Smalltalk has numerous predefined objects and messages. These predefinitions encourage prototyping because a database already exists that need only be adapted to the user's needs. Other features of Smalltalk that resemble prototyping include the ease of implementing alternatives; any changes that are made are equivalent to changing specifications vice changing code.

III. THE DESIGN OF THE DYNAMIC SCHEDULER

A. MODIFICATION TO THE ARCHITECTURE

There is a problem with the conceptualization of the Dynamic Scheduler as it was presented in Chapter II. Because the Translator uses the Kodiyak generator to produce the executable code for the operators, the Dynamic Scheduler, itself an executable Ada program, cannot invoke the Kodiyak Generator. Also, the Static Schedule, an output of the Static Scheduler, is an input to the Dynamic Scheduler. This input represents the schedule by which the critical operators are to be executed. However, this schedule cannot be executed by the Dynamic Scheduler until it is compiled and linked to the output of the Translator. Herein lies another problem. Once an Ada program begins execution, it cannot be suspended to compile an output from one of its units and then resumed to begin execution of that compiled unit. For these reasons, the Execution Support System and the Dynamic Scheduler, in particular, have been modified as reflected in Figure 3-1.

The Execution Support System is revised as follows:

1. The Translator is distinct from the Dynamic Scheduler and therefore is not invoked by it. It is a separate process which can execute in parallel with the Static Scheduler.
2. The Static Scheduler is now part of a system containing the Static Scheduler, its debugging system, and the process by which the non-time-critical operators are transformed into executable code.
3. The Dynamic Scheduler which coordinates the execution of the critical and non-critical operators and its debugging system form another distinct part of the Execution Support System.

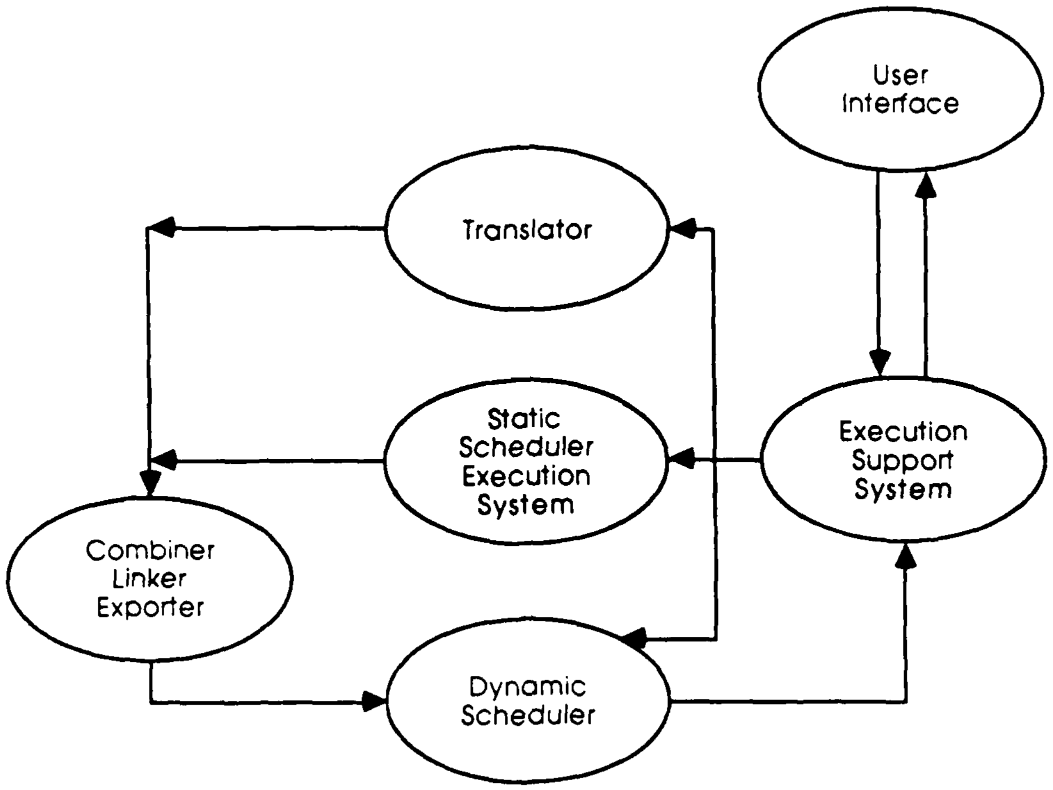

Figure 3-1. **The Execution Support System, as Modified**

It is important that each of the above be distinct from the others. The Translator and the Static Scheduler can then be executed in parallel, either in a single processor or in a multi-processor environment. These two functions produce three outputs: the executable code for the operators, the Static Schedule for the time-critical operators, and the listing of procedure calls for the non-time-critical operators. These outputs must be compiled and linked together before the Dynamic Scheduler can be invoked. The User Interface in CAPS is responsible for invoking the Static Scheduler, the Translator, and the Dynamic Scheduler. The User Interface will also ensure that the outputs above are compiled and linked before invoking the Dynamic Scheduler.

B. DESIGN OF THE DYNAMIC SCHEDULER

The Dynamic Scheduler consists of three processes: the Static Schedule, the non-time-critical operators, and a debugging system. The relationships among these three are shown in Figure 3-2.

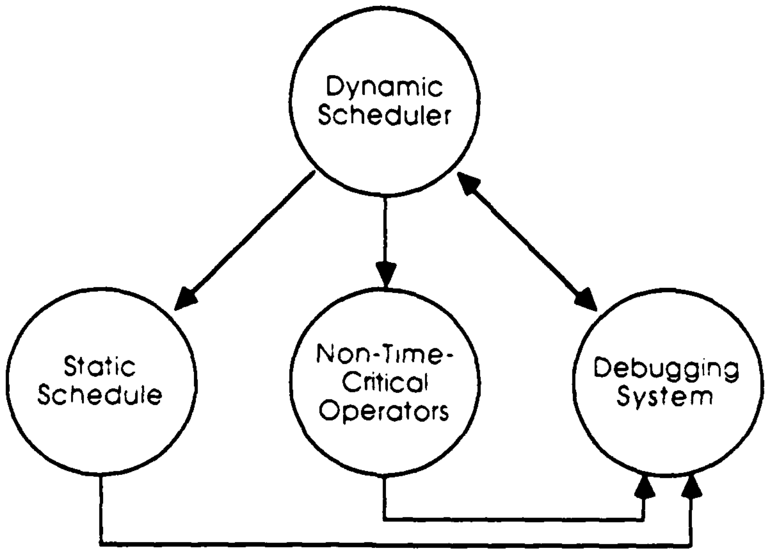

Figure 3-2. **The Dynamic Scheduler**

The Static Schedule is assumed to have the format shown in Figure 3-3. Figure 3-3 shows the minimum amount of information which must be specified for each operator in the Static Schedule. The variable "Exception_Operator" is necessary to inform the debugging system which operator experienced a run-time error. The third line of Figure 3-3 is a procedure call to the code produced by the Translator.

"TL" is the package from the Translator which includes the executable code for the operator. This is the line that actually executes the operator. The *if-elsif* statement is necessary to coordinate the different actions which occur based on the time. These actions are explained below. The Ada constructs in Figure 3-3 will be explained in Chapter V.

```
Exception_Operator := Name_Of_Next_Operator;
Current_Time := CALENDAR.CLOCK;
TL.Name_Of_Next_Operator;
if Current_Time > Next_Start_Time then
     DS_Debug.Runtime_MET_Failure(Exception_Operator);
elsif Current_Time < Next_Start_Time then
     delay Next_Start_Time - Current_Time;
end if;

—The same lines are necessary for each operator in the Static
--Schedule.
```

Figure 3-3. **The Static Schedule**

The Dynamic Scheduler serves to coordinate execution of the time-critical and non-time-critical operators. The time-critical operators are executed through the Static Schedule. Initially, the Static Schedule is invoked. It must be the process which executes first because its first operator is assumed to begin at time zero.

After the first operator finishes execution, the current time is compared to the time the next critical operator must start execution. Depending on the results of that comparison, one of three actions might occur:

1. If the current time is less than the start time for the next critical operator, the static schedule must suspend execution until the next start time, and then the non-time-critical operators begin to execute. At the time the next critical operator must begin execution, the non-time-critical operators are suspended and the Static Schedule is resumed. When the processes resume execution, they do so at the point where they were suspended.

2. If the current time is equal to the start time for the next critical operator, then the Static Schedule continues execution.

3. If the current time is greater than the start time for the next critical operator, then the Static Schedule notifies the debugging system in the Dynamic Scheduler that a run-time execution error has occurred. The user is queried as to whether to continue execution of the prototype. Regardless of his decision, information about the error is written to a file. In this way, a historical record is maintained. After the prototype has finished execution, either normally or abnormally, the user is able to update the execution times for the pertinent operators.

An example may clarify this coordination. Figure 3-4 shows a Static Schedule and a listing of non-time-critical operators. For the Static Schedule, the number on the left side of each solid horizontal line represents the time that the *succeeding operator must start* execution. For example, operator A must start at time 0, B must start at time 15, C at time 22, and so on. On the other hand, the non-time-critical operators are executed sequentially as time allows.

When the Dynamic Scheduler is invoked, operator A will begin to execute, because it must start at time 0. But suppose the operator completes execution at time 12 (represented by the dashed line in Figure 3-4). Operator B does not execute until time 15. Thus, there is an interval of three time units in which the Dynamic Scheduler may execute the non-time-critical operators. Before the first non-time-critical operator may begin execution, the Dynamic Scheduler must suspend the execution of the Static Schedule and save its state of execution. Operator X is then executed. The non-time-critical operators continue to execute until time 15. At that moment, the Dynamic Scheduler suspends execution of the non-time-critical operators, saves their state, and then restores the state of execution for the Static Schedule. In this way, the Static Schedule is resumed at the point where operator A ended and operator B is to begin execution.

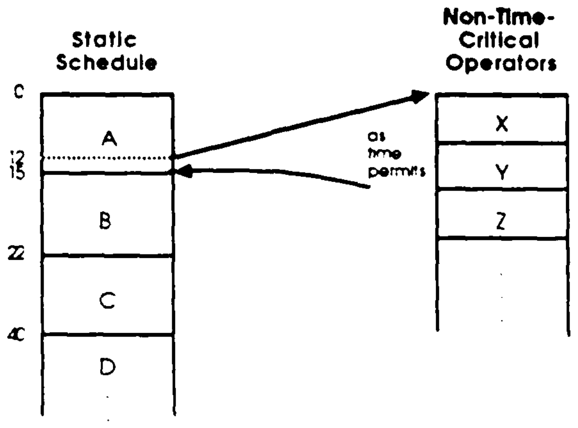

Figure 3-4. **Coordination of Operators**

Now suppose operator B completes execution at time 22. Since this is the time operator C must begin execution, the Static Schedule continues to execute. However, operator C completes execution at time 45—after the required start time (40) for operator D. In this case, the Dynamic Scheduler must suspend execution of the Static Schedule, save the state of the execution, and then inquire of the user whether to continue. If the user wants to continue, the Dynamic Scheduler must adjust the time backwards to the start time required for operator D, restore the state of execution for the Static Schedule, and then resume execution of the Static Schedule.

The process described above continues until all operators have executed.

IV. DESIGN OF THE DEBUGGING SYSTEM

A. PURPOSE OF A DEBUGGING SYSTEM

The purposes of debugging systems are to identify errors and, if possible, correct them. The latter is the more difficult purpose to accomplish. If an error is syntactic in nature, it is fairly easy to correct. For example, if a variable is undeclared, it is easy to correct that error by simply declaring the variable in the proper manner. If, on the other hand, the error is semantic in nature, the error is harder to correct. For example, if an end statement is missing, only the user knows the correct location for it. It is possible in CAPS to have both syntactic and semantic errors. The debugging system for CAPS must then be capable of handling both types of errors.

B. ERROR REPORTING IN THE DEBUGGING SYSTEM

Because the Static Scheduler and the Dynamic Scheduler are distinct from each other, each scheduler must have a debugging system to process those errors encountered by each scheduler. The debugging system in the Static Scheduler will process those errors encountered while creating a Static Schedule. The debugging system in the Dynamic Scheduler will process those errors that occur when the operators execute.

Both debugging systems will report errors in a similar manner. When an error has been encountered, the debugging system should notify the user that an error occurred and the nature of that error. If it is possible to correct or adjust the error, the user should be queried as to whether he wants to terminate or to continue execution. Regardless of the decision, information pertaining to the error should be written to a file. This information should contain sufficient information for the user to understand the error and, possibly, to correct the error.

C. ERRORS IDENTIFIED IN THE TRANSLATOR

When the Translator transforms the PSDL source code into executable code, it identifies three possible errors. Figure 4-1 shows a simple data-flow diagram. The arrow entering the bubble represents a data stream serving as an input source. The bubble represents an operator. The arrow leaving the bubble represents a data stream serving as an output source. When an operator attempts to read the data stream, it expects to find a value there. If no value is present on the data stream, an error exists which must be processed by the debugging system. This error is called Buffer Underflow. Similarly, if the operator is placing a value in the output data stream, it expects the data stream to have room for it. If the data stream is full, an error exists. This error is called Buffer Overflow.

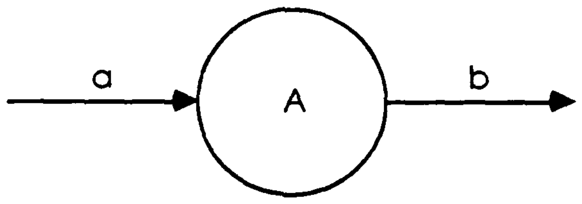

Figure 4-1. **A Simple Dataflow Diagram**

PSDL permits only one exception per data stream. If a data stream has an exception in it and another exception arrives, an error has occurred that must be processed by the debugging system. This type of error is called Exception Error.

The Translator only identifies where these errors may occur. The Buffer_Underflow, Buffer_Overflow, and Exception_Error actually occur only when the operators execute. Therefore, these errors are processed by the debugging system within the Dynamic Scheduler.

D. ERRORS ENCOUNTERED BY THE STATIC SCHEDULER

The Static Scheduler creates a Static Schedule by which the time-critical operators must execute. This schedule ensures that all timing constraints are met. A time-critical operator may have the following timing constraints:

1. A maximum execution time (MET) stating the length of time required by the operator to execute.
2. A maximum response time (MRT) stating how much time passes from the arrival of input values to the placement of the output values into the data streams.
3. A minimum calling period (MCP) stating the time between arrivals of input.

All time-critical operators have a maximum execution time. Only sporadic operators have maximum response times and minimum calling periods. If an operator has a maximum response time, it must also have a minimum calling period. Sporadic operators are executed when new input arrives; periodic operators execute at regular intervals called periods.

During the process of creating a Static Schedule, the Static Scheduler must examine the timing constraints to ensure they are valid for the operators. If the constraints are not valid, an error is reported to the debugging system. Timing constraints are valid if the following relationships occur:

1. The MET for an operator is less than its MRT and MCP. Otherwise, the operator may not complete execution before it must be executed again.
2. The MCP for an operator must be less than its MRT. If it is not, the operator may not produce an output before it must execute again.
3. If an operator has an MET, all operators in its decomposition must have METs.
4. If an operator has an MET, the MET for each operator in its decomposition must be less than or equal to the MET of the operator at the upper level.
5. If an operator has an MET, the sum of the METs of the operators in its decomposition, if applicable, must be less than or equal to the MET of the operator at the upper level.
6. If an operator has a period, MRT, or MCP, it must have an MET.
7. The MET for an operator must be less than its period. Otherwise, the operator may not complete execution before its next execution time occurs.

If any of these relationships is invalid, an error results which must be resolved before the Static Scheduler can continue. These errors are called MET_Not_Less_Than_MRT, MET_Not_Less_Than_MCP, MCP_Not_Less_Than_MRT, MET_Required, MET_GT_Parent, MET_Sum_GT_Parent, Crit_Op_Lacks_MET, and MET_Not_Less_Than_Period.

The debugging system has two options with regard to processing these errors: terminate the Static Scheduler or correct the error. If the error is to be corrected, the user must be queried as to the proper value for the invalid constraint. Changing the value for a constraint on an operator at one level of decomposition may affect constraints for an operator at an upper level of decomposition. Consider Figure 4.2. The METs of the operators in the second level of decomposition are valid; their sum is less than or equal to the MET for operator A. However, the sum of the METs for operators E and F is greater than the MET for operator D. The simple way to correct this error would be to adjust the MET of operator D. By changing the value of the MET for operator D to 45ms, the value of the MET for operator A is now invalid. This rippling effect must be considered when correcting timing constraints. Since the timing constraints for operators at upper levels of decomposition must be reexamined each time a constraint has been altered at a lower level, large amounts of processing time may be spent correcting timing constraints. For this reason, after a specified number of corrections to timing constraints, if the timing constraints are still invalid, the Static Scheduler should notify the debugging system of the situation and processing should be terminated. This error is called Excessive_Constraints_Altered.

The debugging system should maintain a record of the changes that were made to the timing constraints. After the prototype has finished execution, either normally or abnormally, the user is then able to use this historical record to update the PSDL source file.

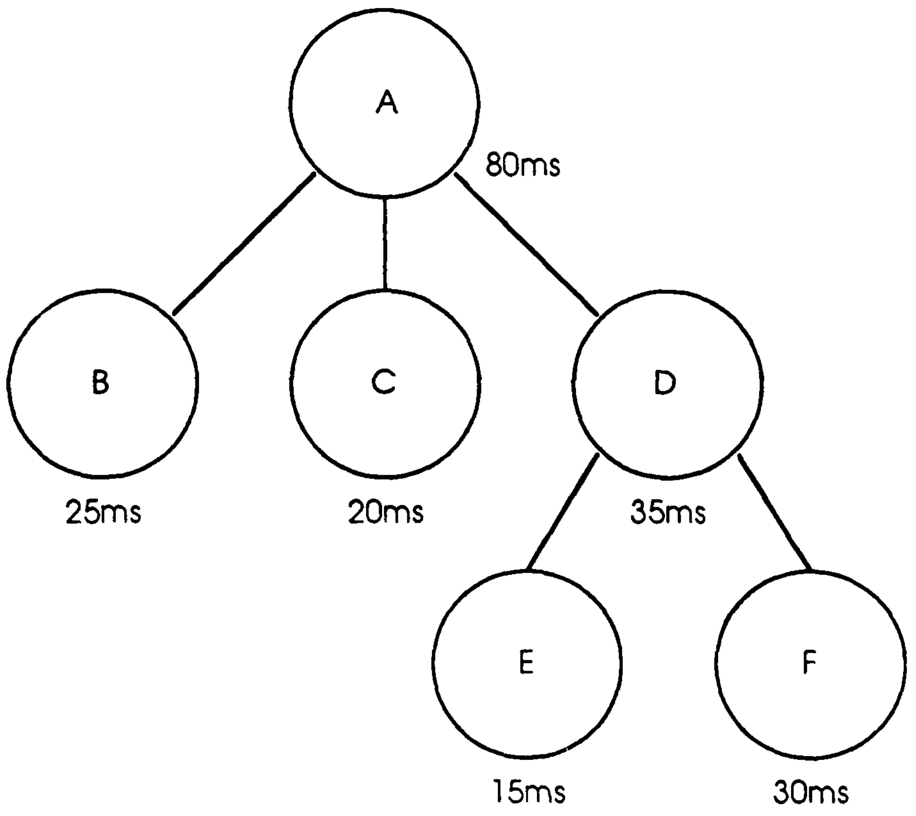

Figure 4-2. **Operator Decomposition**

The Static Schedule may encounter other errors while creating a static schedule. It may not be able to locate the operator to be scheduled first, or it may not be able to locate the successor of an operator. These errors are called No_Initial_Link_Op and No_Matches_Found, respectively.

In order to schedule the operators, the periods of the operators must be exact multiples of some base period. This base period must be determined. An error, called No_Base_Block, results if the base period cannot be determined.

The following errors may occur when the Static Scheduler is calculating the times that the operators must start execution:

1. The MET of an operator is greater than or equal to half of the period for the operator.
2. The total time that the operators need to complete execution is greater than the length of the harmonic block (the set of operators whose periods are multiples of some base period).
3. The ratio of the MET divided by the period for an operator, summed over all the operators, is greater than the number of processors being used.
4. Given the timing constraints, a Static Schedule is not possible.

The first three errors above are called Fail_Half_Period, Bad_Total_Time, and Ratio_Too_Big. The fourth error has been divided into three errors depending on when the determination has been made that a static schedule is not possible. These errors are called Over_Time, Invalid_Schedule, and Schedule_Error.

There is one more error associated with the Static Scheduler. This error is the Runtime_MET_Failure. When the Static Schedule is actually being written by the Static Scheduler, the completion time of the current operator is compared against the start time for the succeeding operator. If the completion time is after the start time, the debugging system is notified of the error. The Runtime_MET_Failure error will occur only during execution of the Static Schedule, and thus this error will be processed by the debugging system in the Dynamic Scheduler.

Figure 4-3 lists all of the errors that can be encountered by the Static Scheduler while creating a Static Schedule. Items 1 through 13 were identified in Reference 11. Reference 13 changed the name of Item 2 from "MET_Equals_Period" (as originally given) to its present name. This change occurred because in a single-processor environment it is permissible for the MET to be equal to the period. The only requirement is that the MET of an operator not be greater than its period. If it is greater, there is no guarantee that the operator will complete execution. Items 14 through 17 were identified in Reference 13.

1. MET_Not_Less_Than_MRT
2. MET_Not_Less_Than_Period
3. No_Initial_Link_Op
4. No_Matches_Found
5. MCP_Not_Less_Than_MRT
6. MET_Not_Less_Than_MCP
7. No_Base_Block
8. Fail_Half_Period
9. Bad_Total_Time
10. Ratio_Too_Big
11. Over_Time
12. Invalid_Schedule
13. Schedule_Error
14. MET_Required
15. MET_GT_Parent
16. MET_Sum_GT_Parent
17. Crit_Op_Lacks_MET
18. Escessive_Constraints_Altered

Figure 4-3. **Errors to be Processed by the Static Scheduler's Debugging Systems**

V. IMPLEMENTATION OF THE DEBUGGING SYSTEMS AND THE DYNAMIC SCHEDULER

A. PROGRAMMING ENVIRONMENT

The programming environment for the implementation is the Unix operating system run on a Sun workstation. The programming language used is Ada. When active, a debugging system will have a dedicated window on the screen to interact with the user. This implementation operates in a single-processor environment.

B. THE STATIC SCHEDULER AND ITS DEBUGGING SYSTEM

The Static Scheduler and its debugging system are implemented as two tasks: the Static_Scheduler and SS_Debug, respectively. These two tasks are dependent on a main program which also includes a procedure, the Create_NTC_Task. Since SS_Debug must cooperate with the Static_Scheduler, they were implemented as tasks because this cooperation among processes is a purpose for tasks. These two tasks cooperate in order to process errors encountered while creating a Static Schedule. The Create_NTC_Task is implemented as a procedure because it only needs to be executed if the Static_ Scheduler was successful in creating a Static Schedule.

This implementation of the debugging system for the Static_ Scheduler does not correct errors. When the Static_Scheduler encounters an error during processing, it notifies the debugging system and then terminates execution. The debugging system will process the error by explaining the error to the user, and then it too will terminate. Appendix A contains the code for the implementation discussed in this section.

1. The Static Scheduler

Reference 14 implemented the task called Static_Scheduler. Since Appendix A includes that task, a brief discussion of the task's activity follows. For more details, refer to Reference 13. Three phases of the Static Scheduler have been implemented. The Read_PSDL phase, implemented on the Kodiyak Generator, produces a text file consisting of the names of the operators, the timing constraints, and the link statements.

The FILE_PROCESSOR package contains two procedures, SEPARATE_DATA and VALIDATE_DATA. The former procedure reads the text file produced by the Kodiyak Generator and separates the time-critical operators. the non-time critical operators, and the link statements. The time-critical operators and their timing constraints are placed in a data structure called an NARY_TREE. The names of the non-critical operators are written to a text file called NON_CRITS. The link statements are placed into a linked list.

The VALIDATE_DATA procedure examines the NARY_TREE to determine whether the timing constraints for the operators are valid. If an invalid constraint is identified, an exception is called. The exception then notifies SS_Debug of the error. An exception is used so that the task terminates gracefully. When an exception is identified, the system looks at the unit in which the exception was identified. If the exception is not found there, the system terminates that unit and goes to the next outer scope to locate the exception. If the main program is reached without locating the exception, the system terminates execution and reports that an exception was not located.

The TOPOLOGICAL_SORTER package contains two procedures, CREATE_LISTS and SORT_REMAINING_OPERATORS. The former procedure locates the operator which must execute first. SORT_REMAINING_OPERATORS identifies those operators which must follow each other. Errors are processed in the same manner as that described for the FILE_PROCESSOR package.

The task called the Static_Scheduler is formed by importing the FILE_PROCESSOR and the TOPOLOGICAL_SORTER packages. The body of the task then calls the procedures in the packages. When the implementation of the Static_Scheduler is completed, the remaining packages and procedure calls will be included in the task Static_Scheduler.

2. The Debugging System

As mentioned previously, the debugging system for the Static_Scheduler is implemented as a task named SS_Debug. The cooperation between SS_Debug and the Static_Scheduler is known in Ada as a *rendezvous*. A rendezvous occurs when one task calls an entry in another task. The entry statements for a task are located in its specification. Each entry statement in the specification has a corresponding accept statement in the body of the task. The accept statement lists the action or actions to be taken for the entry. The accept statement may be either a single statement or a compound statement.

In SS_Debug, the entry statements are the name of the errors listed in Figure 4-3, with the exception of the error Excessive_Constraints_Altered. This error will occur when numerous corrections have been made to the timing constraints. Since SS_Debug does not correct any errors, Excessive_Constraints_Altered is not implemented. The specification for task SS_Debug is shown in Figure 5-1. For those errors triggered by a specific operator, the name of that operator is provided as the value to the parameter Exception_Operator.

```
task SS_Debug is
     entry MET_Not_Less_Than_MRT (Exception_Operator : VSTRING);
     entry MET_Not_Less_Than_Period (Exception_Operator : VSTRING);
     entry No_Initial_Link_Op;
     entry No_Matches_Found (Exception_Operator : VSTRING);
     entry MCP_Not_Less_Than_MRT (Exception_Operator : VSTRING);
     entry MET_Not_Less_Than_MCP (Exception_Operator : VSTRING);
     entry No_Base_Block;
     entry Fail_Half_Period (Exception_Operator : VSTRING);
     entry Bad_Total_Time;
     entry Ratio_Too_Big;
     entry Over_Time;
     entry Invalid_Schedule;
     entry Schedule_Error;
     entry MET_Required (Exception_Operator : VSTRING);
     entry MET_GT_Parent (Exception_Operator : VSTRING);
     entry MET_Sum_GT_Parent (Exception_Operator : VSTRING);
     entry Crit_Op_Lacks_MET (Exception_Operator : VSTRING);
     entry Static_Scheduler_Done;
end SS_Debug;
```

Figure 5-1. **Specification for SS_Debug**

Figure 5-1 also indicates that the specification for SS_Debug includes one other entry statement. This entry is called Static_

Scheduler_Done and is called by the Static_Scheduler task when a schedule of time-critical operators has been successfully created. SS_Debug then knows the Static_Scheduler has terminated and, hence, it too should terminate.

Each of the entries in the specification has a corresponding accept statement in the body of SS_Debug. Because each entry is processed in a similar manner, each accept statement is similar. Because there are two actions to be performed, the accept statements are compound. Additionally, the accept statements are located inside a select loop. When different rendezvous can occur at the same time, a select loop permits SS_Debug to select the accept statement for the current rendezvous. The loop is then re-started to await the next rendezvous. Although in this implementation only one accept statement will be executed, future implementations may choose to correct errors and then the possibility for more rendezvous may occur. Therefore, this method of implementation was chosen with an eye to the future.

The actions to be performed in the accept statement are shown in Figure 5-2. Because each accept statement is similar, only one is shown for illustration purposes. The first action taken is to assign the value of *true* to the variable Error_Exists. The value of this variable is a signal to continue execution (if false) or to terminate execution (if true). The next action is to call a local procedure to print the information pertaining to the error. The explanation of the error and the name of the operator causing the error, if applicable, are written to a file called Information. The accept statement is exited, followed by the exit from the select. A determination is then made to exit the loop based on the value of the variable Error_Exists. If the value of that variable is true, SS_Debug is terminated.

```
loop
     select
               .
               .
               .
          accept No_Initial_Link_Op do
               Error_Exists := true;
               Print_No_Initial_Link_Op_Message (Information);
          end No_Initial_Link_Op;
               .
               .
               .
     end select;
               .
               .
               .
end loop;
```

Figure 5-2. **Accept Statement from the Body of SS_Debug**

The implementation of the accept statement for Static_ Scheduler_Done is only slightly different than that for the other entry statements. The code for Static_Scheduler_Done also has two actions. The first action is to assign the value of *true* to the variable Static_ Scheduler_Finished. Like Error_Exists, this variable is used to determine whether the task should be terminated. The second action in the accept statement is to call the procedure Creat_NTC_Task. This procedure is described in the next section.

In summary, SS_Debug is a task whose main body consists mainly of a select loop which contains an accept statement for each entry. The declarative portion of the task body contains the procedures to print the error messages.

3. The Create_NTC_Task

The Create_NTC_Task is a procedure declared in the main program. It is called from within SS_Debug. Because it is only required to execute if a Static Schedule has been created, the most logical time to have it called is after the Static_Scheduler has completed. It could be called in the main body of the program, but this would require the main program to monitor the Static_Scheduler for termination. This monitoring would be in the form of a "busy wait" which wastes processing time. It is more efficient and more logical to have the Create_NTC_Task called by SS_Debug.

The Create_NTC_Task has the sole function of producing a file called Non_Time_Crits.a. This file will contain the specification and body for an Ada package called Non_Time_Crits_PKG. Inside this package is the specification and body of a task called Non_Time_Critical_Operators. It is important that this package compile and that the included task be capable of executing.

The Create_NTC_Task creates the package by writing the necessary verbiage to the file. The names of the operators which must be written to the output file are located in a file called NON_CRITS. This is the file produced in the Static_Scheduler. The names of the operators in the file must be converted to procedure calls. Because the code for these procedure calls is located in the package produced by the Translator, the names of the operators must be appended to the package name. If the NON_CRITS file is empty, the Create_NTC_Task will not write any procedure calls. The last action taken is a call to DS_Debug to notify the debugging system in the Dynamic Scheduler that all non-time-critical operators have executed.

Figure 5-3 is a sample output from the Create_NTC_Task. Note the verbiage and format of the file. It is similar to the style a programmer would use and should be capable of being compiled. Observe that the specification for the task includes a pragma. This pragma is important when the task is ready to be executed in the Dynamic

```
with TL;   --Translator package
with DS_Debug_PKG;   --Debugging system

package Non_Time_Crits_PKG is
    task Non_Time_Critical_Operators is
        pragma Priority (1);
    end Non_Time_Critical_Operators;
end Non_Time_Crits_PKG;

package body Non_Time_Crits_PKG is
    task body Non_Time_Critical_Operators is
        begin
            TL.First_Operator;
            TL.Second_Operator;
            TL.Third_Operator;
            DS_Debug.Non_Time_Critical_Operators_Done;
        end Non_Time_Critical_Operators;
    end Non_Time_Crits_PKG;
```

Figure 5-3. **Example of a Non_Time_Critical_Schedule**

Scheduler. The section on the Dynamic Scheduler will explain why this pragma is necessary.

In summary, the main program consists of the Static_Scheduler and the SS_Debug tasks and the Create_NTC_Task procedure. When the main program terminates, there will either be two files, one containing a task for the Static Schedule and one containing a task for the non-time-critical operators, or a file containing an error message. If created, the packages will be imported by the Dynamic Scheduler.

C. THE DYNAMIC SCHEDULER AND ITS DEBUGGING SYSTEM

1. Ada Constructs Important to the Dynamic Scheduler

Ada offers facilities which can directly influence the implementation of the Dynamic Scheduler. The more important of these facilities are: the library unit CALENDAR, the sub-program task unit, the pragma PRIORITY, and the reserved word DELAY. Each of these features contributes significantly to the method in which the Dynamic Scheduler can be implemented.

In Ada, there is a library unit called CALENDAR. This unit provides the means to use real time in a program. The unit is a package consisting of the data types DURATION and TIME, and of several procedures which permit manipulation of time. Manipulation of time is critical in applications involving real-time embedded systems. DURATION is a fixed-point type so calculations can be performed without losing accuracy. The procedures included in CALENDAR enable the programmer to add and subtract time, to compare times against each other, to set time, and to split time into its component parts, i.e., month, day, year, and seconds. Again, these operations are critical to embedded systems which monitor time.

Ada also includes tasks which are program units that allow processes to execute in parallel. Tasks also permit cooperation among themselves. This cooperation is particularly important for applications which must communicate with physical systems in real time.

Tasks in Ada have two parts—the specification and the body. The specification is used to describe how a task cooperates with other tasks. The body of a task describes the action to be performed by the task. A task is activated at the end of the parent's declarative section. The body of the main program is considered to be an undeclared task and, as such, is executed first. The dependent tasks are then executed in an unpredictable manner. The execution of tasks is time-shared so that each task is given an opportunity to execute.

In the Dynamic Scheduler, the Static_Schedule and the Non_Time_Critical_Operators must operate in conjunction with each other. While the Static_Schedule is executing, the other task must be suspended but ready to execute. At other times, the Non_Time_Critical_Operators may be operating and the Static_Schedule suspended awaiting some action. Therefore to implement these processes as tasks is appropriate.

The order in which tasks are executed can be controlled by the pragma PRIORITY. A pragma notifies the compiler of comments which are not part of the program. These comments serve as instructions to the compiler. Some of the instructions may include when to start a new page for a listing, when and what to optimize, or whether to list sections of the program. One pragma important to the implementation of the Dynamic Scheduler is the PRIORITY pragma. This pragma specifies the priority of a task or of the main body. It takes an argument which can be either an integer or an expression which evaluates to an integer. A task with a higher priority will be executed before a task with a lower priority. The PRIORITY pragma may only appear in the specification of a task or within the declarative part of the main body.

Figure 5-4 is a sample Ada program that demonstrates the use of the pragma PRIORITY. The priority for a task, if applicable, is declared within the specification for that task. Notice that task Two, although declared second, has a higher priority and, thus, should execute first. Because task One has the next lower priority, it should execute second. Because task Three and the main body have no declared priority, there is no way to determine which will execute third and fourth.

Figure 5-5 is a copy of the output produced when the program in Figure 5-4 is executed. Note that the tasks were executed in the expected order.

One disadvantage with tasks is that they cannot be separately compiled. However, this problem is overcome by including a task inside a package. The package can then be separately compiled, thus accomplishing the desired result.

```
with TEXT_IO;     use TEXT_IO;

procedure Example_Priority is

      task One is
            pragma Priority (1);
      end One;

      task Two is
            pragma Priority (2);
      end Two;

      task Three;

      task body One is
            begin
                  PUT_LINE ("Task One executed.");
            end One;

      task body Two is
            begin
                  PUT_LINE ("Task Two executed.");
            end Two;

      task body Three is
            begin
                  PUT_LINE ("Task THREE executed.");
            end Three;

      begin
            PUT_LINE ("Main body executed.");
      end Example_Priority;
```

Figure 5-4. **Example_Priority Program**

```
Task Two executed.
Task One executed.
Task Three executed.
Main body executed.
```

Figure 5-5. **Output from the Example_Priority Program**

The DELAY statement is used in Ada to suspend execution of a task or main body. DELAY takes an argument which is a constant or an expression that has a value of type DURATION. The value is expressed as the number of or portion of seconds the task or main body will be suspended. After a task is suspended by a DELAY statement, other tasks, including the main body, may execute. However, when the length of the delay is over, the task which was suspended will be ready to execute when the processor is available.

Figure 5-6 illustrates how the DELAY statement is used. Task Two prints a line of text to the screen and then delays for one-tenth of a second. During this time, task One will be executed. After the time specified in the delay statement has expired, task One is suspended and task Two is resumed. The line of text after the DELAY statement in task Two is then executed, and the loop is repeated indefinitely. Figure 5-7 is the program Main again, but this time including Example_Delay. Figure 5-8 is a portion of the output produced by Main. This portion is then repeated indefinitely.

```
with TEXT_IO;     use TEXT_IO;

package Example_Delay is
     task One is
          pragma Priority (1);
     end One;

     task Two is
          pragma Priority (2);
     end Two;

end Example_Delay;

package body Example_Delay is
     task body One is
          begin
               loop
                    PUT_LINE ("One executing.");
               end loop;
          end One;

     task body Two is
          begin
               loop
                    PUT_LINE ("Two entered before delay
                         statement.");
                    delay 0.1;
                    PUT_LINE ("Two executing after delay
                         statement.");
                    NEW_LINE;
               end loop;
          end Two;

end Example_Delay;
```

Figure 5-6. **Example_Delay Package**

```
with Example_Delay;

procedure Main is
     begin
          null;
     end Main;
```

Figure 5-7. **Main Program With Example_Delay Package**

```
Two entered before delay statement,
One executing.
One executing.
One executing.
One executing.
One executing.
One executing.
One executing.
One executing.
One executing.
One executing.
One executing.
One executing.
One executing.
One executing.
Two executing after delay statement.
```

Figure 5-8. **Output from Main Program With Example_Delay Package**

2. The Dynamic Scheduler

The Dynamic Scheduler is implemented as a main program called Dynamic_Scheduler. It imports the packages containing the tasks for Static_Schedule and the Non_Time_Critical_Operators. Because the Static Scheduler is not fully implemented, these tasks do not yet exist. The theory behind the implementation of the Dynamic Scheduler has been demonstrated by writing the small programs discussed previously.

The function of the Dynamic Scheduler is to coordinate the execution of the two tasks of operators. Because the schedule for the time-critical operators must begin at time zero, the Static_Schedule task must be executed first. In order to insure this, the pragma PRIORITY and the reserved word delay are used. The pragma PRIORITY must be included in the specifications of the Static_ Schedule and Non_Time_Critical_Operators tasks. As was seen in Figure 5-3, the pragma is included in the Non_Time_Critical_Operators. When the Static Schedule is written, it too must include a pragma PRIORITY in the task specification. Because the Static_ Schedule must be executed first, it must have a higher priority. This higher priority will ensure that when the Static_Schedule is ready to resume execution, it will be given control of the processor at the first available opportunity. Refer to Figure 3-3. The coordination of the execution of the operators in the two tasks is also obtained by having the delay statement follow the execution of each operator. Note the delay statement in the Static_Schedule is part of an *if-elsif* statement and is only executed when time remains before the next operator must start execution. It is this delay statement that suspends the execution of the Static_Schedule and permits the Non_Time_Critical_Operators to execute. When the length of the delay is over, the Static_Schedule is given the use of the processor at the first available opportunity. The state of execution for both tasks is automatically restored by the processor upon resumption of execution of the task.

3. The Debugging System for the Dynamic Scheduler

The debugging system for the Dynamic Scheduler has been implemented as a task called DS_Debug. Appendix B contains the code for DS_Debug. The specification for the task has six entry statements, one for each expected error and two to indicate when the other tasks have completed. This specification is shown in Figure 5-9.

```
task DS_Debug is
     entry Runtime_MET_Failure (Exception_Operator : VSTRING);
     entry Buffer_Underflow;
     entry Buffer_Overflow;
     entry Exception_Error;
     entry  Non_Time_Critical_Operators_Done;
     entry  Static_Schedule_Done;
end  DS_Debug;
```

Figure 5-9. **Specification for DS_Debug**

The implementation for the body of DS_Debug is similar to that of SS_Debug, except for the Runtime_MET_Failure error. The only difference in the implementation for the latter error is in the actions performed by the accept statement. The user must be queried as to whether to continue or to terminate execution of the prototype. This interaction with the user occurs in a dedicated window on the Sun Workstation. If the user wishes to terminate execution, an error message is printed to the file called Information before termination occurs, in a manner similar to that described for SS_Debug.

On the other hand, if a user wants to continue execution, adjustments must be made to the time and the new time returned to the Static_Schedule. An error message is still printed to a file so as to provide a historical record of needed modifications. Consideration must be given to the fact that an operator may execute numerous times. If an operator which frequently executes exceeds its MET, the error message should not be repeatedly written to the file. To prevent this from happening, a data structure called Operators_Overrun is maintained.

Operators_Overrun is a simple linked list whose nodes are records. Each record contains three fields—one for the name of the operator, one for the number of times it has executed and one for a pointer to the next node. Therefore, when an operator exceeds its runtime MET, it is compared to the Operators_Overrun list to determine the appropriate action. If the operator does not appear in the list, then a node for it is inserted and execution continues. If, on the other hand, the operator appears in the list, and if it has executed less than six times, the second field is updated and the execution of the prototype continues. If, however,the operator appears in the list and it has executed more than five times, then an error message is printed stating that an operator with an invalid MET is executing too frequently, and then execution of the prototype is terminated. Note that the number *five* is an arbitrary limit. When familiarity is gained with the average number of times an operator may execute, this figure may be revised.

DS_Debug does have an inherent disadvantage. Because the tasks for which it must rendezvous are imported, DS_Debug cannot be located inside the Dynamic Scheduler. If it were, it would not be visible to the Static_Schedule and to the Non_Time_Critical_Operators which would be declared before it. For this reason, DS_Debug must be separately compiled, and then the Static_ Schedule and the Non_ Time_Critical_Operators packages must include statements for DS_Debug.

In conclusion, the Dynamic Scheduler is implemented as a main program which relies on imported packages to execute the schedules at the appropriate times. The code for the Dynamic_ Scheduler is shown in Figure 5-10.

```
with Static_Schedule_PKG;
     --package containing Static Schedule
with Non_Time_Crits_PKG;
     --package containing Non_Time_Critical_Operators

procedure Dynamic_Scheduler is
     begin
          null;
     end Dynamic_Scheduler;
```

Figure 5-10. **The Dynamic Scheduler**

VI. CONCLUSIONS AND RECOMMENDATIONS

A. CONCLUSIONS

This thesis provides an implementation for the Dynamic Scheduler and debugging systems for the Execution Support System of CAPS and describes the interactions among the units. The need to divide the Execution Support System into the Static Scheduler Execution System and the Dynamic Scheduler was identified and implemented. The thesis demonstrates how the Static Scheduler executes in conjunction with a debugging system to process errors. It also demonstrates how the Dynamic Scheduler can coordinate the execution of the critical and non-critical operators. The implementation is based on the assumption that the Static Scheduler produces a Static Schedule as described in the thesis. If this assumption proves incorrect when the static Scheduler is fully implemented, then the work in this thesis will have to be re-examined in the new context.

Selecting Ada as the implementation language had a significant effect on the feasibility of implementing both parts of the Execution Support System. The existence of tasks in Ada allows the two schedules of operators to execute in parallel, whether in a single-processor environment or in a multi-processor environment. The delay statement allows the execution of the Static Schedule to be suspended, when applicable, and the Non_Time_Critical_Schedule to be executed in any open time slots. Additionally, the ability to separately compile modules and then import these modules into another program directly affected the implementation of the Dynamic Scheduler. Ada has demonstrated its suitability as a programming language for real-time embedded systems.

B. RECOMMENDATIONS

There are significant opportunities for future work in the aspects of the Execution Support System covered in this thesis. One of the important areas is the expansion of SS_Debug to incorporate more interaction with the user, especially with regard to correcting errors. In particular, the possibility of correcting errors in timing constraints during the validation phase of the Static Scheduler should be examined. The correction of as many errors as possible will make the environment more responsive to the user.

Another possible area of work is to have both debugging systems provide statistical information and debugging facilities. Debugging facilities will support monitoring or tracing relevant information concerning operator execution and displaying a record of events that occurred during execution, including computed values and their associated input and output times. Statistical information collected during execution will include frequency of operator firing, number of exceptions occurring, and statistical data on timing parameters for critical operators. [Ref. 5:p. 40]

A third area of interest is implementing the capability to process hardware or operator interrupts. The Execution Support System should be able to respond to an interrupt about equipment failure or to an interrupt from the operator to abort execution.

APPENDIX A
PROGRAM CODE FOR
SS_DEBUG

```
--     This program implements the debugging system for the Static Scheduler          --
--   in the Computer Aided Prototyping System (CAPS). The program                     --
--   consists of a procedure and two tasks. The procedure                             --
--   Create_NTC_Task produces a file that contains an Ada                             --
--   package which can be compiled. This package contains a task that                 --
--   will call the procedures to execute the operators. One of the tasks              --
--   is SS_Debug. This task processes errors encountered during execution             --
--   of the second task, the Static_Scheduler. The code for the body of               --
--   this task is only partially complete. The procedure mentioned                    --
--   earlier is only called if the Static_Scheduler completes execution               --
--     Included in the FILES package, the library unit VSTRINGS is a generic          --
--   string package. It provides the data type VSTRING and also includes              --
--   procedues/functions to manipulate the strings. Since it is generic,              --
--   VSTRINGS must be instantiatd, and the new name must be made visible.             --
--     The FILE_PRCOCESSOR package includes the procedures SEPARATE_DATA and          --
--   VALIDATE_DATA. The TOPOLOGICAL_SORTER package includes CREATE_LISTS and          --
--   SORT_REMAINING_OPERATORS.                                                        --

with FILES;
with FILE_PROCESSOR;
with TOPOLOGICAL_SORTER;
use FILES;

procedure Main is

  Exception_Operator : VARSTRING.VSTRING :=
         VARSTRING.VSTR("");

  --      The Create_NTC_Task procedure writes lines of              --
  --    text to a file called Non_Time_Crits.a. The procedure also reads   --
  --    lines from the file N0N_CRITS and writes them to the first file.   --

  procedure Create_NTC_Task is
    Non_crits : FILE_TYPE;     --name associated with NON_CRITS file
    Non_Time : FILE_TYPE;       -- name associated with the Non_Time_Crits.a
                         -- file
    Operator_Name : VSTRING;

    begin
      open (Non_crits, IN_FILE. "NON_CRITS");
      create (Non_time, OUT_FILE, "Non_Time_Crits.a");

      PUT_LINE (Non_time, "with TL;      --Translator package");
      PUT_LINE (Non_time, "with DS_Debug_PKG; --Debugging package");
      NEW_LINE (Non_time);
    PUT_LINE (Non_time, "package Non_Time_Crits_PKG is");
    PUT_LINE (Non_time, "  task Non_Time_Critical_Operators is");
    PUT_LINE (Non_time, "     pragma Priority (1);");
    PUT_LINE (Non_time, "  end Non_Time_Critical_Operators;");
    PUT_LINE (Non_time, "end Non_Time_Crits_PKG;");
    NEW_LINE (Non_time);

    PUT_LINE (Non_time, "package body Non_Time_Crits_PKG is");
    PUT_LINE (Non_time, "  task body Non_Time_Critical_Operators is");
    PUT_LINE (Non_time, "     begin");

    if END_OF_FILE (Non_crits) then
      PUT_LINE (Non_time, "        null;");
    else
      while NOT END_OF_FILE (Non_crits) loop
        GET_LINE (Non_crits, Operator_Name);
        PUT (Non_time, "        TL.");
        PUT (Non_time, Operator_Name);
        PUT_LINE (Non_time, ";");
      end loop;
    end if;

    NEW_LINE (Non_time);
    PUT (Non_time, "        DS_Debug.Non_Time_Critical_");
    PUT_LINE (Non_time, "Operators_Done;");
    PUT_LINE (Non_time, "     end Non_Time_Critical_Operators;");
    PUT_LINE (Non_time, "end Non_Time_Crits_PKG;");

    close (Non_crits);
    close (Non_time);
  end Create_NTC_Task;

--   The task specification for task SS_Debug contains an entry        --
--   statement for each error that can be encountered by the           --
--   Static_Scheduler task.  The names of the entries correspond to    --
--   the names of the errors.  The last entry statement indicates that --
--   the Static_Scheduler has successfully completed execution.  The   --
--   parameter to the entry statement, where applicable, will provide  --
--   the name of the operator which caused the error.                  --

task SS_Debug is
  entry MET_Not_Less_Than_MRT (Exception_Operator : VSTRING);
  entry MET_Not_Less_Than_Period (Exception_Operator : VSTRING);
  entry No_Initial_Link_Op;
  entry No_Matches_Found (Exception_Operator : VSTRING);
  entry MCP_Not_Less_Than_MRT (Exception_Operator : VSTRING);
  entry MET_Not_Less_Than_MCP (Exception_Operator : VSTRING);
  entry No_Base_Block;
  entry Fail_Half_Period (Exception_Operator : VSTRING);
  entry Bad_Total_Time;
  entry Ratio_Too_Big;
  entry Over_Time;
  entry Invalid_Schedule;
  entry Schedule_Error;
  entry MET_Required (Exception_Operator : VSTRING);
  entry MET_GT_Parent (Exception_Operator : VSTRING);
  entry MET_Sum_GT_Parent (Exception_Operator : VSTRING);
  entry Crit_Op_Lacks_MET (Exception_Operator : VSTRING);
  entry Static_Scheduler_Done;
end SS_Debug;

--   The task body of SS_Debug contains seventeen procedures - one for   --
--   each expected error. Each procedure will print an error message     --
--   to a file called Information. The two BOOLEAN variables control     --
--   or not the task will terminate. The task body consists of a select  --
--   loop that contains an accept block for each entry statement in the  --
--   specification. In this version, although any call to a accept      --
--   block will terminate execution of the task, the accept blocks were  --
--   placed in a loop with an eye to future revisions.                   --

task body SS_Debug is
  Information : FILE_TYPE;
  Error_Exists : BOOLEAN := false;
  Static_Scheduler_Finished : BOOLEAN := false;

  procedure Print_MET_Not_Less_Than_MRT_Message (Information : FILE_TYPE;
                     Exception_Operator : VSTRING) is
    begin
      PUT (Information, "EXECUTION TERMINATED ABNORMALLY.");
      NEW_LINE (Information);
      PUT (Information, "The maximum execution time (MET) is");
      PUT (Information, " greater than or equal ");
      NEW_LINE (Information);
      PUT (Information, "to the maximum response time (MRT) for the");
      PUT (Information, " operator. The");
      NEW_LINE (Information);
      PUT (Information, "operator can only be scheduled for execution ");
      PUT (Information, "if the MET");
      NEW_LINE (Information);
      PUT (Information, "is less than the MRT. The operator which ");
      PUT (Information, "triggered the");
      NEW_LINE (Information);
      PUT (Information, "error is:");
      NEW_LINE (Information);
      PUT (Information, "    ");
      PUT (Information, Exception_Operator);
      NEW_LINE (Information);
    end Print_MET_Not_Less_Than_MRT_Message;

  procedure Print_MET_Not_Less_Than_Period_Message
        (Information : FILE_TYPE; Exception_Operator : VSTRING) is
    begin
    PUT (Information, "EXECUTION TERMINATED ABNORMALLY.");
    NEW_LINE (Information);
    PUT (Information, "The length of the execution time for the ");
    PUT (Information, "following operator");
    NEW_lINE (Information);
    PUT (Information, "is greater than the length of time that must ");
    PUT (Information, "pass before");
    NEW_LINE (Information);
    PUT (Information, "the operator is executed again.  For an ");
    PUT (Information, "operator to");
    NEW_LINE (Information);
    PUT (Information, "be scheduled for execution, its execution time");
    PUT (Information, " must be less");
    NEW_LINE (Information);
    PUT (Information, "than its period.");
    PUT (Information, "The operator which caused the error is:");
    NEW_LINE (Information);
    PUT (Information, "    ");
    PUT (Information, Exception_Operator);
    NEW_LINE (Information);
  end Print_MET_Not_Less_Than_Period_Message;

procedure Print_No_Initial_Link_Op_Message (Information : FILE_TYPE) is
  begin
    PUT (Information, "EXECUTION TERMINATED ABNORMALLY.");
    NEW_LINE (Information);
    PUT (Information, "An operator could not be found that did ");
    PUT (Information, "not have input");
    NEW_LINE (Information);
    PUT (Information, "into it.  Such an operator must exist before ");
    PUT (Information, "a schedule");
    NEW_LINE (Information);
    PUT (Information, "can be built.");
    NEW_LINE (Information);
  end Print_No_Initial_Link_Op_Message;

procedure Print_No_Matches_Found_Message (Information : FILE_TYPE;
              Exception_Operator : VSTRING) is
  begin
    PUT (Information, "EXECUTION TERMINATED ABNORMALLY.");
    NEW_lINE (Information);
    PUT (Information, "The following operator does not match an");
    PUT (Information, " output");
    NEW_LINE (Information);
    PUT (Information, "operator in a link statement.  The ");
    PUT (Information, "operator which");
    NEW_LINE (Information);
    PUT (Information, "caused the error is:");
    NEW_LINE (Information);
    PUT (Information, "    ");
    PUT (Information, Exception_Operator);
    NEW_LINE (Information);
  end Print_No_Matches_Found_Message;
```

```
procedure Print_MCP_Not_Less_Than_MRT_Message (Information : FILE_TYPE;
                        Exception_Operator : VSTRING) is
  begin
    PUT (Information, "EXECUTION TERMINATED ABNORMALLY.");
    NEW_LINE (Information);
    PUT (Information, "The minimum calling period (MCP) is greater");
    PUT (Information, " than or equal");
    NEW_LINE (Information);
    PUT (Information, "to the maximum response time (MRT) for the ");
    PUT (Information, "following operator.");
    NEW_LINE (Information);
    PUT (Information, "For an operator to be scheduled, its MCP must ");
    PUT (Information, "be less than");
    NEW_LINE (Information);
    PUT (Information, "its MRT.  The operator which caused the error");
    PUT (Information, " is:");
    NEW_LINE (Information);
    PUT (Information, "    ");
    PUT (Information, Exception_Operator);
    NEW_LINE (Information);
  end Print_MCP_Not_Less_Than_MRT_Message;

procedure Print_MET_Not_Less_Than_MCP_Message (Information : FILE_TYPE;
                         Exception_Operator : VSTRING) is
  begin
    PUT (Information, "EXECUTION TERMINATED ABNORMALLY.");
    NEW_LINE (Information);
    PUT (Information, "The maximum execution time (MET) is greater");
    PUT (Information, " than or equal");
    NEW_LINE (Information);
    PUT (Information, "to the minimum calling period (MCP) for the ");
    PUT (Information, "following operator.");
    NEW_LINE (Information);
    PUT (Information, "For an operator to be scheduled, its MET must ");
    PUT (Information, "be less than");
    NEW_LINE (Information);
    PUT (Information, "its MCP.  The operator which caused the error");
    PUT (Information, " is:");
    NEW_LINE (Information);
    PUT (Information, "    ");
    PUT (Information, Exception_Operator);
    NEW_LINE (Information);
   end Print_MET_Not_Less_Than_MCP_Message;

 procedure Print_No_Base_Block_Message (Information : FILE_TYPE) is
   begin
     PUT (Information, "EXECUTION TERMINATED ABNORMALLY.");
     NEW_LINE (Information);
     PUT (Information, "The base block could not be determined.");
     NEW_LINE (Information);
   end Print_No_Base_Block_Message;

 procedure Print_Fail_Half_Period_Message (Information : FILE_TYPE;
                Exception_Operator : VSTRING) is
  begin
    PUT (Information, "EXECUTION TERMINATED ABNORMALLY.");
    NEW_LINE (Information);
    PUT (Information, "The maximum execution time (MET) for the ");
    PUT (Information, "following operator");
    NEW_LINE (Information);
    PUT (Information, "is greater than or equal to half of the ");
    PUT (Information, "operator's period.");
    NEW_LINE (Information);
    PUT (Information, "This relationship cannot hold for a static ");
    PUT (Information, "schedule");
    NEW_LINE (Information);
    PUT (Information, "to be created.  The operator which caused the");
    PUT (Information, " error is:");
    NEW_LINE (Information);
    PUT (Information, "    ");
    PUT (Information, Exception_Operator);
    NEW_LINE (Information);
  end Print_Fail_Half_Period_Message;

procedure Print_Bad_Total_Time_Message (Information : FILE_TYPE) is
  begin
    PUT (Information, "EXECUTION TERMINATED ABNORMALLY.");
    NEW_LINE (Information);
    PUT (Information, "All operators in a block cannot be scheduled ");
    PUT (Information, "according to");
    NEW_LINE (Information);
    PUT (Information, "their timing constraints.  This has been ");
    PUT (Information, "determined by");
    NEW_LINE (Information);
    PUT (Information, "multiplying each operator's maximum ");
    PUT (Information, "execution time");
    NEW_LINE (Information);
    PUT (Information, "by the number of times it is supposed to be ");
    PUT (Information, "scheduled within");
    NEW_LINE (Information);
    PUT (Information, "the block (block length / operator period).  ");
    PUT (Information, "The sum");
    NEW_LINE (Information);
    PUT (Information, "over all the operators must be less than the ");
    PUT (Information, "block length.");
    NEW_LINE (Information);
  end Print_Bad_Total_Time_Message;

procedure Print_Ratio_Too_Big_Message (Information : FILE_TYPE) is
  begin
    PUT (Information, "EXECUTION TERMINATED ABNORMALLY.");
    NEW_LINE (Information);
    PUT (Information, "For a schedule to be created, an operator's ");
    PUT (Information, "execution");
    NEW_LINE (Information);
    PUT (Information, "time divided by its period summed over all ");
```

```
      PUT (Information, "operators must ");
      NEW_LINE (Information);
      PUT (Information, "be less than or equal to the number of ");
      PUT (Information, "processors available.");
      NEW_LINE (Information);
      PUT (Information, "This requirement has been violated, and a ");
      PUT (Information, "Static Schedule");
      NEW_LINE (Information);
      PUT (Information, "cannot be created.");
      NEW_LINE (Information);
    end Print_Ratio_Too_Big_Message;

  procedure Print_Over_Time_Message (Information : FILE_TYPE) is
    begin
      PUT (Information, "EXECUTION TERMINATED ABNORMALLY.");
      NEW_LINE (Information);
      PUT (Information, "A static schedule cannot be created based on ");
      PUT (Information, "the given");
      NEW_LINE (Information);
      PUT_LINE (Information, "timing constraints.");
    end Print_Over_Time_Message;

  procedure Print_Invalid_Schedule_Message (Information : FILE_TYPE) is
    begin
      PUT (Information, "EXECUTION TERMINATED ABNORMALLY.");
      NEW_LINE (Information);
      PUT (Information, "A static schedule cannot be created based on ");
      PUT (Information, "the given ");
      NEW_LINE (Information);
      PUT_LINE (Information, "timing constraints.");
    end Print_Invalid_Schedule_Message;

 procedure Print_Schedule_Error_Message (Information : FILE_TYPE) is
   begin
     PUT (Information, "EXECUTION TERMINATED ABNORMALLY.");
     NEW_LINE (Information);
     PUT (Information, "A static schedule cannot be created based on ");
     PUT (Information, "the given");
     NEW_LINE (Information);
     PUT_LINE (Information, "timing constraints.");
   end Print_Schedule_Error_Message;

 procedure Print_MET_Required_Message (Information : FILE_TYPE;
                          Exception_Operator : VSTRING) is
   begin
     PUT_LINE (Information, "EXECUTION TERMINATED ABNORMALLY.");
     PUT (Information, "The following operator has a maximum ");
     PUT (Information, "execution time (MET).");
     NEW_LINE (Information);
     PUT (Information, "However, in its decomposition, at least one ");
     PUT_LINE (Information, "of the operators");
     PUT (Information, "does not have an MET. The operator with the ");
     PUT_LINE (Information, "incorrect ");
    PUT_LINE (Information, "decomposition is:");
    PUT (Information, "    ");
    PUT (Information, Exception_Operator);
    NEW_LINE (Information);
  end Print_MET_Required_Message;

procedure Print_MET_GT_Parent_Message (Information : FILE_TYPE;
                                       Exception_Operator : VSTRING) is
  begin
    PUT_LINE (Information, "EXECUTION TERMINATED ABNORMALLY.");
    PUT (Information, "An operator in a decomposition has a maximum ");
    PUT (Information, "execution time");
    NEW_LINE (Information);
    PUT (Information, "time that is greater than the pre-decomposed ");
    PUT_LINE (Information, "one.  The ");
    PUT_LINE (Information, "pre_decomposed operator is:");
    PUT (Information, "    ");
    PUT_LINE (Information, Exception_Operator);
  end Print_MET_GT_Parent_Message;

procedure Print_MET_Sum_GT_Parent_Message (Information : FILE_TYPE;
                                           Exception_Operator : VSTRING) is
  begin
    PUT_LINE (Information, "EXECUTION TERMINATED ABNORMALLY.");
    PUT (Information, "An operator which has a maximum execution ");
    PUT_LINE (Information, "time has been");
    PUT (Information, "decomposed.  The sum of the execution times ");
    PUT_LINE (Information, "in the decomposition");
    PUT (Information, "is greater than the pre-decomposed operator's");
    PUT_LINE (Information, " execution time.");
    PUT (Information, "This situation cannot occur.  The operator ");
    PUT_LINE (Information, "whose execution time");
    PUT_LINE (Information, "was exceeded is:");
    PUT (Information, "    ");
    PUT_LINE (Information, Exception_Operator);
  end Print_Met_Sum_GT_Parent_Message;

procedure Print_Crit_Op_Lacks_MET_Message (Information : FILE_TYPE;
                         Exception_Operator : VSTRING) is
  begin
    PUT_LINE (Information, "EXECUTION TERMINATED ABNORMALLY.");
    PUT (Information, "Even though the following operator has some ");
    PUT_LINE (Information, "timing constraints,");
    PUT (Information, "it does not have a maximum execution time.  ");
    PUT_LINE (Information, "This situation");
    PUT (Information, "cannot exist.  The operator causing the error ");
    PUT_LINE (Information, "is:");
    PUT (Information, "    ");
    PUT_LINE (Information, Exception_Operator);
  end Print_Crit_Op_Lacks_MET_Message;
```

```
begin          -- main body of task Debug
  create (FILE => Information,
       MODE => OUT_FILE,
       NAME => "Information");

  loop
    select
      accept MET_Not_Less_Than_MRT (Exception_Operator :
                        VSTRING) do
        Error_Exists := true;
        Print_MET_Not_Less_Than_MRT_Message (Information,
                 Exception_Operator);
      end MET_Not_Less_Than_MRT;
    or
      accept MET_Not_Less_Than_Period
                        (Exception_Operator : VSTRING) do
        Error_Exists := true;
        Print_MET_Not_Less_Than_Period_Message (Information,
                 Exception_Operator);
      end MET_Not_Less_Than_Period;
    or
      accept No_Initial_Link_Op do
        Error_Exists := true;
        Print_No_Initial_Link_Op_Message (Information);
      end No_Initial_Link_Op;
    or
      accept No_Matches_Found (Exception_Operator : VSTRING) do
        Error_Exists := true;
        Print_No_Matches_Found_Message (Information,
              Exception_Operator);
      end No_Matches_Found;
    or
      accept MCP_Not_Less_Than_MRT (Exception_Operator : VSTRING) do
        Error_Exists := true;
        Print_MCP_Not_Less_Than_MRT_Message (Information,
              Exception_Operator);
      end MCP_Not_Less_Than_MRT;
    or
      accept MET_Not_Less_Than_MCP (Exception_Operator : VSTRING) do
        Error_Exists := true;
        Print_MET_Not_Less_Than_MCP_Message (Information,
              Exception_Operator);
      end MET_Not_Less_Than_MCP;
    or
      accept No_Base_Block do
        Error_Exists := true;
        Print_No_Base_Block_Message (Information);
      end No_Base_Block;
    or
      accept Fail_Half_Period (Exception_Operator : VSTRING) do
        Error_Exists := true;
        Print_Fail_Half_Period_Message (Information,
                         Exception_Operator);
  end Fail_Half_Period:
or
  accept Bad_Total_Time do
    Error_Exists := true;
    Print_Bad_Total_Time_Message (Information);
  end Bad_Total_Time;
or
  accept Ratio_Too_Big do
    Error_Exists := true;
    Print_Ratio_Too_Big_Message (Information);
  end Ratio_Too_Big;
or
  accept Over_Time do
    Error_Exists := true;
    Print_Over_Time_Message (Information);
  end Over_Time;
or
  accept Invalid_Schedule do
    Error_Exists := true;
    Print_Invalid_Schedule_Message (Information);
  end Invalid_Schedule;
or
  accept Schedule_Error do
    Error_Exists := true;
    Print_Schedule_Error_Message (Information);
  end Schedule_Error;
or
  accept MET_Required (Exception_Operator : VSTRING) do
    Error_Exists := true;
    Print_MET_Required_Message (Information. Exception_Operator);
  end MET_Required;
or
  accept MET_GT_Parent (Exception_Operator : VSTRING) do
    Error_Exists := true;
    Print_MET_GT_Parent_Message (Information, Exception_Operator);
 end MET_GT_Parent;
or
  accept MET_Sum_GT_Parent (Exception_Operator : VSTRING) do
    Error_Exists := true;
    Print_MET_Sum_GT_Parent_Message (Information,
                       Exception_Operator);
  end MET_Sum_GT_Parent;
or
  accept Crit_Op_Lacks_MET (Exception_Operator : VSTRING) do
    Error_Exists := true;
    Print_Crit_Op_Lacks_MET_Message (Information,
                       Exception_Operator);
  end Crit_Op_Lacks_MET;
or
  accept Static_Scheduler_Done do
    Static_Scheduler_Finished := true;
  end Static_Scheduler_Done;
end select;
```

```
        if Error_Exists or Static_Scheduler_Finished then
          close (Information);
          exit;
        end if;

      end loop;
    end SS_Debug;

    task Static_Scheduler;

--    The task body is the main driver for the Static Scheduler.  It          --
--    calls the procedures within the FILE_PROCESSOR and                     --
--    TOPOLOGICAL_SORTER packages.  When complete it will                    --
--    also call the procedures within HARMONIC_BLOCK_BUILDER                 --
--    and OPERATOR_SCHEDULER.                                                --

    task body Static_Scheduler is

      LNKS        : LINKS_LIST.LIST;
      OPS         : OPERATORS_LIST.NARY_TREE;
      ATOMIC_OPS : ATOMIC_LIST.LIST;
      PRECE       : PRECEDENCE_LIST.LIST;

      begin
        FILE_PROCESSOR.SEPARATE_DATA(LNKS,OPS);
        FILE_PROCESSOR.VALIDATE_DATA(OPS,ATOMIC_OPS);
        TOPOLOGICAL_SORTER.CREATE_LISTS(LNKS,PRECE);
        TOPOLOGICAL_SORTER.SORT_REMAINING_OPERATORS(LNKS,PRECE);

        exception
          when FILE_PROCESSOR.CRIT_OP_LACKS_MET =>
            SS_Debug.Crit_Op_Lacks_MET;
          when FILE_PROCESSOR.MET_REQUIRED =>
            SS_Debug.MET_Required;
          when FILE_PROCESSOR.MET_GT_PARENT =>
            SS_Debug.MET_GT_Parent;
          when FILE_PROCESSOR.MET_SUM_GT_PARENT =>
            SS_Debug.MET_Sum_GT_Parent;
          when FILE_PROCESSOR.MET_NOT_LESS_THAN_MRT =>
            SS_Debug.MET_Not_Less_Than_MRT;
          when FILE_PROCESSOR.MET_NOT_LESS_THAN_PERIOD =>
            SS_Debug.MET_Not_Less_Than_Period;
          when TOPOLOGICAL_SORTER.NO_INITIAL_LINK_OP =>
            SS_Debug.No_Initial_Link_Op;
          when TOPOLOGICAL_SORTER.NO_MATCHES_FOUND =>
            SS_Debug.No_Matches_Found;

        end Static_Scheduler;
    end Main;
```

APPENDIX B
PROGRAM CODE FOR
DS_DEBUG_PKG

```
--     The Global_Declarations package contains an instantiation of the        --
--   generic unit VSTRINGS. The instantiation is called VARSTRING. The unit    --
--   contains the data type VSTRING and procedures/functions to manipulate     --
--   strings.                                                                  --
with Global_Declarations;
use Global_Declarations;

with TEXT_IO, CALENDAR;
use TEXT_IO, CALENDAR;

--     The following package contains the debugging system for the Dynamic    --
--   Scheduler. Implemented as a task, the debugging system is called DS_Debug --
--   and processes errors identified during execution of both the time and     --
--   non-time critical operators.                                              --

package DS_Debug_PKG is
--     The specification for task DS_Debug contains six entry statements.      --
--   The first four statements identify errors that may be enountered when the --
--   operators execute. The last two entry statements identify when the        --
--   Static_Schedule and the Non_Time_Critical_Operators tasks have completed. --

  task DS_Debug is
    entry Runtime_MET_Failure (Exception_Operator : VARSTRING.VSTRING;
                      Current_Time : in out TIME;
                      Next_Start : TIME);
    --     The in value for Current_Time is the time the operator completed    --
    --   execution. The out value for Current_Time is the adjusted time        --
    --   backgrounds. Next_Start has as its value the time the next oper-      --
    --   ator must start execution.                                            --

    entry Buffer_Underflow;  --input queue empty
    entry Buffer_Overflow;   --output queue full
    entry Exception_Error;   --unprocessed exception
    entry Static_Schedule_Done;
    entry Non_Time_Critical_Operators_Done;
  end DS_Debug;
end DS_Debug_PKG;

package body DS_Debug_PKG is
  task body DS_Debug is
    type NODE;
    type LINK is access NODE;

    type NODE is
      record
    Operator : VARSTRING.VSTRING; --name of operator exceeding MET
    Executed_count : NATURAL;    --number of times operator has executed
    Next : LINK;
  end record;

Exception_Operator : VARSTRING.VSTRING; --operator causing error
Information : FILE_TYPE; --file containing error information
Error_Exists : BOOLEAN := FALSE;
Static_Schedule_Finished : BOOLEAN :=FALSE;
Non_Time_Critical_Schedule_Finished : BOOLEAN := FALSE;
Found : BOOLEAN := FALSE; --indicates if operator already in list
Choice : CHARACTER := 'A'; --operator's decision as to continue/terminate
Operators_Overrun : LINK := null; --list of operators that have exceeded
                                  -- their MET
Current : LINK;   --pointer to operator in list
Difference : DURATION;   --time over MET
Max_Executions : CONSTANT NATURAL := 5; --maximum number of times an
                                        --operator whose MET is exceeded
                                        --can operate

--     The Find procedure identifies whether the operator is in the list.      --
--   Name contains the name of the operator with the runtime error. If the    --
--   operator is in the list, Current will point to it. If the operator is    --
--   not in the list, Current will point to the last node in the list. The    --
--   value of Found will identify if the operator is already in the list.     --

procedure Find (Head : in LINK; Name : in VARSTRING.VSTRING;
          Current : in out LINK; Found : out BOOLEAN) is
  begin
    Current := Head;

    if Current = null then    --if no nodes in list
      Found := FALSE;
    elsif Current.Next = null then    --if only one node in list
      if VARSTRING.equal (Current.Operator, Name) then
        Found := TRUE;
      else
        Found := FALSE;
      end if;
    else            --traverse list
      while Current.Next /= null
        loop
          if VARSTRING.equal (Current.Operator, Name) then
            Found := TRUE;
          end if;
          Current := Current.Next;
        end loop;

      -- when traversing list, the last node will not be examined.
      -- following "if" ensures last node examined
      if Current.Next = null then
```

```
            if VARSTRING.equal (Current.Operator, Name) then
              Found := TRUE;
            else
              Found := FALSE;
            end if;
          end if;
        end if;
      end Find;

--      The Insert procedure will place a node at the end of the list.  The      --
--    node will contain the name of the operator with the error and the number   --
--    of times the operator has executed.  The number is initialized to one.     --

    procedure Insert (Head : in out LINK; Name : VARSTRING.VSTRING) is
      Temp_Pt : LINK;
      New_Node : LINK;

      begin
        New_Node := new NODE' (Name, 1, null);

        if Head = null then
          Head := New_Node;
        else
          Temp_Pt := Head;
          while Temp_Pt.Next /= null
            loop
              Temp_Pt := Temp_Pt.Next;
            end loop;
          Temp_Pt.Next := New_Node;
        end if;
      end Insert;

--      The next five procedures print an error message to the file Informa-    --
--    tion. The name of each procedure indicates the name of the error it is    --
--    processing.  The last procedure is called when an operator has excecuted  --
--    more frequently than the permitted number of executions (for an operator  --
--    exceeding its MET).                                                        --

    procedure Print_Buffer_Underflow_Message (Information : FILE_TYPE) is
      begin
        PUT (Information, "EXECUTION TERMINATED ABNORMALLY.");
        NEW_LINE (Information);
        PUT (Information, "There was an attempt to read a data buffer ");
        PUT (Information, "that");
        NEW_LINE (Information);
        PUT (Information, "contained no data.");
        NEW_LINE (Information);
      end Print_Buffer_Underflow_Message;

    procedure Print_Buffer_Overflow_Message (Information : FILE_TYPE) is
      begin
      PUT (Information, "EXECUTION TERMINATED ABNORMALLY.");
      NEW_LINE (Information);
      PUT (Information, "There was an attempt to store data into a ");
      PUT (Information, "data buffer");
      NEW_LINE (Information);
      PUT (Information, "that was already full.");
      NEW_LINE (Information);
    end Print_Buffer_Overflow_Message;

procedure Print_Exception_Error_Message (Information : FILE_TYPE) is
    begin
      PUT (Information, "EXECUTION TERMINATED ABNORMALLY.");
      NEW_LINE (Information);
      PUT (Information, "One exception was not processed when another ");
      PUT (Information, "one was");
      NEW_LINE (Information);
      PUT (Information, "raised.");
      NEW_LINE (Information);
    end Print_Exception_Error_Message;

procedure Print_Runtime_MET_Failure_Message (Information : FILE_TYPE;
                   Exception_Operator : VARSTRING.VSTRING) is
   begin
     PUT (Information, "EXECUTION HAS BEEN SUSPENDED OR HAS ");
     PUT_LINE (Information, "TERMINATED ABNORMALLY");
     NEW_LINE (Information);
     PUT (Information, "The following operator did not complete ");
     PUT (Information, "execution ");
     NEW_LINE (Information);
     PUT (Information, "before its maximum execution time was ");
     PUT_LINE (Information, "expired.  The operator");
     PUT (Information, "which caused the error is:");
     NEW_LINE (Information);
     PUT (Information, "     ");
     VARSTRING.PUT (Information, Exception_Operator);
     NEW_LINE (Information);
     NEW_LINE (Information);
   end Print_Runtime_MET_Failure_Message;

procedure Print_Too_Many_Executions_Message (Information : FILE_TYPE;
                        Exception_Operator : VARSTRING.VSTRING) is
   begin
     PUT (Information, "EXECUTION TERMINATED ABNORMALLY.");
     PUT (Information, "The following operator, which executes ");
     PUT_LINE (Information, "frequently, has a maximum");
     PUT (Information, "execution time that is not long enough.  ");
     PUT_LINE (Information, "Execution has been");
     PUT (Information, "terminated because processing time is being ");
     PUT_LINE (Information, "wasted by having");
     PUT (Information, "to handle the error each time the operator ");
     PUT_LINE (Information, "executes.  The operator is:");
     PUT (Information, "     ");
     VARSTRING.PUT_LINE (Information, Exception_Operator);
    end Print_Too_Many_Executions_Message;

--    The following procedure is called when an operator first exceeds its      --
--  MET.  The procedure queries the user as to whether to terminate or not.    --
--  The user is given three attempts to input valid data - either A or B.      --
--  If he has not provided valid data, the procedure will return a value       --
--  of A to terminate execution.  Also, the procedure will print a message     --
--  stating that execution has been terminated due to invalid input.           --

  procedure Obtain_User_Choice (Exception_Operator : VARSTRING.VSTRING;
                                Choice : in out CHARACTER) is
    Count : INTEGER;

    procedure Print_Too_Many_Tries_Message is
      begin
        NEW_LINE;
        PUT ("You exceeded the number of attempts authorized to ");
        PUT ("enter data.");
        NEW_LINE;
        PUT ("Therefore, execution of the prototype has been ");
        PUT ("terminated.");
        NEW_LINE;
      end Print_Too_Many_Tries_Message;

    begin
      Count := 1;
      NEW_LINE;
      NEW_LINE;
      PUT ("Execution of the prototype has been suspended because an");
      NEW_LINE;
      PUT ("operator exceeded its maximum execution time.  The");
      NEW_LINE;
      PUT_LINE ("operator causing the error is: ");
      PUT ("     ");
      VARSTRING.PUT (Exception_Operator);
      NEW_LINE;
      NEW_LINE;
      PUT_LINE ("Do you want to ");
      PUT_LINE ("A.  Terminate execution of the prototype?");
      PUT ("B.  Adjust the execution time of the operator and continue");
      NEW_LINE;
      PUT ("    execution of the prototype?");
      NEW_LINE;
      NEW_LINE;
      PUT_LINE ("Type the letter preceding the option you want.");

      loop
        GET (Choice);
        NEW_LINE;
        NEW_LINE;
      if Choice = 'a' then
        Choice := 'A';
      end if;

      if Choice = 'b' then
        Choice := 'B';
      end if;

      exit when Choice = 'A' or Choice = 'B' or Count = 3;

      PUT ("You typed: ");
      PUT (Choice);
      NEW_LINE;
      PUT_LINE ("You must type either A or B.");

      Count := Count + 1;
    end loop;

    if Choice /= 'A' and Choice /= 'B' then
      Choice := 'A';
      Print_Too_Many_Tries_Message;
    end if;
  end Obtain_User_Choice;



begin          -- main body of task DS_Debug
  create (FILE => Information,
          MODE => OUT_FILE,
          NAME => "Information");

  loop
    select
      accept Buffer_Underflow do
        Error_Exists := true;
        Print_Buffer_Underflow_Message (Information);
      end Buffer_Underflow;
    or
      accept Buffer_Overflow do
        Error_Exists := true;
        Print_Buffer_Overflow_Message (Information);
      end Buffer_Overflow;
    or
      accept Exception_Error do
        Error_Exists := true;
        Print_Exception_Error_Message (Information);
      end Exception_Error;
    or
      accept Runtime_MET_Failure
                    (Exception_Operator : VARSTRING.VSTRING;
                     Current_Time : in out TIME;
                     Next_Start : TIME) do
          Find (Operators_Overrun, Exception_Operator, Current, Found);
            --is operator in Operators_Overrun list?

          if Found then              --check number of executions
            --if operator executed less than that authorized, update
            if Current.Executed_count <= Max_Executions then
              Current.Executed_count := Current.Executed_count + 1;
            else          --terminate and print error message
              Error_Exists := true;
              PUT_LINE ("EXECUTION TERMINATED ABNORMALLY.");
              Print_Too_Many_Executions_Message (Information,
                       Exception_Operator);
            end if;
          else            --query user as to terminate/continue
            Obtain_User_Choice (Exception_Operator, Choice);

            case Choice is
              when 'A' => Error_Exists := true;  --terminate
              when 'B' => Insert (Operators_Overrun,
                             Exception_Operator);
                      --insert operator into Operators_Overrun list
              when others => null;
            end case;

            Print_Runtime_MET_Failure_Message (Information,
                           Exception_Operator);
                --print error message first time operator exceeds MET
          end if;

          Difference := Current_Time - Next_Start;
            --calculate time over MET
          Current_Time := Current_Time - Difference;
            --reset time to the start time of the next operator
        end Runtime_MET_Failure;
      or
        accept Static_Schedule_Done do
          Static_Schedule_Finished := true;
        end Static_Schedule_Done;
      or
        accept Non_Time_Critical_Operators_Done do
          Non_Time_Critical_Operators_Finished := TRUE;
        end Non_Time_Critical_Operators_Done;
      end select;

      if Error_Exists or (Static_Schedule_Finished and
                  Non_Time_Critical_Operators_Finished) then
        close (Information);
        exit;
      end if;

    end loop;
  end DS_Debug;
end Ds_Debug_PKG;
```

LIST OF REFERENCES

1. Booch, G., *Software Engineering With Ada*, 2nd ed., The Benjamin/ Cummings Publishing Company, Inc., 1986.

2. Whitten, J. L., Bentley, L. D.,and Ho, T. I. M., *Systems Analysis and Design Methods*, Times Mirror/Mosby College Publishing, 1986.

3. Luqi and Ketabchi, M., "A Computer-Aided Prototyping System," *IEEE Software*, pp. 66-72, March 1988.

4. Discussion among Software Engineering with Ada class, 29 July 1988.

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

6. Raum, H., *The Design and Implementation of an Expert User Interface for the Computer Aided Prototyping System*, Master's Thesis, Naval Postgraduate School, Monterey, California, December 1988.

7. Thorstenson, R., *A Graphical Editor for the Computer Aided Prototyping System (CAPS)*, Master's Thesis, Naval Postgraduate School, Monterey, California, December 1988.

8. Galik, D., *A Conceptual Design of a Software Base Management System for the Computer Aided Prototyping System*, Master's Thesis, Naval Postgraduate School, Monterey, California, December 1988.

9. Moffitt II, C., *A Language Translator for a Computer Aided Rapid Prototyping System*, Master's Thesis, Naval Postgraduate School, Monterey, California, March 1988.

10. Altizer, C., *Implementation of a Language Translator for a Computer Aided Rapid Prototyping System*, Master's Thesis, Naval Postgraduate School, Monterey, California, December 1988.

11. Janson, D. M., *A Static Scheduler for the Computer Aided Prototyping System: An Implementation Guide*, Master's Thesis, Naval Postgraduate School, Monterey, California, March 1988.

12. O'Hern, J. T., *Conceptual Level Design for a Static Scheduler for Hard Real-Time Systems*, Master's Thesis, Naval Postgraduate School, Monterey, California, March 1988.

13. Marlowe, L., *A Scheduler for Critical Timing Constraints*, Master's Thesis, Naval Postgraduate School, Monterey, California, December 1988.

14. Isoda, S., Shimomura, T., and Ono, Y., "VIPS: A Visual Debugger," *IEEE Software*, pp. 8–18, May 1987.

15. Seviora, R. E., "Knowledge-Based Program Debugging Systems," *IEEE Software*, pp. 20–31, May 1987.

16. Knudsen, J. L., "Better Exception-Handling in Block-Structured Systems," *IEEE Software*, pp. 40–49, May 1987.

17. Diederich, J., and Milton, J., "Experimental Prototyping in Smalltalk," *IEEE Software*, pp. 50–64, May 1987.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria, VA 22304-6145	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, CA 93943-5002	2
3.	Office of Naval Research Office of the Chief of Naval Research ATTN: CDR Michael Gehl, Code 1224 800 N. Quincy Street Arlington, VA 22217-5000	1
4.	Space and Naval Warfare Systems Command ATTN: Dr. Knudsen, Code PD 50 Washington, DC 20363-5100	1
5.	Ada Joint Program Office OUSDRE(R&AT) Pentagon Washington, DC 20363-5100	1
6.	Naval Sea Systems command ATTN: CAPT Joel Crandall National Center #2, Suite 7N06 Washington, DC 20363-5100	1
7.	Office of the Secretary of Defense ATTN: CDR Barber STARS Program Office Washington, DC 20301	1
8.	Office of the Secretary of Defense ATTN: Mr. Joel Trimble STARS Program Office Washington, DC 20301	1

9.	Commanding Officer Naval Research Laboratory Code 5150 ATTN: Dr. Elizabeth Wald Washington, DC 20375-5000	1
10.	Navy Ocean System Center ATTN: Linwood Sutton, Code 423 San Diego, CA 92152-5000	1
11.	National Science Foundation ATTN: Dr. Wiliam Wulf Washington, DC 20550	1
12.	National Science Foundation Division of Computer and Computation Research ATTN: Dr. Tom Keenan Washington, DC 20550	1
13.	National Science Foundation Director, PYI Program ATTN: Dr. C. Tan Washington, DC 20550	1
14.	Office of Naval Research Computer Science Division, Code 1133 ATTN: Dr. Van Tilborg 800 N. Quincy Street Arlington, VA 22217-5000	1
15.	Office of Naval Research Applied Mathematics and Computer Science, Code 1211 ATTN: Mr. J. Smith 800 N. Quincy Street Arlington, VA 22217-5000	1
16.	Defense Advanced Research Projects Agency (DARPA) Integrated Strategic Technology Office (ISTO) ATTN: Dr. Jacob Schwartz 1400 Wilson Boulevard Arlington, VA 22209-2308	1
17.	Defense Advanced Research Projects Agency (DARPA) Integrated Strategic Technology Office (ISTO) ATTN: Dr. Squires 1400 Wilson Boulevard Arlington, VA 22209-2308	1

18.	Defense Advanced Research Projects Agency (DARPA) Integrated Strategic Technology Office (ISTO) ATTN: MAJ Mark Pullen, USAF 1400 Wilson Boulevard Arlington, VA 22209-2308	1
19.	Defense Advanced Research Projects Agency (DARPA) Director, Naval Technology Office 1400 Wilson Boulevard Arlington, VA 22209-2308	1
20.	Defense Advanced Research Projects Agency (DARPA) Director, Strategic Technology Office 1400 Wilson Boulevard Arlington, VA 22209-2308	1
21.	Defense Advanced Research Projects Agency (DARPA) Director, Prototype Projects Office 1400 Wilson Boulevard Arlington, VA 22209-2308	1
22.	Defense Advanced Research Projects Agency (DARPA) Director, Tactical Technology Office 1400 Wilson Boulevard Arlington, VA 22209-2308	1
23.	COL C. Cox, USAF JCS (J-8) Nuclear Force Analysis Division Pentagon Washington, DC 20318-8000	1
24.	LTC Kirk Lewis, USA JCS (J-8) Nuclear Force Analysis Division Pentagon Washington, DC 20318-8000	1
25.	U.S. Air Force Systems Command Rome Air Development Center RADC/COE ATTN: Mr. Samuel A. DiNitto, Jr. Griffis Air Force Base, NY 13441-5700	1

26.	U.S. Air Force Systems Command Rome Air Development Center RADC/COE ATTN: Mr. William E. Rzepka Griffis Air Force Base, NY 13441-5700	1
27.	Professor Luqi, Code 52LQ Computer Science Department Naval Postgraduate School Monterey, CA 93943-5100	1
28.	MaryLou Wood NARDAC, NAS Jacksonville Jacksonville, FL 32212-0111	1