INTEGRATION OF THE EXECUTION SUPPORT SYSTEM FOR THE COMPUTER-AIDED PROTOTYPING SYSTEM (CAPS)

by

Frank V. Palazzo

September 1990

Thesis Advisor: Luqi

Approved for public release; distribution is unlimited.
# Integration of the Execution Support System for the Computer-Aided Prototyping System (CAPS)

**Personal Author(s):**
Frank V. Palazzo

**Type of Report:**
Master's Thesis

**Time Covered:**
September 1990

**Date of Report (Year, Month, Day):**
1990, September

**Page Count:**
138

---

**Abstract**

With the rapidly falling cost of computer hardware continuing to drive software expenses up, attention has turned to ways to effect savings. One approach that shows particular promise is rapid prototyping. Rapid prototyping is the use of executable models of a software system to firm up the requirements before a significant amount of time and effort has been invested in implementation. The computer-aided prototyping system (CAPS) is a rapid prototyping system that automates many of the manual processes of prototyping, thus allowing for quicker prototype construction and even further cost savings. Within CAPS there exists an execution support system. The purpose of the execution support system is to take the description of a prototype written in the prototyping language PSDL and to convert this into an executable prototype which can then be shown to the user. Previous research resulted in separate implementations for the components of the execution support system, but these components were never integrated into a functioning system. It is the development of this tool which is the subject of this thesis.

---

**Address:**
Monterey, CA 93943-5000

**Office Symbol:**
CS

---

**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 United States Government.**

---

**COSATI Codes**

<table>
<thead>
<tr>
<th>Field Group</th>
<th>Sub-Group</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

**Distribution Availability of Abstract**

- Unclassified
- Limited
- Same as Report
- Do not distribute

---

**Telephone:**
(408) 646-2733

---

**Accession Number:**
092-014-6603

---

**Unclassified**
Integration of the Execution Support System for the Computer-Aided Prototyping System (CAPS)

by

Frank V. Palazzo
B.S., Fordham University, 1980

Submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
September 1990

Author: Frank V. Palazzo

Approved by: Luqi, Thesis Advisor

Valdis Berzins, Second Reader

Robert B. McGhee, Chairman
Department of Computer Science
ABSTRACT

With the rapidly falling cost of computer hardware continuing to drive software expenses up, attention has turned to ways to effect savings. One approach that shows particular promise is rapid prototyping. Rapid prototyping is the use of executable models of a software system to firm up the requirements before a significant amount of time and effort has been invested in implementation. The computer-aided prototyping system (CAPS) is a rapid prototyping system that automates many of the manual processes of prototyping, thus allowing for quicker prototype construction and even further cost savings.

Within CAPS there exists an execution support system. The purpose of the execution support system is to take the description of a prototype written in the prototyping language PSDL and to convert this into an executable prototype which can then be shown to the user. Previous research resulted in separate implementations for the components of the execution support system, but these components were never integrated into a functioning system.
It is the development of this tool which is the subject of this thesis.
# TABLE OF CONTENTS

I. INTRODUCTION ............................................................................. 1

II. LANGUAGE AND METHOD .......................................................... 9
   A. LANGUAGE .............................................................................. 9
      1. The PSDL Computational Model ..................................... 9
         a. Operators ....................................................................... 10
         b. Data Streams .................................................................. 11
         c. Timing Constraints ......................................................... 14
         d. Control Constraints ......................................................... 20
      2. Abstractions ...................................................................... 23
         a. Operator Abstractions ................................................. 23
         b. Data Abstractions ......................................................... 24
         c. Control Abstractions ..................................................... 26
   B. THE PROTOTYPING METHOD ................................................. 26

III. COMPUTER-AIDED PROTOTYPING SYSTEM .............................. 29
   A. USER INTERFACE ............................................................... 29
   B. REWRITE SYSTEM ............................................................... 31
   C. DESIGN-MANAGEMENT SYSTEM ....................................... 32
   D. SOFTWARE BASE .................................................................. 32
   E. DESIGN DATABASE ............................................................ 33
APPENDIX E. ADDITIONAL RUNTIME SUPPORT.................96

LIST OF REFERENCES........................................123

INITIAL DISTRIBUTION LIST.................................125
LIST OF TABLES

TABLE 1. PSDL TYPE CONSTRUCTORS.............................25
TABLE 2. SAMPLE REWRITE-SYSTEM RULE TABLE...............31
LIST OF FIGURES

Fig. 1. Traditional Software Life Cycle ...................... 4
Fig. 2. Rapid Prototyping Loop .............................. 6
Fig. 3. A Simple State Machine .............................. 12
Fig. 4. Timing Constraints for a Periodic Operator ...... 16
Fig. 5. Timing Constraints for a Sporadic Operator ...... 18
Fig. 6. PSDL Triggering Conditions .......................... 21
Fig. 7. The Prototyping Method .............................. 28
Fig. 8. CAPS System Architecture ............................ 30
Fig. 9. Execution Support System ............................ 37
Fig. 10. Integration of Reusable Components ................ 42
Fig. 11. Execution Support System, Previous ............... 49
Fig. 12. Sample Dynamic Schedule ........................... 51
Fig. 13. Sample Static Schedule .............................. 54
Fig. 14. Sample Static Schedule .............................. 59
Fig. 15. Sample User Query ................................. 61
Fig. 16. Contents of File Information ........................ 61
Fig. 17. Dataflow Diagram ................................. 64
Fig. 18. Prototype Description .............................. 65
Fig. 19. Reusable Components .............................. 67
Fig. 20. DRIVER Procedures..............................................68
Fig. 21. Dynamic Schedule.............................................70
Fig. 22. Static Schedule..................................................70
Fig. 23. Sample Track Data............................................72
Fig. 24. Corresponding Output........................................73
I. INTRODUCTION

As the cost of computer hardware continues to decline, total software costs continue to grow rapidly as we uncover more and more problem domains that demand an automated solution. In the early 1970s, total software costs for the Department of Defense exceeded $3 billion; it is predicted that in 1990 total software costs for embedded computer systems alone will exceed $32 billion [Ref. 1:p. 8]. According to one economic study, costs related to software accounted for about 2% of the U.S. gross national product in 1980, or about $40 billion [Ref. 2]. According to a more recent estimate, software related expenses had risen to about 5% of the U.S. GNP in 1986, or about $228 billion [Ref. 3].

As the rapidly falling cost of computer hardware continues to drive software costs up, where can savings be effected? One area where significant savings can probably be realized is in the area of software maintenance. Software maintenance is defined as the changes that are made to an existing software system. Reports vary on the actual
figures, but its estimated that industry spends from 40% to 75% of its total hardware and software monies on software maintenance [Ref. 4]. But why are software systems changed?

Software systems are changed for the following reasons:

New situations

Changes in the environment of the system have introduced new requirements. Examples of such changes are new external systems, new policies, new technologies, and new competitive pressures.

User education

Customer requirements have changed because experience with the current version of the system has changed their perception of how computers can be used to solve their problems.

Phased delivery

A partial implementation has been delivered because the customer cannot wait until a complete implementation is available.

Requirements errors

The developers have incorrectly understood the requirements and have produced a system that does not meet user needs.

Implementation errors

A faulty design or implementation does not correspond to the specification. [Ref. j:p. 2-3]

One approach that can help reduce maintenance costs is rapid prototyping. A prototype is an executable model or
pilot version of the intended software system. A prototype is usually a partial representation of the intended system, used as an aid in requirements analysis rather than as production software. The rapid construction activity leading to such a prototype is called rapid prototyping. Prototypes can help customers visualize and test consequences of their requirements, thus allowing the requirements to be stabilized before a significant amount of time and effort has been invested in implementation. [Ref. 6]

The traditional software life cycle consists of a series of phases which yield runnable software only late in the process. One view of the traditional life cycle is illustrated in Fig. 1. A major problem with the traditional approach is that there is no guarantee that the resulting product will meet the customer's needs. Often users will be able to indicate the true requirements only by observing the operation of the system, and the traditional life cycle yields executable programs late in the process, when too much money has already been spent. [Ref. 5:p. 5-6]
This problem is even more exacerbated in the case of hard real-time or embedded systems, where the potential for inconsistencies is greater. One of the major differences between a real-time computer system and a conventional system is the required precision and accuracy of the application software. The response time of each individual
operation may be a significant aspect of the associated requirements, especially for operations whose purpose is to maintain the state of some external system within a specified region, as is common in these types of systems. These response times, or deadlines, must be met or the system will fail to function, possibly with catastrophic consequences. These requirements, which will often exceed the intellectual capacity of a single software engineer, can be very difficult to determine. [Ref. 5:p. 6-7]

Current research suggests a revised software development life cycle based on rapid prototyping. This prototyping life cycle is an alternative to the traditional life cycle which has been proposed to alleviate problems stemming from incorrect requirements, especially when designing hard real-time or embedded systems. As a software development methodology, rapid prototyping provides the user and designer with a fast, efficient and easy-to-use stepwise process. When utilized during the early stages of the development life cycle, rapid prototyping allows validation of the requirements, specifications and initial design before valuable time and effort are expended on
implementation software. Fig. 2 graphically depicts this methodology as a feedback loop. Rapid prototyping is an iterative process that starts out with the user defining the requirements for the critical aspects of the envisioned system. Based on these requirements, the designer then
constructs a model or prototype of the system in a high-
level, prototype description language and examines the 
execution of this prototype together with the user. If the 
prototype fails to execute properly, the user then redefines 
the requirements and the prototype is modified accordingly. 
This process continues until the user determines that the 
prototype successfully meets the critical aspects of the 
envisioned system. Following this validation, the designer 
uses the validated requirements as a basis for the design of 
the production software. [Ref. 5:p. 7-9]

While rapid prototyping has been found to be an effective 
technique for clarifying requirements and eliminating a 
large amount of wasted effort currently spent on developing 
software to meet incorrect or inappropriate requirements, 
the addition of computer aid would make rapid prototyping 
even more beneficial. The computer-aided prototyping system 
(CAPS) is a prototyping system which provides this 
capability. Designed to operate on the prototyping language 
PSDL (Prototype System Description Language), CAPS provides 
the designer with an integrated set of tools to support 
prototyping of complex software systems which may include
hard real-time constraints [Ref. 5:p. 9]. These tools include 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. 7:p. 66]. It is the integration of the execution support system which is the focus of this thesis.

The prototyping language PSDL, along with the prototyping method are examined in Chapter II. An overview of CAPS is given in Chapter III. Integration of the execution support system is discussed in Chapter IV. A complete prototype example is provided in Chapter V. Conclusions are presented in Chapter VI.
II. LANGUAGE AND METHOD

A. LANGUAGE

A good language for expressing design thoughts in terms of a precise model is important for rapid prototyping. It is impossible to do a good design without a language designed especially for this purpose. A powerful, easy-to-use, and portable prototype-description language is a critical part of a computer-aided prototyping system. Such a language is needed before the tools in the system can be built. [Ref. 7:p. 68]

PSDL was designed together with the prototyping method to ensure the most efficient use of the language. It serves as an executable prototyping language at a specification or design level and has special features for real-time system design. [Ref. 7:p. 68]

1. The PSDL Computational Model

The PSDL language is based on a computational model which treats software systems as networks of operators communicating via data streams. The computational model is an augmented directed graph
G=(V,E,T(v),C(v))

where V is the set of vertices, E is the set of edges, T(v) is the set of timing constraints for each vertex v, and C(v) is the set of control constraints for each vertex v. Each vertex is an operator and each edge is a data path. Each of the four components of the graph are described in more detail below. The semantics of a PSDL system description is determined by the associated augmented graph and the semantics of the operators appearing in the diagram. [Ref. 8:p. 2]

a. Operators

All PSDL operators are either FUNCTIONS or STATE MACHINES. When an operator fires, it reads one data value from each of its input streams, and writes at most one data value on each of its output streams. The output objects produced when a function fires depend only on the current input values. The output values produced when a state machine fires depend only on the current input values and the current values of a finite number of internal STATE VARIABLES. [Ref. 6:p. 1411-1412]

Each operator is either ATOMIC or COMPOSITE. Atomic
operators are operators which cannot be decomposed any further. Composite operators are operators which have realizations as data and control flow networks of lower level operators. A composite operator whose network contains cycles is a state machine. In such a case, one of the data streams in each cycle is designated as the state variable controlling the feedback loop, and an initial value is specified for it. The state variables serve to break the circular precedence relationships among the operators which would otherwise be implied. In the example shown below, there is a cycle consisting of the streams S and Y. The stream X is the only input to the network, and Z is the only output. [Ref. 6:p. 1412]

In PSDL, S would be designated as the state variable of this cycle by including the following

states S initially S-0

in the specification part of the composite operator. S-0 gives the initial value for S. [Ref. 6:p. 1412]

b. Data Streams

PSDL operators communicate by means of data streams. A data stream is a communication link connecting exactly one
producer operator to exactly one consumer operator. In Fig. 3, Y is a data stream from producer op-1 to consumer op-2. Each stream carries a sequence of data values of an abstract data type. Streams have the pipeline property: if a and b are two data values in data stream Y and the data value a is generated by op-1 before value b is generated then it is
impossible for a to be delivered to op-2 after b is delivered. [Ref. 6:p. 1412]

There are two types of data streams in PSDL-DATA FLOW STREAMS and SAMPLED STREAMS. Dataflow streams are used in applications where the values in the stream must not be lost or replicated, while sampled streams are used in applications where a value must be available at all times and values can be replicated without affecting their meaning. A dataflow stream can be thought of as a fifo queue of length one, while a sampled stream can be thought of as a cell capable of containing just one value, which is updated whenever the producer generates a new value. [Ref. 6:p. 1412]

Dataflow streams guarantee that each of the data values written into the stream is read exactly once. Computation sequences that would require a value to be written into a full queue or to be read from an empty queue result in an error message. While computation sequences that would require a value to be written to a non-empty sampled stream are valid, attempts to read from an uninitialized sampled stream are not, and will result in an
error message. Thus, this type of error can be avoided by declaring initial values for sampled streams. [Ref. 8:p. 3-4]

c. Timing Constraints

Any PSDL operator can have timing constraints associated with it. An operator is time-critical if it has at least one timing constraint associated with it, and is non time-critical otherwise. The timing constraints together with the control constraints determine when the operator can be fired, and when it must be fired. There are several different kinds of timing constraints. [Ref. 8:p. 4] The most basic are given in the specification part of a PSDL module, and consist of the MAXIMUM EXECUTION TIME, the MAXIMUM RESPONSE TIME, and the MINIMUM CALLING PERIOD [Ref. 6:p. 1416].

Every time-critical operator must have a maximum execution time (MET) to allow the construction of a static schedule. The MET of an operator is an upper bound on the length of time between the instant when a module begins execution and the instant when it completes, called the execution interval (EI). All of the actions that may be
required to fire an operator once must fit into the execution interval. These actions are listed below.

(1) Reading values from input data streams.
(2) Evaluating triggering conditions.
(3) Calculating output values.
(4) Evaluating output guards.
(5) Writing values into output streams. [Ref. 8:p. 4-5]

Operators triggered by temporal events are periodic in PSDL. Every periodic operator must have a period (PERIOD) and may have a deadline (FINISH_WITHIN). These two timing constraints partially determine the set of scheduling intervals (SI) for the operator. Each periodic operator must be fired exactly once in each scheduling interval, and must complete execution before the end of the scheduling interval. The period is the length of time between the start of any scheduling interval and the start of the next scheduling interval. The deadline is the length of each scheduling interval. The starting time of the first scheduling interval for each operator is determined by the static scheduler. [Ref. 8:p. 5]

The relation between the timing constraints, scheduling intervals, and execution intervals for a periodic
operator is illustrated in Fig. 4. The execution intervals and scheduling intervals in the diagram are indexed by integers in the order of their occurrence. Thus $SI[n]$ denotes the $n$-th scheduling interval for the operator and $EI[n]$ denotes the $n$-th execution interval for the operator.

**Fig. 4 Timing Constraints for a Periodic Operator**
The static scheduler takes the length of each execution interval to be equal to the maximum execution time to allow for worst case conditions. If a time-critical operator completes before the end of the execution interval reserved for it by the static scheduler, the remaining time in the execution interval is used by the dynamic scheduler for the execution of a non time-critical operator. [Ref. 8:p. 5]

Operators triggered by the arrival of new data values are sporadic. Timing constraints for sporadic operators are optional. Sporadic operators with timing constraints must have both a maximum response time (MRT) and a minimum calling period (MCP) in addition to a MET. The MRT is an upper bound on the time between the arrival of a new data value and the time when the last value is put into the output streams of the operator in response to the arrival of the new data value. The MCP is a constraint on the environment of a sporadic operator, consisting of a lower bound on the delay between the arrival of one set of inputs and the arrival of the next set. The relation between these quantities is illustrated in Fig. 5. SI[n] denotes the n-th scheduling interval for the consumer operator,
which is sporadic and time-critical. CEI[n] denotes the n-th execution interval for the consumer operator, and PEI[n] denotes the n-th execution interval for the producer operator, which is assumed here to be time-critical also.

The response time associated with a consumer operator is measured from the end of the execution interval for the

![Diagram](image)

- **PEI[n]** = n-th producer execution interval
- **CEI[n]** = n-th consumer execution interval
- **SI[n]** = n-th scheduling interval

*Fig. 5 Timing Constraints for a Sporadic Operator*
producer operator of the triggering data value to the end of the execution interval for the consumer operator of the triggering data value. [Ref. 8:p. 6]

Unlike the MET, the MRT includes a scheduling delay. The MRT gives the length of the scheduling interval. The static scheduler may not be able to use the entire scheduling interval if the producer is non time-critical, because the ending time of the producer’s execution interval is not known to the static scheduler in that case. [Ref. 8:p. 6]

The calling period of an operator is the length of time between the end of the execution interval for the producer of the triggering data value and the end of the execution interval for the producer of the next triggering data value. The calling period must not be less than the MCP. The MCP of an operator constrains the behavior of the producers of the triggering data values rather than constraining the behavior of the operator itself. An MCP constraint is needed to allow the realization of a maximum response time constraint with a fixed amount of computational resources, via a limit on the frequency with
which new data can arrive. Violation of an MCP constraint should result in a warning message. [Ref. 8:p. 6-7]

d. Control Constraints

The control aspect of a PSDL operator is specified implicitly, via control constraints. Control constraints are used for the following purposes:

(1) Controlling operator execution.
(2) Controlling output.
(3) Controlling exceptions.
(4) Controlling timers. [Ref. 8:p. 7]

Exceptions are discussed in Chapter IV. Readers interested in a discussion of timers can refer to [Ref. 11:pp. 18-20].

Control constraints controlling operator execution are called triggering conditions. The forms of PSDL triggering conditions are shown in Fig. 6. A triggering condition has two parts, the trigger and the guard, both of which are optional. The trigger defines the conditions under which an operator can be fired. The keywords "TRIGGERED BY ALL" indicate the operator is ready to fire whenever new data values have arrived on all of the input data streams named in the id_list. The id_list can contain any non-empty subset of the input data streams for the
operator. [Ref. 8:p. 7] This kind of trigger can be used to ensure that the output of the operator is always based on fresh data for all of the inputs in the list, and can be used to synchronize the processing of corresponding input values from a number of input streams [Ref. 6:p. 1414]. The natural dataflow firing rule corresponds to a "TRIGGERED BY ALL" triggering condition that lists all of the input data streams of the operator. [Ref. 8:p. 7-8]

```
triggering_condition =
 "OPERATOR" id "TRIGGERED" ["BY" trigger] ["IF" guard]
 trigger = "ALL" id_list | "SOME" id_list
```

Fig. 6 PSDL Triggering Conditions

The keywords "TRIGGERED BY SOME" indicate the operator can be fired if there is a new data value on at least one of the data streams named in the id_list [Ref. 8:p. 8]. This kind of trigger can be used to keep software estimates of sensor data up to date [Ref. 6:p. 1414]. A null trigger means that the operator can fire at any time, whether or not any new data values have arrived. Null triggers are most useful for periodic operators.

The triggering conditions of the operators implicitly determine the types of the data streams. A
stream is a dataflow stream if it appears in a "TRIGGERED BY ALL" constraint of the consumer operator and is a sampled stream otherwise. [Ref. 8:p. 8]

The second part of the triggering condition is the guard, a boolean expression which can depend only on the input values to the operator and locally available state information. A null guard is always true. When an operator fires, it reads one data value from each of its input streams. If the predicate is satisfied, then the output values of the operator are calculated, otherwise the firing of the operator terminates immediately without producing any output. [Ref. 8:p. 8]

Constraints for controlling output are based on output guards. An output guard is a boolean expression that can depend on the input data of the operator, locally available state information, and the calculated output values. A null output guard is always true. [Ref. 8:p. 8] An example of a control constraint specifying a conditional output is shown below.

OPERATOR t OUTPUT z IF 1 < z AND z < max
The example shows an operator with an output guard, which depends on the input value MAX and the output value z [Ref. 6:p. 1415]. An output value is written to the output stream provided that the output guard evaluates to true. If the output guard evaluates to false, then nothing is written into the stream. [Ref. 8:p. 8]

2. Abstractions

Abstractions are an important means for controlling complexity, which is especially important in rapid prototyping because a system must appear to be simple to be built or analyzed quickly. PSDL supports three kinds of abstractions: data abstractions, operator abstractions, and control abstractions. [Ref. 6:p. 1413]

a. Operator Abstractions

There are two types of operator abstractions: functional abstractions and state machine abstractions. PSDL supports both types. A PSDL operator consists of two parts: a SPECIFICATION part and an IMPLEMENTATION part. The specification part contains attributes describing the form of the interface, the timing characteristics, and descriptions of the observable behavior of the operator.
The attributes both specify the operator and form the basis for retrievals from a software base. The set of attributes consists of GENERIC PARAMETERS, INPUT, OUTPUT, STATES, EXCEPTIONS, and TIMING INFORMATION. [Ref. 6:p. 1413]

A PSDL operator corresponds to a state machine abstraction if its specification part contains a STATES declaration, otherwise it corresponds to a functional abstraction. The STATES declaration gives the types of the state variables and also their initial values. [Ref. 6:p. 1413]

The implementation part of the operator signifies whether it is atomic or composite. Atomic operators have a keyword specifying the underlying programming language, followed by the name of the module implementing the operator. This name is filled in as the result of a successful retrieval from the software base, or is supplied by the designer in cases where the module cannot be constructed from reusable components and must be coded manually. [Ref. 6:p. 1413]

b. Data Abstractions

All of the PSDL data types are immutable. For
immutable types the set of instances and the properties of each instance are fixed. All PSDL variables are local. Both mutable data types and global variables can lead to coupling problems in large prototype systems and thus have been excluded from PSDL. [Ref. 6:p. 1413]

The PSDL data types include the immutable subset of the built-in types of Ada, user defined abstract types, the special types TIME and EXCEPTION, and the types that can be built using the immutable type constructors of PSDL. The PSDL type constructors (see Table 1) were chosen to provide powerful data modeling facilities. [Ref. 6:p. 1414]

<table>
<thead>
<tr>
<th>TABLE 1. PSDL TYPE CONSTRUCTORS</th>
</tr>
</thead>
<tbody>
<tr>
<td>set[item: type]</td>
</tr>
<tr>
<td>sequence[item: type]</td>
</tr>
<tr>
<td>map[from: type, to: type]</td>
</tr>
<tr>
<td>tuple[tag_1: T_1,..., tag_n: T_n]</td>
</tr>
<tr>
<td>one_of[tag_1: T_1,..., tag_n: T_n]</td>
</tr>
<tr>
<td>relation[tag_1: T_1,..., tag_n: T_n]</td>
</tr>
</tbody>
</table>

Finite sets, sequences, and mappings correspond to the usual mathematical concepts. Tuples are finite Cartesian products, with operations for constructing tuples and for extracting components. One_ofs are tagged disjoint unions of a finite number of other types, with operations for constructing one_of values with a given tag.
(injections), for testing whether a one_of value has a given tag (domain predicates), and for extracting the data component of a one_of (projections). Relations are n-ary mathematical relations, with operations that are commonly used in relational databases (select, project, join, union, set difference, etc.). [Ref. 6:p. 1414]

c. Control Abstractions

Enhanced data flow diagrams, augmented by control constraints represent the control abstractions of PSDL. Periodic execution is supported explicitly, as well as concurrent execution in certain cases. Order of execution is determined from the data flow relations given in the enhanced data flow diagrams, based on the rule that an operator consuming a data value must not start until after the operator producing the data value has completed. This applies to periodic operators only at synchronization points, which occur at intervals equal to the least common multiple of the periods of the two operators.

B. THE PROTOTYPEING METHOD

The PSDL prototyping method is a decomposition strategy which results in a hierarchically structured prototype. The
prototyping method provides a strategy for filling in more
details at any level of the prototype design and helps the
designer focus on the critical subsystems that must be
refined to resolve the problems that motivated the rapid
prototyping effort. [Ref. 9:p. 26]

Fig. 7 illustrates the major steps in the prototyping
method. The designer begins by entering the specifications
of the intended software component. A rewrite subsystem
maps the specification into an internal abstract form that
is used by the design-management system to search for a
reusable component to implement the specification. If a
reusable component is found, the design-management system
retrieves it; if it finds several reusable components that
meet the specification, the designer must choose one.
Otherwise, the specification cannot be met by an existing
component and the designer should decompose the
specification into simpler specifications or create a hand-
coded implementation if the component is so simple that
decomposition does not make sense. [Ref. 7:p. 67]

When a specification is decomposed into a network of
simpler components, the required interconnections are
recorded in the design database with a dataflow diagram. This helps serve as design documentation. After the designer decomposes the specification into simpler specifications, the entire prototyping method is then applied to those specifications. [Ref. 7:p. 67]
III. COMPUTER-AIDED PROTOTYPING SYSTEM

The computer-aided prototyping system (CAPS) is a set of software tools which reduces the designer's efforts by automating time-consuming tasks in conventional prototyping, such as turning specifications into prototypes, modifying prototypes, and searching for available reusable components [Ref. 7:p. 67-68]. Designed to operate on the prototyping language PSDL, CAPS consists of a syntax-directed editor with graphics capabilities, a rewrite system, a design-management system, a software base, a design database, and an execution support system [Ref. 7:p. 66]. Fig. 8 shows the architecture of the computer-aided prototyping system. Following is a short description of the functions of each of the components of CAPS.

A. USER INTERFACE

The user interface consists of a syntax-directed editor for PSDL and a graphics tool to construct and display dataflow diagrams. The syntax-directed editor expedites the designer's data entry at a terminal by eliminating syntax errors, automatically supplying keywords, and prompting
with a choice of legal syntactic alternatives at each point. [Ref. 9:p. 29]

The graphics tool provides a graphical view of the enhanced data flow diagram in the PSDL implementation of a composite module. It helps you visualize the relationships between the components of a decomposition through a two-
dimensional dataflow diagram and provides a convenient way to enter and update the decomposition information in the dataflow diagram. [Ref. 9:p. 29]

B. REWRITE SYSTEM

The rewrite system translates semantically equivalent specifications into a common (normalized) form that is used by the design-management system to search for components. Normalized components are easier to retrieve because there are fewer keys to search for in the software base. This is a more practical approach than trying to generate all variations of a description and searching the software base for each variation. Table 2 shows an example of an informal rewriting system. [Ref. 7:p. 69]

<table>
<thead>
<tr>
<th>Term</th>
<th>Aliases</th>
</tr>
</thead>
<tbody>
<tr>
<td>update</td>
<td>change, modify, refresh, replace, substitute</td>
</tr>
<tr>
<td>read</td>
<td>fetch, obtain, input, get, retrieve</td>
</tr>
</tbody>
</table>

The rewrite system would replace all occurrences of the aliases by the associated basic terms. The sentence "Fetch the order from the transaction file and modify the
C. DESIGN-MANAGEMENT SYSTEM

The design-management system is responsible for organizing, retrieving, and instantiating reusable components from the software base and for managing the versions, refinements, and alternatives of the prototypes in the design database. The design-management system is essentially a database-management system that can efficiently manage long transactions, data describing complex objects (such as software components), the iterative and tentative nature of the design process that leads to versions, refinements, and alternatives of the design objects, and concurrent design operations in a distributed computing environment. It also provides special-purpose operations to compose components, and a browsing capability similar to the one provided by the Smalltalk environment. [Ref. 7:p. 69]

D. SOFTWARE BASE

The software base consists of PSDL descriptions and code
for all available reusable software components. Each component in the software base must have a PSDL specification. This PSDL specification is organized as a set of distinct but related attributes. The design-management system provides component retrieval based on partial matches of these attributes. The software base contains a relatively complete set of general-purpose components to perform the functions common to many systems, such as managing displays, sorting and searching, parsing input strings, and managing look-up tables. [Ref. 9:p. 29]

E. DESIGN DATABASE

The design database in the prototyping environment contains the PSDL prototype descriptions for each software development project using CAPS. This design history consists of the relationship between each version of the requirements and the corresponding versions of parts of the prototype. This information is useful in cases where parts of the requirements are returned to previous configurations, because it enables the system to help restore the corresponding parts of the prototype to their previous configurations. The design database will support retrievals
of the form

- given a requirement, find all the PSDL components that implement it, or

- given a PSDL component, find all the requirements it implements. [Ref. 9:p. 29]

F. EXECUTION SUPPORT SYSTEM

Within the CAPS architecture there exists an execution support system which allows the designer to execute the prototype. This support system consists of three major components: a translator, a static scheduler, and a dynamic scheduler. The translator generates code binding together the reusable components extracted from the software base. Its main functions are to implement data streams and control constraints. The static scheduler allocates time slots for operators with real-time constraints before execution begins. If the allocation succeeds, all operators are guaranteed to meet their deadlines even with worst-case execution times. The dynamic scheduler invokes operators without real-time constraints in the time slots not used by the operators with real-time constraints as execution proceeds. [Ref. 5:p. 16] [Ref. 10:p. 15-16]
The execution support system is covered extensively in the remaining chapters.
IV. INTEGRATION OF THE EXECUTION SUPPORT SYSTEM

A. OVERVIEW

The execution support system as conceptualized by this author is shown in Fig. 9. It consists of three processes: the translator, the static scheduler and the dynamic scheduler. The translator takes as input the PSDL source code generated by the designer and produces DRIVER procedures, contained in package TL, which call the reusable components in package SB. The static scheduler takes as input the PSDL source code and produces two outputs: the static schedule for the time-critical operators and a file called non_crits containing the names of the non-time-critical operators. The dynamic scheduler takes as input the file non_crits and uses this to produce an Ada task containing procedure calls for the non-time-critical operators.

The user interface is responsible for invoking the translator, static scheduler and dynamic scheduler. The translator and the static scheduler can operate in parallel, however the dynamic scheduler must wait for completion of
the static scheduler before it can run. Once the translator, static scheduler and dynamic scheduler have completed, the user interface takes their outputs as well as the packages: PSDL_SYSTEM, PSDL_EXCEPTIONS, SB, Global Declarations, DS_Debug_PKG, TIMERS, VSTRINGS, List_Single_
Unbounded_Unmanaged and PRIORITY_DEFINITIONS and compiles and links them, thus forming the executable prototype. Note, the latter four packages are the ADDITIONAL RUNTIME SUPPORT.

Previous research resulted in separate implementations for the translator [Ref. 11], static scheduler [Ref. 12] [Ref. 13] [Ref. 14] [Ref. 15], and dynamic scheduler [Ref. 16]. These individual components were not previously integrated though into a functioning execution support system which enables the execution of prototypes written in PSDL. It is the development of this tool that is the focus of this thesis.

B. MODIFICATIONS TO THE TRANSLATOR

1. Data Stream Implementation

Data streams are the means by which PSDL operators communicate. A data stream is a communication link connecting exactly one producer operator to exactly one consumer operator. Altizer [Ref. 11:p. 40] describes an implementation for data streams in which data streams are implemented as an Ada generic package containing an embedded task. Inside the task are the various operations necessary
to support data triggers and data stream reads and writes. Four separate packages are provided to implement the variations of data streams. All of these packages are then grouped into a package called PSDL_SYSTEM.

While Altizer's handling of data streams was found to be essentially adequate, several modifications were made to the original design. Appendix A contains the new version of PSDL_SYSTEM.

One change made was to replace the literal value in the pragma PRIORITY with the constant BUFFER_PRIORITY. The value of BUFFER_PRIORITY is declared in the package PRIORITY_DEFINITIONS along with the values of the constants STATIC_SCHEDULE_PRIORITY and DYNAMIC_SCHEDULE_PRIORITY. The purpose of doing this is to consolidate all the PRIORITY pragma values for the prototype into one package, so that future design changes and enhancements will be easier to make.

Another change made to the original design was to eliminate the record type DATA_STREAM_TOKEN. DATA_STREAM_TOKEN was determined to be unnecessary and was replaced instead with two local variables: BUFFER and FRESH.
The names of several identifiers in PSDL_SYSTEM were changed. For instance, SAMPLED_STREAM was changed to SAMPLED_BUFFER. DATA_STREAM was changed to BUFFER. GET was changed to READ. PUT was changed to WRITE, etc. This was done in the interest of improving clarity.

Finally, probably the most significant change made was to consolidate the generic packages SAMPLED_STATE_VAR and DATAFLOW_STATE_VAR into one package STATE_VARIABLE. Having both SAMPLED_STATE_VAR and DATAFLOW_STATE_VAR was found to be redundant.

2. Integration of Reusable Components

In Chapter II. of this thesis, a short description is given of the prototyping method, which is the process by which a PSDL atomic operator specification is used to extract a reusable Ada component from the software base. These components, which represent the implementations of the atomic operators, are critical to the functioning of the prototype. Yet, up until Altizer [Ref. 11:p. 38], no provision was made for their insertion into the prototype.

Altizer proposed that when a reusable module is extracted from the software base, that it be inserted into
the PSDL source code at the IMPLEMENTATION statement. Then, when the PSDL source code is processed by the translator this component would be extracted and placed in the TL package, which is the output of the translator.

This approach, though it seems viable, was never actually implemented by Altizer. An alternative approach that actually was implemented during the course of this thesis, makes use of the Ada with and use clauses. In this approach all components extracted from the software base are assembled into a package called SB which is placed into a file called sb.a. The output of the translator, package TL, consists then of DRIVER procedures which call these components. Visibility to package SB is provided via the Ada with and use clauses. This is illustrated in Fig. 10.

3. Exception Handling

PSDL exceptions are described by Luqi in [Ref. 17] as special data types that may be written to any data stream regardless of the stream’s normal data type. When a PSDL exception is raised in an operator it is immediately output onto all of the operator’s output data streams. All operators immediately downstream from this operator receive
the exception on one or more of their input streams and are then responsible for its handling.

```ada
with TEXT_IO; use TEXT_IO;
with PSDL_EXCEPTIONS; use PSDL_EXCEPTIONS;
package SB is
  .
  .
end SB;

package body SB is
  .
  .
  reusable Ada components
  .
  .
end SB;

package TL is
  .
  .
end TL;

with PSDL_SYSTEM; use PSDL_SYSTEM;
with PSDL_EXCEPTIONS; use PSDL_EXCEPTIONS;
with SB; use SB;
package body TL is
  .
  .
instantiations of data streams
  .
  .
DRIVER procedures
  .
  .
end TL;
```

Fig. 10 Integration of Reusable Components

Altizer in [Ref. 11] describes an alternative method for handling PSDL exceptions. In his approach, a distinction is made between streams which carry normal data
values and streams which carry PSDL exceptions. Streams which carry PSDL exceptions are of type PSDL_EXCEPTIOM and must be declared explicitly by an operator. An operator allowed to raise a PSDL exception can only communicate that exception to operators which are directly connected to it via an exception data stream. An operator receiving the PSDL exception on its input data stream is then by definition an exception handler.

While Altizer's method was determined to be easier to implement than the method originally proposed by Luqi, it was also determined to be inadequate. The approach to exception handling eventually decided upon in this thesis draws on this approach but differs somewhat. The following paragraphs will describe an example of a PSDL to Ada translation involving PSDL exceptions. At that time the method adopted in this thesis should become clear. Deviations from previous research in both method and implementation will be noted.

Appendix B contains a sample prototype which performs no particular function. The example consists of an enhanced data flow diagram, a prototype description, parts of its Ada
implementation and a supporting Ada package. Operator C1 is a composite operator. OP1, OP2 and OP3 are atomic operators. OP1, OP2 and OP3 are subcomponents of C1. Operator EXCEPTION_HANDLER is an atomic operator. Its purpose is to handle any exceptions raised by OP1. The data streams visible in this diagram are A, B and EXCP. A and B are data streams of normal data types. EXCP is an exception data stream. It is of type PSDL_EXCEPTION.

Referring to Section B of Appendix B, it can be seen that data streams A and B are declared to be of type INTEGER. Data stream EXCP is declared to be of type PSDL_EXCEPTION. The statement

```
EXCEPTION OUT_OF_RANGE IF A > 100
```

is a PSDL EXCEPTION statement. It is interpreted as "If the value which is to be output to A is greater than 100, then output the PSDL exception OUT_OF_RANGE onto all of the operator's output data streams declared to be of the PSDL_EXCEPTION data type. Do not output the calculated value onto A." This represents a departure from previous research. Prior to this thesis, the syntax of the EXCEPTION statement required that a target data stream be specified.
In the interest of efficiency, it was decided that the default be all data streams of type PSDL_EXCEPTION and that if the designer wished to suppress outputting the exception to any particular stream he could do so via an Output Condition [Ref. 6:p. 1415-1416].

The statement

TRIGGERED BY ALL EXCP IF EXCP = OUT_OF_RANGE

represents another departure from previous research. It is a PSDL triggering condition which is interpreted here as "If there is a new data value on input data stream EXCP, then consume the data value. If the value consumed equals the PSDL exception OUT_OF_RANGE, then allow the operator EXCEPTION_HANDLER to execute. If the expression EXCP = OUT_OF_RANGE is not satisfied, then EXCEPTION_HANDLER is not allowed to execute any farther and is terminated." Altizer introduced a special operation in his thesis to perform the above function, which is to determine if a particular exception has been raised. This was determined to be unnecessary. A special note here. In the above statement, the condition EXCP = OUT_OF_RANGE is optional. If this condition is null, then the presence of any PSDL exception
on EXCP will cause EXCEPTION_HANDLER to execute.

One last item of interest in Section B of Appendix B is the statement

EXCEPTIONS NEGATIVE_NUMBER

This statement is interpreted as "During the execution of operator OP1, it's possible for an Ada exception NEGATIVE_NUMBER to be raised. If this happens convert NEGATIVE_NUMBER into a PSDL exception and output it onto all data streams of type PSDL_EXCEPTION, which in this case is only one stream, EXCP." This provides a mechanism for interfacing between Ada exceptions and PSDL exception handlers. Ada exceptions that are not declared in the PSDL specifications are treated as errors and cause abnormal termination of the prototype.

Section C of Appendix B is a partial Ada implementation of the prototype described in Section B. It consists of two packages: TL and SB. Package SB is as described earlier in this thesis. It is a package containing the reusable components extracted from the software base. Package TL is the output of the translator. It consists of DRIVER procedures which call the components in SB.
In both package TL and package SB visibility is established to a package PSDL_EXCEPTIONS via the Ada with and use clauses,

    with PSDL_EXCEPTIONS; use PSDL_EXCEPTIONS;

Package PSDL_EXCEPTIONS is an Abstract Data Type (ADT). Its purpose is to define the type PSDL_EXCEPTION and various operations applicable to objects of the type. When the reusable components were placed into a separate package SB, it was found necessary to abstract out the type PSDL_EXCEPTION and to create the ADT. PSDL_EXCEPTIONS is listed in Section D of Appendix B.

C. IMPLEMENTATION OF THE DYNAMIC SCHEDULER

As stated at the beginning of this chapter, the execution support system consists of three major components: a translator, a static scheduler and a dynamic scheduler. Previous research resulted in implementations for these components, but they were never integrated into a functioning execution support system which enables the execution of prototypes written in PSDL. Providing this tool is the purpose of this thesis.
In the case of the translator, Altizer [Ref. 11] provided an implementation which when modified as described earlier, was assimilated into the execution support environment. In the case of the static scheduler, Kilic [Ref. 15] provided an implementation which when modified was assimilated into the execution support environment. Not so with the dynamic scheduler though.

The dynamic scheduler as implemented by Wood [Ref. 16] presented a problem. The problem is that Wood's dynamic scheduler is based on a conceptualization of the execution support system which differs from the one adopted in this thesis. As envisioned by Wood, Fig. 11, the translator is a separate process which can execute in parallel with the static scheduler. The static scheduler is 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. The dynamic scheduler coordinates the execution of the critical and non-critical operators. It and its debugging system form another distinct part of the execution support system.
According to Wood, the translator and the static scheduler 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 also ensures that the outputs above are compiled and linked before invoking the dynamic scheduler.

Owing to the incompatibility between Wood's conceptualization and the one in this thesis, it was necessary then to come up with a new implementation for the dynamic scheduler. This is presented in Appendix C. As can be seen, the dynamic scheduler was implemented as a procedure. It has one input and one output. Input consists of the names of the non-time-critical operators. Output consists of a task containing procedure calls to the non-time-critical operator DRIVERS in package TL. Visibility to TL, as well as to the package PRIORITY_DEFINITIONS, is provided via the Ada with and use clauses. The output of the dynamic scheduler is referred to as a dynamic schedule in this thesis. Fig. 12 is an example schedule.

One item to take note of in the sample schedule is the statement

```plaintext
pragma priority (DYNAMIC_SCHEDULE_PRIORITY);
```
with TL; use TL;
with PRIORITY_DEFINITIONS; use PRIORITY_DEFINITIONS
package DS_PACKAGE is
  task DYNAMIC_SCHEDULE is
    pragma priority (DYNAMIC_SCHEDULE_PRIORITY);
  end DYNAMIC_SCHEDULE;
end DS_PACKAGE;

package body DS_PACKAGE is
  task body DYNAMIC_SCHEDULE is
  begin
    loop
      non_critical_op1_DRIVER;
      non_critical_op2_DRIVER;
      null;
    end loop;
  end DYNAMIC_SCHEDULE;
end DS_PACKAGE;

Fig. 12 Sample Dynamic Schedule

A pragma is a directive to the Ada compiler. The priority pragma specifies the priority of a task or main program. It takes as a single argument either an integer or an expression which evaluates to an integer. A task with a higher priority will execute before a task with a lower priority.

The static schedule is a task which contains procedure calls to the time-critical operator DRIVERS in TL. It too has a priority pragma statement. The pragma statements in the dynamic and static schedules are set so that the static schedule always has the higher priority. This is to ensure that when the static schedule needs the processor it will get it. The use of Ada's tasking facilities to control
operator execution is a feature which originally appeared in Wood's implementation. Though as mentioned above, the dynamic scheduler implementations are different, this was a feature that was found to be especially useful and thus was incorporated here.

D. MODIFICATIONS TO THE STATIC SCHEDULER

Integration into the execution support system of the static scheduler Kilic [Ref. 15] presented few problems for this author. Test cases run through the scheduler during integration identified only a single minor error in the actual functioning of the scheduler; this was quickly corrected. Subsequent modifications during integration consisted entirely of changes of an interface nature. These are described below.

Fig. 13 is a sample static schedule. When Kilic was implementing the static scheduler, the work of integrating the debugger into the execution support system was still in its infancy. Static schedules created by the static scheduler consequently contained no reference to the debugger. This was changed. Added to static schedules now are the statements
with GLOBAL_DECLARATIONS; use GLOBAL_DECLARATIONS;
with DS_DEBUG_PKG; use DS_DEBUG_PKG;

which establish visibility to the debugger and the statement

DS_DEBUG.RUNTIME_MET_FAILURE (VARSTRING.VSTR("CRITICAL_OP3"),
CURRENT_TIME, CRITICAL_OP3_STOP_TIME3, PERIOD);

which executes an entry call to Runtime_MET_Failure. The actual functioning of the debugger is described in the next section and will not be covered here.

One item that does need to be covered here though is the use of the Ada delay statement in the static schedule. It was mentioned in the previous section that priority pragma statements are utilized in the dynamic and static schedules to ensure that when the static schedule needs the processor it gets it. The question now is "How does the dynamic schedule get the processor then?" The answer is via the Ada delay statement.

The delay statement is used in Ada to suspend execution of a task or main body. When a time-critical operator completes execution a check is made to determine if the operator's scheduled stop time is greater than the current time. If it is, then the static schedule delays for the
time remaining till its stop time. Once it does this, it effectively gives up the processor to the dynamic schedule. When the length of the delay is over, the static schedule is given the processor back.

```vhdl
with GLOBAL_DECLARATIONS; use GLOBAL_DECLARATIONS;
with DS_DEBUG_PKG; use DS_DEBUG_PKG;
with TL; use TL;
with DS_PACKAGE; use DS_PACKAGE;
with PRIORITY_DEFINITIONS; use PRIORITY_DEFINITIONS;
with CALENDAR; use CALENDAR;
with TEXT_IO; use TEXT_IO;
procedure STATIC_SCHEDULE is
  CRITICAL_01_TIMING_ERROR : exception;
  CRITICAL_02_TIMING_ERROR : exception;
  CRITICAL_03_TIMING_ERROR : exception;
  task SCHEDULE is
    pragma priority (STATIC_SCHEDULE_PRIORITY);
end SCHEDULE;

task body SCHEDULE is

begin
  loop
    begin
      CRITICAL_01_DRIVER;
      SLACK_TIME := START_OF_PERIOD + CRITICAL_01_STOP_TIME1 - CLOCK;
      if SLACK_TIME >= 0.0 then
        delay (SLACK_TIME);
      else
        raise CRITICAL_01_TIMING_ERROR;
      end if;
      delay (START_OF_PERIOD + CRITICAL_01_STOP_TIME1 - CLOCK);
    end;

    CRITICAL_02_DRIVER;

    delay (START_OF_PERIOD + CRITICAL_02_STOP_TIME2 - CLOCK);

    CRITICAL_03_DRIVER;

    START_OF_PERIOD := START_OF_PERIOD + PERIOD;
    delay (START_OF_PERIOD - clock);
end loop;
```

Fig. 13 Sample Static Schedule
exception
  when CRITICAL_OP3_TIMING_ERROR =>
      PUT_LINE("timing error from operator CRITICAL_OP3");
      CURRENT_TIME:= clock - START_OF_PERIOD;
      DS_DEBUG.RUNTIME_MET_FAILURE (VARSTRING.VSTR("CRITICAL_OP3"),
         CURRENT_TIME, CRITICAL_OP3_STOP_TIME3, PERIOD);
      START_OF_PERIOD := clock;
  when CRITICAL_OP2_TIMING_ERROR =>
      .
      .
  when CRITICAL_OP1_TIMING_ERROR =>
      .
      .
end;
end loop
end SCHEDULE;

begin
    null;
end STATIC_SCHEDULE;

Fig. 13 Continued

E. THE DEBUGGER

1. Background

There are three kinds of exceptions defined in this thesis: Ada exceptions not declared in the prototype description, PSDL exceptions defined by the prototype designer to include Ada exceptions declared in the prototype description and Ada exceptions declared by the PSDL runtime system. Ada exceptions not declared in the prototype description include the predefined exceptions declared in package STANDARD. The raising of this type of exception during prototype execution results in abnormal termination.
PSDL exceptions defined by the prototype designer were discussed in section B of this chapter. Handling of these exceptions is by the exception handler which the designer must provide in package SB. Ada exceptions declared by the PSDL runtime system, of which there are three, are CAPS specific exceptions which can occur during the execution of any prototype. They are: BUFFER_UNDERFLOW, BUFFER_OVERFLOW and OPERATOR_TIMING_ERROR, where OPERATOR is the name that was given to the operator in the prototype description. Exception BUFFER_UNDERFLOW is raised when an operator attempts to read from an empty data stream. Exception BUFFER_OVERFLOW is raised when an operator attempts to write to a dataflow stream which currently contains a value. Exception OPERATOR_TIMING_ERROR is raised when an operator exceeds its MET. Ada exceptions declared by the PSDL runtime system are handled by the debugger which is described below.

As part of the earlier research effort on CAPS, in addition to developing the translator, the static scheduler and the dynamic scheduler, work was expended on the development of a runtime debugger for the execution support.
system. This work was done primarily by Wood and is described in her thesis [Ref. 16]. Wood calls her debugger the Dynamic Scheduler debugging system and implements it as a task consisting of six entries. They are: Runtime_MET_Failure, Buffer_Underflow, Buffer_Overflow, Exception_Error, Non_Time_Critical_Operators_Done and Static_Schedule_Done. The first four entries were provided to serve as handlers for the Ada exceptions declared by the PSDL runtime system. The latter two entries were provided to allow for graceful termination of the prototype.

During the course of integrating Wood's debugging system into the execution support system, entry Exception_Error was determined to be unnecessary and was eliminated. Entries Non_Time_Critical_Operators_Done and Static_Schedule_Done were determined to be predicated on the false assumption that all the operators execute only once and likewise were eliminated. Thus the modified version of the Dynamic Scheduler debugging system called simply the debugger in this thesis, consists of only three entry statements. The code is included as Appendix D.
2. Sample Session

The purpose of any debugger is usually a twofold one-to identify errors and if possible, to correct them. The purpose of this debugger is no different. In the case of Buffer_Underflow and Buffer_Overflow, the function performed is one of identification—a message is printed to a file, the prototype terminates. In the case of Runtime_MET_Failure though, the function performed is not only one of identification, but also one of correction.

Reference is made to Fig. 14 which is repeated from the previous section. When a critical operator exceeds its MET during prototype execution, a TIMING_ERROR exception is raised. At this point, control passes to the exception handler, an error message is printed to the screen and an entry call is made to Runtime_MET_Failure. Prototype execution has in effect been suspended. Reference is now made to Appendix D. Upon accepting the entry, the first action the debugger takes is to determine if the operator which exceeded its MET is in the Operators_Overrun list. The Operators_Overrun list is a list of records where each record contains the name of an operator that has exceeded its MET and a count of the number of times it has done so.
with GLOBAL_DECLARATIONS; use GLOBAL_DECLARATIONS;
with DS_DEBUG_PKG; use DS_DEBUG_PKG;
with TL; use TL;
with DS_PACKAGE; use DS_PACKAGE;
with PRIORITY_DEFINITIONS; use PRIORITY_DEFINITIONS;
with CALENDAR; use CALENDAR;
with TEXT_10; use TEXT_10;
procedure STATIC_SCHEDULE is
  CRITICAL_OP3_TIMING_ERROR : exception;
  CRITICAL_OP2_TIMING_ERROR : exception;
  CRITICAL_OP1_TIMING_ERROR : exception;
  task SCHEDULE is
    pragma priority (STATIC_SCHEDULE_PRIORITY);
end SCHEDULE;

  task body SCHEDULE is
    ...
    begin
      loop
        begin
          CRITICAL_OP1_DRIVER;
          SLACK_TIME := START_OF_PERIOD + CRITICAL_OP1_STOP_TIME1 - CLOCK;
          if SLACK_TIME >= 0.0 then
            delay (SLACK_TIME);
          else
            raise CRITICAL_OP1_TIMING_ERROR;
          end if;
          delay (START_OF_PERIOD + CRITICAL_OP1_STOP_TIME1 - CLOCK);
        
        CRITICAL_OP2_DRIVER;
          ...
          ...
        delay (START_OF_PERIOD + CRITICAL_OP2_STOP_TIME2 - CLOCK);
        
        CRITICAL_OP3_DRIVER;
          ...
          ...
        START_OF_PERIOD := START_OF_PERIOD + PERIOD;
        delay (START_OF_PERIOD - clock);
        exception
          when CRITICAL_OP3_TIMING_ERROR =>
            PUT_LINE("timing error from operator CRITICAL_OP3");
            CURRENT_TIME := clock - START_OF_PERIOD;
            DS_DEBUG.RUNTIME_MET_FAILURE (VARSTRING.VSTR("CRITICAL_OP3"),
              CURRENT_TIME, CRITICAL_OP3_STOP_TIME3, PERIOD);
            START_OF_PERIOD := clock;
            when CRITICAL_OP2_TIMING_ERROR =>
              ...
        end exception;
        ...
      end loop;
    end SCHEDULE;
end STATIC_SCHEDULE;

Fig. 14 Sample Static Schedule
If the operator appears in the list, a check is then made to determine if the operator has exceeded its authorized number of executions. If it has, an error message is printed and execution of the prototype terminates abnormally. If it hasn’t exceeded its authorized number of executions, the Executed_count field in the record is then incremented by one, adjustments are made to the operator’s MET, a record of the adjustments is printed to a file called Information and execution of the prototype is resumed.

Assume for the moment that this is the first time this operator has exceeded its MET. Consequently, it will not be found in the Operators_Overrun list. At this point the user is queried as to which of two possible courses of action he would like to take. This is illustrated in Fig. 15. If the user opts for A, prototype execution terminates. If the user opts for B, a record for the operator is then inserted.
in the Operators Overrun list, an error message is printed to file Information, adjustments are made to the operator’s MET and a record of this is printed to Information. Fig. 16

Timing error from operator CRITICAL.OP2

Execution of the prototype has been suspended because an operator exceeded its maximum execution time. The operator causing the error is:

CRITICAL.OP2

Do you want to
A. Terminate execution of the prototype?
B. Adjust the execution time of the operator and continue execution of the prototype?

Type the letter preceding the option you want.

Fig. 15 Sample User Query

EXECUTION HAS BEEN SUSPENDED OR HAS TERMINATED ABNORMALLY

The following operator did not complete execution before its maximum execution time was expired. The operator which caused the error is:

CRITICAL.OP2

The maximum execution time for operator CRITICAL.OP2 was increased by 3.0200000E+00.
The new ending time for operator CRITICAL.OP2 from the start of the period within the static schedule is 1.5020000E+01.

EXECUTION HAS BEEN SUSPENDED OR HAS TERMINATED ABNORMALLY

The following operator did not complete execution before its maximum execution time was expired. The operator which caused the error is:

CRITICAL.OP3

The maximum execution time for operator CRITICAL.OP3 was increased by 1.7400000E+00.
The new ending time for operator CRITICAL.OP3 from the start of the period within the static schedule is 1.5740000E+01.

Fig. 16 Contents of File Information
is an illustration of the possible contents of file Information. Note, the printing of the adjustment times to file Information represents a modification to the earlier implementation.

F. ADDITIONAL RUNTIME SUPPORT

In the description of what constitutes a prototype at the beginning of this chapter, reference was made to a group of packages called simply, ADDITIONAL RUNTIME SUPPORT. With the exception of PRIORITY_DEFINITIONS, which was discussed already, these packages provide various types and operations which are essential to the prototype's execution. In the interest of clarity and completeness, the code for these packages is provided as Appendix E.
CHAPTER V. PROTOTYPE EXAMPLE

This chapter contains a prototype description and the corresponding prototype actually generated by the execution support system of CAPS. The example is based on the command and control system for a HAWK Air Defense Guided Missile System Battery. The mission of the HAWK system is to detect and destroy low to medium altitude, hostile, airborne targets. The HAWK missile system detects, identifies, tracks, engages, and destroys airborne targets. The system performs these functions against extremely low-altitude and medium-altitude targets while retaining a high degree of mobility for rapid deployment with the Army in the field.

Fig. 17 is a very small section of the overall dataflow diagram for the command and control system. This particular diagram is modeling the part of the system that provides to the Tactical Control Officer (TCO) a continuously updated tactical summary. Track data are read, sorted by category and then written to a tactical display located by the TCO. C1 is a composite operator. READ_TRACKS, SORT_TRACKS and WRITE_TRACKS are atomic operators which represent the
realization of C1.

Fig. 17 Dataflow Diagram

Fig. 18 is the prototype description corresponding to the dataflow diagram in Fig. 17. As can be seen, operators READ TRACKS, SORT TRACKS and WRITE TRACKS are time-critical. More specifically though, they're periodic. Every 20
seconds they will execute. A and B are data streams. By virtue of the fact A does not appear in a "TRIGGERED BY ALL" constraint of SORT_TRACKS, it is a sampled stream and will be implemented as such. Likewise for B. By virtue of the fact it does not appear in a "TRIGGERED BY ALL" constraint of WRITE_TRACKS, it too is a sampled stream. STR_LIST is a user defined type.

```
OPERATOR C1
SPECIFICATION
    MAXIMUM EXECUTION TIME 14 sec
END
IMPLEMENTATION GRAPH
A.READ_TRACKS->SORT_TRACKS
B.SORT_TRACKS->WRITE_TRACKS
DATA STREAM A : STR_LIST,
    B : STR_LIST
CONTROL CONSTRAINTS
    OPERATOR READ_TRACKS
        PERIOD 20 sec
    OPERATOR SORT_TRACKS
        PERIOD 20 sec
    OPERATOR WRITE_TRACKS
        PERIOD 20 sec
    DESCRIPTION ( text )
END
OPERATOR READ_TRACKS
```

Fig. 18 Prototype Description
Fig. 18 Continued

Fig. 19 is package SB which contains the reusable components extracted from the software base. Procedure READ_TRACKS basically just keeps reading track data from file INPUT_FILE until it encounters the END_OF_FILE condition. SORT_TRACKS makes use of the reusable component QUICKSORT to sort the tracks by category. And finally, WRITE_TRACKS writes the data to the screen.
Fig. 20 is the output of the translator after it reads and processes the prototype description given in Fig. 18.

The statement

```perl
package DS_A is new SAMPLED_BUFFER(STRLIST);
```

is the instantiation for data stream A. The statement

```perl
package DS_B is new SAMPLED_BUFFER(STRLIST);
```

with TEXT_IO; use TEXT_IO;
with ORDERING;
with PSDL_EXCEPTIONS; use PSDL_EXCEPTIONS;
package SB is
  type STRING_REC is record
    NAME : STRING (1..80);
  end record;
  type STR_LIST is array (INTEGER range 1..10) of STRINGREC;
  procedure READTRACKS (DATA : in out STRLIST);
  procedure SORTTRACKS (L in STR_LIST; M : in out STR_LIST);
  procedure WRITETRACKS (L in STRLIST);
end SB;

package body SB is
  function "<" (X : in STRING_REC;
      Y : in STRING_REC) return Boolean is
    begin
      return X.NAME (1) < Y.NAME (1);
    end "<";
  package INT_INC is new ORDERING.QUICKSORT (STRING_REC, INTEGER, "<");
  use INT_INC;
  procedure READTRACKS (DATA : in out STR_LIST) is
    INPUT_FILE : FILE_TYPE;
    I, LAST : INTEGER;
    TEMP_STRING : STRING (1..80);
    begin
      OPEN (INPUT_FILE, IN_FILE, "INPUT_FILE");
      for J in 1..10 loop
        for K in 1..80 loop
          DATA (J).NAME (K) := ' ';
        end loop;
      end loop;
      I := 1;
      while (not (END_OF_FILE (INPUT_FILE))) loop
        GET_LINE (INPUT_FILE, TEMP_STRING, LAST);
        DATA (I).NAME (1..LAST) := TEMP_STRING (1..LAST);
      end loop;
  end procedure READTRACKS;
end SB;
```

Fig. 19 Reusable Components
I := I + 1;
end loop;
CLOSE (INPUT_FILE);
end READ_TRACKS;

procedure SORT_TRACKS (L : in STR_LIST; M : in out STR_LIST) is
begin
M := L;
QUICKSORT (INT_INC.List (M)); -- Sort in increasing sequence
end SORT_TRACKS;

procedure WRITE_TRACKS (L : in STR_LIST) is
begin
NEW_LINE;
for I in 1..10 loop
PUT_LINE (L (I).NAME (1..80));
NEW_LINE;
end loop;
end WRITE_TRACKS;
end SB;

Fig. 19 Continued

package TL is
procedure READ_TRACKSDRIVER;
procedure SORTTRACKSDRIVER;
procedure WRITETRACKSDRIVER;
end TL;

with SB; use SB;
with PSDL_SYSTEM; use PSDL_SYSTEM;
package body TL is

package ClSPEC is

package DSA is new SAMPLEDBUFFER(STRLIST);
package DSB is new SAMPLED_BUFFER(STR_LIST);

end ClSPEC;

procedure READ_TRACKSDRIVER is
A: STR_LIST;
begin
READ_TRACKS(A);
Cl_SPEC.DSA.BUFFER.WRITE(A);
end READ_TRACKS_DRIVER;

procedure SORT_TRACKSDRIVER is
A: STR_LIST;
B: STR_LIST;
begin
Cl_SPEC.DSA.BUFFER.READ(A);
SORT_TRACKS(A, B);
end SORT_TRACKS_DRIVER;
end ClSPEC;

procedure WRITE_TRACKSDRIVER is

Fig. 20 DRIVER Procedures
C1_SPEC.DS_B.BUFFER.WRITE(B);
end SORT_TRACKS_DRIVER;

procedure WRITE_TRACKS_DRIVER is
  B: STR_LIST;
begin
  C1_SPEC.DS_B.BUFFER.READ(B);
  WRITE_TRACKS(B);
end WRITE_TRACKS_DRIVER;
end TL;

is the instantiation for data stream B. The procedures READ_TRACKS_DRIVER, etc. are the drivers for the reusable components given in Fig. 19.

Fig. 21 is the output of the dynamic scheduler. It consists of a package DS_PACKAGE containing a task DYNAMIC_SCHEDULE. DYNAMIC_SCHEDULE contains procedure calls to the non-time-critical operator DRIVERS in package TL. In the example in this chapter, there are no non-time-critical operators, consequently, DYNAMIC_SCHEDULE contains no procedure calls. Another important item to take note of is the statement

pragma priority (DYNAMIC_SCHEDULE_PRIORITY);

This statement is needed to ensure that if the static schedule and the dynamic schedule are both vying for the processor, the static schedule will get it. The value of this priority must be less than the value of the priority in
the static schedule.

```vhdl
with TL; use TL;
with PRIORITY_DEFINITIONS; use PRIORITY_DEFINITIONS;
package DS_PACKAGE is
task DYNAMIC_SCHEDULE is
  pragma priority (DYNAMIC_SCHEDULE_PRIORITY);
end DYNAMIC_SCHEDULE;
end DS_PACKAGE;

package body DS_PACKAGE is
task body DYNAMIC_SCHEDULE is
begin
  loop
    null;
  end loop;
end DYNAMIC_SCHEDULE;
end DS_PACKAGE;
```

Fig. 21 Dynamic Schedule

Fig. 22 is the output of the static scheduler. It consists of procedure calls to the time-critical operator DRIVERS in package TL. Note that the value of the priority here has to be greater than the value of the priority in the dynamic schedule.

```vhdl
with GLOBAL DECLARATIONS; use GLOBAL DECLARATIONS;
with DS_DEBUG_PKG; use DS_DEBUG_PKG;
with TL; use TL;
with DS_PACKAGE; use DS_PACKAGE;
with PRIORITY_DEFINITIONS; use PRIORITY_DEFINITIONS;
with CALENDAR; use CALENDAR;
with TEXT IO; use TEXT IO;
procedure STATIC_SCHEDULE is
  WRITE_TRACKS_TIMING_ERROR : exception;
  READ_TRACKS_TIMING_ERROR : exception;
  task body SCHEDULE is
  pragma priority (STATIC_SCHEDULE_PRIORITY);
end SCHEDULE;

  begin
    null;
  end loop;
end DYNAMIC_SCHEDULE;
end DS_PACKAGE;
```

Fig. 22 Static Schedule
begin
loop
begin
READ_TRACKS_DRIVER;
SLACK_TIME := START_OF_PERIOD + READ_TRACKS_STOP_TIME1 - CLOCK;
if SLACK_TIME >= 0.0 then
  delay (SLACK_TIME);
else
  raise READTRACKS_TIMING_ERROR;
end if;
delay (START_OF_PERIOD + READ_TRACKS_STOP_TIME1 - CLOCK);
SORT_TRACKS_DRIVER;
SLACK_TIME := START_OF_PERIOD + SORT_TRACKS_STOP_TIME2 - CLOCK;
if SLACK_TIME >= 0.0 then
  delay (SLACK_TIME);
else
  raise SORTTRACKS_TIMING_ERROR;
end if;
delay (START_OF_PERIOD + SORT_TRACKS_STOP_TIME2 - CLOCK);
WRITE_TRACKS_DRIVER;
SLACK_TIME := START_OF_PERIOD + WRITE_TRACKS_STOP_TIME3 - CLOCK
if SLACK_TIME >= 0.0
  delay (SLACK_TIME);
else
  raise WRITE_TRACKS_TIMING_ERROR;
end if;
START_OF_PERIOD := START_OF_PERIOD + PERIOD;
delay (START_OF_PERIOD - clock);
end
exception
when WRITE_TRACKS_TIMING_ERROR =>
  PUT_LINE("timing error from operator WRITE_TRACKS");
  CURRENT_TIME := clock - START_OF_PERIOD;
  DS_DEBUG.RUNTIME_MET_FAILURE (VARSTRING.VSTR("WRITE TRACKS"),
                              CURRENT_TIME, WRITE_TRACKS_STOP_TIME3, PERIOD);
  START_OF_PERIOD := clock;
when SORT_TRACKS_TIMING_ERROR =>
  PUT_LINE("timing error from operator SORT_TRACKS");
  CURRENT_TIME := clock - START_OF_PERIOD;
  DS_DEBUG.RUNTIME_MET_FAILURE (VARSTRING.VSTR("SORT TRACKS"),
                              CURRENT_TIME, SORT_TRACKS_STOP_TIME2, PERIOD);
  START_OF_PERIOD := clock;
when READ_TRACKS_TIMING_ERROR =>
  PUT_LINE("timing error from operator READ_TRACKS");
  CURRENT_TIME := clock - START_OF_PERIOD;
  DS_DEBUG.RUNTIME_MET_FAILURE (VARSTRING.VSTR("READ TRACKS"),
                              CURRENT_TIME, READ_TRACKS_STOP_TIME1, PERIOD);
  START_OF_PERIOD := clock;
end
VARSTRING.VSTR("READ_TRACKS"),
    CURRENT_TIME, READ_TRACKS_STOP_TIME1, PERIOD);
START_OF_PERIOD := clock;
    end;
end loop;
end SCHEDULE;

begin
null;
end STATIC_SCHEDULE;

Fig. 22 Continued

Once the translator, static scheduler and dynamic scheduler have completed, the user interface takes their outputs as well as the packages: PSDL_SYSTEM, PSDL_EXCEPTIONS, SB, Global_Declarations, DS_Debug_PKG, TIMERS, VSTRINGS, List_Single_Unbounded_Unmanaged and PRIORITY_DEFINITIONS and compiles and links them, thus forming the executable prototype. This prototype is then run. Fig. 23 is a sample input for the prototype. Fig. 24

<table>
<thead>
<tr>
<th>Identified</th>
<th>Azimuth</th>
<th>Range (km)</th>
<th>Heading</th>
<th>Speed (mph)</th>
</tr>
</thead>
<tbody>
<tr>
<td>hostile</td>
<td>320</td>
<td>425</td>
<td>140</td>
<td>1500</td>
</tr>
<tr>
<td>unknown</td>
<td>40</td>
<td>250</td>
<td>240</td>
<td>750</td>
</tr>
<tr>
<td>friendly</td>
<td>80</td>
<td>75</td>
<td>360</td>
<td>1000</td>
</tr>
<tr>
<td>friendly</td>
<td>100</td>
<td>20</td>
<td>270</td>
<td>1000</td>
</tr>
<tr>
<td>friendly</td>
<td>250</td>
<td>130</td>
<td>90</td>
<td>500</td>
</tr>
<tr>
<td>hostile</td>
<td>330</td>
<td>450</td>
<td>150</td>
<td>1500</td>
</tr>
<tr>
<td>hostile</td>
<td>300</td>
<td>475</td>
<td>120</td>
<td>1500</td>
</tr>
<tr>
<td>unknown</td>
<td>45</td>
<td>750</td>
<td>180</td>
<td>1000</td>
</tr>
<tr>
<td>unknown</td>
<td>30</td>
<td>650</td>
<td>210</td>
<td>1000</td>
</tr>
</tbody>
</table>

Fig. 23 Sample Track Data

is the corresponding output.
<table>
<thead>
<tr>
<th>Identified</th>
<th>Azimuth</th>
<th>Range (km)</th>
<th>Heading</th>
<th>Speed (mph)</th>
</tr>
</thead>
<tbody>
<tr>
<td>friendly</td>
<td>100</td>
<td>20</td>
<td>270</td>
<td>1000</td>
</tr>
<tr>
<td>friendly</td>
<td>80</td>
<td>75</td>
<td>360</td>
<td>1000</td>
</tr>
<tr>
<td>friendly</td>
<td>250</td>
<td>130</td>
<td>90</td>
<td>500</td>
</tr>
<tr>
<td>hostile</td>
<td>300</td>
<td>475</td>
<td>120</td>
<td>1500</td>
</tr>
<tr>
<td>hostile</td>
<td>320</td>
<td>425</td>
<td>140</td>
<td>1500</td>
</tr>
<tr>
<td>hostile</td>
<td>330</td>
<td>450</td>
<td>150</td>
<td>1500</td>
</tr>
<tr>
<td>unknown</td>
<td>30</td>
<td>650</td>
<td>210</td>
<td>1000</td>
</tr>
<tr>
<td>unknown</td>
<td>45</td>
<td>750</td>
<td>180</td>
<td>1000</td>
</tr>
<tr>
<td>unknown</td>
<td>40</td>
<td>250</td>
<td>240</td>
<td>750</td>
</tr>
</tbody>
</table>

Fig. 24 Corresponding Output
VI. CONCLUSIONS AND RECOMMENDATIONS

A. CONCLUSIONS

Integration is sort of a nebulous term. Integration as used throughout this thesis has meant: 1. make modifications to the component to make it work if it doesn't already and 2. then make modifications to the component to make it function as part of the execution support system. In the case of the translator, we saw the need for both types of modifications. In the case of the static scheduler and of the debugger, just the latter type of modification was required. Finally, in the case of the dynamic scheduler we had a situation where modifications would have been so extensive, that a new implementation was deemed more efficient.

The purpose of this thesis was to provide a tool that enables the execution of prototypes written in PSDL. That has been accomplished. With the addition of the debugger the designer also has the capability of changing an operator's MET dynamically and of recording all operator time adjustments.
B. RECOMMENDATIONS

There are significant opportunities for future research in the area of the execution support system. One possibility is the expansion of the debugger. As it is currently implemented, entries Buffer_Underflow and Buffer_Overflow provide only an identification type function. If a call is made to either of them, the only action taken is to print an error message to file Information and then terminate the prototype's execution. Perhaps the capability of correcting these error dynamically, as it is for Runtime_MET_Failure, can be built in to the debugger.

Another possibility for future research is in the way that the user interface interacts with the execution support system. The way that CAPS is implemented now, when the designer finishes construction of the prototype description and wishes to invoke the execution support system, he or she does so by selecting an Execute option off of a menu. What happens then is the user interface submits the prototype description to the translator for creation of package TL. Once the translator finishes, the user interface next submits the prototype description to the static scheduler.
for processing. After the static scheduler finishes, the user interface then invokes the dynamic scheduler. After the dynamic scheduler finishes the user interface next compiles and links the necessary files, creating the executable prototype which is immediately run. At no point in this process though does the user interface query the designer as to whether or not they would like to continue, even if something should go awry along the way like the static scheduler not being able to derive a schedule. The user interface needs to be changed so that the Execute option is broken down into perhaps three options like: Translate, Compile and Run.
APPENDIX A. DATA STREAM IMPLEMENTATION

with PRIORITY_DEFINITIONS; use PRIORITY_DEFINITIONS;
with vstrings, TIMERS;
package PSDL_SYSTEM is

package PSDL_STRINGS is new vstrings(50);
type PSDL_TIMER is new TIMERS.TIMER;

BUFFER_UNDERFLOW, BUFFER_OVERFLOW: exception;

generic
type ELEMENT_TYPE is private;
package SAMPLED_BUFFER is
task BUFFER is
  pragma PRIORITY(BUFFER_PRIORITY);
  entry CHECK(NEW_DATA: out BOOLEAN);
  entry WRITE(VALUE: in ELEMENT_TYPE);
  entry READ(VALUE: out ELEMENT_TYPE);
end BUFFER;
  function NEW_DATA return BOOLEAN;
end SAMPLED_BUFFER;

generic
type ELEMENT_TYPE is private;
package FIFO_BUFFER is
task BUFFER is
  pragma PRIORITY(BUFFER_PRIORITY);
  entry CHECK(NEW_DATA: out BOOLEAN);
  entry WRITE(VALUE: in ELEMENT_TYPE);
  entry READ(VALUE: out ELEMENT_TYPE);
end BUFFER;
  function NEW_DATA return BOOLEAN;
end FIFO_BUFFER;

generic
type ELEMENT_TYPE is private;
INITIAL_VALUE: ELEMENT_TYPE;
package STATE_VARIABLE is
task BUFFER is
  pragma PRIORITY(BUFFER_PRIORITY);
  entry CHECK(NEW_DATA: out BOOLEAN);
  entry READ(VALUE: out ELEMENT_TYPE);
  entry WRITE(VALUE: in ELEMENT_TYPE);
end BUFFER;
  function NEW_DATA return BOOLEAN;
end STATE_VARIABLE;
end PSDL_SYSTEM;

package body PSDL_SYSTEM is

package body SAMPLED_BUFFER is

task body BUFFER is
    BUFFER: ELEMENT_TYPE;
    FRESH: BOOLEAN := false;

begin
    loop
        select
            accept CHECK(NEWDATA: out BOOLEAN) do
                NEWDATA := FRESH;
                end CHECK;
        or
            accept READ(VALUE: out ELEMENT_TYPE) do
                if FRESH then
                    VALUE := BUFFER; FRESH := false;
                else raise BUFFERUNDERFLOW; end if;
            end READ;
        or
            accept WRITE(VALUE: in ELEMENTTYPE) do
                BUFFER := VALUE; FRESH := true;
            end WRITE;
        end select;
    end loop;
end BUFFER;

function NEWDATA return BOOLEAN is
begin
    BUFFER.CHECK(RESULT);
    return (RESULT);
end NEWDATA;

end SAMPLED_BUFFER;

package body FIFO_BUFFER is

task body BUFFER is
    BUFFER: ELEMENT_TYPE;
    FRESH: BOOLEAN := false;

begin
    loop
        select
            accept CHECK(NEWDATA: out BOOLEAN) do
                NEWDATA := FRESH;
                end CHECK;
        or
            accept READ(VALUE: out ELEMENT_TYPE) do
                VALUE := BUFFER; FRESH := false;
                end READ;
        end select;
    end loop;
end BUFFER;

end FIFO_BUFFER;
end READ;

or
accept WRITE(VALUE: in ELEMENT_TYPE) do
  if not FRESH then
    BUFFER := VALUE; FRESH := true;
  else raise BUFFER_OVERFLOW; end if;
end WRITE;
end select;
end loop;
end BUFFER;

function NEW_DATA return BOOLEAN is
  RESULT: BOOLEAN;
begin
  BUFFER.CHECK(RESULT);
  return RESULT;
end NEW_DATA;
end FIFO_BUFFER;

package body STATE_VARIABLE is
  task body BUFFER is
    BUFFER: ELEMENT_TYPE := INITIAL_VALUE;
    FRESH: BOOLEAN := true;
begin
  loop
    select
      accept CHECK(NEW_DATA: out BOOLEAN) do
        NEW_DATA := FRESH;
      end CHECK;
    or
      accept READ(VALUE: out ELEMENT_TYPE) do
        VALUE := BUFFER; FRESH := false;
      end READ;
    or
      accept WRITE(VALUE: in ELEMENT_TYPE) do
        BUFFER := VALUE; FRESH := true;
      end WRITE;
    end select;
  end loop;
end BUFFER;

function NEW_DATA return BOOLEAN is
  RESULT: BOOLEAN;
begin
  BUFFER.CHECK(RESULT);
  return RESULT;
end NEW_DATA;
end STATE_VARIABLE;
end PSDL_SYSTEM;
APPENDIX B. EXCEPTION HANDLING EXAMPLE

This appendix contains a sample prototype which performs no particular function. The purpose of this example is to illustrate the approach to exception handling adopted in this thesis.

A. DATA FLOW DIAGRAM
B. PROTOTYPE DESCRIPTION

OPERATOR C1

SPECIFICATION

MAXIMUM EXECUTION TIME 14 sec

END

IMPLEMENTATION GRAPH

A. OP1->OP2
   EXCP.OP1->EXCEPTION_HANDLER
B. OP2->OP3

DATA STREAM A : INTEGER,
   EXCP : PSDL_EXCEPTION,
   B : INTEGER

CONTROL CONSTRAINTS

OPERATOR OP1
   PERIOD 20 sec
   EXCEPTION OUT_OF_RANGE IF A > 100

OPERATOR OP2
   PERIOD 20 sec

OPERATOR OP3
   PERIOD 20 sec

OPERATOR EXCEPTION_HANDLER
   TRIGGERED BY ALL EXCP IF EXCP = OUT_OF_RANGE
   PERIOD 20 sec

END

OPERATOR OP1

SPECIFICATION

OUTPUT A : INTEGER
OUTPUT EXCP : PSDL_EXCEPTION
MAXIMUM EXECUTION TIME 10 sec
EXCEPTIONS NEGATIVE_NUMBER

END

IMPLEMENTATION ADA OP1
OPERATOR OP2

SPECIFICATION

   INPUT A : INTEGER
   OUTPUT B : INTEGER
   MAXIMUM EXECUTION TIME 2 sec

END

IMPLEMENTATION ADA OP2
END

OPERATOR OP3

SPECIFICATION

   INPUT B : INTEGER
   MAXIMUM EXECUTION TIME 2 sec

END

IMPLEMENTATION ADA OP3
END

OPERATOR EXCEPTION_HANDLER

SPECIFICATION

   INPUT EXCP : PSDL_EXCEPTION
   MAXIMUM EXECUTION TIME 2 sec

END

IMPLEMENTATION ADA EXCEPTION_HANDLER
END

C. PARTIAL ADA IMPLEMENTATION OF PROTOTYPE DESCRIPTION

package TL is
   procedure OP1_DRIVER;
   procedure OP2_DRIVER;
   procedure OP3_DRIVER;
   procedure EXCEPTION_HANDLER_DRIVER;
end TL;

with PSDL_SYSTEM; use PSDL_SYSTEM;

82
with PSDL_EXCEPTIONS; use PSDL_EXCEPTIONS;
with SB; use SB;
package body TL is

package Cl_SPEC is

package DS_A is new SAMPLED_BUFFER(INTEGER);
package DS_B is new SAMPLED_BUFFER(INTEGER);
package DS_EXCP is new FIFO_BUFFER(PSDL_EXCEPTION);
end Cl_SPEC;

procedure OP1_DRIVER is
A : INTEGER;
begin
  OP1(A);
  if A > 100 then
    Cl_SPEC.DSEXCP.BUFFER.WRITE(INDEX(CHARACTER_STRING.
    VSTR("OUT_OF_RANGE")));
  else
    Cl_SPEC.DSA.BUFFER.WRITE(A);
  end if;

exception
  when NEGATIVE_NUMBER =>
    Cl_SPEC.DSEXCP.BUFFER.WRITE(INDEX(CHARACTER_STRING.
    VSTR("NEGATIVE_NUMBER")));
end OP1_DRIVER;

procedure OP2_DRIVER is
A : INTEGER;
B : INTEGER;
begin
  Cl_SPEC.DSA.BUFFER.READ(A);
  OP2(A, B);
  Cl_SPEC.DSB.BUFFER.WRITE(B);
end OP2_DRIVER;

procedure OP3_DRIVER is
B : INTEGER;
begin
  Cl_SPEC.DSB.BUFFER.READ(B);
  OP3(B);
end OP3_DRIVER;

procedure EXCEPTION_HANDLER_DRIVER is
EXCP : PSDL_EXCEPTION;

begin
  if Cl_SPEC.DS_EXCP.NEW_DATA then
    Cl_SPEC.DS_EXCP.BUFFER.READ(EXCP);
    if EXCP = (INDEX(CHARACTER_STRING.VSTR("OUT_OF_RANGE"))) then
      EXCEPTION_HANDLER(EXCP);
      end if;
    end if;
  end EXCP_HANDLER_DRIVER;
begin
  DECLARE_EXCEPTION(CHARACTER_STRING.VSTR("OUT_OF_RANGE"));
  DECLARE_EXCEPTION(CHARACTER_STRING.VSTR("NEGATIVE_NUMBER")
  );
end TL;

with TEXT_IO; use TEXT_IO;
with PSDL_EXCEPTIONS; use PSDL_EXCEPTIONS;
package SB is
  NEGATIVE_NUMBER : exception;
  procedure OPl (DATA : in out INTEGER);
  procedure OP2 (L : in INTEGER; M : in out INTEGER);
  procedure OP3 (L : in INTEGER);
  procedure EXCEPTION_HANDLER (L : in PSDL_EXCEPTION);
end SB;
package body SB is
  procedure OPl (DATA : in out INTEGER) is
    begin
      if (DATA < 0) then
        raise NEGATIVE_NUMBER;
      end if;
    end OPl;
  procedure OP2 (L : in INTEGER; M : in out INTEGER) is
    begin
     ..
    end OP2;
  procedure OP3 (L : in INTEGER) is
    begin
      ..
    end OP3;
procedure EXCEPTION_HANDLER (L : in PSDL_EXCEPTION) is
begin
  newline;
  if (CHARACTER_STRING.STR (NAME (L))) = "OUT_OF_RANGE"
    then
    put_line("Out of Range");
  elsif (CHARACTER_STRING.STR (NAME (L))) = "NEGATIVE_NUMBER" then
    put_line("Negative Number");
  end if;
  newline;
end EXCEPTION_HANDLER;
end SB;

D. SUPPORTING ADA PACKAGE

with VSTRINGS;
with List_Single_Unbounded_Unmanaged;
package PSDL_EXCEPTIONS is
  -- Definition of Abstract Data Type
  type PSDL_EXCEPTION is private;
  package CHARACTER_STRING is new VSTRINGS (80);
  procedure DECLARE_EXCEPTION (E : in CHARACTER_STRING.VSTRING);
  function NAME (E in PSDL_EXCEPTION) return CHARACTER_STRING.VSTRING;
  function INDEX (E in CHARACTER_STRING.VSTRING) return PSDL_EXCEPTION;
private
  type PSDL_EXCEPTION is range 1..1000;
end PSDL_EXCEPTIONS;

package body PSDL_EXCEPTIONS is

  package EXCEPTION_LIST is new
      List_Single_Unbounded_Unmanaged (Item => CHARACTER_STRING.VSTRING);

  DECLARED_EXCEPTIONS : EXCEPTION_LIST.List;

  function IS_EQUAL (LEFT : in CHARACTER_STRING.VSTRING;
                     RIGHT : in CHARACTER_STRING.VSTRING)
    return BOOLEAN is
begin
  return (CHARACTER_STRING."=" (LEFT, RIGHT));
end IS_EQUAL;

procedure DECLARE_EXCEPTION (E : in CHARACTER_STRING.VSTRING) is
begin
  EXCEPTIONLIST.Construct (E, DECLARED_EXCEPTIONS);
end DECLARE_EXCEPTION;

function NAME (E : in PSDL_EXCEPTION.T) return CHARACTER_STRING.VSTRING is
  TEMPORARYLIST : EXCEPTIONLIST.List := DECLARED_EXCEPTIONS;
begin
  for INDEX in l..(E-1) loop
    TEMPORARYLIST := EXCEPTIONLIST.Tail_Of (TEMPORARYLIST);
  end loop;
  return (EXCEPTIONLIST.Head_Of (TEMPORARYLIST));
end NAME;

function INDEX (E : in CHARACTER_STRING.VSTRING) return PSDL_EXCEPTION is
  TEMPORARYLIST : EXCEPTIONLIST.List := DECLARED_EXCEPTIONS;
  I : PSDL_EXCEPTION := 1;
begin
  while not EXCEPTIONLIST.Is_Null (TEMPORARYLIST) loop
    if IS_EQUAL (E, EXCEPTIONLIST.Head_Of (TEMPORARYLIST)) then
      return I;
    else
      TEMPORARYLIST := EXCEPTIONLIST.Tail_Of (TEMPORARYLIST);
      I := I + 1;
    end if;
  end loop;
end INDEX;

end PSDL_EXCEPTIONS;
APPENDIX C. DYNAMIC SCHEDULER IMPLEMENTATION

with TEXT_IO; use TEXT_IO;

procedure DYNAMIC_SCHEDULER is
  NON_CRITS : FILE_TYPE;
  DSV3      : FILE_TYPE;
  IN_STRING : STRING(1..72);
  LAST      : NATURAL;

begin
  OPEN(NON_CRITS, IN_FILE,
  "/n/suns2/work/caps/prototypes/non_crits");
  CREATE(DSV3, OUT_FILE,
  "/n/suns2/work/caps/prototypes/ds.a");
  PUT_LINE(DSV3, "with TL; use TL;");
  PUT_LINE(DSV3, "with PRIORITY_DEFINITIONS; use
  
PRIORITY_DEFINITIONS;");
  PUT_LINE(DSV3, "package DS_PACKAGE is");
  PUT_LINE(DSV3, "task DYNAMIC_SCHEDULE is");
  -- system defined priority for dynamic schedule
  PUT_LINE(DSV3, "pragma priority
  (DYNAMIC_SCHEDULE_PRIORITY);");
  PUT_LINE(DSV3, "end DYNAMIC_SCHEDULE;");
  PUT_LINE(DSV3, "end DS_PACKAGE;");
  NEW_LINE(DSV3);
  PUT_LINE(DSV3, "package body DS_PACKAGE is");
  PUT_LINE(DSV3, "task body DYNAMIC_SCHEDULE is");
  PUT_LINE(DSV3, "begin");
  PUT_LINE(DSV3, "delay (1.0);");
  PUT_LINE(DSV3, "loop");
  while not END_OF_FILE(NON_CRITS) loop
    begin
      GET_LINE(NON_CRITS, IN_STRING, LAST);
      PUT(DSV3, " ");
      for INDEX in 1..LAST loop
        PUT(DSV3, IN_STRING(INDEX));
      end loop;
      PUT_LINE(DSV3, ";");
    end loop;
    PUT_LINE(DSV3, " null");
    PUT_LINE(DSV3, " end loop");
    PUT_LINE(DSV3, " end DYNAMIC_SCHEDULE");
    PUT_LINE(DSV3, "end DS_PACKAGE");
  end DYNAMIC_SCHEDULER;
This package contains information that must be declared globally for the Dynamic Scheduler procedure. The library unit VSTRINGS is a generic string package. It contains the data type VSTRING and procedures to manipulate these strings. Since VSTRINGS is generic, it must be instantiated. The name of the instantiation must then be made visible. The instantiation is declared in a package so that type compatibility will be ensured among variables that the Dynamic Scheduler will share among tasks.

with VSTRINGS;

package Global_Declarations is

    package VARSTRING is new VSTRINGS (160);
    use VARSTRING;
end Global_Declarations;

with Global_Declarations;
use Global_Declarations;

with TEXT_10, CALENDAR;
use TEXT_10, CALENDAR;

package DS_Debug_PKG is

    --* The specification for task DS_Debug contains three entry statements. --*
    --* These statements identify errors that may be encountered when the operators execute. --* 

    task DS_Debug is
        pragma priority (10);
        entry Runtime_MET_Failure (Exception_Operator : VARSTRING.VSTRING;
            Current_Time : in duration;
            Operator_Ending_Time : in out duration;
            Period : in out duration);

        ** 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 operator must start execution. **

        entry Buffer_Underflow; -- input queue empty
        entry Buffer_Overflow; -- output queue full
package body DS_Debug_PKG is

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.equat (Current.Operator, Name) then
      Found := TRUE;
    else
      Found := FALSE;
    end if;
  else
    -- traverse list
    while Current.Next /= null

end Find;

package body DS_Debug_PKG is

task body DS_Debug is

  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;

package FLOATIO is new TEXTIO.FLOAT 10 (FLOAT);

Exception_Operator : VARSTRING.VSTRING; -- operator causing error
Information : FILE_TYPE; -- file containing error information
Error_Exists : 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.*--

-- end DS_Debug;
-- end DS_Debug_PKG;
loop
  if VARSTRING.equat (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.equat (Current.Operator, Name) then
    Found := TRUE;
  else
    Found := FALSE;
  end if;
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 executed*--
--*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.");
procedure Print_Buffer_Underflow_Message (Information : FILE_TYPE) is
begin
  PUT (Information, "There was an attempt to read a data buffer ");
  NEW_LINE (Information);
  PUT (Information, "that 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_Runtime_MEF_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_MEF_Failure_Message;

procedure Print_Adjustments_Made_Message (Information : FILE_TYPE; Exception_Operator : VARSTRING.VSTRING; Difference : in duration; Operator_Ending_Time : in duration; Period : in duration) is
begin
  PUT (Information, "The maximum execution time for operator ");
  VARSTRING.PUT_LINE (Information, Exception_Operator);
  PUT (Information, "was increased by ");
  FLOATIO.PUT (Information, FLOAT(Difference), 3, 7);
  PUT_LINE (Information, ".");
  PUT (Information, "The new ending time for operator ");
  VARSTRING.PUT_LINE (Information, Exception_Operator);
  PUT (Information, "from the start of the period within the ");
  PUT_LINE (Information, "static schedule ");
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 (Information, "frequently, has a maximum ");
  PUT (Information, "execution time that is not long enough. ");
  PUT (Information, "Execution has been");
  PUT (Information, "terminated because processing time is being ");
  PUT (Information, "wasted by having");
  PUT (Information, "to handle the error each time the operator ");
  PUT (Information, "executes. The operator is ");
  VARSTRING.PUTLINE (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_UserChoice (Exception_Operator : VARSTRING.VSTRING;
Choice : in out CHARACTER) is
  Count : INTEGER;

begin
  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';
    PrintToo_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 Runtime_MET_Failure
        (Exception_Operator : VARSTRING; VSTRING;
         Current_Time : in duration;
         Operator_Ending_Time : in out duration;
         Period : in out duration) 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
                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 - Operator_Ending_Time;
        -- calculate time over MET
        Operator_Ending_Time := Operator_Ending_Time + Difference;
        -- reset time to the new operator ending time
Period := Period + Difference;
Print_Adjustments_Made_Message (Information,
                Exception_Operator, Difference,
                Operator_Ending_Time, Period);
    end runtime_FAIL;
end select;

if Error_Exists then
    close (Information); 
    exit;
end if;

end loop;
end DS_Debug;
end DS_Debug_PKG;
A. Timers

with Calendar;
package TIMERS is

    subtype MICROSEC is NATURAL;
    type TIMER is private;

    procedure RESET (NAME : in out TIMER);
    procedure START (NAME : in out TIMER);
    procedure STOP (NAME : in out TIMER);
    function READ (NAME : in TIMER) return MICROSEC;
    function LT (PT : TIMER; T : MICROSEC) return BOOLEAN;
    function LT (T : MICROSEC; PT : TIMER) return BOOLEAN;
    function GT (PT : TIMER; T : MICROSEC) return BOOLEAN;
    function GT (T : MICROSEC; PT : TIMER) return BOOLEAN;
    function LTE (PT : TIMER; T : MICROSEC) return BOOLEAN;
    function LTE (T : MICROSEC; PT : TIMER) return BOOLEAN;
    function GTE (PT : TIMER; T : MICROSEC) return BOOLEAN;
    function GTE (T : MICROSEC; PT : TIMER) return BOOLEAN;
    function EQU (PT : TIMER; T : MICROSEC) return BOOLEAN;
    function EQU (T : MICROSEC; PT : TIMER) return BOOLEAN;

private
    type STATE is (INITIAL, RUNNING, STOPPED);
    type TIMER is record
        START_TIME, STOP_TIME : CALENDAR.TIME;
        ELAPSED_TIME : DURATION;
        PRESENT_STATE : STATE;
    end record;
end TIMERS;

with Calendar;
use Calendar;
package body TIMERS is

    CONVERSION_FACTOR : constant DURATION := 1000000.0; -- converts elapsed
                      -- time to microsec's
function READ (NAME : in TIMER) return MICROSEC is
begin
    case NAME.PRESENT_STATE is
    when RUNNING => return MICROSEC(CLOCK - NAME.START_TIME + NAME.ELAPSED_TIME);
    when others => return MICROSEC(NAME.ELAPSED_TIME);
    end case;
end READ;

procedure RESET (NAME : in out TIMER) is
begin
    case NAME.PRESENT_STATE is
    when STOPPED => NAME.ELAPSED_TIME := 0.0;
    NAME.PRESENT_STATE := INITIAL;
    when others => null;
    end case;
end RESET;

procedure START (NAME : in out TIMER) is
begin
    case NAME.PRESENT_STATE is
    when RUNNING => null;
    when others => NAME.START_TIME := CALENDAR.CLOCK;
    NAME.PRESENT_STATE := RUNNING;
    end case;
end START;

procedure STOP (NAME : in out TIMER) is
begin
    case NAME.PRESENT_STATE is
    when RUNNING => NAME.ELAPSED_TIME := CALENDAR.CLOCK - NAME.START_TIME;
    end case;
end STOP;
+ NAME.ELAPSED_TIME;
NAME.PRESENT_STATE := STOPPED;

when others => null;
end case;
end STOP;

function LT (PT : TIMER; T : MICROSEC) return BOOLEAN is
begin
return FLOAT (PT.ELAPSED_TIME * CONVERSION_FACTOR) < FLOAT(T);
end LT;

function LT (T : MICROSEC; PT : TIMER) return BOOLEAN is
begin
return LT (T,PT) = GTE (PT,T);
end LT;

function GT (PT : TIMER; T : MICROSEC) return BOOLEAN is
begin
return FLOAT (PT.ELAPSED_TIME * CONVERSION_FACTOR) > FLOAT(T);
end GT;

function GT (T : MICROSEC; PT : TIMER) return BOOLEAN is
begin
return GT (T,PT) = LTE (PT,T);
end GT;

function EQU (PT : TIMER; T : MICROSEC) return BOOLEAN is
begin
return not LT(PT,T) and not GT (PT,T);
end EQU;

function EQU (T : MICROSEC; PT : TIMER) return BOOLEAN is
begin
return not LT (PT,T) and not GT (PT,T);
end EQU;

function LTE (PT : TIMER; T : MICROSEC) return BOOLEAN is
function LTE (T : MICROSEC; PT : TIMER) return BOOLEAN is
begin
  return not GT (PT,T);
end LTE;

function GTE (PT : TIMER; T : MICROSEC) return BOOLEAN is
begin
  return not LT (PT,T);
end GTE;

function GTE (T : MICROSEC; PT : TIMER) return BOOLEAN is
begin
  return not LT (PT,T);
end GTE;
end TIMERS;
B. VSTRINGS

-- UNIT: generic package spec of VSTRINGS
--
-- FILES: vstring_spec.a in pubtictib
-- related file is vstring_body.a in pubtictib
--
-- PURPOSE: An implementation of the abstract data type "variable-length string."
--
-- DESCRIPTION: This package provides a private type VSTRING. VSTRING objects are "strings" that have a length between zero and LAST, where LAST is the generic parameter supplied in the package instantiation.
--
-- In addition to the type VSTRING, a subtype and two constants are declared. The subtype STRINDEX is an index to a VSTRING. The STRINDEX constant FIRST is an index to the first character of the string, and the VSTRING constant NUL is a VSTRING of length zero. NUL is the default initial value of a VSTRING.
--
-- The following sets of functions, procedures, and operators are provided as operations on the type VSTRING:
--
-- ATTRIBUTE FUNCTIONS: LEN, MAX, STR, CHAR
-- The attribute functions return the characteristics of a VSTRING.
--
-- COMPARISON OPERATORS: "="", "/="", "<"", ">"", "<="", ">=""
-- The comparison operators are the same as for the predefined type STRING.
--
-- INPUT/OUTPUT PROCEDURES: GET, GET_LINE, PUT, PUT_LINE
-- The input/output procedures are similar to those for the predefined type STRING, with the following exceptions:
--
-- - GET has an optional parameter LENGTH, which indicates the number of characters to get (default is LAST).
--
-- - GET_LINE does not have a parameter to return the length of the string (the LEN function should be used instead).
--
-- EXTRACTION FUNCTIONS: SLICE, SUBSTR, DELETE
-- The SLICE function returns the slice of a VSTRING between two indices (equivalent to STR(X)(A .. B)).
--
-- SUBSTR returns a substring of a VSTRING taken from a given index and extending a given length.
The DELETE function returns the VSTRING which results from removing the slice between two indices.

EDITING FUNCTIONS: INSERT, APPEND, REPLACE
The editing functions return the VSTRING which results from inserting, appending, or replacing at a given index with a VSTRING, STRING, or CHARACTER. The index must be in the current range of the VSTRING; i.e., zero cannot be used.

CONCATENATION OPERATOR: "&"
The concatenation operator is the same as for the type STRING. It should be used instead of APPEND when the APPEND would always be after the last character.

POSITION FUNCTIONS: INDEX, RINDEX
The position functions return an index to the Nth occurrence of a VSTRING, STRING, or CHARACTER from the front or back of a VSTRING. Zero is returned if the search is not successful.

CONVERSION FUNCTIONS AND OPERATOR: VSTR, CONVERT, "++"
VSTR converts a STRING or a CHARACTER to a VSTRING.

CONVERT is a generic function which can be instantiated to convert from any given variable-length string to another, provided the FROM type has a function equivalent to STR defined for it, and that the TO type has a function equivalent to VSTR defined for it. This provides a means for converting between VSTRINGs declared in separate instantiations of VSTRINGs. When instantiating CONVERT for VSTRINGs, the STR and VSTR functions are implicitly defined, provided that they have been made visible (by a use clause).

Note: CONVERT is NOT implicitly associated with the type VSTRING declared in this package (since it would not be a derivable function (see RM 3.4(11))).

Caution: CONVERT cannot be instantiated directly with the names VSTR and STR, since the name of the subprogram being declared would hide the generic parameters with the same names (see RM 8.3(16)). CONVERT can be instantiated with the operator "++", and any instantiation of CONVERT can subsequently be renamed VSTR or STR.

Example: Given two VSTRINGs instantiations X and Y:
function "++" is new X.CONVERT(X.VSTRING, Y.VSTRING);
function "++" is new X.CONVERT(Y.VSTRING, X.VSTRING);
(Y.CONVERT could have been used in place of X.CONVERT)
function VSTR(A : X.VSTRING) return Y.VSTRING renames "++";
function VSTR(A : Y.VSTRING) return X.VSTRING renames "+";

"+" is equivalent to VSTR. It is supplied as a short-hand notation for the function. The "+" operator cannot immediately follow the "&" operator; use ... & (+ ...) instead.

DISCUSSION:

This package implements the type "variable-length string" (vstring) using generics. The alternative approaches are to use a discriminant record in which the discriminant controls the length of a STRING inside the record, or a record containing an access type which points to a string, which can be deallocated and reallocated when necessary.

Advantages of this package:

* The other approaches force the vstring to be a limited private type. Thus, their vstrings cannot appear on the left side of the assignment operator; i.e., their vstrings cannot be given initial values or values by direct assignment. This package uses a private type; therefore, these things can be done.

* The other approach stores the vstring in a string whose length is determined dynamically. This package uses a fixed length string. This difference might be reflected in faster and more consistent execution times (this has NOT been verified).

Disadvantages of this package:

* Different instantiations must be used to declare vstrings with different maximum lengths (this may be desirable, since CONSTRAINT_ERROR will be raised if the maximum is exceeded).

* A second declaration is required to give the type declared by the instantiation a name other than "VSTRING."

* The storage required for a vstring is determined by the generic parameter LAST and not the actual length of its contents. Thus, each object is allocated the maximum amount of storage, regardless of its actual size.

MISCELLANEOUS:

Constraint checking is done explicitly in the code; thus, it cannot be suppressed. On the other hand, constraint checking is not lost if pragma suppress is supplied to the compilation (-S option) (The robustness of the explicit constraint checking has NOT been determined).

Compiling with the optimizer (-O option) may significantly reduce the size (and possibly execution time) of the resulting executable.

Compiling an instantiation of VSTRINGS is roughly equivalent to recompiling VSTRINGS. Since this takes a significant amount of time, and the instantiation does not depend on any other library units,
it is STRONGLY recommended that the instantiation be compiled
separately, and thus done only ONCE.

-- USAGE: with VSTRINGS;
-- package package_name is new VSTRINGS(maximum_length);
-- ................................................................. --
with TEXT_IO; use TEXT_IO;
generic
LAST : NATURAL;
package VSTRINGS is

subtype STRINDEX is NATURAL;
FIRST : constant STRINDEX := STRINDEX'FIRST + 1;
type VSTRING is private;
NUL : constant VSTRING;

-- Attributes of a VSTRING

function LEN(FROM : VSTRING) return STRINDEX;
function MAX(FROM : VSTRING) return STRINDEX;
function STR(FROM : VSTRING) return STRING;
function CHAR(FROM: VSTRING; POSITION : STRINDEX := FIRST)
return CHARACTER;

-- Comparisons

function "<" (LEFT: VSTRING; RIGHT: VSTRING) return BOOLEAN;
function ">" (LEFT: VSTRING; RIGHT: VSTRING) return BOOLEAN;
function "<=" (LEFT: VSTRING; RIGHT: VSTRING) return BOOLEAN;
function ">=\" (LEFT: VSTRING; RIGHT: VSTRING) return BOOLEAN;
function equal (LEFT: VSTRING; RIGHT: VSTRING) return BOOLEAN;
f.. ion notequal (LEFT: VSTRING; RIGHT: VSTRING) return BOOLEAN;

-- Input/Output

procedure PUT(FILE : in FILE_TYPE; ITEM : in VSTRING);
procedure PUT(ITEM : in VSTRING);

procedure PUT_LINE(FILE : in FILE_TYPE; ITEM : in VSTRING);
procedure PUT_LINE(ITEM : in VSTRING);

procedure GET(FILE : in FILE_TYPE; ITEM : out VSTRING;
LENGTH : in STRINDEX := LAST);
procedure GET(ITEM : out VSTRING; LENGTH : in STRINDEX := LAST);

procedure GET_LINE(FILE : in FILE_TYPE; ITEM : in out VSTRING);
procedure GET_LINE(ITEM : in out VSTRING);

-- Extraction
function SLICE(FROM: VSTRING; FRONT, BACK : STRINDEX) return VSTRING;
function SUBSTR(FROM: VSTRING; START, LENGTH: STRINDEX) return VSTRING;
function DELETE(FROM: VSTRING; FRONT, BACK : STRINDEX) return VSTRING;

-- Editing

function INSERT(TARGET: VSTRING; ITEM: VSTRING;
    POSITION: STRINDEX := FIRST) return VSTRING;
function INSERT(TARGET: VSTRING; ITEM: STRING;
    POSITION: STRINDEX := FIRST) return VSTRING;
function INSERT(TARGET: VSTRING; ITEM: CHARACTER;
    POSITION: STRINDEX := FIRST) return VSTRING;

function APPEND(TARGET: VSTRING; ITEM: VSTRING;
    POSITION: STRINDEX) return VSTRING;
function APPEND(TARGET: VSTRING; ITEM: STRING;
    POSITION: STRINDEX) return VSTRING;
function APPEND(TARGET: VSTRING; ITEM: CHARACTER;
    POSITION: STRINDEX) return VSTRING;

function REPLACE(TARGET: VSTRING; ITEM: VSTRING;
    POSITION: STRINDEX := FIRST) return VSTRING;
function REPLACE(TARGET: VSTRING; ITEM: STRING;
    POSITION: STRINDEX := FIRST) return VSTRING;
function REPLACE(TARGET: VSTRING; ITEM: CHARACTER;
    POSITION: STRINDEX := FIRST) return VSTRING;

-- Concatenation

function "&" (LEFT: VSTRING; RIGHT : VSTRING) return VSTRING;
function "&" (LEFT: VSTRING; RIGHT : STRING) return VSTRING;
function "&" (LEFT: VSTRING; RIGHT : CHARACTER) return VSTRING;
function "&" (LEFT: STRING; RIGHT : VSTRING) return VSTRING;
function "&" (LEFT: CHARACTER; RIGHT : VSTRING) return VSTRING;

-- Determine the position of a substring

function INDEX(WHOLE: VSTRING; PART: VSTRING; OCCURRENCE : NATURAL := 1)
    return STRINDEX;
function INDEX(WHOLE: VSTRING; PART: STRING; OCCURRENCE : NATURAL := 1)
    return STRINDEX;
function INDEX(WHOLE: VSTRING; PART: CHARACTER; OCCURRENCE : NATURAL := 1)
    return STRINDEX;

function RINDEX(WHOLE: VSTRING; PART: VSTRING; OCCURRENCE : NATURAL := 1)
    return STRINDEX;
function RINDEX(WHOLE : VSTRING; PART : STRING; OCCURRENCE : NATURAL := 1)
return STRINDEX;

function RINDEX(WHOLE : VSTRING; PART : CHARACTER; OCCURRENCE : NATURAL := 1)
return STRINDEX;

-- Conversion from other associated types

function VSTR(FROM : STRING) return VSTRING;
function VSTR(FROM : CHARACTER) return VSTRING;
function "=" (FROM : STRING) return VSTRING;
function "=" (FROM : CHARACTER) return VSTRING;

generic
  type FROM is private;
  type TO is private;
  with function STR(X FROM) return STRING is <>;
  with function VSTR(Y : STRING) return TO is <>;
  function CONVERT(X FROM) return TO;

private
  type VSTRING is
    record
      LEN : STRINDEX := STRINDEX'FIRST;
      VALUE : STRING(FIRST .. LAST) := (others => ASCII.NUL);
    end record;

  NUL : constant VSTRING := (STRINDEX'FIRST, (others => ASCII.NUL));
end VSTRINGS;

--
-- DISTRIBUTION AND COPYRIGHT:
--
-- This software is released to the Public Domain (note:
-- software released to the Public Domain is not subject
-- to copyright protection).
-- Restrictions on use or distribution: NONE
--
-- DISCLAIMER:
--
-- This software and its documentation are provided "AS IS" and
-- without any expressed or implied warranties whatsoever.
-- No warranties as to performance, merchantability, or fitness
-- for a particular purpose exist.
--
-- Because of the diversity of conditions and hardware under
-- which this software may be used, no warranty of fitness for
-- a particular purpose is offered. The user is advised to
-- test the software thoroughly before relying on it. The user
-- must assume the entire risk and liability of using this
-- software.
-- In no event shall any person or organization of people be
-- held responsible for any direct, indirect, consequential
-- or inconsequential damages or lost profits.

-- UNIT: generic package body of VSTRINGS
--
-- FILES: vstring_body.a in publiclib
-- related file is vstring_spec.a in publiclib
--
-- PURPOSE: An implementation of the abstract data type "variable-length
-- string."
--
-- DESCRIPTION: This package provides a private type VSTRING. VSTRING objects
-- are "strings" that have a length between zero and LAST, where
-- LAST is the generic parameter supplied in the package
-- instantiation.
--
-- In addition to the type VSTRING, a subtype and two constants
-- are declared. The subtype STRINDEX is an index to a VSTRING,
-- The STRINDEX constant FIRST is an index to the first character
-- of the string, and the VSTRING constant NUL is a VSTRING of
-- length zero. NUL is the default initial value of a VSTRING.
--
-- The following sets of functions, procedures, and operators
-- are provided as operations on the type VSTRING:
--
-- ATTRIBUTE FUNCTIONS: LEN, MAX, STR, CHAR
-- The attribute functions return the characteristics of
-- a VSTRING.
--
-- COMPARISON OPERATORS: "=", "/=", ",", ":", ":=", ":=";
-- The comparison operators are the same as for the predefined
-- type STRING.
--
-- INPUT/OUTPUT PROCEDURES: GET, GET_LINE, PUT, PUT_LINE
-- The input/output procedures are similar to those for the
-- predefined type STRING, with the following exceptions:
--
-- - GET has an optional parameter LENGTH, which indicates
-- the number of characters to get (default is LAST).
--
-- - GET_LINE does not have a parameter to return the length
-- of the string (the LEN function should be used instead).
--
-- EXTRACTION FUNCTIONS: SLICE, SUBSTR, DELETE
-- The SLICE function returns the slice of a VSTRING between
-- two indices (equivalent to STR(X)(A .. B)).
SUBSTR returns a substring of a VSTRING taken from a given index and extending a given length.

The DELETE function returns the VSTRING which results from removing the slice between two indices.

EDITING FUNCTIONS: INSERT, APPEND, REPLACE

The editing functions return the VSTRING which results from inserting, appending, or replacing at a given index with a VSTRING, STRING, or CHARACTER. The index must be in the current range of the VSTRING; i.e., zero cannot be used.

CONCATENATION OPERATOR: "&"

The concatenation operator is the same as for the type STRING. It should be used instead of APPEND when the APPEND would always be after the last character.

POSITION FUNCTIONS: INDEX, RINDEX

The position functions return an index to the Nth occurrence of a VSTRING, STRING, or CHARACTER from the front or back of a VSTRING. Zero is returned if the search is not successful.

CONVERSION FUNCTIONS AND OPERATOR: VSTR, CONVERT, "+

VSTR converts a STRING or a CHARACTER to a VSTRING.

CONVERT is a generic function which can be instantiated to convert from any given variable-length string to another, provided the FROM type has a function equivalent to STR defined for it, and that the TO type has a function equivalent to VSTR defined for it. This provides a means for converting between VSTRINGS declared in separate instantiations of VSTRINGS. When instantiating CONVERT for VSTRINGS, the STR and VSTR functions are implicitly defined, provided that they have been made visible (by a use clause).

Note: CONVERT is NOT implicitly associated with the type VSTRING declared in this package (since it would not be a derivable function (see RM 3.4(11))).

Caution: CONVERT cannot be instantiated directly with the names VSTR and STR, since the name of the subprogram being declared would hide the generic parameters with the same names (see RM 8.3(16)). CONVERT can be instantiated with the operator "+", and any instantiation of CONVERT can subsequently be renamed VSTR or STR.

Example: Given two VSTRINGS instantiations X and Y:

function "+" is new X.CONVERT(X.VSTRING, Y.VSTRING);
function "+" is new X.CONVERT(Y.VSTRING, X.VSTRING);
(Y.CONVERT could have been used in place of X.CONVERT)

function VSTR(A : X.VSTRING) return Y.VSTRING renames "+";
function VSTR(A : Y.VSTRING) return X.VSTRING renames "+";

"+" is equivalent to VSTR. It is supplied as a short-hand notation for the function. The "+" operator cannot immediately follow the "&" operator; use ... & (+ ...) instead.

DISCUSSION:

This package implements the type "variable-length string" (vstring) using generics. The alternative approaches are to use a discriminant record in which the discriminant controls the length of a STRING inside the record, or a record containing an access type which points to a string, which can be deallocated and reallocated when necessary.

Advantages of this package:

* The other approaches force the vstring to be a limited private type. Thus, their vstrings cannot appear on the left side of the assignment operator; i.e., their vstrings cannot be given initial values or values by direct assignment. This package uses a private type; therefore, these things can be done.

* The other approach stores the vstring in a string whose length is determined dynamically. This package uses a fixed length string. This difference might be reflected in faster and more consistent execution times (this has NOT been verified).

Disadvantages of this package:

* Different instantiations must be used to declare vstrings with different maximum lengths (this may be desirable, since CONSTRAINT_ERROR will be raised if the maximum is exceeded).

* A second declaration is required to give the type declared by the instantiation a name other than "VSTRING."

* The storage required for a vstring is determined by the generic parameter LAST and not the actual length of its contents. Thus, each object is allocated the maximum amount of storage, regardless of its actual size.

MISCELLANEOUS:

Constraint checking is done explicitly in the code; thus, it cannot be suppressed. On the other hand, constraint checking is not lost if pragma suppress is supplied to the compilation (-S option) (The robustness of the explicit constraint checking has NOT been determined).

Compiling with the optimizer (-O option) may significantly reduce the size (and possibly execution time) of the resulting executable.
Compiling an instantiation of VSTRINGS is roughly equivalent to
recompiling VSTRINGS. Since this takes a significant amount of time,
and the instantiation does not depend on any other library units,
it is STRONGLY recommended that the instantiation be compiled
separately, and thus done only ONCE.

-- USAGE: with VSTRINGS;
-- package package_name is new VSTRINGS(maximum_length);
-- ........................................................................
package body VSTRINGS is

-- local declarations

FILL_CHAR : constant CHARACTER := ASCII.NUL;

procedure FORMAT(THESTRING in out VSTRING; OLDLEN : in STRINDEX := LAST) is
-- fill the string with FILL_CHAR to null out old values

begin -- FORMAT (Local Procedure)
THESTRING.VALUE(THESTRING.LEN + 1 .. OLDLEN) :=
(others => FILL_CHAR);
end FORMAT;

-- bodies of visible operations

function LEN(FROM : VSTRING) return STRINDEX is

begin -- LEN
return(FROM.LEN);
end LEN;

function MAX(FROM : VSTRING) return STRINDEX is
begin -- MAX
return(LAST);
end MAX;

function STR(FROM : VSTRING) return STRING is

begin -- STR
return(FROM.VALUE(FIRST .. FROM.LEN));
end STR;

function CHAR(FROM : VSTRING; POSITION : STRINDEX := FIRST) return CHARACTER is

begin -- CHAR
if POSITION not in FIRST .. FROM.LEN
then raise CONSTRAINT_ERROR;
end if;
end CHAR;
function "<" (LEFT: VSTRING; RIGHT: VSTRING) return BOOLEAN is
begin -- "<"
  return(LEFT.VALUE < RIGHT.VALUE);
end "<";

function ")" (LEFT: VSTRING; RIGHT: VSTRING) return BOOLEAN is
begin -- "")"
  return(LEFT.VALUE > RIGHT.VALUE);
end ")";

function "<=" (LEFT: VSTRING; RIGHT: VSTRING) return BOOLEAN is
begin -- "<="
  return(LEFT.VALUE <= RIGHT.VALUE);
end "<=";

function ">=" (LEFT: VSTRING; RIGHT: VSTRING) return BOOLEAN is
begin -- ">="
  return(LEFT.VALUE >= RIGHT.VALUE);
end ">=";

function equal (LEFT: VSTRING; RIGHT: VSTRING) return BOOLEAN is
begin -- equal
  return(LEFT.VALUE = RIGHT.VALUE);
end equal;

function notequal (LEFT: VSTRING; RIGHT: VSTRING) return BOOLEAN is
begin -- notequal
  return(LEFT.VALUE /= RIGHT.VALUE);
end notequal;

procedure PUT(FILE : in FILE_TYPE; ITEM : in VSTRING) is
begin -- PUT
  PUT(FILE, ITEM.VALUE(FIRST .. ITEM.LEN));
end PUT;

procedure Put(ITEM : in VSTRING) is
begin -- PUT
  PUT(ITEM.VALUE(FIRST .. ITEM.LEN));
end Put;

procedure PUT_LINE(FILE : in FILE_TYPE; ITEM : in VSTRING) is
begin -- PUT_LINE
    PUT_LINE(FILE, ITEM.VALUE(FIRST .. ITEM.LEN));
end PUT_LINE;

procedure PUT_LINE(ITEM : in VSTRING) is
begin -- PUT_LINE
    PUT_LINE(ITEM.VALUE(FIRST .. ITEM.LEN));
end PUT_LINE;

procedure GET(FILE : in FILETYPE; ITEM : out VSTRING;
    LENGTH : in STRINDEX := LAST) is
begin -- GET
    if LENGTH not in FIRST .. LAST
    then raise CONSTRAINTERROR;
    end if;

    ITEM := NUL;
    for INDEX in FIRST .. LENGTH loop
        GET(FILE, ITEM.VALUE(INDEX));
        ITEM.LEN := INDEX;
    end loop;
end GET;

procedure GET(ITEM : out VSTRING; LENGTH in STRINDEX := LAST) is
begin -- GET
    if LENGTH not in FIRST .. LAST
    then raise CONSTRAINTERROR;
    end if;

    ITEM := NUL;
    for INDEX in FIRST .. LENGTH loop
        GET(ITEM.VALUE(INDEX));
        ITEM.LEN := INDEX;
    end loop;
end GET;

procedure GET_LINE(FILE : in FILETYPE; ITEM in out VSTRING) is
    OLDLEN : constant STRINDEX := ITEM.LEN;

begin -- GET_LINE
    GET_LINE(FILE, ITEM.VALUE, ITEM.LEN);
    FORMAT(ITEM, OLDLEN);
end GET_LINE;

procedure GET_LINE(ITEM : in out VSTRING) is
    OLDLEN : constant STRINDEX := ITEM.LEN;
begin -- GET_LINE
    GET_LINE(ITEM.VALUE, ITEM.LEN);
    FORMAT(ITEM, OLDLEN);
end GET_LINE;

function SLICE(FROM : VSTRING; FRONT, BACK : STRINDEX) return VSTRING is
begin -- SLICE
    if ((FRONT not in FIRST . . FROM.LEN) or else
        (BACK not in FIRST . . FROM.LEN)) and then FRONT <= BACK
    then raise CONSTRAINTERROR;
    end if;

    return(Vstr(FROM.VALUE(FRONT .. BACK)));
end SLICE;

function SUBSTR(FROM : VSTRING; START, LENGTH : STRINDEX) return VSTRING is
begin -- SUBSTR
    if (START not in FIRST . . FROM.LEN) or else
        ((START + LENGTH - 1 not in FIRST . . FROM.LEN)
         and then (LENGTH > 0))
    then raise CONSTRAINTERROR;
    end if;

    return(Vstr(FROM.VALUE(START .. START + LENGTH -1)));
end SUBSTR;

function DELETE(FROM : VSTRING; FRONT, BACK : STRINDEX) return VSTRING is
    TEMP : VSTRING := FROM;

    begin -- DELETE
        if ((FRONT not in FIRST .. FROM.LEN) or else
            (BACK not in FIRST .. FROM.LEN)) and then FRONT <= BACK
        then raise CONSTRAINTERROR;
        end if;

        if FRONT > BACK then return(FROM); end if;
        TEMP.LEN := FROM.LEN - (BACK - FRONT) - 1;

        TEMP.VALUE(FRONT .. TEMP.LEN) := FROM.VALUE(BACK + 1 .. FROM.LEN);
        FORMAT(TEMP, FROM.LEN);
        return(TEMP);
    end DELETE;

function INSERT(TARGET: VSTRING; ITEM: VSTRING;
POSITION : STRINDEX := FIRST() return VSTRING is

TEMP : VSTRING;
begin -- INSERT
  if POSITION not in FIRST .. TARGET.LEN
    then raise CONSTRAINT_ERROR;
  end if;
  if TARGET.LEN + ITEM.LEN > LAST
    then raise CONSTRAINT_ERROR;
    else TEMP.LEN := TARGET.LEN + ITEM.LEN;
  end if;
  TEMP.VALUE(FIRST .. POSITION - 1) := TARGET.VALUE(FIRST .. POSITION - 1);
  TEMP.VALUE(POSITION .. (POSITION + ITEM.LEN - 1)) :=
    ITEM.VALUE(FIRST .. ITEM.LEN);
  TEMP.VALUE((POSITION + ITEM.LEN) .. TEMP.LEN) :=
    TARGET.VALUE(POSITION .. TARGET.LEN);
  return(TEMP);
end INSERT;

function INSERT(TARGET: VSTRING; ITEM: STRING;
  POSITION : STRINDEX := FIRST) return VSTRING is
begin -- INSERT
  return INSERT(TARGET, VSTR(ITEM), POSITION);
end INSERT;

function INSERT(TARGET: VSTRING; ITEM: CHARACTER;
  POSITION : STRINDEX := FIRST) return VSTRING is
begin -- INSERT
  return INSERT(TARGET, VSTR(ITEM), POSITION);
end INSERT;

function APPEND(TARGET: VSTRING; ITEM: VSTRING;
  POSITION : STRINDEX) return VSTRING is

TEMP : VSTRING;
POS : STRINDEX := POSITION;
begin -- APPEND
  if POSITION not in FIRST .. TARGET.LEN
    then raise CONSTRAINT_ERROR;
  end if;
  if TARGET.LEN + ITEM.LEN > LAST
    then raise CONSTRAINT_ERROR;
    else TEMP.LEN := TARGET.LEN + ITEM.LEN;
  end if;

113
TEMP.VALUE(FIRST .. POS) := TARGET.VALUE(FIRST .. POS);
TEMP.VALUE(POS + 1 .. (POS + ITEM.LEN)) := ITEM.VALUE(FIRST .. ITEM.LEN);
TEMP.VALUE((POS + ITEM.LEN + 1) .. TEMP.LEN) :=
  TARGET.VALUE(POS + 1 .. TARGET.LEN);

return(TEMP);
end APPEND;

function APPEND(TARGET: VSTRING; ITEM: STRING; POSITION : STRINDEX)
  return VSTRING is
begin -- APPEND
  return APPEND(TARGET, VSTR(ITEM), POSITION);
end APPEND;

function APPEND(TARGET: VSTRING; ITEM: CHARACTER; POSITION : STRINDEX)
  return VSTRING is
begin -- APPEND
  return APPEND(TARGET, VSTR(ITEM), POSITION);
end APPEND;

function APPEND(TARGET: VSTRING; ITEM: VSTRING) return VSTRING is
begin -- APPEND
  return(APPEND(TARGET, ITEM, TARGET.LEN));
end APPEND;

function APPEND(TARGET: VSTRING; ITEM: STRING) return VSTRING is
begin -- APPEND
  return(APPEND(TARGET, VSTR(ITEM), TARGET.LEN));
end APPEND;

function APPEND(TARGET: VSTRING; ITEM: CHARACTER) return VSTRING is
begin -- APPEND
  return(APPEND(TARGET, VSTR(ITEM), TARGET.LEN));
end APPEND;

function REPLACE(TARGET: VSTRING; ITEM: VSTRING;
  POSITION : STRINDEX := FIRST) return VSTRING is

  TEMP : VSTRING;

begin -- REPLACE
  if POSITION not in FIRST .. TARGET.LEN
    then raise CONSTRAINT_ERROR;
  end if;

  if POSITION + ITEM.LEN - 1 <= TARGET.LEN
    then TEMP.LEN := TARGET.LEN;
  elsif POSITION + ITEM.LEN - 1 > LAST
then raise CONSTRAINT_ERROR;
else TEMP.LEN := POSITION + ITEM.LEN - 1;
end if;

TEMP.VALUE(FIRST .. POSITION - 1) := TARGET.VALUE(FIRST .. POSITION - 1);
TEMP.VALUE(POSITION .. (POSITION + ITEM.LEN - 1)) :=
ITEM.VALUE(FIRST .. ITEM.LEN);
TEMP.VALUE((POSITION + ITEM.LEN) .. TEMP.LEN) :=
TARGET.VALUE((POSITION + ITEM.LEN) .. TARGET.LEN);

return(TEMP);
end REPLACE;

function REPLACE(TARGET: VSTRING; ITEM: STRING;
POSITION : STRINDEX :=FIRST) return VSTRING is
begin -- REPLACE
return REPLACE(TARGET, VSTR(ITEM), POSITION);
end REPLACE;

function REPLACE(TARGET: VSTRING; ITEM: CHARACTER;
POSITION : STRINDEX :=FIRST) return VSTRING is
begin -- REPLACE
return REPLACE(TARGET, VSTR(ITEM), POSITION);
end REPLACE;

function "&"(LEFT:VSTRING; RIGHT : VSTRING) return VSTRING is
 TEMP : VSTRING;

 begin -- 
 if LEFT.LEN + RIGHT.LEN > LAST
 then raise CONSTRAINT_ERROR;
 else TEMP.LEN := LEFT.LEN + RIGHT.LEN;
 end if;

 TEMP.VALUE(FIRST .. TEMP.LEN) := LEFT.VALUE(FIRST .. LEFT.LEN) &
 RIGHT.VALUE(FIRST .. RIGHT.LEN);
 return(TEMP);
end "&";

function "&"(LEFT:VSTRING; RIGHT : STRING) return VSTRING is
 begin -- 
 return LEFT & VSTR(RIGHT);
end "&";

function "&"(LEFT:VSTRING; RIGHT : CHARACTER) return VSTRING is
 begin -- 
 return LEFT & VSTR(RIGHT);
end "&";
function "&" (LEFT : STRING; RIGHT : VSTRING) return VSTRING is
begin -- "&"
    return VSTR(LEFT) & RIGHT;
end "&";

function "&" (LEFT : CHARACTER; RIGHT : VSTRING) return VSTRING is
begin -- "&"
    return VSTR(LEFT) & RIGHT;
end "&";

Function INDEX (WHOLE : VSTRING; PART : VSTRING; OCCURRENCE : NATURAL := 1)
return STRINDEX is

    NOT_FOUND : constant NATURAL := 0;
    INDEX : NATURAL := FIRST;
    COUNT : NATURAL := 0;

begin -- INDEX
    if PART = NUL then return (NOT_FOUND); -- by definition
         end if;
    while INDEX + PART.LEN - 1 <= WHOLE.LEN and then COUNT < OCCURRENCE loop
        if WHOLE.VALUE(INDEX .. PART.LEN + INDEX - 1) =
            PART.VALUE(1 .. PART.LEN)
            then COUNT := COUNT + 1;
                end if;
            INDEX := INDEX + 1;
        end loop;

    if COUNT = OCCURRENCE
        then return(INDEX - 1);
        else return (NOT_FOUND);
        end if;
    end INDEX;

Function INDEX (WHOLE : VSTRING; PART : STRING; OCCURRENCE : NATURAL := 1)
return STRINDEX is

    begin -- Index
            return (INDEX(WHOLE, VSTR(PART), OCCURRENCE));
    end INDEX;

Function INDEX (WHOLE : VSTRING; PART : CHARACTER; OCCURRENCE : NATURAL := 1)
return STRINDEX is

    begin -- Index
            return (INDEX(WHOLE, VSTR(PART), OCCURRENCE));
    end INDEX;
function RINDEX(WHOLE: VSTRING; PART: VSTRING; OCCURRENCE: NATURAL := 1)
return STRINDEX is

NOT_FOUND : constant NATURAL := 0;
INDEX : INTEGER := WHOLE.LEN - (PART.LEN - 1);
COUNT : NATURAL := 0;

begin -- RINDEX
if PART = NUL then return(NOT_FOUND); -- by definition
end if;

while INDEX >= FIRST and then COUNT < OCCURRENCE loop
if WHOLE.VALUE(INDEX .. PART.LEN + INDEX - 1) =
PART.VALUE(INDEX .. PART.LEN)
then COUNT := COUNT + 1;
end if;
INDEX := INDEX - 1;
end loop;

if COUNT = OCCURRENCE
then
if COUNT > 0
then return(INDEX + 1);
else return(NOT_FOUND);
end if;
else return(NOT_FOUND);
end if;
end RINDEX;

Function RINDEX(WHOLE : VSTRING; PART : STRING; OCCURRENCE : NATURAL := 1)
return STRINDEX is

begin -- Rindex
return(RINDEX(WHOLE, VSTR(PART), OCCURRENCE));
end RINDEX;

Function RINDEX(WHOLE : VSTRING; PART : CHARACTER; OCCURRENCE : NATURAL := 1)
return STRINDEX is

begin -- Rindex
return(RINDEX(WHOLE, VSTR(PART), OCCURRENCE));
end RINDEX;

function VSTR(FROM : CHARACTER) return VSTRING is

TEMP : VSTRING;

begin -- VSTR


if LAST < 1
  then raise CONSTRAINT_ERROR;
else TEMP.LEN := 1;
end if;

TEMP.VALUE(FIRST) := FROM;
return(TEMP);
end VSTR;

function VSTR(FROM : STRING) return VSTRING is

  TEMP : VSTRING;

begin -- VSTR
  if FROM.LN > LAST
    then raise CONSTRAINT_ERROR;
  else TEMP.LEN := FROM.LN;
  end if;

  TEMP.VALUE(FIRST .. FROM.LN) := FROM;
  return(TEMP);
end VSTR;

Function "*" (FROM : STRING) return VSTRING is
begin -- "*"
  return(VSTR(FROM));
end "*";

Function "*" (FROM : CHARACTER) return VSTRING is
begin
  return(VSTR(FROM));
end "*";

function CONVERT(X : FROM) return TO is

begin -- CONVERT
  return(VSTR(STR(X)));
end CONVERT;
end VSTRINGS;

-- DISTRIBUTION AND COPYRIGHT:
--
-- This software is released to the Public Domain (note:
-- software released to the Public Domain is not subject
-- to copyright protection).
-- Restrictions on use or distribution:  NONE
--
-- DISCLAIMER:
This software and its documentation are provided "AS IS" and without any expressed or implied warranties whatsoever. No warranties as to performance, merchantability, or fitness for a particular purpose exist.

Because of the diversity of conditions and hardware under which this software may be used, no warranty of fitness for a particular purpose is offered. The user is advised to test the software thoroughly before relying on it. The user must assume the entire risk and liability of using this software.

In no event shall any person or organization of people be held responsible for any direct, indirect, consequential or inconsequential damages or lost profits.
C. List_Single_Unbounded_Unmanaged

generic
  type Item is private;
package List_Single_Unbounded_Unmanaged is

  type List is private;

  Null_list : constant List;

  procedure Construct (The_Item : in Item;
      And_The_List : in out List);

  function Is_Null (The_List : in List) return Boolean;
  function Head_Of (The_List : in List) return Item;
  function Tail_Of (The_List : in List) return List;

  Overflow : exception;
  List_Is_Null : exception;

private
  type Node;
  type List is access Node;
  Null_List : constant List := null;
end List_Single_Unbounded_Unmanaged;

package body List_Single_Unbounded_Unmanaged is

  type Node is
 discrod
    The_Item : Item;
    Next : List;
  end record;

  procedure Construct (The_Item : in Item;
      And_The_List : in out List) is
  begin
    And_The_List := new Node(The_Item => The_Item,
      Next => And_The_List);
  exception
    when Storage_Error =>
      raise Overflow;
  end Construct;

  function Is_Null (The_List : in List) return Boolean is
  begin
    return (The_List = null);
  end Is_Null;

  function Head_Of (The_List : in List) return Item is
begin
   return The_List.The_Item;
exception
   when Constraint_Error =>
      raise List_Is_Null;
end Head_Of;

function Tail_Of (The_List : in List) return List is
begin
   return The_List.Next;
exception
   when Constraint_Error =>
      raise List_Is_Null;
end Tail_Of;

end List_Single_Unbounded_Unmanaged;
D. PRIORITY_DEFINITIONS

with SYSTEM;
use SYSTEM;
package PRIORITY_DEFINITIONS is
  BUFFER_PRIORITY : constant priority := 10;
  STATIC_SCHEDULE_PRIORITY : constant priority := 8;
  DYNAMIC_SCHEDULE_PRIORITY : constant priority := 1;
end PRIORITY_DEFINITIONS;
LIST OF REFERENCES


**INITIAL DISTRIBUTION LIST**

<table>
<thead>
<tr>
<th></th>
<th>Distributor</th>
<th>Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Defense Technical Information Center</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>Cameron Station</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Alexandria, Virginia 22304-6145</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Dudley Knox Library</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>Code 52</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Naval Postgraduate School</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Monterey, California 93943-5002</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Director of Research Administration</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Attn: Prof. Howard</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Code 81</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Naval Postgraduate School</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Monterey, California 93943-5100</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Prof. McGhee</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Code CS/Mz</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Naval Postgraduate School</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Monterey, California 93943-5100</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Chief of Naval Research</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>800 N. Quincy Street</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Arlington, Virginia 22217-5000</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Center for Naval Analysis</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>4401 Ford Avenue</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Alexandria, Virginia 22302-0268</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>National Science Foundation</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Division of Computer and Computation Research</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Attn: Dr. K.C. Tai</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Washington, D.C. 20550</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Ada Joint Program Office</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>OUSDRE(R&amp;AT)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Pentagon</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Washington, D.C. 20301</td>
<td></td>
</tr>
</tbody>
</table>

125
9. Naval Sea Systems Command  
   Attn. CAPT A. Thompson  
   National Center #2, Suite 7N06  
   Washington, D. C. 22202

10. Navy Ocean System Center  
    Attn. Linwood Sutton, Code 423  
    San Diego, California 92152-5000

11. Office of Naval Research  
    Computer Science Division, Code 1133  
    Attn. Dr. R. Wachter  
    800 N. Quincy Street  
    Arlington, Virginia 22217-5000

12. Office of Naval Research  
    Applied Mathematics and Computer Science, Code 1211  
    Attn. Mr. J. Smith  
    800 N. Quincy Street  
    Arlington, Virginia 22217-5000

13. Defense Advanced Research Projects Agency (DARPA)  
    Director, Naval Technology Office  
    1400 Wilson Boulevard  
    Arlington, Virginia 22209-2308

14. Defense Advanced Research Projects Agency (DARPA)  
    Director, Prototype Projects Office  
    1400 Wilson Boulevard  
    Arlington, Virginia 22209-2308

15. Defense Advanced Research Projects Agency (DARPA)  
    Director, Tactical Technology Office  
    1400 Wilson Boulevard  
    Arlington, Virginia 22209-2308

16. Chief of Naval Operations  
    Attn: Dr. R. M. Carroll (OP-01B2)  
    Washington, D.C. 20350

17. Chief of Naval Operations  
    Attn: Dr. Earl Chavis (OP-162)  
    Washington, D.C. 20350
18. Naval Surface Warfare Center
   Code K54
   Attn: Dr. William McCoy
   Dahlgren, Virginia 22448

19. Naval Surface Warfare Center
   Code U33
   Attn: Phil Hwang
   Silver Spring, Maryland 20903-5000

20. Professor Luqi
    Code CS/Lq
    Naval Postgraduate School
    Computer Science Department
    Monterey, California 93943-5100

21. Frank V. Palazzo
    Defense Manpower Data Center
    99 Pacific Street
    Monterey, California 93940

22. Defense Manpower Data Center
    Attn. Mr. Robert Brandewie
    99 Pacific Street
    Monterey, California 93940