DTIC FILE COPY

1

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1a	1b RESTRICTIVE MARKINGS NONE
2a AD-A218 735	3 DISTRIBUTION/AVAILABILITY OF REPORT APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.
2b	
4 PERFORMING ORGANIZATION REPORT NUMBER(S)	5 MONITORING ORGANIZATION REPORT NUMBER(S) AFIT/CI/CIA- 89-108
6a NAME OF PERFORMING ORGANIZATION AFIT STUDENT AT [illegible] — 6b OFFICE SYMBOL (If applicable)	7a NAME OF MONITORING ORGANIZATION AFIT/CIA
6c ADDRESS (City, State, and ZIP Code)	7b ADDRESS (City, State, and ZIP Code) Wright-Patterson AFB OH 45433-6583
8a NAME OF FUNDING / SPONSORING ORGANIZATION — 8b OFFICE SYMBOL (If applicable)	9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c ADDRESS (City, State, and ZIP Code)	10 SOURCE OF FUNDING NUMBERS: PROGRAM ELEMENT NO / PROJECT NO / TASK NO / WORK UNIT ACCESSION NO

11 TITLE (Include Security Classification) (UNCLASSIFIED)
[illegible]

12 PERSONAL AUTHOR(S)
[illegible]

13a TYPE OF REPORT	13b TIME COVERED	14 DATE OF REPORT (Year, Month, Day)	15 PAGE COUNT
THESIS/~~DISSERTATION~~	FROM ____ TO ____	[illegible]	109

16 SUPPLEMENTARY NOTATION
APPROVED FOR PUBLIC RELEASE IAW AFR 190-1
ERNEST A. HAYGOOD, 1st Lt, USAF
Executive Officer, Civilian Institution Programs

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

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

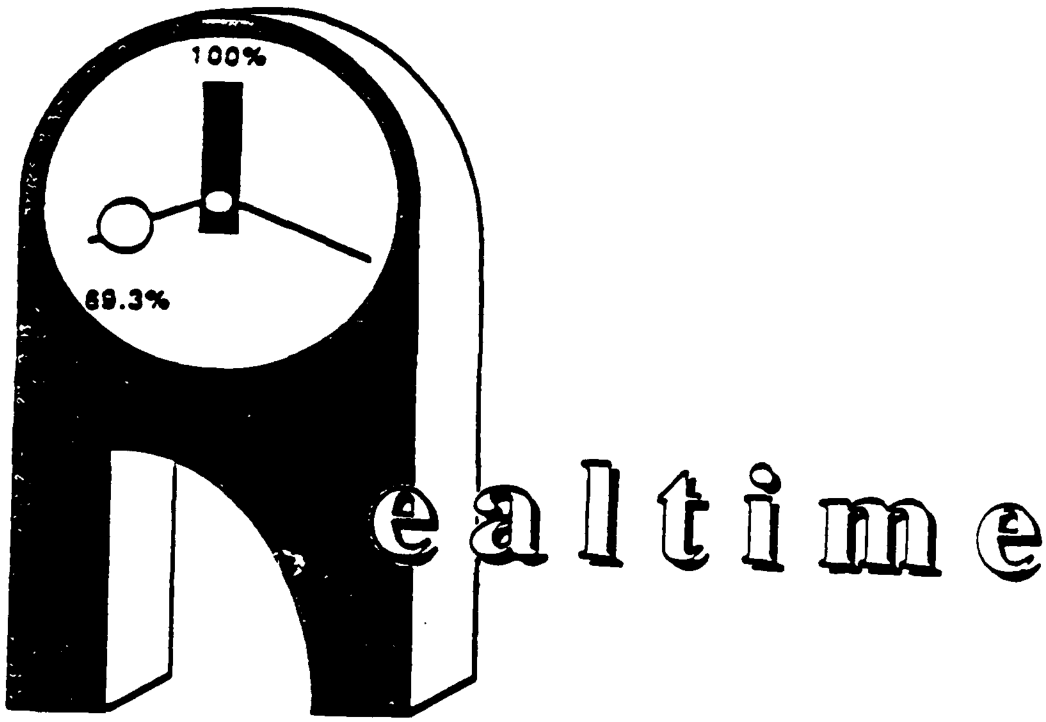
DTIC
S ELECTE FEB 15 1990 D
D

90 02 14 058

20 DISTRIBUTION/AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS	21 ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED
22a NAME OF RESPONSIBLE INDIVIDUAL ERNEST A. HAYGOOD, 1st Lt, USAF	22b TELEPHONE (Include Area Code) (513) 255-2259 — 22c OFFICE SYMBOL AFIT/CI

DD Form 1473, JUN 86 — Previous editions are obsolete. — SECURITY CLASSIFICATION OF THIS PAGE

AFIT/CI "OVERPRINT"



Imprecise Computations in Ada

Gary James Gregory

November 1989

IMPRECISE COMPUTATIONS IN ADA

BY

GARY JAMES GREGORY

B.S., United States Air Force Academy, 1983

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1990

Urbana, Illinois

Table of Contents

Acknowledgments

I am indebted to several people who helped me directly and indirectly in my research efforts. First, I want to thank my thesis advisor Dr. Kwei-Jay Lin for his guidance and his answers to my many questions. Second, I want to thank Capt Sean Doran and SSgt Keith Kepner of the 83rd Fighter Weapons Squadron for their tireless efforts. Without their painstaking editing and in-depth operating system knowledge, I could not have completed the real-time tests. Finally, I want to thank my family for their encouragement and my wife, Sharon, for her loving support during the life of this project.

1 Introduction

1.1 Ada

Ada is the United States Department of Defense's (DoD) newest programming language. Ada was born in an era of rising software costs and a proliferation of programming languages within the DoD. To halt this software crisis, the DoD developed Ada. Ada was to become the common, high-order programming language for all organizations within the DoD. Since the majority of software costs in the DoD were connected with embedded systems [5], it is not surprising that Ada was designed with real-time programming in mind.

Current estimates [18] show that the DoD spends $12 billion a year on software for embedded, real-time computer systems for missile guidance, communications control, and weapons firing. This value is growing at a compound annual rate of 17%. The Ada share of this market is increasing as Ada receives acceptance and older languages are phased out. Initially, Ada received staunch opposition and required the DoD to take steps to ensure Ada's acceptance.

DoD Directives 3405.1 and 3405.2 [21,22] were drafted and signed into effect in 1987 making Ada the single, common, high-order programming language within the DoD. Additionally, these directives mandated the use of Ada in intelligence systems, command and control systems, and weapons systems. The North Atlantic Treaty Organization (NATO) has also established policies that mandate the use of Ada abroad

[4]. Ada will be used on numerous European projects, including the European Space Agency's space station project, Airbus flight avionics, and air traffic control systems. Within the DoD, Ada will be used to program the on-board, embedded, real-time computer systems of the Air Force's Advanced Tactical Fighter, the Army's Light Helicopter Experimental, and the Navy's Advanced Tactical Aircraft [18].

The committment to Ada is strong, not only in the United States but abroad also. This committment is especially strong in the area of embedded, real-time systems.

1.2 Imprecise Computation

The concept of imprecise computation is quite straightforward [11-13]. For some applications, approximate results are adequate when the nature of the computation involves lengthy computation time. Under real-time computation constraints, these lengthy computations may never be able to finish. When the degree of accuracy of the intermediate results of a computation is non-decreasing as more processor time is spent to obtain a result, the process is called a monotone process [13]. If the monotone process completes normally, it will produce a precise result. However, if the monotone process times out prior to completion, it produces a result that is not precise, or imprecise. Although the imprecise result is not as precise as originally desired, it may still be of use to the application.

Acceptable imprecise results can be returned when the structure of a computation is iterative [20]. Many iterative numerical computations fall into this category [15,16]. Monte Carlo simulation is another prime iterative target. Monte Carlo simulation has been used to determine radiation shielding and nuclear reactor criticality [9]. An example of the Jacobi method used to solve linear systems of equations and an example of Monte Carlo simulation used to perform integration of a curve are presented later in this paper as imprecise computation examples. Iterative computations are not the only type of computation that can be implemented as an imprecise computation. Additionally, some non-iterative computations can be reformulated as iterative computations and used as an imprecise computation [3].

There are two language primitives required by the programmer to implement an imprecise computation application. These primitives are "impreturn" and "impresult" [11,12]. Impreturn sends imprecise results and error indicators from the callee to the caller. Impresult binds a handler procedure to an imprecise computation. The handler is called to "massage" the imprecise result before it is returned.

Imprecise computation is especially applicable to real-time computer systems. Under severe time constraints, the imprecise computation will return an imprecise, though correct value. It is here that the link between Ada and imprecise computation lies.

1.3 Ada and Imprecise Computations

There has been no in-depth, published work accomplished in an effort to correctly implement imprecise computations in Ada. Although this topic has been cursorily addressed [2,13], no immediately implementable solutions have been identified. The committment to Ada in the real-time arena is sharply growing, while the concept of imprecise computation continues to increase its following. By implementing imprecise computations in Ada, the real-time system designer is given a viable tool in designing state of the art, fault tolerant, real-time computer systems. It was for this reason that this research project was undertaken.

2 Design

2.1 Goals and Criteria

This research effort began with the goal of investigating all possible approaches to implementing imprecise computations in standard Ada. This included actual implementation and testing of feasible approaches. Because Ada supports the software engineering principles of information hiding, modularity, and localization [5], the imprecise computation implementation would be a self contained module that is cohesively strong. Utmost consideration was given to creating a well-structured software system. This in turn would translate to ease of use on the part of the real time system designer who would ultimately use the imprecise computation module. The criteria for evaluating each approach to implementing imprecise computations in Ada evolved from these considerations.

The evaluation criteria represented varied requirements, desires and concerns. Because this implementation would be employed in a real-time system, efficiency was a key requirement. An inefficient implementation would not be tolerable. Portability was another vital concern. Ada was designed to be portable. Because the name "Ada" is trademarked, no dialects or subsets are legally allowed. The implementation should in no way rely on the underlying machine or operating system. If the implementation were too unruly or difficult to understand, it would probably not be utilized. Therefore, ease of programming was a crucial criterion. It is easy to find a solution to a problem when the constraints on the problem are changed in midstream. Likewise, it is easy for someone approaching the problem of implementing imprecise computations in Ada to come up with an easy solution, but one which involves changes to the Ada standard. The goal of this research was to implement imprecise computations in standard Ada, without additions that are contrary to the standard. Finally, the implementation would have to produce correct results according to the tenets of imprecise computation.

In summary, each approach was analyzed and evaluated based on the following criteria: efficiency; portability; ease of programming; whether or not it could be implemented using the current (standard) version of Ada, and; correctness. Three general approaches to implementing imprecise computations in Ada were identified.

2.2 Approaches to Implementation

There were three approaches to implementation of imprecise computations in Ada identified. Subsequently, each approach was analyzed and evaluated based on the criteria defined in Section 2.1. The approaches identified involved shared memory and variables, asynchronous transfer of control, and atomic computation loops.

2.2.1 Shared Memory / Shared Variables

In this method, cooperating tasks share memory locations containing common variables such as boolean flags. A timeout flag location would be established wherein a timer task would flag a timeout condition. The computation task would be required to check this flag repeatedly during its execution. This requires that each computation task contain a polling mechanism. Polling not only violates the principle of modularity, but it also imposes significant overhead if done frequently enough to guarantee fast response [2]. Polling reduces the efficiency of the executing code. An additional problem lies in the use of the pragma "SHARED". The Ada standard [23] provides pragma SHARED to allow two tasks to communicate via shared variables. These shared variables are identified as such by the pragma SHARED statement. This ensures that the tasks are properly synchronized when accessing the shared variable. However, the Ada development environment available to us, the Verdix Ada Development System (VADS), does not implement the pragma SHARED [24]. Due to these insurmountable problems, this approach was rejected for implementation.

2.2.2 Asynchronous Transfer of Control

There are basically two different ways any task can influence another task. A task can abort another task or it can rendezvous with it. The abort statement is not an effective communication means between tasks and can be costly in terms of execution time [2]. The rendezvous is by definition a synchronous means of communication. However, there is no means for one task to asynchronously interrupt another task. Broegel [6] points out that this is a result of a conflict between the two goals of having uninterruptible critical regions and short interrupt latency. In a preliminary version of Ada [10], the exception "FAILURE" was capable of being raised in other tasks. However, this feature was phased out in the Ada standard.

Some research has attempted to extend the Ada tasking model and allow a task to asynchronously interrupt another task. At Delft University of Technology, researchers have constructed a custom implementation of Ada that allows asynchronous interrupts [19]. Baker [2] presents a possible implementation of imprecise computations, but relies on a non-standard package to asynchronously raise exceptions. Both of these approaches are comprised of non-standard Ada and hence are not portable.

Asynchronously raising an exception in a computation would be a straightforward mechanism towards implementing imprecise computations in Ada. Unfortunately, there exists no standard Ada way to accomplish this. Any non-standard solution would not be portable and not acceptable. This approach was summarily rejected.

2.2.3 Atomic Computation Loop

This approach started with the concept of treating the computation loop as an atomic unit. The loop would be triggered each iteration by a monitor task when sufficient time was available prior to its deadline. The computation loop would not be interrupted once it started the current iteration. This approach introduced strict timing concerns because of the performance of the Ada tasking model implementations.

The Ada tasking model has been sharply criticized due to its alleged inefficiency. The designers of the new Hellfire missile [14] opted not to use the tasking features because of critical time constraints. A recent study was conducted by Burger and Nielsen [7] to determine the overhead of Ada tasking facilities. The measurements were made on a Digital Equipment Corporation (DEC) VAX 8600 running DEC Ada V1.2. As a baseline, a simple procedure call required 11 microseconds. But a simple, non-parameter rendezvous required 503 microseconds. This disparity mandated a judicious use of the rendezvous in the implementation of imprecise computations in Ada. For this reason, the atomic computation loop approach was broken down into a synchronous version and an asynchronous version.

In both versions, a computation task is created that performs the required function. This task contains the computation loop that refines the precision of the result.

The computation loop contains IMPRETURN statements that return the current, imprecise result. When the deadline occurs before the computation loop achieves a precise result, the appropriate handler is invoked and the computation loop is stopped. If the computation loop runs to completion, it signals via the compute rendezvous in the synchronous version and via a boolean flag in the asynchronous version.

As the name implies, the asynchronous version does not interfere nor does it rendezvous with the computation task. The asynchronous version initializes the computation loop with input parameters by way of a TIMER task. After initialization, the TIMER task starts the computation loop. The loop continues unmolested until it either completes or is stopped due to the deadline. The TIMER task has a higher priority than the computation task which guarantees that the TIMER task will execute when necessary. The TIMER task monitors the progression of time as it approaches the deadline by delaying a duration proportional to the amount of time left before the deadline. In the meantime, the computation task is storing imprecise results and error indicators via the IMPRETURN call. When TIMER times out, it grabs the most recent copy of the imprecise results, invokes the appropriate handler, and then returns the result. If the computation loop completes, it signals via an IMPRETURN call with the final result value and zero error indicator.

The synchronous version relies on frequent rendezvous.

This version initializes the computation loop with input parameters and then calls for a rendezvous with the computation task each time the compute loop is to be executed. When the deadline occurs, the appropriate handler is invoked and the computation loop is stopped. If the computation loop runs to completion, it signals via the compute entry call and is subsequently stopped.

Both versions have their respective advantages and disadvantages. The synchronous version is less efficient because of the frequency of rendezvous, but maintains more control over the computation loop. Conversely, the asynchronous version requires no rendezvous with the computation loop and relies on the run-time system's efficient and correct implementation of the Ada "delay" statement. Both versions required no modifications to standard Ada. Efficiency, a key design criterion, was initially a major detractor of the synchronous version. A system spending more time completing rendezvous and less time computing was intolerable. However, study of potential imprecise computation targets such as the Jacobi method for solving linear systems of equations [15,16] showed that these computations may only loop about 10 to 50 times before a precise result is calculated. The rendezvous overhead is trivial compared to a Monte Carlo application which might loop about 10000 times before a precise result is obtained. It was apparent that both versions were viable approaches to implementing imprecise computations in Ada.

3 Implementation

3.1 Ada Specifics

To promote sound software engineering principles, the data type definitions, variable declarations, and associated procedures of the imprecise computation system are located in a single, strongly cohesive module. In Ada, such a module is known as a package. Further, because the result type of each imprecise computation is unique, the imprecise computation package would have to allow differing result types. For example, the Jacobi imprecise computation requires a 3-element array of floating point numbers as its result, while the Monte Carlo imprecise computation of the area of a circle merely requires a single floating point number for its result. It would be quite unruly to construct and maintain an imprecise computation package for any conceivable result type. Fortunately, Ada provides a means to circumvent this situation.

Ada provides the "generic" package. This allows the designer to implement a mechanism without ties to specific data types. According to Booch [5], generic program units define a unit template, along with generic parameters that provide the facility for tailoring that template to particular needs at compilation time. At compile time, a generic package is instantiated by specifying the actual parameters to be substituted for the generic parameters, thus creating an instance of the package. Generic parameters can be types, values, objects, and/or subprograms [5].

One of the generic parameter types represents the imprecise computation. Because it is necessary for the imprecise computation to maintain its state information, the imprecise computation must be constructed as a task. Accordingly, one of the generic parameters is the imprecise computation task type and another parameter is an access type that points to the task type. Other generic parameter types include the result type, the error indicator type, and the input type. Generic parameter subprograms are used to call the entry points in the imprecise computation task. These procedures are necessary because the imprecise computation package has no knowledge of the specific task structure until instantiation. Therefore, the task entry points cannot be hard-coded into the imprecise computation package, even if the entry names are standardized. The actual procedures corresponding to the generic subprograms are simple, one line programs that call the appropriate entry points. These entry points vary between the asynchronous and synchronous imprecise computation packages.

Through the use of the generic package, single synchronous and asynchronous imprecise computation packages can be constructed. At compilation, new instances of these packages can be created by specifying the appropriate generic parameters. This allows the luxury of having one asynchronous package and one synchronous package to modify and maintain, but at the same time allowing unlimited instances based on the specific computation.

3.2 Synchronous Imprecise Computation

The package SYNCHRONOUS_IMPRECISE_COMPUTATION has been implemented as a generic package. This package is composed of generic parameters required for instantiation and procedures visible from outside the package.

3.2.1 Generic Parameters

The package SYNCHRONOUS_IMPRECISE_COMPUTATION contains the following generic parameter list:

```
type COMPUTATION is limited private;

type COMPUTATION_PTR is access COMPUTATION;

type RESULT_TYPE is private;

type ERROR_INDICATOR_TYPE is private;

type INPUT_TYPE is private;

with procedure INITIALIZE(THE_COMPUTATION : in
                              COMPUTATION_PTR;
                          INPUT           : in
                              INPUT_TYPE);

with procedure COMPUTE(THE_COMPUTATION      : in
                           COMPUTATION_PTR;
                       COMPUTATION_COMPLETE : out
                           boolean);

with procedure HANDLE(THE_COMPUTATION      : in
                          COMPUTATION_PTR;
                      HANDLER_NUMBER       : in
                          integer;
                      LAST_VALUE           : in
                          RESULT_TYPE;
                      LAST_ERROR_INDICATOR : in
```

```
                    ERROR_INDICATOR_TYPE);
   with procedure STOP(THE_COMPUTATION : in COMPUTATION_PTR);
```

The type COMPUTATION corresponds to the task type of the desired imprecise computation. The task type serves as a template that is used to create instances of task objects [5]. In this way, multiple imprecise computation tasks may be active simultaneously. The task type is declared a limited type because neither assignment nor the predefined comparison for equality and inequality are defined for objects of task types [23].

The type COMPUTATION_PTR provides an access type to the task type COMPUTATION. When a pointer of type COMPUTATION_PTR is allocated using the "new" statement, a task in the form of task type COMPUTATION is created. The pointer variable now points to the active task and is used to reference the task entry points. This pointer is needed in the imprecise computation package because it effectively allows a task to be passed as an argument to a procedure. Actually, the pointer is being passed but the result is the same. In this way, an allocated pointer variable of type COMPUTATION_PTR is an effective and efficient means of manipulating the computation task.

The generic parameter RESULT_TYPE is merely the data type of the result that the imprecise computation generates. Here lies the beauty of Ada's generic facility, for any valid data type can be used to instantiate the generic package.

The type ERROR_INDICATOR_TYPE provides the means of determining the exact precision of an imprecise computation's result. It can be instantiated with the data type that is most applicable to the imprecise computation.

The generic parameter INPUT_TYPE is the data type used to initialize the computation task. Often, several items are needed to properly initialize a computation task. In this case, INPUT_TYPE should be instantiated with a record type composed of the necessary items. The remaining generic parameters in the package SYNCHRONOUS_IMPRECISE_COMPUTATION are generic subprograms.

Each generic subprogram is needed in order to rendezvous with various entry points of the computation task. The user of the SYNCHRONOUS_IMPRECISE_COMPUTATION generic package must construct his own computation task type. This task type must include several entry points. An initialization entry point receives input data. A compute entry point performs one loop of the computation. One or more handler entry points are required to manipulate the imprecise result. Finally, an entry point to stop the task is required in lieu of the abort option. The names of these entry points are not relevant, but must be properly reflected in the procedures used to instantiate the generic package. For example, consider a task type with the following structure:

```
task type EXAMPLE is
     entry INITIALIZE_THE_TASK(...);
     entry COMPUTE_ONE_LOOP(...);
```

```
      entry HANDLER(...);
      entry HALT_THE_TASK;
   end EXAMPLE;
```

The procedure for stopping the task that would be used to instantiate the generic package would look like the following:

```
   procedure STOP_TASK(COMP_PTR : in COMPUTATION_PTR) is
      begin
         COMP_PTR.HALT_THE_TASK;
      end STOP_TASK;
```

Note that the procedure can have any name. At instantiation, the procedure name is bound to the generic subprogram STOP. So whenever STOP is called in the imprecise computation mechanism, STOP_TASK will be called at run-time. Each generic subprogram has clearly defined purposes.

The procedure INITIALIZE takes two parameters, THE_COMPUTATION and INPUT. THE_COMPUTATION references the computation task calculating the imprecise computation. INPUT is the data required to properly initialize the computation task. This procedure must be instantiated with a simple procedure that merely requests a rendezvous with the initialization entry call of the computation task.

The procedure COMPUTE initiates a rendezvous with the compute entry point of the computation task. Procedure COMPUTE takes two parameters, THE_COMPUTATION and COMPUTATION_COMPLETE. The former is a pointer to the computation task. The latter is a boolean flag that is set by the computation task to alert the imprecise computation mechanism that a precise result has been produced. If the computation task does not produce a precise result by its deadline, a handler task must be called.

The procedure HANDLE initiates a rendezvous with a specified handler entry point within the computation task. The parameters for this procedure are THE_COMPUTATION, HANDLER_NUMBER, LAST_VALUE, and LAST_ERROR_INDICATOR. Again, THE_COMPUTATION is a pointer referencing the computation task. There may be more than one handler entry point in the computation task. The parameter HANDLER_NUMBER specifies which handler entry point to call. The handler entry points may be implemented as a family of entry calls with a discrete range [23]. If not, procedure HANDLE will be required to decipher the value of HANDLER_NUMBER and call the appropriate entry point. The parameters LAST_VALUE and LAST_ERROR_INDICATOR represent the most current imprecise result and error indicator returned by the imprecise computation task. They are passed to the handler entry point where they can be modified if necessary. A modified imprecise result and error indicator is saved in the standard method by issuing an IMPRETURN call at the end of the handler rendezvous. After a precise result has been computed or a handler executed, the computation task must be stopped.

The procedure STOP initiates a rendezvous with the stop entry point of the computation task. A boolean flag is then set and subsequently causes an exit from the internal loop structure. Procedure STOP requires one parameter, THE_COMPUTATION. This parameter is a pointer referencing the computation task.

At compilation time, all of the preceding generic types and subprograms are instantiated with the data types and procedures developed by the user. After instantiation, a custom synchronous imprecise computation package exists in the user's library. Now, the user has the capability of accessing the procedures bundled in the synchronous imprecise computation package.

3.2.2 Procedures

There are two procedures in the generic package SYNCHRONOUS_IMPRECISE_COMPUTATION as dictated by the tenets of imprecise computation [11-13]. However, the name of the procedure IMPRESULT has been changed to IMPCALL because it seemed more fitting of its role. The other procedure remains as IMPRETURN. The procedure declarations are defined in the generic package specification in the following manner:

```
procedure IMPCALL(THE_COMPUTATION : in out
                                COMPUTATION_PTR;
                  THE_HANDLER     : in     integer;
                  DEADLINE        : in
                                CALENDAR.TIME;
                  INPUT           : in     INPUT_TYPE;
                  FINAL_RESULT    :    out
                                RESULT_TYPE);
```

```
procedure IMPRETURN(INTERMEDIATE_RESULT : in
                        RESULT_TYPE;
                    ERROR_INDICATOR     : in
                        ERROR_INDICATOR_TYPE);
```

Procedure IMPCALL requires five parameters. Parameter THE_COMPUTATION is a pointer to the computation task that will be computed in an imprecise fashion. In the event the computation task does not complete before its deadline, parameter THE_HANDLER indicates which handler routine to call. Parameter DEADLINE specifies the absolute time when computation should cease. The computation task is initialized with the contents of the parameter INPUT. Finally, the out parameter FINAL_RESULT is the precise result if the computation task completes, or the imprecise result after being passed through the handler routine.

Procedure IMPRETURN is the means by which the computation task returns imprecise results and error indicators to the imprecise computation mechanism. The two parameters of procedure IMPRETURN reflect this design. Parameter INTERMEDIATE_RESULT is the current imprecise result, while parameter ERROR_INDICATOR indicates the precision of this result.

An imprecise computation application can only interface with an instantiated imprecise computation package via the two procedures IMPCALL and IMPRETURN as specified in the package specification. The package body contains the code that implements these two procedures. However, the data types, variables, and procedures in the package body are invisible to the user. These entities are only visible within the package body itself [23]. The package body of SYNCHRONOUS_IMPRECISE_COMPUTATION contains two variables that are global within the package body.

```
CURRENT_VALUE           : RESULT_TYPE;
CURRENT_ERROR_INDICATOR : ERROR_INDICATOR_TYPE;
```

These variables reflect the current imprecise result and its associated error indicator. These variables are updated solely by the IMPRETURN procedure. Procedure IMPCALL is implemented in the package body of SYNCHRONOUS_IMPRECISE_COMPUTATION in the following way:

```
procedure IMPCALL(THE_COMPUTATION : in out
                                             COMPUTATION_PTR;
                  THE_HANDLER     : in        integer;
                  DEADLINE        : in        CALENDAR.TIME;
                  INPUT           : in        INPUT_TYPE;
                  FINAL_RESULT    :    out RESULT_TYPE) is

     COMPUTATION_COMPLETED : boolean;
     TIME_HACK             : CALENDAR.TIME;

     begin
          INITIALIZE(THE_COMPUTATION,
                     INPUT);
          loop
               COMPUTE(THE_COMPUTATION,
                       COMPUTATION_COMPLETED);
               exit when COMPUTATION_COMPLETED;
               TIME_HACK := CALENDAR.CLOCK;

               if CALENDAR.">"(TIME_HACK,
                               DEADLINE) then
                   HANDLE(THE_COMPUTATION,
                          THE_HANDLER,
                          CURRENT_VALUE,
                          CURRENT_ERROR_INDICATOR);
                   exit;
```

```
        end if;
    end loop;
    STOP(THE_COMPUTATION);
    FINAL_RESULT := CURRENT_VALUE;
end IMPCALL;
```

The algorithm involved is straightforward. The computation task is first initialized by calling the procedure INITIALIZE. This generic subprogram in turn completes a rendezvous with the computation task, passing it the appropriate data in parameter INPUT. The algorithm then enters a loop. This loop will be executed when the computation completes or the deadline is reached. First, the procedure COMPUTE is called. This generic subprogram in turn enters into a rendezvous with the computation task at the compute entry point. Remember, this rendezvous causes the computation task to complete one iteration of the computation. If this causes the computation to complete, it signals so via the COMPUTATION_COMPLETED parameter. After COMPUTE finishes, the loop will be exited if the computation has completed. If not, the system clock is sampled and compared to the deadline. If the deadline has expired, the procedure HANDLE is called which in turn initiates a rendezvous with the computation task at the handler entry point. The loop is exited after the procedure HANDLE completes. If the deadline has not expired, control returns to the top of the loop. After termination of the loop, procedure STOP is called, ultimately completing a rendezvous with the computation task at the stop entry point. The final precise or imprecise result is then copied to the parameter FINAL_RESULT and subsequently passed back to the caller.

The procedure IMPRETURN is the means by which the computation task returns imprecise results and error indicators. There is no algorithm required because this process merely involves the passing and subsequent storing of data. This procedure is implemented in the following way:

```
procedure IMPRETURN(INTERMEDIATE_RESULT : in
                                        RESULT_TYPE;
                    ERROR_INDICATOR     : in
                                        ERROR_INDICATOR_TYPE) is

    begin
      CURRENT_VALUE              := INTERMEDIATE_RESULT;
      CURRENT_ERROR_INDICATOR    := ERROR_INDICATOR;
    end IMPRETURN;
```

The input parameters INTERMEDIATE_RESULT and ERROR_INDICATOR are copied to the hidden variables CURRENT_VALUE and CURRENT_ERROR_INDICATOR, respectively.

The complete package specification and package body of SYNCHRONOUS_IMPRECISE_COMPUTATION can be found in Appendix A. Figure 1 presents the synchronous imprecise computation mechanism in a graphical manner, using the symbols defined in [5]. This approach to imprecise computations has been implemented in standard Ada code and should compile on any validated compiler. The user need only instantiate this package with his own data types and subprograms. Actual imprecise computation examples using this package are given in Section 4.

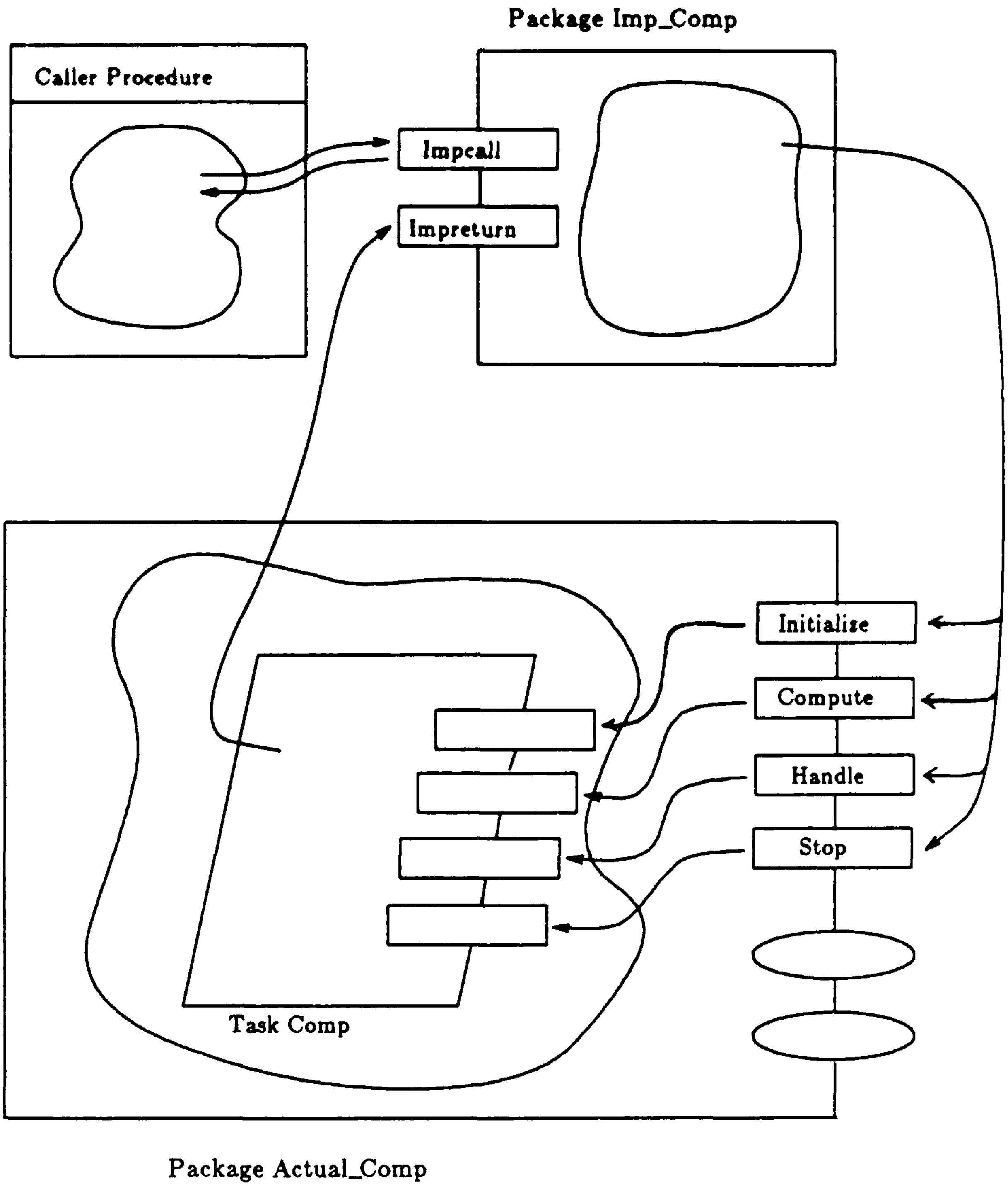


Figure 1. Synchronous Imprecise Computation

3.3 Asynchronous Imprecise Computation

The package ASYNCHRONOUS_IMPRECISE_COMPUTATION has been implemented as a generic package. This package is very similar to the synchronous implementation in terms of user interface, but internally is quite different. Like the synchronous version, this package is composed of generic parameters required for instantiation and procedures visible from outside the package.

3.3.1 Generic Parameters

The package ASYNCHRONOUS_IMPRECISE_COMPUTATION contains the following generic parameter list:

```
type COMPUTATION is limited private;
type COMPUTATION_PTR is access COMPUTATION;
type RESULT_TYPE is private;
type ERROR_INDICATOR_TYPE is private;
type INPUT_TYPE is private;
with procedure START_COMPUTATION(THE_COMPUTATION : in
                                         COMPUTATION_PTR;
                                 INPUT           : in
                                         INPUT_TYPE);
with procedure HANDLE(LAST_VALUE           : in out
                               RESULT_TYPE;
                      LAST_ERROR_INDICATOR : in out
                               ERROR_INDICATOR_TYPE);
```

The generic data types are identical to those in the generic package SYNCHRONOUS_IMPRECISE_COMPUTATION. However, the generic subprograms are quite different. Not all of the generic subprograms in this asynchronous version are used to rendezvous with the computation task. The user of the generic package ASYNCHRONOUS_IMPRECISE_COMPUTATION must construct a task type that contains a single entry point. When this entry point is called, input data is passed to the task. After initialization, the task begins iterating and producing imprecise results. The task proceeds without any further interruption or rendezvous, asynchronously.

The generic procedure START_COMPUTATION is the procedure called by the imprecise computation mechanism to initialize the computation task. The computation task receives input, initializes, and then starts iterating. Procedure START_COMPUTATION requires two parameters. Parameter THE_COMPUTATION is a pointer to an active task. The necessary input data is passed via parameter INPUT. Because the computation task type has only a single entry point, START_COMPUTATION is the only generic subprogram needed to initiate a rendezvous.

By virtue of the definition of a rendezvous [23], an asynchronous approach to imprecise computations cannot utilize this synchronous mechanism. In the synchronous implementation, handler entry points are included in the computation task type. This is possible because the synchronous imprecise computation mechanism closely governs the executing computation task. However, in the asynchronous version, the computation task is turned loose. When a deadline is reached, the imprecise result must be immediately passed to a handler. For this reason, the handler routine is not part of the computation task type, but is a separate procedure. Therefore, the generic procedure HANDLE does not require the task pointer variable required by the synchronous handler. Also, the synchronous version includes a handler number which facilitates the use of entry families when the handler is an entry call. This parameter has not been included in the asynchronous version.

The generic procedure HANDLE requires two parameters. The parameter LAST_VALUE supplies the handler routine with the last imprecise result returned via an IMPRETURN call. Likewise, the parameter LAST_ERROR_INDICATOR provides a means of determining the precision of LAST_VALUE. Note that both of these parameters are of mode "in out". This is necessary because the asynchronous handler is a separate procedure and not a part of the task environment as it is in the synchronous version.

When the preceding generic types and generic subprograms are instantiated with appropriate data types and procedures at compilation time, a custom asynchronous imprecise computation package is created and placed in the user's library. This package contains the bundled procedures that form the crux of the asynchronous imprecise computation mechanism.

3.3.2 Procedures

In accordance with the theory of imprecise computations [11-13], there are two visible procedures in the generic package ASYNCHRONOUS_IMPRECISE_COMPUTATION. Like the synchronous version, the name of the procedure IMPRESULT has been changed to IMPCALL. The other procedure remains as IMPRETURN. The procedure declarations are defined in the generic package specification in the following manner:

```
procedure IMPCALL(THE_COMPUTATION : in out
                                      COMPUTATION_PTR;
                  DEADLINE        : in
                                      CALENDAR.TIME;
                  INPUT           : in
                                      INPUT_TYPE;
                  FINAL_RESULT    :     out
                                      RESULT_TYPE);

procedure IMPRETURN(INTERMEDIATE_RESULT : in
                                     RESULT_TYPE;
                    ERROR_INDICATOR     : in
                                     ERROR_INDICATOR_TYPE;
                    STOP_FLAG           : in out
                                     boolean);
```

Procedure IMPCALL requires four parameters. The parameter THE_COMPUTATION is a pointer to the imprecise computation task. Parameter DEADLINE specifies the absolute time when computation should cease. The computation task is initialized with the value of parameter INPUT. Lastly, the final result of the computation, whether precise or imprecise, is received via the parameter FINAL_RESULT.

Procedure IMPRETURN is called by the computation task in order to return an imprecise result and its associated error indicator. The first two parameters, INTERMEDIATE_RESULT and ERROR_INDICATOR, carry the imprecise result and error indicator from the computation task to the asynchronous imprecise computation mechanism. Unlike the IMPRETURN in the synchronous version, this IMPRETURN contains a third parameter. The parameter STOP_FLAG functions as a two-way communication flag between the computation task and the asynchronous imprecise computation mechanism. If the computation task achieves a precise result, it issues an IMPRETURN call with STOP_FLAG set to "true". If the deadline has occurred, the computation task is signalled to stop via STOP_FLAG when the next IMPRETURN call is issued. In this way, the asynchronous imprecise computation mechanism does not have to explicitly stop the computation task. It merely sets a flag which is communicated to the task when the task makes its next IMPRETURN call. The package body contains the code that implements these mechanisms.

In addition to the procedure bodies for IMPCALL and IMPRESULT, the ASYNCHRONOUS_IMPRECISE_COMPUTATION package body contains other variables and a task. These entities are not visible to the user of the package. They are only visible within the package body itself [23]. This package body, because it embodies the implementation of a concept, is quite different from the synchronous version. The following variables are included in the package body:

```
CURRENT_VALUE            : RESULT_TYPE;
```

```
CURRENT_ERROR_INDICATOR : ERROR_INDICATOR_TYPE;
STOP_COMPUTATION_FLAG   : boolean := FALSE;
```

The variables CURRENT_VALUE and CURRENT_ERROR_INDICATOR hold the last imprecise result and error indicator sent by the IMPRETURN call. These variables are updated by the procedure IMPRETURN in the course of computation or the procedure HANDLE when the deadline has expired. The variable STOP_COMPUTATION_FLAG is a boolean flag that holds the current state of the computation. The flag is initially set to "false", so the computation is not to be stopped. The flag will be set to "true" when the deadline expires or when the computation task achieves a precise result. If the deadline expires, the flag is set by the asynchronous imprecise computation mechanism. If a precise result is achieved, the flag is set during a call to procedure IMPRETURN. A local task is also contained in the package body of ASYNCHRONOUS_IMPRECISE_COMPUTATION.

While the computation task is iterating towards a precise result, it is necessary to have another task monitoring the system time as the deadline approaches. This monitor has been implemented as a task because it requires the use of task priorities. If this monitor were implemented as a called procedure, it could not be assigned a priority [23]. During the period of imprecise computation, there are two tasks in the application executing. The computation task is computing imprecise results while the monitor task is checking the deadline and then delaying. It is necessary for the monitor task to have a higher priority so that when the deadline occurs, the monitor task gets immediate control of the processor. The monitor task in the package body of ASYNCHRONOUS_IMPRECISE_COMPUTATION has the following task specification:

```
task TIMER is
    pragma PRIORITY(7);
    entry RUN_JOB(THE_JOB  : in out COMPUTATION_PTR;
                  INPUT    : in     INPUT_TYPE;
                  DEADLINE : in     CALENDAR.TIME);
end TIMER;
```

The monitor task has been called task TIMER to reflect its function. The first statement of the specification sets the task priority to 7, the highest priority allowed by the VADS software used for development [24]. It is imperative that the user include the following statement in the computation task:

```
pragma PRIORITY(0);
```

This will ensure that task TIMER can gain control of the priority-driven processor.

Task TIMER contains a single entry point called RUN_JOB. This entry point is called from procedure IMPCALL when it wants a particular computation task executed as an imprecise computation. Entry point RUN_JOB receives three parameters from procedure IMPCALL during the rendezvous. The parameter THE_JOB is a pointer to a computation task. The computation task is initialized with the information stored in INPUT. The parameter DEADLINE informs task TIMER of the point in time when a result is expected. The backbone of the asynchronous approach to imprecise computation is the body of task TIMER.

The task body of task TIMER from the package body of ASYNCHRONOUS_IMPRECISE_COMPUTATIONS has been implemented in the following manner:

```
task body TIMER is

    COMPUTATION_COMPLETED : boolean;
    TIME_HACK             : CALENDAR.TIME;
    TIME_LEFT             : float;
    DELAY_TIME            : DURATION;

begin
    accept RUN_JOB(THE_JOB  : in out COMPUTATION_PTR;
                   INPUT    : in     INPUT_TYPE;
                   DEADLINE : in     CALENDAR.TIME) do
        START_COMPUTATION(THE_JOB, INPUT);
        loop
            TIME_HACK := CALENDAR.CLOCK;
            TIME_LEFT := float(CALENDAR."-"(DEADLINE,
                                            TIME_HACK));
            DELAY_TIME := DURATION(TIME_LEFT / 2.0);
            if DELAY_TIME < DURATION'SMALL AND THEN
              DELAY_TIME > 0.0 then
                DELAY_TIME := 0.0;
            end if;
            if DELAY_TIME > 0.0 then
                delay DELAY_TIME;
            else
                STOP_COMPUTATION_FLAG := TRUE;
                HANDLE(CURRENT_VALUE,
                       CURRENT_ERROR_INDICATOR);
            end if;
            exit when STOP_COMPUTATION_FLAG;
        end loop;
    end RUN_JOB;
end TIMER;
```

The body of task TIMER is basically one rendezvous. Procedure IMPCALL calls the RUN_JOB entry point of task TIMER when an asynchronous imprecise computation task is to be run. Task TIMER then initializes and starts the computation task by calling the generic procedure START_COMPUTATION. After entering the main loop, the system clock is sampled and compared to the deadline. The time remaining is used to compute a delay amount. Task TIMER will suspend itself via the "delay" statement if sufficient time remains before deadline. If the deadline has expired, the flag STOP_COMPUTATION_FLAG will be set so that during the next IMPRETURN call the computation task will terminate itself. The generic procedure HANDLE will then be called and the loop exited. If the computation task achieves a precise result and subsequently signals via the procedure IMPRETURN, the flag STOP_COMPUTATION_FLAG will be set and the loop exited. When the loop is exited, the rendezvous completes, task TIMER terminates, and the final result is left stored in the variable CURRENT_VALUE.

Note that the variable DELAY_TIME is assigned a duration value that is only one-half of the time remaining before the deadline. This heuristic is necessary because of an anomaly with the "delay" statement. The statement

```
delay 1.0;
```

suspends the task for at least one second. However, there is no guarantee on the upper bound of the delay. While task

TIMER is delaying itself, the computation task has control of the processor. When TIMER's delay is complete, task TIMER is ready to be run again. Because TIMER was given a higher priority than the computation task, task TIMER should gain control of the processor. However, the scheduler only checks the list of ready tasks at a specified frequency. VADS checks at one second intervals [24]. This time slice is much too large for real-time systems. Digital Equipment Corporation's (DEC) VAX Ada provides the statement

```
pragma TIME_SLICE(static_expression);
```

where static_expression is a duration amount in seconds [8]. The DEC manual [8] points out that the amount of scheduling overhead needed to support round-robin task scheduling increases as the value of a time slice decreases. The minimum recommended time slice is 0.01 seconds. A test was constructed to evaluate this feature and the effects of background tasks on the delay statement.

In order to determine the effect of background tasks on the delay statement, the procedures DELAY_TEST and DELAY_TEST_NO_TASK were designed. These procedures were run on a VADS computer system and then augmented with

```
pragma TIME_SLICE(0.01 or 1.00);
```

and run on DEC Ada machines to investigate the best performance (0.01) and to compare the DEC Ada run-time system with the VADS run-time system (1.00). The procedure DELAY_TEST_NO_TASK was constructed in the following way:

```
with CALENDAR;           use CALENDAR;
with TEXT_IO;            use TEXT_IO;
with FLOAT_IO;           use FLOAT_IO;
procedure DELAY_TEST_NO_TASK is

   HACK1, HACK2 : time;
   TOTAL        : float := 0.0;


begin
    for COUNT in 1 .. 100 loop
        HACK1 := clock;
        delay 1.0;
        HACK2 := clock;
        TOTAL := TOTAL + float(HACK2 - HACK1);
        put("Time difference for 1 second delay=>");
        put(float(HACK2 - HACK1));
        put_line(" secs.");
    end loop;
    new_line(3);
    put("AVERAGE DELAY WAS => ");
    put(TOTAL / 100.0);
    put_line(" secs.");
end DELAY_TEST_NO_TASK;
```

This procedure merely samples the system clock before and after a one-second delay statement. The actual delay is averaged over 100 delay statements. The procedure DELAY_TEST includes a background task:

```
with CALENDAR;           use CALENDAR;
with TEXT_IO;            use TEXT_IO;
with FLOAT_IO;           use FLOAT_IO;
procedure DELAY_TEST is

    pragma PRIORITY(7);

    HACK1, HACK2 : time;
    TOTAL        : float := 0.0;

    task EAT is
        pragma PRIORITY(0);
        entry STOP;
```

```
        end EAT;
        task body EAT is
            COUNT    : integer := 0;
            FINISHED : boolean := false;
        begin
            loop
                select
                    accept STOP do
                        FINISHED := true;
                    end STOP;
                else
                    COUNT := COUNT + 1;
                end select;
                exit when FINISHED;
            end loop;
        end EAT;

    begin
        for COUNT in 1 .. 100 loop
            HACK1 := clock;
            delay 1.0;
            HACK2 := clock;
            TOTAL := TOTAL + float(HACK2 - HACK1);
            put("Time difference for 1 second delay=>");
            put(float(HACK2 - HACK1));
            put_line(" secs.");
        end loop;
        EAT.STOP;
        new_line(3);
        put("AVERAGE DELAY WAS => ");
        put(TOTAL / 100.0);
        put_line(" secs.");
    end DELAY_TEST;
```

Note that the task EAT has a lower priority, thus simulating the asynchronous imprecise computation task. Both of these procedures were run on a VAX-11/780 under VADS, a VAX-11/780 under DEC VAX Ada, and a VAX 8700 under DEC VAX Ada. Additionally, the DEC VAX Ada tests incorporated the time slice pragma. The average delays, in seconds, for a one second delay statement are summarized in Table 1. The VERDIX system did not perform well in comparison to the DEC configurations. Even with TIME_SLICE set to one second in an

CONFIGURATION	TASK	NO TASK
VAX Ada TIME_SLICE(1.00)		
Average	1.00995(8700) 1.01796(11/780)	1.00995(8700) 1.00995(11/780)
Standard Deviation	0.0(8700) 0.01866(11/780)	0.0(8700) 0.0(11/780)
Variance	0.0(8700) 0.00035(11/780)	0.0(8700) 0.0(11/780)
VAX Ada TIME_SLICE(0.01)		
Average	1.00995(8700) 1.01676(11/780)	1.00995(8700) 1.01005(11/780)
Standard Deviation	0.0(8700) 0.00469(11/780)	0.0(8700) 0.001(11/780)
Variance	0.0(8700) 0.00002(11/780)	0.0(8700) 0.000001(11/780)
VERDIX Ada Development System		
Average	1.84434(11/780)	1.27862(11/780)
Standard Deviation	1.13453(11/780)	0.23124(11/780)
Variance	1.28716(11/780)	0.05347(11/780)

Table 1. Comparison of Configurations and Task/No Task Option

effort to mimic the VADS inherent time slice the DEC Ada run-time system clearly performed better.

It is apparent that some Ada run-time systems are better geared for real-time applications. A serious real-time designer would not implement his hard, real-time system in an Ada development system such as VADS. In proving the viability of the package ASYNCHRONOUS_IMPRECISE_COMPUTATION, it became obvious that the testing would have to be accomplished within the realm of a genuine, real-time Ada development system. The package body however still contains standard Ada code.

The procedure body for procedure IMPCALL in the package body of ASYNCHRONOUS_IMPRECISE_COMPUTATION is implemented in the following manner:

```
procedure IMPCALL(THE_COMPUTATION : in out
                                      COMPUTATION_PTR;
                  DEADLINE        : in
                                      CALENDAR.TIME;
                  INPUT           : in
                                      INPUT_TYPE;
                  FINAL_RESULT    :    out
                                      RESULT_TYPE) is

    begin
        TIMER.RUN_JOB(THE_COMPUTATION,
                      INPUT,
                      DEADLINE);
        FINAL_RESULT := CURRENT_VALUE;
    end IMPCALL;
```

Procedure IMPCALL first calls the RUN_JOB entry point of task TIMER, passing it a pointer to the computation task to run, the initialization input, and the deadline. Procedure IMPCALL remains in the rendezvous with task TIMER until a final result is produced. Remember, when task TIMER terminates, the final result is left in the variable CURRENT_VALUE. Procedure IMPCALL copies the final result into its output variable FINAL_RESULT and then completes.

The body of procedure IMPRETURN is implemented in the package body of ASYNCHRONOUS_IMPRECISE_COMPUTATION in the following way:

```
    procedure IMPRETURN(INTERMEDIATE_RESULT : in
                                  RESULT_TYPE;
                        ERROR_INDICATOR     : in
                                  ERROR_INDICATOR_TYPE;
                        STOP_FLAG           : in out
                                  boolean) is

    begin
        if not STOP_COMPUTATION_FLAG then
           CURRENT_VALUE           := INTERMEDIATE_RESULT;
           CURRENT_ERROR_INDICATOR:= ERROR_INDICATOR;
        end if;
        if not STOP_FLAG then
            STOP_FLAG := STOP_COMPUTATION_FLAG;
        else
            STOP_COMPUTATION_FLAG := STOP_FLAG;
        end if;
    end IMPRETURN;
```

The first action IMPRETURN takes is checking the state of the flag variable STOP_COMPUTATION_FLAG that is local to the package body. If this flag has not been set by task TIMER, then the deadline has not occurred and the computation task should continue. The local variables CURRENT_VALUE and CURRENT_ERROR_INDICATOR are updated accordingly. If the flag has been set by task TIMER, then the deadline has occurred and no further updates to CURRENT_VALUE and CURRENT_ERROR_INDICATOR are required. If the incoming parameter STOP_FLAG is false, then the IMPRETURN call is merely returning an imprecise result and its error indicator. Parameter STOP_FLAG is set to the state of STOP_COMPUTATION_FLAG so that the computation task is informed when a deadline passes. If the parameter STOP_FLAG is true, then the computation task is signalling that the computation task has completed. STOP_COMPUTATION_FLAG is set to true which in turn signals task TIMER to terminate.

The complete package specification and package body for ASYNCHRONOUS_IMPRECISE_COMPUTATION can be found in Appendix B. Figure 2 presents the asynchronous imprecise computation mechanism in a graphical manner, using the symbols outlined in [5]. This approach to imprecise computations has been implemented in standard Ada code and should compile on any validated compiler. However, this approach requires an adequate run-time system to perform correctly. Actual imprecise computation examples using this package are given in the following section.

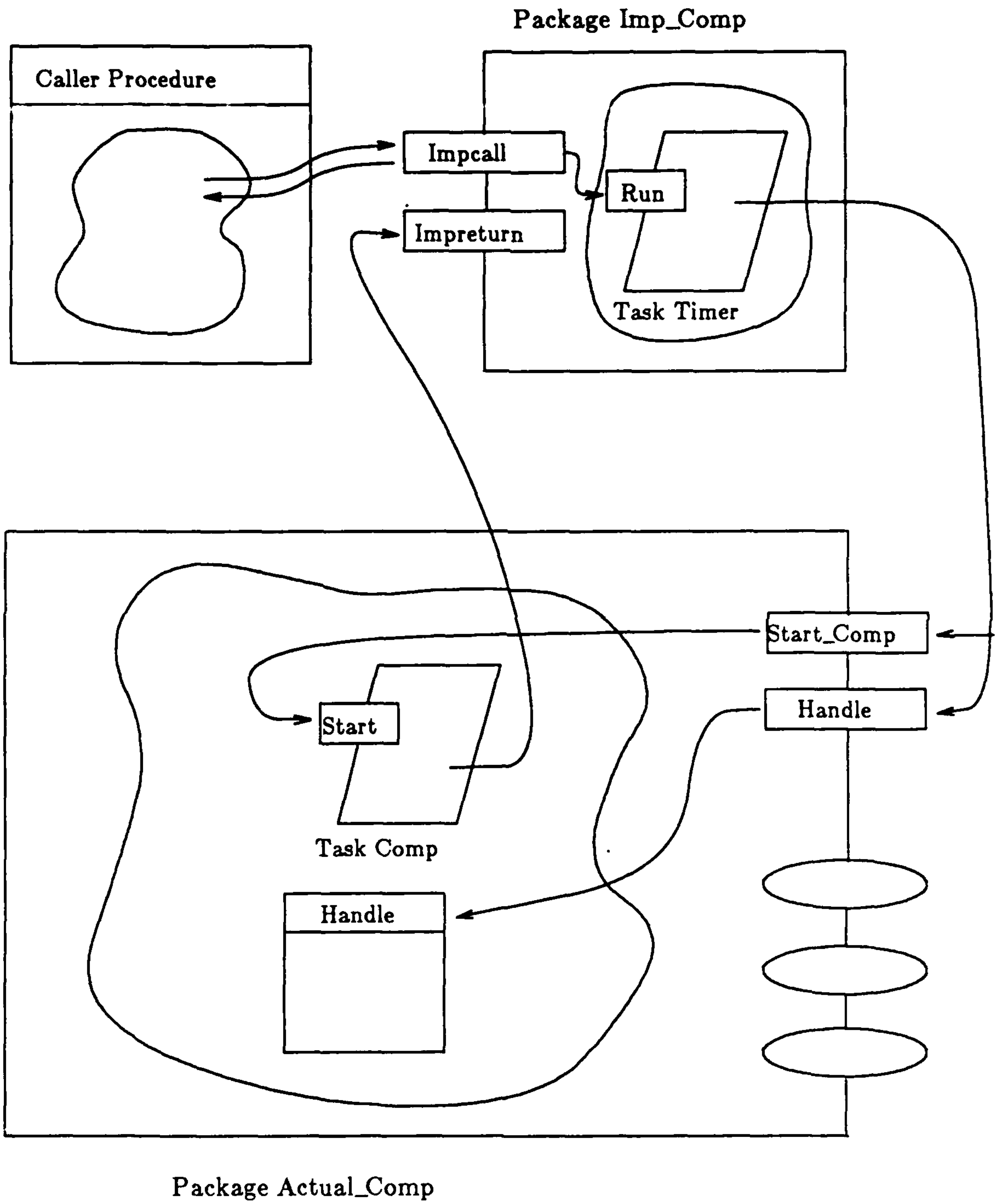


Figure 2. Asynchronous Imprecise Computation

4 Imprecise Computation Examples

The examples in this section demonstrate how the generic packages SYNCHRONOUS_IMPRECISE_COMPUTATION and ASYNCHRONOUS_IMPRECISE_COMPUTATION are used to construct imprecise computation applications.

4.1 Monte Carlo Simulation

The Monte Carlo method can be used to simulate a myriad of problems. Theoretical examples include the solution of partial differential equations, the evaluation of multiple integrals, and the study of particle diffusion [9]. Practical examples include the simulation of industrial and economic problems, the simulation of biomedical systems, and the simulation of war strategies and tactics [17]. The Monte Carlo method is based on the general idea of using sampling to estimate a desired result [17].

The area of a circle can be computed by the Monte Carlo method [17]. The idea is to construct a square about the circle such that the square encloses and is tangent to the circle. Accordingly, the square has sides equal in length to the diameter of the circle. Then, random coordinate pairs are generated that are within the square. Each coordinate pair is tested to determine if it is within the circle. The total number of coordinate pairs generated are counted and divided into the number of coordinate pairs that fell within the boundary of the circle. This fraction is then multiplied by the area of the square to yield an estimate of the area of the circle.

This Monte Carlo method of determining the area of a circle has been used to create synchronous and asynchronous imprecise computation examples. The use of the generic packages *SYNCHRONOUS_IMPRECISE_COMPUTATION* and *ASYNCHRONOUS_IMPRECISE_COMPUTATION* is clearly demonstrated, along with the necessary user-written code.

4.1.1 Synchronous Circle Imprecise Computation

The first file constructed contains the data types, computation task type, and procedure declarations that will be used to instantiate *SYNCHRONOUS_IMPRECISE_COMPUTATION*. Because this file contains related types and procedures, it is fashioned as a package specification. Its package body will contain the task and procedure bodies. The package specification for SYNCHRONOUS_CIRCLE_COMPUTATION includes the following declarations:

```
subtype RESULT_TYPE is float;

subtype ERROR_TYPE is integer;

type INPUT_TYPE is record
    LOOPS_TO_COMPLETE : integer;
    RADIUS            : float;
end record;

task type TEST_TASK is
    entry INITIALIZE(INPUT : in INPUT_TYPE);
    entry COMPUTE(COMPUTATION_COMPLETE :out boolean);
    entry HANDLER(1 .. 2)(LAST_RESULT:in RESULT_TYPE;
                          LAST_ERROR :in ERROR_TYPE);
    entry STOP;
```

```
end TEST_TASK;

type TEST_PTR is access TEST_TASK;

procedure INITIALIZE(THE_TASK : in TEST_PTR;
                     INPUT    : in INPUT_TYPE);

procedure COMPUTE(THE_TASK                : in TEST_PTR;
                  COMPUTATION_COMPLETE :out boolean);

procedure HANDLE(THE_TASK             :in TEST_PTR;
                 HANDLER_NUMBER       :in integer;
                 LAST_VALUE           :in RESULT_TYPE;
                 LAST_ERROR_INDICATOR:in ERROR_TYPE);

procedure STOP(THE_TASK : in TEST_PTR);
```

The result of the computation will be a floating point value representing an estimate of the area of a circle, so RESULT_TYPE is made a subtype of float. To monitor the precision of the imprecise result, a counter will count the number of random coordinate pairs generated. Therefore, ERROR_TYPE is created as a subtype of integer. At initialization, the computation task will need two pieces of information. Represented by the elements in type INPUT_TYPE, this information is the number of random coordinate pairs to generate before a precise result is achieved, and the radius of the circle. The task type TEST_TASK is the computation task. It includes the necessary entry points to initialize the task, cause one iteration, handle an imprecise result, and stop the task. Note that entry HANDLER has been implemented as a family of entries. The type TEST_P"R is a pointer to task type TEST_TASK. The four procedures INITIALIZE, COMPUTE, HANDLE, and STOP are required to allow the imprecise computation mechanism, which has no prior knowledge of the computation task type, to call specific entry points within the computation task. When this package specification is compiled, it is entered into the user's Ada library where it can be further referenced.

Now that the required data types, task type, and procedures have been declared, an instantiation of the generic package SYNCHRONOUS_IMPRECISE_COMPUTATION can be made. The declarations in the package specification of SYNCHRONOUS_CIRCLE_COMPUTATION will be used to create the package SYNCHRONOUS_CIRCLE_IMPRECISE_COMPUTATION in the following manner:

```
with SYNCHRONOUS_CIRCLE_COMPUTATION;
use SYNCHRONOUS_CIRCLE_COMPUTATION;
with SYNCHRONOUS_IMPRECISE_COMPUTATION;
package SYNCHRONOUS_CIRCLE_IMPRECISE_COMPUTATION is
    new SYNCHRONOUS_IMPRECISE_COMPUTATION
        (COMPUTATION            => TEST_TASK,
         COMPUTATION_PTR        => TEST_PTR,
         RESULT_TYPE            => RESULT_TYPE,
         ERROR_INDICATOR_TYPE   => ERROR_TYPE,
         INPUT_TYPE             => INPUT_TYPE,
         INITIALIZE             => INITIALIZE,
         COMPUTE                => COMPUTE,
         HANDLE                 => HANDLE,
         STOP                   => STOP);
```

When this file is compiled, a new synchronous imprecise computation package is created that includes the declarations in SYNCHRONOUS_CIRCLE_COMPUTATION's package specification. The next file to compose and compile is the package body. The new imprecise computation package is instantiated before the computation package body is compiled for a crucial reason.

The imprecise computation mechanism employs a circular calling pattern. Procedure IMPCALL calls procedures that call entry points of the computation task. In the meantime, the computation task is calling procedure IMPRETURN with imprecise results and error indicators. When the new package is instantiated after the declarations are made in the computation package specification, the new IMPCALL is supplied with the procedure declarations it needs to get its job done. The implementation, or body of these procedures is of no consequence to IMPCALL. After instantiation, a valid IMPRETURN exists in the new imprecise computation package. At this point, the computation task body which relies on IMPRETURN can be coded. In this way, a single package can house the synchronous imprecise computation mechanism, even though circular calling exists.

The package body for SYNCHRONOUS_CIRCLE_COMPUTATION contains the following procedure bodies:

```
procedure INITIALIZE(THE_TASK : in TEST_PTR;
                     INPUT    : in INPUT_TYPE) is
    begin
        THE_TASK.INITIALIZE(INPUT);
    end INITIALIZE;

procedure COMPUTE(THE_TASK             : in TEST_PTR;
                  COMPUTATION_COMPLETE : out
                                          boolean) is
    begin
        THE_TASK.COMPUTE(COMPUTATION_COMPLETE);
    end COMPUTE;

procedure HANDLE(THE_TASK             : in TEST_PTR;
                 HANDLER_NUMBER       : in integer;
```

```
                    LAST_VALUE            : in RESULT_TYPE;
                    LAST_ERROR_INDICATOR : in ERROR_TYPE) is
        begin
            THE_TASK.HANDLER(HANDLER_NUMBER)
                         (LAST_VALUE,
                          LAST_ERROR_INDICATOR);
        end HANDLE;

    procedure STOP(THE_TASK : in TEST_PTR) is
        begin
            THE_TASK.STOP;
        end STOP;
```

These procedures call their respective entry points in the computation task. Although the procedure names and the entry points have exact or similar names, the names are independent of the synchronous imprecise computation mechanism. The only names it needs are the names used to instantiate SYNCHRONOUS_CIRCLE_IMPRECISE_COMPUTATION. The task body for task type TEST_TASK has the following structure:

```
    task body TEST_TASK is

        ... local variable declarations ...

    begin
        accept INITIALIZE(INPUT : in INPUT_TYPE) do
            ... initialize variables with input ...
        end INITIALIZE;
        loop
            select
                accept COMPUTE(COMPUTATION_COMPLETE : out
                                                 boolean) do
                    ... generate random coord pairs ...
                    ... check circle boundary ...
                    ... compute area ...
                    ... check if precise,
                          set COMPUTATION_COMPLETE ...
                    ... IMPRETURN ...
                end COMPUTE;
            or
                accept HANDLER(1)(LAST_RESULT : in
                                                  RESULT_TYPE;
                                  LAST_ERROR   : in
```

```
                                          ERROR_TYPE) do
                  ... handler #1 code ...
                  ... IMPRETURN ...
              end HANDLER;
          or
              accept HANDLER(2)(LAST_RESULT : in
                                          RESULT_TYPE;
                                LAST_ERROR  : in
                                          ERROR_TYPE) do
                  ... handler #2 code ...
                  ... IMPRETURN ...
              end HANDLER;
          or
              accept STOP do
                  FINISHED := true;
              end STOP;
          end select;
          exit when FINISHED;
      end loop;
  end TEST_TASK;
```

After initialization, the task continuously loops through a select statement. The select statement causes the task to wait for a call to any one of the entry points. The COMPUTE entry point contains the code that implements the random sampling of the Monte Carlo method and the area calculation code. The HANDLER entry family contains the code necessary to further manipulate the final imprecise result. The loop is exited by a rendezvous with the STOP entry point. No other entities are required in the computation package body. After this package body is compiled and subsequently entered into the user's Ada library, a self-contained, operational synchronous imprecise computation package exists and can be used. The following VAX Ada program exercises the synchronous package:

```
  with SYNCHRONOUS_CIRCLE_COMPUTATION;
```

```
use  SYNCHRONOUS_CIRCLE_COMPUTATION;
with SYNCHRONOUS_CIRCLE_IMPRECISE_COMPUTATION;
use  SYNCHRONOUS_CIRCLE_IMPRECISE_COMPUTATION;
with CALENDAR;                     use CALENDAR;
with TEXT_IO;                      use TEXT_IO;
with FLOAT_TEXT_IO;                use FLOAT_TEXT_IO;
with INTEGER_TEXT_IO;              use INTEGER_TEXT_IO;
procedure SYNCHRONOUS_CIRCLE_TEST is

    MY_TASK_PTR : TEST_PTR       := new TEST_TASK;
    DEAD        : CALENDAR.TIME;
    RESULT      : RESULT_TYPE;
    COMP_TIME   : float;
    MY_INPUT    : SYNCHRONOUS_CIRCLE_COMPUTATION.
                      INPUT_TYPE;
begin
    put("Enter the circle radius => ");
    get(MY_INPUT.RADIUS);
    put("Enter number of iterations to complete =>");
    get(MY_INPUT.LOOPS_TO_COMPLETE);
    put("Enter computation duration in seconds =>");
    get(COMP_TIME);
    put_line("Synchronous CIRCLE TEST starting...");
    DEAD := CALENDAR.CLOCK + DURATION(COMP_TIME);
    SYNCHRONOUS_CIRCLE_IMPRECISE_COMPUTATION.
        IMPCALL(MY_TASK_PTR,
                1,
                DEAD,
                MY_INPUT,
                RESULT);
    put("TEST ending...  RESULT => ");
    put(RESULT, EXP => 0, AFT => 2);
    new_line;
end SYNCHRONOUS_CIRCLE_TEST;
```

Once the computation package is built and compiled correctly, using it is quite simple. After the input parameters are determined, the imprecise computation is run by merely invoking the IMPCALL procedure in the newly instantiated SYNCHRONOUS_CIRCLE_IMPRECISE_COMPUTATION package. The complete file listings for this example are located in Appendix C.

4.1.2 Asynchronous Circle Imprecise Computation

A similar sequence of files is used to build an asynchronous imprecise computation application because the asynchronous approach also relies on a circular calling mechanism. The first file constructed contains data types, task type, and procedure declarations required for instantiation of generic package ASYNCHRONOUS_IMPRECISE_COMPUTATION. These declarations are located in the package specification for ASYNCHRONOUS_CIRCLE_COMPUTATION in the following format:

```
subtype RESULT_TYPE is float;

subtype ERROR_TYPE is integer;

type INPUT_TYPE is record
    LOOPS_TO_COMPLETE : integer;
    RADIUS            : float;
end record;

task type TEST_TASK is
    pragma PRIORITY(0);
    entry START_COMPUTATION(INPUT : in INPUT_TYPE);
end TEST_TASK;

type TEST_PTR is access TEST_TASK;

procedure START_COMPUTATION(THE_TASK :in TEST_PTR;
                            INPUT    :in INPUT_TYPE);

procedure HANDLE(LAST_VALUE           : in out
                                         RESULT_TYPE;
                 LAST_ERROR_INDICATOR : in out
                                         ERROR_TYPE);
```

The subtypes RESULT_TYPE, ERROR_TYPE, and INPUT_TYPE are the same as in the synchronous example. However, the task type TEST_TASK is quite different. The task must contain the priority pragma statement with a priority lower than that of the TIMER task in ASYNCHRONOUS_IMPRECISE_COMPUTATION. Task type TEST_TASK contains a single entry point where the task is initialized and then turned loose. The type TEST_PTR remains as an access type pointing to TEST_TASK. The procedure START_COMPUTATION is required to allow the asynchronous imprecise computation mechanism, which has no knowledge of the internal structure of the computation task, to indirectly call the START_COMPUTATION entry point. The procedure HANDLE is not affiliated with the computation task as it is in the synchronous version, but accomplishes the same function of manipulating the final imprecise result. The compiled ASYNCHRONOUS_CIRCLE_COMPUTATION package specification is entered into the user's Ada library where it can be further referenced by the application.

Once the required data types, task type, and procedures have been declared, a new package can be created by instantiating the generic package ASYNCHRONOUS_IMPRECISE_COMPUTATION. This is accomplished in the following file:

```
with ASYNCHRONOUS_CIRCLE_COMPUTATION;
use  ASYNCHRONOUS_CIRCLE_COMPUTATION;
with ASYNCHRONOUS_IMPRECISE_COMPUTATION;
package ASYNCHRONOUS_CIRCLE_IMPRECISE_COMPUTATION is
    new ASYNCHRONOUS_IMPRECISE_COMPUTATION
        (COMPUTATION          => TEST_TASK,
         COMPUTATION_PTR      => TEST_PTR,
         RESULT_TYPE          => RESULT_TYPE,
         ERROR_INDICATOR_TYPE => ERROR_TYPE,
         INPUT_TYPE           => INPUT_TYPE,
         START_COMPUTATION    => START_COMPUTATION,
         HANDLE               => HANDLE);
```

When this file is compiled, a new asynchronous imprecise computation package is created. This new package contains the declarations from the package specification of ASYNCHRONOUS_CIRCLE_COMPUTATION substituted in for the generic parameters. The next step is to implement the body of the computation package.

The package body of ASYNCHRONOUS_CIRCLE_COMPUTATION contains the following procedure bodies:

```
procedure START_COMPUTATION(THE_TASK : in TEST_PTR;
                            INPUT    : in INPUT_TYPE) is
    begin
        THE_TASK.START_COMPUTATION(INPUT);
    end START_COMPUTATION;

procedure HANDLE(LAST_VALUE           : in out
                                         RESULT_TYPE;
                 LAST_ERROR_INDICATOR : in out
                                         ERROR_TYPE) is
    begin
        ... handler routine ...
    end HANDLE;
```

Procedure START_COMPUTATION merely calls THE_TASK at the START_COMPUTATION entry point. During the rendezvous, the initialization INPUT is passed to the compute task. Procedure HANDLE is a standalone procedure that manipulates the final imprecise result. Also included in the computation package body is the body of task type TEST_TASK. It has the following structure:

```
task body TEST_TASK is

    ... local variable declarations ...

begin
    accept START_COMPUTATION(INPUT :in INPUT_TYPE) do
        ... initialize local variables ...
    end START_COMPUTATION;
    delay DURATION'SMALL;
```

```
        loop
            ... generate random coordinate pairs ...
            ... test boundary of circle ...
            ... compute area ...
            ... check if precise, set FINISHED ...
            ... IMPRETURN ...
            exit when FINISHED;
        end loop;
    end TEST_TASK;
```

The purpose of the delay statement with the minimal amount of delay is to allow the TIMER task to regain immediate control of the processor after the rendezvous with TEST_TASK. The delay statement causes TEST_TASK to be blocked and allows the higher priority TIMER task to execute. After the TIMER task determines its delay amount and suspends itself, the task TEST_TASK regains control of the processor and proceeds with the computation. The TEST_TASK loop is not exited until it achieves a precise result or is signalled to exit via the IMPRETURN call. No other entities are required in the computation package body. After this package is compiled and entered into the user's library, a fully operational asynchronous imprecise computation mechanism is available by referencing ASYNCHRONOUS_CIRCLE_IMPRECISE_COMPUTATION. This new package is used in the following VAX Ada procedure:

```
with ASYNCHRONOUS_CIRCLE_COMPUTATION;
use  ASYNCHRONOUS_CIRCLE_COMPUTATION;
with ASYNCHRONOUS_CIRCLE_IMPRECISE_COMPUTATION;
use  ASYNCHRONOUS_CIRCLE_IMPRECISE_COMPUTATION;
with CALENDAR;                          use CALENDAR;
with TEXT_IO;                           use TEXT_IO;
with FLOAT_TEXT_IO;                     use FLOAT_TEXT_IO;
with INTEGER_TEXT_IO;                   use INTEGER_TEXT_IO;
procedure ASYNCHRONOUS_CIRCLE_TEST is
```

```
pragma TIME_SLICE(0.01);

    MY_TASK_PTR : TEST_PTR       := new TEST_TASK;
    DEAD        : CALENDAR.TIME;
    RESULT      : RESULT_TYPE;
    COMP_TIME   : float;
    MY_INPUT    : ASYNCHRONOUS_CIRCLE_COMPUTATION.
                       INPUT_TYPE;
begin
    put("Enter the circle radius => ");
    get(MY_INPUT.RADIUS);
    put("Enter number of iterations to complete =>");
    get(MY_INPUT.LOOPS_TO_COMPLETE);
    put("Enter computation duration in seconds => ");
    get(COMP_TIME);
    put_line("Asynchronous CIRCLE TEST starting...");
    DEAD := CALENDAR.CLOCK + DURATION(COMP_TIME);
    ASYNCHRONOUS_CIRCLE_IMPRECISE_COMPUTATION.
        IMPCALL(MY_TASK_PTR,
                DEAD,
                MY_INPUT,
                RESULT);
    put("CIRCLE TEST ending...CIRCLE AREA RESULT=>");
    put(RESULT, EXP => 0, AFT => 2); new_line;
end ASYNCHRONOUS_CIRCLE_TEST;
```

Like its synchronous version, the asynchronous imprecise computation package is quite easy to use. Once compiled correctly, it can be accessed by simply invoking IMPCALL with the necessary parameters. A complete listing of the example files for asynchronously computing the area of a circle can be found in Appendix D.

4.2 Iterative Numerical Methods

Iterative numerical methods involve the repeated application of an operator. These methods include Newton's method (nonlinear equations), the Jacobi method (linear equations), and the Newton divided-difference method (infinite series approximation) among others [15,16]. This example demonstrates another use of the synchronous and asynchronous approaches in implementing an imprecise computation application. SYNCHRONOUS_IMPRECISE_COMPUTATION and ASYNCHRONOUS_IMPRECISE_COMPUTATION are generic packages used to implement a synchronous and an asynchronous linear system of equations solver that utilizes the Jacobi method.

4.2.1 Synchronous Jacobi Imprecise Computation

The sequence of file generation and compilation is identical to the previous example. The first file generated is the package specification. This file contains the data types, task type, and procedure declarations that will be used to instantiate a custom synchronous imprecise computation package. The following declarations are used for the Jacobi method:

```
N : constant integer := 3;

type RESULT_TYPE is array(1 .. N) of float;

subtype ERROR_TYPE is integer;

type COEFFICIENT_TYPE is array(1 .. N, 1 .. N) of float;

type INPUT_TYPE is record
    COEFFICIENTS       : COEFFICIENT_TYPE;
    RIGHT_HAND_SIDE    : RESULT_TYPE;
    XOLD               : RESULT_TYPE;
    TOL                : float;
end record;

task type TEST_TASK is
    entry INITIALIZE(INPUT : in INPUT_TYPE);
    entry COMPUTE(COMPUTATION_COMPLETE :out boolean);
    entry HANDLER(1 .. 2)(LAST_RESULT:in RESULT_TYPE;
                          LAST_ERROR :in ERROR_TYPE);
    entry STOP;
```

```
end TEST_TASK;

type TEST_PTR is access TEST_TASK;

procedure INITIALIZE(THE_TASK : in TEST_PTR;
                     INPUT    : in INPUT_TYPE);

procedure COMPUTE(THE_TASK             : in TEST_PTR;
                  COMPUTATION_COMPLETE : out boolean);

procedure HANDLE(THE_TASK             : in TEST_PTR;
                 HANDLER_NUMBER       : in integer;
                 LAST_VALUE           : in RESULT_TYPE;
                 LAST_ERROR_INDICATOR : in ERROR_TYPE);

procedure STOP(THE_TASK : in TEST_PTR);
```

The integer constant N represents the number of equations in the linear system. Likewise, N also represents the number of coefficients in each equation. The type RESULT_TYPE indicates that a solution vector with N floating point components will be the result of the computation. The subtype ERROR_TYPE is defined as an integer, for the error will be represented by an integer counter indicating the number of iterations accomplished. The type COEFFICIENT_TYPE defines an N by N matrix of floating point values. This type is not directly used in instantiation, but is used in the definition of the input to the computation. Type INPUT_TYPE is a record type containing 4 fields. The field COEFFICIENTS is an N by N matrix containing the coefficients of the equations in the linear system. The right hand side of these equations is stored in the field RIGHT_HAND_SIDE. The field XOLD contains an initial guess at the solution vector. This gives the Jacobi method a place to start. The field TOL is the tolerance used to determine whether a new solution vector, when compared to the previous one, can be considered a precise result. The task type TEST_TASK contains the appropriate entry calls required by the synchronous imprecise computation mechanism to initialize the task, cause an iteration of the computation, handle an imprecise result, and stop the task. A pointer type to this task type is defined as type TEST_PTR. Finally, the procedures INITIALIZE, COMPUTE, HANDLE, and STOP are declared so that the synchronous imprecise computation mechanism, when instantiated with these declarations, can call the entry points of a task. These procedures are necessary because the synchronous mechanism has no prior knowledge of the task TEST_TASK or its structure. Once this package specification is compiled, it is entered into the user's Ada library where it can be further referenced by the application.

With the data types, task type, and procedures declared in the package specification, an instantiation of the generic package SYNCHRONOUS_IMPRECISE_COMPUTATION can be made. The declarations in the SYNCHRONOUS_JACOBI_COMPUTATION package specification are used to create the new package SYNCHRONOUS_JACOBI_IMPRECISE_COMPUTATION in the following way:

```
with SYNCHRONOUS_JACOBI_COMPUTATION;
use SYNCHRONOUS_JACOBI_COMPUTATION;
with SYNCHRONOUS_IMPRECISE_COMPUTATION;
package SYNCHRONOUS_JACOBI_IMPRECISE_COMPUTATION is
    new SYNCHRONOUS_IMPRECISE_COMPUTATION
```

```
(COMPUTATION             => TEST_TASK,
 COMPUTATION_PTR         => TEST_PTR,
 RESULT_TYPE             => RESULT_TYPE,
 ERROR_INDICATOR_TYPE    => ERROR_TYPE,
 INPUT_TYPE              => INPUT_TYPE,
 INITIALIZE              => INITIALIZE,
 COMPUTE                 => COMPUTE,
 HANDLE                  => HANDLE,
 STOP                    => STOP);
```

After this file is compiled, a new synchronous imprecise computation package exists in the user's library. This new package contains the same mechanism, but with the new declarations substituted in for the generic parameters. The next file defines the bodies for the task type and the procedures declared in the package specification.

The package body of SYNCHRONOUS_JACOBI_COMPUTATION contains the implementation details of the task type and procedure bodies. The procedure bodies are implemented in the following way:

```
procedure INITIALIZE(THE_TASK : in TEST_PTR;
                     INPUT    : in INPUT_TYPE) is
    begin
        THE_TASK.INITIALIZE(INPUT);
    end INITIALIZE;

procedure COMPUTE(THE_TASK                : in TEST_PTR;
                  COMPUTATION_COMPLETE : out boolean) is
   begin
        THE_TASK.COMPUTE(COMPUTATION_COMPLETE);
    end COMPUTE;

procedure HANDLE(THE_TASK              :in TEST_PTR;
                 HANDLER_NUMBER        :in integer;
                 LAST_VALUE            :in RESULT_TYPE;
                 LAST_ERROR_INDICATOR:in ERROR_TYPE) is
    begin
        THE_TASK.HANDLER(HANDLER_NUMBER)
            (LAST_VALUE, LAST_ERROR_INDICATOR);
    end HANDLE;
```

```
    procedure STOP(THE_TASK : in TEST_PTR) is
        begin
            THE_TASK.STOP;
        end STOP;
```

These procedures call their respective entry points in the Jacobi computation task. The procedures and entry points can have any names. The only requirement is that the procedures used to instantiate the new synchronous imprecise computation package call the appropriate entry point in the Jacobi computation task. The task body for task type TEST_TASK has the following structure:

```
    task body TEST_TASK is

        ... local variable declarations ...

    begin
        accept INITIALIZE(INPUT : in INPUT_TYPE) do
            ... initialize local variables with input ...
            ... normalize coefficient matrix ...
        end INITIALIZE;
        loop
            select
                accept COMPUTE(COMPUTATION_COMPLETE :
                                            out boolean) do
                        ... compute new solution vector
                            using method in [15,16] ...
                        ... find absolute difference
                            between old and new elements..
                        ... let present estimate be
                            improved estimate ...
                        ... report current result
                            with IMPRETURN ...
                end COMPUTE;
            or
                accept HANDLER(1)
                        (LAST_RESULT : in RESULT_TYPE;
                         LAST_ERROR  : in ERROR_TYPE) do
                    ... handler #1 code ...
                    ... IMPRETURN ...
                end HANDLER;
            or
                accept HANDLER(2)
```

```
                    (LAST_RESULT : in RESULT_TYPE;
                     LAST_ERROR  : in ERROR_TYPE) do
                  ... handler #2 code ...
                  ... IMPRETURN ...
              end HANDLER;
          or
              accept STOP do
                  FINISHED := true;
              end STOP;
          end select;
          exit when FINISHED;
      end loop;
  end TEST_TASK;
```

During initialization, the values of the input record are copied to local variables. The input record fields cannot be used directly in the computation task because their scope is limited to the INITIALIZE rendezvous. The coefficient matrix is then normalized and the rendezvous is complete. The task then enters a loop that contains a select and an exit statement. The task waits for an entry call, performs the operation in the rendezvous, and then checks if it should exit the loop. The COMPUTE entry point contains the implementation of the Jacobi method as specified in [15,16]. The HANDLER entry family contains the code necessary to further manipulate the final imprecise result. The STOP entry point sets the flag that triggers the loop exit. No other entities are required in the SYNCHRONOUS_JACOBI_COMPUTATION package body. After this package body is compiled and entered into the user's Ada library, a self-contained synchronous imprecise Jacobi computation mechanism exists. Real-time programs in need of an imprecise Jacobi computation package can call SYNCHRONOUS_JACOBI_IMPRECISE_COMPUTATION in the following manner:

```
with SYNCHRONOUS_JACOBI_COMPUTATION;
use  SYNCHRONOUS_JACOBI_COMPUTATION;
with SYNCHRONOUS_JACOBI_IMPRECISE_COMPUTATION;
use  SYNCHRONOUS_JACOBI_IMPRECISE_COMPUTATION;
with CALENDAR;                        use CALENDAR;
with TEXT_IO;                         use TEXT_IO;
with FLOAT_TEXT_IO;                   use FLOAT_TEXT_IO;
with INTEGER_TEXT_IO;                 use INTEGER_TEXT_IO;
procedure SYNCHRONOUS_JACOBI_TEST is

    MY_TASK_PTR : TEST_PTR      := new TEST_TASK;
    DEAD        : CALENDAR.TIME;
    RESULT      : RESULT_TYPE;
    COMP_TIME   : FLOAT;
    INPUT       : INPUT_TYPE;
begin
    for INDEX in 1 .. N loop
        put_line("Enter the coefficients and " &
                 "right hand side for equation " &
                 integer'image(INDEX));
        for NUM_COEFF in 1 .. N loop
            get(INPUT.COEFFICIENTS(INDEX,NUM_COEFF));
        end loop;
        get(INPUT.RIGHT_HAND_SIDE(INDEX));
    end loop;
    for INDEX in 1 .. N loop
        INPUT.XOLD(INDEX) := 0.0;
    end loop;
    put("Enter tolerance factor => ");
    get(INPUT.TOL);
    put("Enter the computation duration(secs.) => ");
    get(COMP_TIME);
    put_line("Synchronous Jacobi test starting...");
    DEAD := CALENDAR.CLOCK + DURATION(COMP_TIME);
    SYNCHRONOUS_JACOBI_IMPRECISE_COMPUTATION.
            IMPCALL(MY_TASK_PTR,
                    1,
                    DEAD,
                    INPUT,
                    RESULT);
    put_line("Jacobi TEST ending... ");
    for INDEX in 1 .. N loop
        put("X"); put(INDEX,WIDTH => 1); put(" => ");
        put(RESULT(INDEX), EXP => 0);
        new_line;
    end loop;
end SYNCHRONOUS_JACOBI_TEST;
```

After the synchronous imprecise Jacobi computation mechanism is built, using it is quite simple. After the input variables are determined, a single IMPCALL runs the entire imprecise computation. The complete file listings for this example, including the implementation of the Jacobi method, are located in Appendix E.

4.2.2 Asynchronous Jacobi Imprecise Computation

The sequence of files is again the same because of the circular calling mechanism employed. The package specification for the asynchronous computation looks like this:

```
N : constant integer := 3;

type RESULT_TYPE is array (1 .. N) of float;

subtype ERROR_TYPE is integer;

type COEFFICIENT_TYPE is array(1 .. N, 1 .. N) of float;

type INPUT_TYPE is record
    COEFFICIENTS     : COEFFICIENT_TYPE;
    RIGHT_HAND_SIDE : RESULT_TYPE;
    XOLD             : RESULT_TYPE;
    TOL              : float;
end record;

task type TEST_TASK is
    pragma PRIORITY(0);
    entry START_COMPUTATION(INPUT : in INPUT_TYPE);
end TEST_TASK;

type TEST_PTR is access TEST_TASK;

procedure START_COMPUTATION(THE_TASK : in TEST_PTR;
                            INPUT    : in INPUT_TYPE);

procedure HANDLE(LAST_VALUE           : in out
                                          RESULT_TYPE;
                 LAST_ERROR_INDICATOR : in out
                                          ERROR_TYPE);
```

The integer constant N represents the number of equations in the linear system. The type RESULT_TYPE represents the form of the final result which will be a solution vector with N elements. Type ERROR_TYPE will again be an integer count of the number of iterations. The type COEFFICIENT_TYPE will not be used directly for instantiation, but represents an N by N matrix of coefficients. The type INPUT_TYPE is the same as the synchronous input. The coefficient matrix COEFFICIENTS, the values to the right of the equal operator RIGHT_HAND_SIDE, the initial solution guess XOLD, and the tolerance TOL are passed to the computation task at initialization. The specification of task type TEST_TASK contains the compiler directive to give a task object of this task type the lowest possible priority. This allows the asynchronous imprecise computation mechanism, operating at the highest priority, to gain control of the processor. Task type TEST_TASK also contains a single entry call, START_COMPUTATION. The procedure START_COMPUTATION is needed to rendezvous with the computation task and initialize it. The procedure HANDLE is a standalone procedure that manipulates the final, imprecise result.

Again, once the asynchronous computation package specification is compiled and entered into the user's library, an instantiation of the generic package ASYNCHRONOUS_IMPRECISE_COMPUTATION can be made using the declarations from the newly constructed package specification. This is accomplished in the following way:

```
with ASYNCHRONOUS_JACOBI_COMPUTATION;
use ASYNCHRONOUS_JACOBI_COMPUTATION;
with ASYNCHRONOUS_IMPRECISE_COMPUTATION;
package ASYNCHRONOUS_JACOBI_IMPRECISE_COMPUTATION is
    new ASYNCHRONOUS_IMPRECISE_COMPUTATION
        (COMPUTATION           => TEST_TASK,
         COMPUTATION_PTR       => TEST_PTR,
         RESULT_TYPE           => RESULT_TYPE,
         ERROR_INDICATOR_TYPE  => ERROR_TYPE,
         INPUT_TYPE            => INPUT_TYPE,
         START_COMPUTATION     => START_COMPUTATION,
         HANDLE                => HANDLE);
```

The package ASYNCHRONOUS_JACOBI_IMPRECISE_COMPUTATION is created from the generic template, substituting the new declarations for the generic parameters. This new package contains valid IMPCALL and IMPRETURN procedures, the latter needed by the computation task to return imprecise results. At this point, the computation package body containing the procedure bodies and task type body is constructed.

The package body of ASYNCHRONOUS_JACOBI_COMPUTATION contains the following procedure bodies:

```
procedure START_COMPUTATION(THE_TASK : in TEST_PTR;
                            INPUT    : in INPUT_TYPE) is
    begin
        THE_TASK.START_COMPUTATION(INPUT);
    end START_COMPUTATION;

procedure HANDLE(LAST_VALUE           : in out
                                           RESULT_TYPE;
                 LAST_ERROR_INDICATOR : in out
                                           ERROR_TYPE) is
    begin
        put_line("HANDLE called ...");
        put("Computation looped ");
        put(LAST_ERROR_INDICATOR);
        put_line(" times.");
    end HANDLE;
```

The procedure START_COMPUTATION merely calls the entry point START_COMPUTATION in THE_TASK. During the rendezvous, parameter INPUT is used to initialize the computation task. Procedure HANDLE, in this example, merely displays the number of iterations the computation completed. Additional statements could be included to manipulate the imprecise result based on this number. The computation package body also contains the body of task type TEST_TASK:

```
    task body TEST_TASK is

         ... local variable declarations ...

    begin
         accept START_COMPUTATION(INPUT:in INPUT_TYPE) do
              ... initialize local variables with input...
         end START_COMPUTATION;
         delay duration'small;
         ... normalize matrix ...
         loop
              ... iterate improvement until required
                   accuracy is achieved ...
              ... compute new solution vector
                   using method in [15,16] ...
              ... find absolute difference
                   between old and new elements..
              ... let present estimate be
                   improved estimate ...
              ... set finished flag if within accuracy ...
              ... report current result
                   with IMPRETURN ...
              exit when FINISHED;
         end loop;

         exception
              when NUMERIC_ERROR =>
                   put_line("NUMERIC ERROR... " &
                            "Diverging solution.");
    end TEST_TASK;
```

During the START_COMPUTATION rendezvous, local variables are assigned the values of INPUT fields. The task then delays the smallest possible amount of time. This delay allows task TIMER to determine its initial delay amount and then delay itself. After the coefficient matrix is normalized, the task enters a loop. This loop contains the Jacobi algorithm as specified in [15,16]. If the required accuracy is achieved, the FINISHED flag will be set. An IMPRETURN call returns the current imprecise result, error indicator, and state of the FINISHED flag. If FINISHED is set, the loop is exited and the task completes. Appropriate exception handlers are set up as required by the particular computation. With the ASYNCHRONOUS_JACOBI_COMPUTATION package body compiled and in the user's library, the following VAX Ada procedure can use the imprecise Jacobi computation mechanism:

```
with ASYNCHRONOUS_JACOBI_COMPUTATION;
use ASYNCHRONOUS_JACOBI_COMPUTATION;
with ASYNCHRONOUS_JACOBI_IMPRECISE_COMPUTATION;
use  ASYNCHRONOUS_JACOBI_IMPRECISE_COMPUTATION;
with CALENDAR;                     use CALENDAR;
with TEXT_IO;                      use TEXT_IO;
with FLOAT_TEXT_IO;                use FLOAT_TEXT_IO;
with INTEGER_TEXT_IO;              use INTEGER_TEXT_IO;
procedure ASYNCHRONOUS_JACOBI_TEST is

pragma TIME_SLICE(0.01);

    MY_TASK_PTR : TEST_PTR      := new TEST_TASK;
    DEAD        : CALENDAR.TIME;
    RESULT      : RESULT_TYPE;
    COMP_TIME   : FLOAT;
    INPUT       : INPUT_TYPE;

begin
    for INDEX in 1 .. N loop
        put_line("Enter the coefficients and " &
                 "right hand side for equation " &
                 integer'image(INDEX));
        for NUM_COEFF in 1 .. N loop
            get(INPUT.COEFFICIENTS(INDEX,NUM_COEFF));
```

```
        end loop;
        get(INPUT.RIGHT_HAND_SIDE(INDEX));
    end loop;
    for INDEX in 1 .. N loop
        INPUT.XOLD(INDEX) := 0.0;
    end loop;
    put("Enter tolerance factor => ");
    get(INPUT.TOL);
    put("Enter the computation duration(secs) => ");
    get(COMP_TIME);
    put_line("Asynchronous Jacobi test starting...");
    DEAD := CALENDAR.CLOCK + DURATION(COMP_TIME);
    ASYNCHRONOUS_JACOBI_IMPRECISE_COMPUTATION.
        IMPCALL(MY_TASK_PTR,
                DEAD,
                INPUT,
                RESULT);
    put_line("Jacobi TEST ending... ");
    for INDEX in 1 .. N loop
        put("X"); put(INDEX,WIDTH => 1); put(" => ");
        put(RESULT(INDEX), EXP => 0);
        new_line;
    end loop;
end ASYNCHRONOUS_JACOBI_TEST;
```

After the input variables are given their appropriate values, the imprecise computation is run by merely calling IMPCALL and passing it the necessary parameters. When the computation completes, the final result is passed back in the parameter RESULT and IMPCALL terminates. A complete listing of the files for this asynchronous Jacobi example can be found in Appendix F.

4.3 Running the Examples

All of the preceding examples were compiled and run on a VAX-11/780 at the 83rd Fighter Weapons Squadron's Range Support Facility (RSF), Tyndall Air Force Base, Florida. The RSF VAX runs the VMS operating system and uses the DEC Ada compiler. All test files compiled and linked correctly. The example tests were run at a real-time priority, giving them privilege over system processes such as the swapper and all other user processes. The results of these tests are summarized in Tables 2 through 5. Each table contains the duration of the imprecise computation (TIME), the number of iterations completed (ITERATIONS COMPLETED), and the amount of time the computation took past its deadline (PAST DEADLINE).

As expected, the asynchronous approach proved much faster, almost by an order of magnitude, than the synchronous approach in the circle test. This algorithm involves a short, simple loop that must be repeated 10000 times to produce a result considered precise. In this example, the synchronous approach yielded more consistent and lower deadline expiration times. This is expected because the synchronous approach maintains total control over the computation loop. In the Jacobi test, solving a linear system of three equations with three unknowns required only 15 iterations. This example represents the other side of the iteration spectrum as compared to the circle test's 10000 iterations. In addition to the synchronous approach maintaining its lower and consistent deadline expiration times, it also produced a precise result ahead of the asynchronous approach.

TIME	ITERATIONS COMPLETED	PAST DEADLINE(sec)
0.01	0	1.00098E-02
0.05	1830	1.00098E-02
0.10	4080	1.00098E-02
0.15	5540	0.00000E+00
0.20	8510	0.00000E+00
0.25	10000 (complete)	-

Table 2. Asynchronous Circle Test Results

TIME	ITERATIONS COMPLETED	PAST DEADLINE(sec)
0.10	290	0.00000E+00
0.25	730	9.99500E-03
0.50	1510	9.99500E-03
1.00	3040	9.99500E-03
2.00	6300	9.99500E-03
2.50	7470	9.99500E-03
3.00	8880	9.99500E-03
3.50	10000 (complete)	-

Table 3. Synchronous Circle Test Results

TIME	ITERATIONS COMPLETED	PAST DEADLINE(sec)
0.022	0	7.9956E-03
0.023	0	7.0190E-03
0.024	0	5.9814E-03
0.025	15 (complete)	-

Table 4. Asynchronous Jacobi Test Results

TIME	ITERATIONS COMPLETED	PAST DEADLINE(sec)
0.0010	1	8.97000E-03
0.0050	3	5.00000E-03
0.0075	4	2.50000E-03
0.0085	4	1.46000E-03
0.0100	15 (complete)	-

Table 5. Synchronous Jacobi Test Results

The circle and Jacobi imprecise computation examples are indicative of real-time applications. The results of these examples show the relative merits of both the synchronous and asynchronous approaches.

5 Analysis and Conclusion

5.1 Analysis of the Test Results

In analyzing the results of the circle and Jacobi imprecise computation tests, several key observations can be made. First and foremost, the synchronous and asynchronous approaches have been implemented and shown to be feasible. Both approaches have demonstrated their consistent behavior within these example tests. Second, it is apparent that the approach used for a particular application should depend on the nature of the computation involved. The asynchronous approach demonstrated its capability to outdistance the synchronous approach in the simple, short, highly repetitive computation loop of the Monte Carlo circle test. On the other hand, the synchronous approach was able to achieve a precise result four times faster than the asynchronous approach in the computation-intensive loop of the Jacobi test. Finally, respectable deadline expiration times were turned in without either the synchronous or asynchronous approaches employing any deadline checking heuristic algorithms. For example, the synchronous mechanism could maintain a running average of the execution time of each iteration. This average time could then be used in deciding whether or not another iteration should be triggered. Another possible enhancement is changing the division factor of the calculated delay time in the asynchronous approach. Altering this constant can help compensate for a lagging run-time system.

The results turned in by the RSF VAX will undoubtedly vary between dissimilar systems. The more a run-time system is geared for real-time performance the better the results will be. Conversely, the less a run-time system is geared for real-time performance the worse the results will be. The same circle and Jacobi tests run on a VADS machine produced totally unreliable results. It was not uncommon to observe deadline past times of one or two seconds! These observations added to the list of lessons learned in this project.

5.2 Lessons Learned

Through the course of this research effort, several problems related to the Ada programming language and its run-time environment were identified. First, the rendezvous is too costly in terms of execution time. The rendezvous has been shown to require fifty times the execution time of a procedure call [7]. This is the one major drawback to the synchronous approach to imprecise computations. The asynchronous approach identified more severe and less deterministic problems.

Although the Ada tasking model is priority driven, it is not preemptive. For this reason, priority inversion can occur and render the priority system useless. In the context of the Monte Carlo circle example, when the higher priority TIMER task becomes ready to run after its prescribed delay amount, it should not have to wait while the lower priority compute task continues to execute. In this environment, deadlines can be missed by staggering amounts of time. Time slices can be used to compensate for this problem.

An Ada run-time system should allow the user to specify the time slice, or the amount of time a given task can hold onto the processor. VAX Ada provides the non-standard pragma TIME_SLICE. The documentation [8] suggests a minimum value of 0.01 seconds. The VADS implementation is hard-wired to an unrealistic one second [24]. Running the asynchronous imprecise computation tests on both systems demonstrated that a VAX Ada implementation can achieve consistent deadline past times while those achieved by the VADS implementation were unruly and totally unacceptable. The bottom line is the lower the time slice, the less priority inversion effects the computation.

The final problem area is the sense of time in Ada. The delay statement only gives a minimum delay. When this problem is coupled with large time slices and an environment fostering priority inversion, delays can be observed orders of magnitude greater than the requested delay. The VAX Ada asynchronous imprecise computation results show acceptable, consistent results. With the time slice capability, maximum delay can be kept in check.

These problems areas do not spell the death of Ada, nor the death of any project implemented in Ada. The synchronous and asynchronous approaches to imprecise computation have been implemented despite these drawbacks. The problems are not insurmountable. Rather, they form an agenda for the evolution of the Ada programming language.

5.3 Conclusion

The goal of this research effort was to investigate all possible approaches to implementing imprecise computations in Ada. Two approaches emerged out of a central idea. The synchronous and asynchronous versions of the atomic computation loop approach were distinguished because of early timing concerns regarding the rendezvous. Both versions were implemented in standard Ada code. Each version was demonstrated using the Monte Carlo circle example and the Jacobi example. Each example was painstakingly constructed in a straightforward manner. These examples illustrated that the synchronous and asynchronous approaches were better suited for different imprecise computation applications. But more importantly, the examples showed that implementing imprecise computations in Ada is entirely possible.

Appendix A

SYNCHRONOUS_IMPRECISE_COMPUTATION

```
with CALENDAR;
generic
    -- the task type --
    type COMPUTATION is limited private;

    -- the pointer type to the task type --
    type COMPUTATION_PTR is access COMPUTATION;

    -- the result type of the computation --
    type RESULT_TYPE is private;

    -- the error indicator type --
    type ERROR_INDICATOR_TYPE is private;

    -- the input argument type --
    type INPUT_TYPE is private;

    -- procedure to initialize the compute task --
    with procedure INITIALIZE(THE_COMPUTATION : in
                                             COMPUTATION_PTR;
                              INPUT           : in
                                             INPUT_TYPE);

    -- procedure to call a rendezvous with compute loop --
    with procedure COMPUTE(THE_COMPUTATION      : in
                                             COMPUTATION_PTR;
                           COMPUTATION_COMPLETE : out
                                             boolean);

    -- procedure to call a rendezvous with a handler --
    with procedure HANDLE(THE_COMPUTATION      : in
                                      COMPUTATION_PTR;
                          HANDLER_NUMBER       : in
                                      integer;
                          LAST_VALUE           : in
                                      RESULT_TYPE;
                          LAST_ERROR_INDICATOR : in
                                      ERROR_INDICATOR_TYPE);

    -- procedure to stop the compute task --
    with procedure STOP(THE_COMPUTATION : in
                                            COMPUTATION_PTR);


package SYNCHRONOUS_IMPRECISE_COMPUTATION is

    procedure IMPCALL(THE_COMPUTATION : in out
                                            COMPUTATION_PTR;
                      THE_HANDLER     : in  integer;
                      DEADLINE        : in  CALENDAR.TIME;
                      INPUT           : in  INPUT_TYPE;
                      FINAL_RESULT    : out RESULT_TYPE);
```

```
    procedure IMPRETURN(INTERMEDIATE_RESULT : in
                                        RESULT_TYPE;
                         ERROR_INDICATOR     : in
                                        ERROR_INDICATOR_TYPE);

end SYNCHRONOUS_IMPRECISE_COMPUTATION;
```

```
with TEXT_IO;           use TEXT_IO;
with FLOAT_TEXT_IO;     use FLOAT_TEXT_IO;
package body SYNCHRONOUS_IMPRECISE_COMPUTATION is

    CURRENT_VALUE            : RESULT_TYPE;
    CURRENT_ERROR_INDICATOR  : ERROR_INDICATOR_TYPE;

    procedure IMPCALL(THE_COMPUTATION : in out
                                           COMPUTATION_PTR;
                      THE_HANDLER     : in  integer;
                      DEADLINE        : in  CALENDAR.TIME;
                      INPUT           : in  INPUT_TYPE;
                      FINAL_RESULT    : out RESULT_TYPE) is

        COMPUTATION_COMPLETED : boolean;
        TIME_HACK             : CALENDAR.TIME;

    begin
        INITIALIZE(THE_COMPUTATION, INPUT);
        loop
            COMPUTE(THE_COMPUTATION,
                    COMPUTATION_COMPLETED);
            exit when COMPUTATION_COMPLETED;
            TIME_HACK := CALENDAR.CLOCK;

            if CALENDAR.">"(TIME_HACK, DEADLINE) then
                put("deadline expired by ");
                put(float(calendar."-"(TIME_HACK,
                                       deadline)), exp=>0);
              put_line("secs.  Calling handler...");
                HANDLE(THE_COMPUTATION,
                       THE_HANDLER,
                       CURRENT_VALUE,
                       CURRENT_ERROR_INDICATOR);
                exit;
            end if;
        end loop;
        STOP(THE_COMPUTATION);
        FINAL_RESULT := CURRENT_VALUE;
    end IMPCALL;

    procedure IMPRETURN(INTERMEDIATE_RESULT : in
                                   RESULT_TYPE;
                        ERROR_INDICATOR     : in
                                   ERROR_INDICATOR_TYPE) is

    begin
        CURRENT_VALUE           := INTERMEDIATE_RESULT;
        CURRENT_ERROR_INDICATOR := ERROR_INDICATOR;
    end IMPRETURN;

end SYNCHRONOUS_IMPRECISE_COMPUTATION;
```

Appendix B

ASYNCHRONOUS_IMPRECISE_COMPUTATION

```
with CALENDAR;
generic
    -- the task type --
    type COMPUTATION is limited private;

    -- the pointer type to the task type --
    type COMPUTATION_PTR is access COMPUTATION;

    -- the result type of the computation --
    type RESULT_TYPE is private;

    -- the error indicator type --
    type ERROR_INDICATOR_TYPE is private;

    -- the input argument type --
    type INPUT_TYPE is private;

    -- procedure to start compute loop --
    with procedure START_COMPUTATION(THE_COMPUTATION : in
                                                  COMPUTATION_PTR;
                                     INPUT           : in
                                                  INPUT_TYPE);

    -- procedure to call a handler --
    with procedure HANDLE(LAST_VALUE            : in out
                                            RESULT_TYPE;
                          LAST_ERROR_INDICATOR : in out
                                            ERROR_INDICATOR_TYPE);


package ASYNCHRONOUS_IMPRECISE_COMPUTATION is

    procedure IMPCALL(THE_COMPUTATION : in out
                                             COMPUTATION_PTR;
                      DEADLINE         : in  CALENDAR.TIME;
                      INPUT            : in  INPUT_TYPE;
                      FINAL_RESULT     : out RESULT_TYPE);

    procedure IMPRETURN(INTERMEDIATE_RESULT : in
                                         RESULT_TYPE;
                        ERROR_INDICATOR     : in
                                         ERROR_INDICATOR_TYPE;
                        STOP_FLAG           : in out
                                         boolean);

end ASYNCHRONOUS_IMPRECISE_COMPUTATION;
```

```
with TEXT_IO;           use TEXT_IO;
with FLOAT_TEXT_IO;     use FLOAT_TEXT_IO;
package body ASYNCHRONOUS_IMPRECISE_COMPUTATION is

    CURRENT_VALUE           : RESULT_TYPE;
    CURRENT_ERROR_INDICATOR : ERROR_INDICATOR_TYPE;
    STOP_COMPUTATION_FLAG   : boolean := FALSE;

    task TIMER is
        pragma PRIORITY(7);
        entry RUN_JOB(THE_JOB  : in out COMPUTATION_PTR;
                      INPUT    : in     INPUT_TYPE;
                      DEADLINE : in     CALENDAR.TIME);
    end TIMER;

    task body TIMER is

        COMPUTATION_COMPLETED : boolean;
        TIME_HACK             : CALENDAR.TIME;
        TIME_LEFT             : float;
        DELAY_TIME            : DURATION;
        HACK1, HACK2          : CALENDAR.TIME;

    begin
        accept RUN_JOB(THE_JOB  : in out COMPUTATION_PTR;
                       INPUT    : in     INPUT_TYPE;
                       DEADLINE : in     CALENDAR.TIME) do
            START_COMPUTATION(THE_JOB, INPUT);
            loop
                TIME_HACK := CALENDAR.CLOCK;
                TIME_LEFT := float(CALENDAR."-"(DEADLINE,
                                                TIME_HACK));
                DELAY_TIME := DURATION(TIME_LEFT / 2.0);
                if DELAY_TIME < DURATION'SMALL and then
                        DELAY_TIME > 0.0 then
                    DELAY_TIME := 0.0;
                end if;
                if DELAY_TIME > 0.0 then
                    put("delaying ");
                    put(float(DELAY_TIME));
                    put_line(" secs.");
                    HACK1 := CALENDAR.CLOCK;
                    delay DELAY_TIME;
                    HACK2 := CALENDAR.CLOCK;
                    put("Actual delay was ");
                    put(float(CALENDAR."-"(HACK2,HACK1)));
                    put_line(" secs.");
                else
                    put("DEADLINE expired by ");
                    put(float(CALENDAR."-"(TIME_HACK,
                                           DEADLINE)));
                    put_line(" secs.");
```

```
                    STOP_COMPUTATION_FLAG := TRUE;
                    HANDLE(CURRENT_VALUE,
                           CURRENT_ERROR_INDICATOR);
                end if;
                exit when STOP_COMPUTATION_FLAG;
            end loop;
        end RUN_JOB;
    end TIMER;

    procedure IMPCALL(THE_COMPUTATION : in out
                                           COMPUTATION_PTR;
                      DEADLINE        : in CALENDAR.TIME;
                      INPUT           : in INPUT_TYPE;
                      FINAL_RESULT    : out RESULT_TYPE) is

    begin
        TIMER.RUN_JOB(THE_COMPUTATION,
                      INPUT,
                      DEADLINE);
        FINAL_RESULT := CURRENT_VALUE;
    end IMPCALL;

    procedure IMPRETURN(INTERMEDIATE_RESULT : in
                                       RESULT_TYPE;
                        ERROR_INDICATOR     : in
                                       ERROR_INDICATOR_TYPE;
                        STOP_FLAG           : in out
                                       boolean) is
    begin
        if not STOP_COMPUTATION_FLAG then
            CURRENT_VALUE           := INTERMEDIATE_RESULT;
            CURRENT_ERROR_INDICATOR := ERROR_INDICATOR;
        end if;

        -- If incoming stop flag is FALSE, then this is
        -- merely a classic IMPRETURN call.  If TRUE, then
        -- this is a signal that the computation has
        -- completed.
        if not STOP_FLAG then
            STOP_FLAG := STOP_COMPUTATION_FLAG;
        else
            STOP_COMPUTATION_FLAG := STOP_FLAG;
        end if;
    end IMPRETURN;

end ASYNCHRONOUS_IMPRECISE_COMPUTATION;
```

Appendix C

Synchronous Circle Test Files

```
with CALENDAR;                        use CALENDAR;
package SYNCHRONOUS_CIRCLE_COMPUTATION is

    subtype RESULT_TYPE is float;

    subtype ERROR_TYPE is integer;

    type INPUT_TYPE is record
        LOOPS_TO_COMPLETE : integer;
        RADIUS            : float;
    end record;

    task type TEST_TASK is
        entry INITIALIZE(INPUT : in INPUT_TYPE);
        entry COMPUTE(COMPUTATION_COMPLETE : out boolean);
        entry HANDLER(1 .. 2)(LAST_RESULT : in RESULT_TYPE;
                              LAST_ERROR  : in ERROR_TYPE);
        entry STOP;
    end TEST_TASK;

    type TEST_PTR is access TEST_TASK;

    procedure INITIALIZE(THE_TASK : in TEST_PTR;
                         INPUT    : in INPUT_TYPE);

    procedure COMPUTE(THE_TASK             : in  TEST_PTR;
                      COMPUTATION_COMPLETE : out boolean);

    procedure HANDLE(THE_TASK              : in TEST_PTR;
                     HANDLER_NUMBER        : in integer;
                     LAST_VALUE            : in RESULT_TYPE;
                     LAST_ERROR_INDICATOR  : in ERROR_TYPE);

    procedure STOP(THE_TASK : in TEST_PTR);

end SYNCHRONOUS_CIRCLE_COMPUTATION;
```

```
with TEXT_IO;                                   use TEXT_IO;
with FLOAT_TEXT_IO;                             use FLOAT_TEXT_IO;
with INTEGER_TEXT_IO;                           use INTEGER_TEXT_IO;
with SYNCHRONOUS_CIRCLE_IMPRECISE_COMPUTATION;
use  SYNCHRONOUS_CIRCLE_IMPRECISE_COMPUTATION;
with RANDOM_NUMBER_GENERATOR;
use  RANDOM_NUMBER_GENERATOR;
package body SYNCHRONOUS_CIRCLE_COMPUTATION is

    procedure INITIALIZE(THE_TASK : in TEST_PTR;
                         INPUT    : in INPUT_TYPE) is
        begin
            THE_TASK.INITIALIZE(INPUT);
        end INITIALIZE;

    procedure COMPUTE(THE_TASK : in TEST_PTR;
                      COMPUTATION_COMPLETE : out boolean) is
        begin
            THE_TASK.COMPUTE(COMPUTATION_COMPLETE);
        end COMPUTE;

    procedure HANDLE(THE_TASK              : in TEST_PTR;
                     HANDLER_NUMBER        : in integer;
                     LAST_VALUE            : in RESULT_TYPE;
                     LAST_ERROR_INDICATOR : in ERROR_TYPE) is
        begin
            THE_TASK.HANDLER(HANDLER_NUMBER)
                            (LAST_VALUE,
                             LAST_ERROR_INDICATOR);
        end HANDLE;

    procedure STOP(THE_TASK : in TEST_PTR) is
        begin
            THE_TASK.STOP;
        end STOP;


    task body TEST_TASK is

        FINISHED        : boolean    := false;
        ERROR           : ERROR_TYPE := 0;
        M               : integer    := 0;
        N               : integer    := 0;
        RADIUS          : float;
        RADIUS_SQUARED  : float;
        DIAMETER        : float;
        SQUARE_AREA     : float;
        X, Y            : float;
        AREA            : RESULT_TYPE;
        LOOP_NUM        : integer;
        SEED            : integer;
```

```
begin
    accept INITIALIZE(INPUT : in INPUT_TYPE) do
        RADIUS          := INPUT.RADIUS;
        RADIUS_SQUARED := RADIUS ** 2;
        DIAMETER        := 2.0 * RADIUS;
        SQUARE_AREA     := DIAMETER ** 2;
        LOOP_NUM        := INPUT.LOOPS_TO_COMPLETE;
        SEED            := 1;
    end INITIALIZE;
    loop
        select
            accept COMPUTE(COMPUTATION_COMPLETE : out
                                                boolean) do
                RANDOM(X,SEED);
                RANDOM(Y,SEED);
                X := X * DIAMETER - RADIUS;
                Y := Y * DIAMETER - RADIUS;
                N := N + 1;
                if (X**2 + Y**2) <= RADIUS_SQUARED then
                    M := M + 1;
                end if;
                ERROR := ERROR + 1;
                if ERROR > LOOP_NUM then
                    COMPUTATION_COMPLETE := TRUE;
                else
                    COMPUTATION_COMPLETE := FALSE;
                end if;
                if ERROR rem 10 = 0 or
                   ERROR > LOOP_NUM then
                    AREA := SQUARE_AREA *
                                 float(M) / float(N);
                    IMPRETURN(AREA, ERROR);
                end if;
            end COMPUTE;
        or
            accept HANDLER(1)
                       (LAST_RESULT : in RESULT_TYPE;
                        LAST_ERROR  : in ERROR_TYPE) do
                -- output number of iterations --
                put("Computation looped ");
                put(LAST_ERROR);
                put_line(" times.");
                -- IMPRETURN if modification made --
            end HANDLER;
        or
            accept HANDLER(2)
                       (LAST_RESULT : in RESULT_TYPE;
                        LAST_ERROR  : in ERROR_TYPE) do
                null;  -- this handler does nothing --
                -- IMPRETURN if modification made --
            end HANDLER;
        or
```

```
                    accept STOP do
                        FINISHED := true;
                    end STOP;
                end select;
                exit when FINISHED;
            end loop;
        end TEST_TASK;

    end SYNCHRONOUS_CIRCLE_COMPUTATION;
```

```
with SYNCHRONOUS_CIRCLE_COMPUTATION;
use  SYNCHRONOUS_CIRCLE_COMPUTATION;
with SYNCHRONOUS_IMPRECISE_COMPUTATION;
package SYNCHRONOUS_CIRCLE_IMPRECISE_COMPUTATION is
    new SYNCHRONOUS_IMPRECISE_COMPUTATION
         (COMPUTATION          => TEST_TASK,
          COMPUTATION_PTR      => TEST_PTR,
          RESULT_TYPE          => RESULT_TYPE,
          ERROR_INDICATOR_TYPE => ERROR_TYPE,
          INPUT_TYPE           => INPUT_TYPE,
          INITIALIZE           => INITIALIZE,
          COMPUTE              => COMPUTE,
          HANDLE               => HANDLE,
          STOP                 => STOP);
```

```
with SYNCHRONOUS_CIRCLE_COMPUTATION;
use  SYNCHRONOUS_CIRCLE_COMPUTATION;
with SYNCHRONOUS_CIRCLE_IMPRECISE_COMPUTATION;
use  SYNCHRONOUS_CIRCLE_IMPRECISE_COMPUTATION;
with CALENDAR;                              use CALENDAR;
with TEXT_IO;                               use TEXT_IO;
with FLOAT_TEXT_IO;                         use FLOAT_TEXT_IO;
with INTEGER_TEXT_IO;                       use INTEGER_TEXT_IO;
procedure SYNCHRONOUS_CIRCLE_TEST is

pragma TIME_SLICE(0.01);

    MY_TASK_PTR : TEST_PTR      := new TEST_TASK;
    DEAD        : CALENDAR.TIME;
    RESULT      : RESULT_TYPE;
    COMP_TIME   : float;
    MY_INPUT    : INPUT_TYPE;

begin
    put("Enter the circle radius => ");
    get(MY_INPUT.RADIUS);
    put("Enter the number of iterations to complete => ");
    get(MY_INPUT.LOOPS_TO_COMPLETE);
    put("Enter the computation duration in seconds => ");
    get(COMP_TIME);
    put_line("Synchronous CIRCLE TEST starting...");
    DEAD := CALENDAR.CLOCK + DURATION(COMP_TIME);
    IMPCALL(MY_TASK_PTR,
            1,
            DEAD,
            MY_INPUT,
            RESULT);
    put("TEST ending...  RESULT => ");
    put(RESULT, EXP => 0, AFT => 2);
    new_line;
end SYNCHRONOUS_CIRCLE_TEST;
```

Appendix D

Asynchronous Circle Test Files

```
with CALENDAR;   use CALENDAR;
with SYSTEM;     use SYSTEM;
package ASYNCHRONOUS_CIRCLE_COMPUTATION is

    subtype RESULT_TYPE is float;

    subtype ERROR_TYPE is integer;

    type INPUT_TYPE is record
        LOOPS_TO_COMPLETE : integer;
        RADIUS            : float;
    end record;

    task type TEST_TASK is
        pragma PRIORITY(0);
        entry START_COMPUTATION(INPUT : in INPUT_TYPE);
    end TEST_TASK;

    type TEST_PTR is access TEST_TASK;

    procedure START_COMPUTATION(THE_TASK : in TEST_PTR;
                                INPUT    : in INPUT_TYPE);

    procedure HANDLE(LAST_VALUE           : in out
                                             RESULT_TYPE;
                     LAST_ERROR_INDICATOR : in out
                                             ERROR_TYPE);

end ASYNCHRONOUS_CIRCLE_COMPUTATION;
```

```
with TEXT_IO;                                   use TEXT_IO;
with FLOAT_TEXT_IO;                             use FLOAT_TEXT_IO;
with INTEGER_TEXT_IO                            use INTEGER_TEXT_IO;
with ASYNCHRONOUS_CIRCLE_IMPRECISE_COMPUTATION;
use  ASYNCHRONOUS_CIRCLE_IMPRECISE_COMPUTATION;
with RANDOM_NUMBER_GENERATOR;
use  RANDOM_NUMBER_GENERATOR;
package body ASYNCHRONOUS_CIRCLE_COMPUTATION is

    procedure START_COMPUTATION(THE_TASK : in TEST_PTR;
                                INPUT    : in INPUT_TYPE)
                                                       is
        begin
            THE_TASK.START_COMPUTATION(INPUT);
        end START_COMPUTATION;

    procedure HANDLE(LAST_VALUE            : in out
                                                 RESULT_TYPE;
                     LAST_ERROR_INDICATOR : in out
                                               ERROR_TYPE) is
        begin
            put("Computation looped ");
            put(LAST_ERROR_INDICATOR);
            put_line(" times.");
        end HANDLE;


    task body TEST_TASK is

        FINISHED       : boolean    := false;
        ERROR          : ERROR_TYPE := 0;
        M              : integer    := 0;
        N              : integer    := 0;
        RADIUS         : float;
        RADIUS_SQUARED : float;
        DIAMETER       : float;
        SQUARE_AREA    : float;
        X, Y           : float;
        AREA           : float;
        LOOP_NUM       : integer;
        SEED           : integer;

    begin
        accept START_COMPUTATION(INPUT : in INPUT_TYPE) do
            RADIUS         := INPUT.RADIUS;
            LOOP_NUM       := INPUT.LOOPS_TO_COMPLETE;
            RADIUS_SQUARED := RADIUS ** 2;
            DIAMETER       := 2.0 * RADIUS;
            SQUARE_AREA    := DIAMETER ** 2;
            SEED           := 1;
        end START_COMPUTATION;
        delay DURATION'SMALL;
```

```
        loop
            RANDOM(X,SEED);
            RANDOM(Y,SEED);
            X := X * DIAMETER - RADIUS;
            Y := Y * DIAMETER - RADIUS;
            N := N + 1;
            if (X**2 + Y**2) <= RADIUS_SQUARED then
                M := M + 1;
            end if;
            ERROR := ERROR + 1;
            if ERROR > LOOP_NUM then
                FINISHED := TRUE;
            end if;
            if (ERROR rem 10 = 0) or FINISHED then
                AREA := SQUARE_AREA * float(M) / float(N);
                IMPRETURN(AREA, ERROR, FINISHED);
            end if;
            exit when FINISHED;
        end loop;
    end TEST_TASK;

end ASYNCHRONOUS_CIRCLE_COMPUTATION;
```

```
with ASYNCHRONOUS_CIRCLE_COMPUTATION;
use  ASYNCHRONOUS_CIRCLE_COMPUTATION;
with ASYNCHRONOUS_IMPRECISE_COMPUTATION;
package ASYNCHRONOUS_CIRCLE_IMPRECISE_COMPUTATION is
    new ASYNCHRONOUS_IMPRECISE_COMPUTATION
        (COMPUTATION           => TEST_TASK,
         COMPUTATION_PTR       => TEST_PTR,
         RESULT_TYPE           => RESULT_TYPE,
         ERROR_INDICATOR_TYPE => ERROR_TYPE,
         INPUT_TYPE            => INPUT_TYPE,
         START_COMPUTATION     => START_COMPUTATION,
         HANDLE                => HANDLE);
```

```
with ASYNCHRONOUS_CIRCLE_COMPUTATION;
use  ASYNCHRONOUS_CIRCLE_COMPUTATION;
with ASYNCHRONOUS_CIRCLE_IMPRECISE_COMPUTATION;
use  ASYNCHRONOUS_CIRCLE_IMPRECISE_COMPUTATION;
with CALENDAR;                           use CALENDAR;
with TEXT_IO;                            use TEXT_IO;
with FLOAT_TEXT_IO;                      use FLOAT_TEXT_IO;
with INTEGER_TEXT_IO;                    use INTEGER_TEXT_IO;
procedure ASYNCHRONOUS_CIRCLE_TEST is

pragma TIME_SLICE(0.01);

    MY_TASK_PTR : TEST_PTR      := new TEST_TASK;
    DEAD        : CALENDAR.TIME;
    RESULT      : RESULT_TYPE;
    COMP_TIME   : float;
    MY_INPUT    : INPUT_TYPE;

begin
    put("Enter the circle radius => ");
    get(MY_INPUT.RADIUS);
    put("Enter the number of iterations to complete => ");
    get(MY_INPUT.LOOPS_TO_COMPLETE);
    put("Enter the computation duration in seconds => ");
    get(COMP_TIME);
    put_line("Asynchronous CIRCLE TEST starting...");
    DEAD := CALENDAR.CLOCK + DURATION(COMP_TIME);
    IMPCALL(MY_TASK_PTR,
            DEAD,
            MY_INPUT,
            RESULT);
    put("CIRCLE TEST ending...  CIRCLE AREA RESULT => ");
    put(RESULT, EXP => 0, AFT => 2);
    new_line;
end ASYNCHRONOUS_CIRCLE_TEST;
```

Appendix E

Synchronous Jacobi Test Files

```
with CALENDAR;                    use CALENDAR;
package SYNCHRONOUS_JACOBI_COMPUTATION is

    N : constant integer := 3;

    type RESULT_TYPE is array(1 .. N) of float;

    subtype ERROR_TYPE is integer;

    type COEFFICIENT_TYPE is array(1 .. N, 1 .. N)
        of float;

    type INPUT_TYPE is record
        COEFFICIENTS        : COEFFICIENT_TYPE;
        RIGHT_HAND_SIDE     : RESULT_TYPE;
        XOLD                : RESULT_TYPE;
        TOL                 : float;
    end record;

    task type TEST_TASK is
        entry INITIALIZE(INPUT : in INPUT_TYPE);
        entry COMPUTE(COMPUTATION_COMPLETE : out boolean);
        entry HANDLER(1 .. 2)
            (LAST_RESULT : in RESULT_TYPE;
             LAST_ERROR  : in ERROR_TYPE);
        entry STOP;
    end TEST_TASK;

    type TEST_PTR is access TEST_TASK;

    procedure INITIALIZE(THE_TASK : in TEST_PTR;
                         INPUT    : in INPUT_TYPE);

    procedure COMPUTE(THE_TASK             : in  TEST_PTR;
                      COMPUTATION_COMPLETE : out boolean);

    procedure HANDLE(THE_TASK             : in TEST_PTR;
                     HANDLER_NUMBER       : in integer;
                     LAST_VALUE           : in RESULT_TYPE;
                     LAST_ERROR_INDICATOR : in ERROR_TYPE);

    procedure STOP(THE_TASK : in TEST_PTR);

end SYNCHRONOUS_JACOBI_COMPUTATION;
```

```
with TEXT_IO;                                use TEXT_IO;
with FLOAT_TEXT_IO;                          use FLOAT_TEXT_IO;
with INTEGER_TEXT_IO;                        use INTEGER_TEXT_IO;
with SYNCHRONOUS_JACOBI_IMPRECISE_COMPUTATION;
use  SYNCHRONOUS_JACOBI_IMPRECISE_COMPUTATION;
package body SYNCHRONOUS_JACOBI_COMPUTATION is

    procedure INITIALIZE(THE_TASK : in TEST_PTR;
                         INPUT    : in INPUT_TYPE) is

        begin
            THE_TASK.INITIALIZE(INPUT);
        end INITIALIZE;

    procedure COMPUTE(THE_TASK             : in  TEST_PTR;
                      COMPUTATION_COMPLETE : out boolean) is

        begin
            THE_TASK.COMPUTE(COMPUTATION_COMPLETE);
        end COMPUTE;

    procedure HANDLE(THE_TASK             : in TEST_PTR;
                     HANDLER_NUMBER       : in integer;
                     LAST_VALUE           : in RESULT_TYPE;
                     LAST_ERROR_INDICATOR : in ERROR_TYPE) is

        begin
            THE_TASK.HANDLER(HANDLER_NUMBER)
                (LAST_VALUE, LAST_ERROR_INDICATOR);
        end HANDLE;

    procedure STOP(THE_TASK : in TEST_PTR) is

        begin
            THE_TASK.STOP;
        end STOP;

    task body TEST_TASK is

        FINISHED        : boolean    := false;
        ERROR           : ERROR_TYPE := 0;
        COEFF           : COEFFICIENT_TYPE;
        R_H_S           : RESULT_TYPE; -- right-hand-side --
        XOLD            : RESULT_TYPE; -- solution guess --
        TOL             : float;       -- tolerance --
        XNEW            : RESULT_TYPE; -- new solution --
        C               : COEFFICIENT_TYPE; -- norm coeff
        D               : RESULT_TYPE; -- normalized r-h-s
        MAXNEW,
        NNEW,      --"NEW" in text but an Ada reserved word.
        MAXDIF,
        DIFF            : float;
```

```
begin
    accept INITIALIZE(INPUT : in INPUT_TYPE) do
        COEFF := INPUT.COEFFICIENTS;
        R_H_S := INPUT.RIGHT_HAND_SIDE;
        XOLD  := INPUT.XOLD;
        TOL   := INPUT.TOL;

        -- Normalize matrix --
        for J in 1 .. N loop
            for K in 1 .. J - 1 loop
                C(J,K) := COEFF(J,K) / COEFF(J,J);
            end loop;
            for K in J + 1 .. N loop
                C(J,K) := COEFF(J,K) / COEFF(J,J);
            end loop;
            D(J) := R_H_S(J) / COEFF(J,J);
        end loop;
    end INITIALIZE;

    loop
        select
            accept COMPUTE(COMPUTATION_COMPLETE : out
                                                  boolean) do
                MAXNEW := 0.0;
                MAXDIF := 0.0;
                for J in 1 .. N loop
                    XNEW(J) := D(J);
                    for K in 1 .. J - 1 loop
                        XNEW(J) := XNEW(J) - C(J,K)
                                               * XOLD(K);
                    end loop;
                    for K in J + 1 .. N loop
                        XNEW(J) := XNEW(J) - C(J,K)
                                               * XOLD(K);
                    end loop;

                    -- Find max absolute difference
                    -- between old and new elements.
                    DIFF := ABS(XNEW(J) - XOLD(J));
                    if DIFF > MAXDIF then
                        MAXDIF := DIFF;
                    end if;
                    NNEW := ABS(XNEW(J));
                    if NNEW > MAXNEW then
                        MAXNEW := NNEW;
                    end if;
                end loop;
                ERROR := ERROR + 1;

                -- Let present estimate be improved
                -- estimate
                XOLD(1 .. N) := XNEW(1 .. N);
```

```
                    if MAXNEW /= 0.0 and then
                            (MAXDIF / MAXNEW) <= TOL then
                        COMPUTATION_COMPLETE := TRUE;
                    else
                        COMPUTATION_COMPLETE := FALSE;
                    end if;

                    -- Report current result --
                    IMPRETURN(XNEW, ERROR);

                end COMPUTE;
            or
                accept HANDLER(1)
                           (LAST_RESULT : in RESULT_TYPE;
                            LAST_ERROR  : in ERROR_TYPE) do
                    put("Computation looped ");
                    put(LAST_ERROR);
                    put_line(" times.");
                    -- IMPRETURN if modification made --
                end HANDLER;
            or
                accept HANDLER(2)
                           (LAST_RESULT : in RESULT_TYPE;
                            LAST_ERROR  : in ERROR_TYPE) do
                    null;  -- this handler does nothing --
                    -- IMPRETURN if modification made --
                end HANDLER;
            or
                accept STOP do
                    FINISHED := true;
                end STOP;
            end select;
            exit when FINISHED;
        end loop;
    end TEST_TASK;

end SYNCHRONOUS_JACOBI_COMPUTATION;
```

```
with SYNCHRONOUS_JACOBI_COMPUTATION;
use  SYNCHRONOUS_JACOBI_COMPUTATION;
with SYNCHRONOUS_IMPRECISE_COMPUTATION;
package SYNCHRONOUS_JACOBI_IMPRECISE_COMPUTATION is
    new SYNCHRONOUS_IMPRECISE_COMPUTATION
        (COMPUTATION          => TEST_TASK,
         COMPUTATION_PTR      => TEST_PTR,
         RESULT_TYPE          => RESULT_TYPE,
         ERROR_INDICATOR_TYPE => ERROR_TYPE,
         INPUT_TYPE           => INPUT_TYPE,
         INITIALIZE           => INITIALIZE,
         COMPUTE              => COMPUTE,
         HANDLE               => HANDLE,
         STOP                 => STOP);
```

```
with SYNCHRONOUS_JACOBI_COMPUTATION;
use  SYNCHRONOUS_JACOBI_COMPUTATION;
with SYNCHRONOUS_JACOBI_IMPRECISE_COMPUTATION;
use  SYNCHRONOUS_JACOBI_IMPRECISE_COMPUTATION;
with CALENDAR;                          use CALENDAR;
with TEXT_IO;                           use TEXT_IO;
with FLOAT_TEXT_IO;                     use FLOAT_TEXT_IO;
with INTEGER_TEXT_IO;                   use INTEGER_TEXT_IO;
procedure SYNCHRONOUS_JACOBI_TEST is

    MY_TASK_PTR : TEST_PTR      := new TEST_TASK;
    DEAD        : CALENDAR.TIME;
    RESULT      : RESULT_TYPE;
    COMP_TIME   : float;
    INPUT       : INPUT_TYPE;

begin
    for INDEX in 1 .. N loop
        put_line("Enter the coefficients and " &
                 "right hand side for equation " &
                  integer'image(INDEX));
        for NUM_COEFF in 1 .. N loop
            get(INPUT.COEFFICIENTS(INDEX,NUM_COEFF));
        end loop;
        get(INPUT.RIGHT_HAND_SIDE(INDEX));
    end loop;
    for INDEX in 1 .. N loop
        INPUT.XOLD(INDEX) := 0.0;
    end loop;
    put("Enter tolerance factor => ");
    get(INPUT.TOL);
    put("Enter the computation duration in seconds => ");
    get(COMP_TIME);
    put_line("Synchronous Jacobi test starting...");
    DEAD := CALENDAR.CLOCK + DURATION(COMP_TIME);
    IMPCALL(MY_TASK_PTR,
            1,
            DEAD,
            INPUT,
            RESULT);
    put_line("Jacobi TEST ending... ");
    for INDEX in 1 .. N loop
        put("X");
        put(INDEX,WIDTH => 1);
        put(" => ");
        put(RESULT(INDEX), EXP => 0);
        new_line.
    end loop;
end SYNCHRONOUS_JACOBI_TEST;
```

Appendix F

Asynchronous Jacobi Test Files

```
with CALENDAR;   use CALENDAR;
with SYSTEM;     use SYSTEM;
package ASYNCHRONOUS_JACOBI_COMPUTATION is

    N : constant integer := 3;

    type RESULT_TYPE is array (1 .. N) of float;

    subtype ERROR_TYPE is integer;

    type COEFFICIENT_TYPE is array(1 .. N, 1 .. N) of float;

    type INPUT_TYPE is record
        COEFFICIENTS     : COEFFICIENT_TYPE;
        RIGHT_HAND_SIDE  : RESULT_TYPE;
        XOLD             : RESULT_TYPE;
        TOL              : float;
    end record;

    task type TEST_TASK is
        pragma PRIORITY(0);
        entry START_COMPUTATION(INPUT : in INPUT_TYPE);
    end TEST_TASK;

    type TEST_PTR is access TEST_TASK;

    procedure START_COMPUTATION(THE_TASK : in TEST_PTR;
                                INPUT    : in INPUT_TYPE);

    procedure HANDLE(LAST_VALUE            : in out
                                                 RESULT_TYPE;
                     LAST_ERROR_INDICATOR  : in out
                                                 ERROR_TYPE);

end ASYNCHRONOUS_JACOBI_COMPUTATION;
```

```
with TEXT_IO;                                   use TEXT_IO;
with FLOAT_TEXT_IO;                             use FLOAT_TEXT_IO;
with INTEGER_TEXT_IO;                           use INTEGER_TEXT_IO;
with ASYNCHRONOUS_JACOBI_IMPRECISE_COMPUTATION;
use  ASYNCHRONOUS_JACOBI_IMPRECISE_COMPUTATION;
package body ASYNCHRONOUS_JACOBI_COMPUTATION is

    procedure START_COMPUTATION(THE_TASK : in TEST_PTR;
                                INPUT    : in INPUT_TYPE) is

        begin
            THE_TASK.START_COMPUTATION(INPUT);
        end START_COMPUTATION;

    procedure HANDLE(LAST_VALUE            : in out
                                             RESULT_TYPE;
                     LAST_ERROR_INDICATOR : in out
                                             ERROR_TYPE) is
        begin
            put("Computation looped ");
            put(LAST_ERROR_INDICATOR);
            put_line(" times.");
        end HANDLE;

    task body TEST_TASK is

        FINISHED : boolean    := false;
        ERROR    : ERROR_TYPE := 0;
        COEFF    : COEFFICIENT_TYPE; -- coefficient input
        R_H_S    : RESULT_TYPE;      -- right-hand-side
        XOLD     : RESULT_TYPE;      -- solution guess
        TOL      : float;            -- tolerance
        XNEW     : RESULT_TYPE;      -- new solution vector
        C        : COEFFICIENT_TYPE; -- norm input coeff
        D        : RESULT_TYPE;      -- normalized r_h_s
        MAXNEW,
        NNEW,    -- "new" in text but reserved
        MAXDIF,
        DIFF     : float;

    begin
        accept START_COMPUTATION(INPUT : in INPUT_TYPE) do
            COEFF := INPUT.COEFFICIENTS;
            R_H_S := INPUT.RIGHT_HAND_SIDE;
            XOLD  := INPUT.XOLD;
            TOL   := INPUT.TOL;
        end START_COMPUTATION;
        delay duration'small;

        -- Normalize matrix --
        for J in 1 .. N loop
            for K in 1 .. J - 1 loop
```

```
            C(J,K) := COEFF(J, K) / COEFF(J, J);
        end loop;
        for K in J + 1 .. N loop
            C(J,K) := COEFF(J, K) / COEFF(J, J);
        end loop;
        D(J) := R_H_S(J) / COEFF(J, J);
    end loop;

    -- Iterate improvement until required
    -- accuracy is achieved
    loop
        MAXNEW := 0.0;
        MAXDIF := 0.0;
        for J in 1 .. N loop
            XNEW(J) := D(J);
            for K in 1 .. J - 1 loop
                XNEW(J) := XNEW(J) - C(J,K) * XOLD(K);
            end loop;
            for K in J + 1 .. N loop
                XNEW(J) := XNEW(J) - C(J,K) * XOLD(K);
            end loop;

            -- Find max absolute difference
            -- between old and new elements.
            DIFF := ABS(XNEW(J) - XOLD(J));
            if DIFF > MAXDIF then
                MAXDIF := DIFF;
            end if;
            NNEW := ABS(XNEW(J));
            if NNEW > MAXNEW then
                MAXNEW := NNEW;
            end if;
        end loop;

        -- Let present estimate be improved estimate
        XOLD(1 .. N) := XNEW(1 .. N);
        ERROR := ERROR + 1;
        if MAXNEW /= 0.0 and then
                      (MAXDIF / MAXNEW) <= TOL then
            FINISHED := TRUE;
        end if;
        IMPRETURN(XNEW, ERROR, FINISHED);
        exit when FINISHED;
    end loop;

exception
    when CONSTRAINT_ERROR | NUMERIC_ERROR =>
        put_line("NUMERIC ERROR - Diverging solution");

end TEST_TASK;

end ASYNCHRONOUS_JACOBI_COMPUTATION;
```

```
with ASYNCHRONOUS_JACOBI_COMPUTATION;
use  ASYNCHRONOUS_JACOBI_COMPUTATION;
with ASYNCHRONOUS_IMPRECISE_COMPUTATION;
package ASYNCHRONOUS_JACOBI_IMPRECISE_COMPUTATION is
    new ASYNCHRONOUS_IMPRECISE_COMPUTATION
        (COMPUTATION          => TEST_TASK,
         COMPUTATION_PTR      => TEST_PTR,
         RESULT_TYPE          => RESULT_TYPE,
         ERROR_INDICATOR_TYPE => ERROR_TYPE,
         INPUT_TYPE           => INPUT_TYPE,
         START_COMPUTATION    => START_COMPUTATION,
         HANDLE               => HANDLE);
```

```
with ASYNCHRONOUS_JACOBI_COMPUTATION;
use  ASYNCHRONOUS_JACOBI_COMPUTATION;
with ASYNCHRONOUS_JACOBI_IMPRECISE_COMPUTATION;
use  ASYNCHRONOUS_JACOBI_IMPRECISE_COMPUTATION;
with CALENDAR;                         use CALENDAR;
with TEXT_IO;                          use TEXT_IO;
with FLOAT_TEXT_IO;                    use FLOAT_TEXT_IO;
with INTEGER_TEXT_IO;                  use INTEGER_TEXT_IO;
procedure ASYNCHRONOUS_JACOBI_TEST is

pragma TIME_SLICE(0.01);

    MY_TASK_PTR : TEST_PTR      := new TEST_TASK;
    DEAD        : CALENDAR.TIME;
    RESULT      : RESULT_TYPE;
    COMP_TIME   : float;
    INPUT       : INPUT_TYPE;

begin
    for INDEX in 1 .. N loop
        put_line("Enter the coefficients and " &
                 "right hand side for equation " &
                  integer'image(INDEX));
        for NUM_COEFF in 1 .. N loop
            get(INPUT.COEFFICIENTS(INDEX,NUM_COEFF));
        end loop;
        get(INPUT.RIGHT_HAND_SIDE(INDEX));
    end loop;
    for INDEX in 1 .. N loop
        INPUT.XOLD(INDEX) := 0.0;
    end loop;
    put("Enter tolerance factor => ");
    get(INPUT.TOL);
    put("Enter the computation duration in seconds => ");
    get(COMP_TIME);
    put_line("Asynchronous Jacobi TEST starting...");
    DEAD := CALENDAR.CLOCK + DURATION(COMP_TIME);
    IMPCALL(MY_TASK_PTR,
            DEAD,
            INPUT,
            RESULT);
    put_line("Jacobi TEST ending... ");
    for INDEX in 1 .. N loop
        put("X");
        put(INDEX,WIDTH => 1);
        put(" => ");
        put(RESULT(INDEX), EXP => 0);
        new_line;
    end loop;
end ASYNCHRONOUS_JACOBI_TEST;
```

List of References

[1] Baker, Ted P. and Kevin Jeffay. "Corset and Lace." Proceedings Real-Time Systems Symposium, 1-3 Dec. 1987, pp. 158-167.

[2] Baker, Ted P. "Improving Immediacy in Ada." Ada Letters, Vol. 8, No. 7 (1988), pp. 50-56.

[3] Basu, Sanat K. "On Development of Iterative Programs from Function Specifications." IEEE Transactions on Software Engineering, Vol. SE-6, No. 2 (March 1980), pp. 170-182.

[4] Booch, Grady. "Ada Scores in the International Market." Defense Computing, Vol. 1, No. 5 (September/October 1988), pp. 19-24.

[5] Booch, Grady. Software Engineering with Ada. Menlo Park: The Benjamin/Cummings Publishing Co., Inc., 1983.

[6] Brosgol, Benjamin. "International Workshop on Real-Time Ada Issues: Summary Report." Ada Letters, Vol. 8, No. 1 (January/February 1988), pp. 91-107.

[7] Burger, Thomas and Kjell Nielsen. "An Assessment of the Overhead Associated with Tasking Facilities and Task Paradigms in Ada." Ada Letters, Vol. 7, No. 1 (January/February 1988), pp. 49-58.

[8] Digital Equipment Corporation. VAX Ada Programmer's Run-Time Reference Manual, Maynard, MA, 1985.

[9] Hammersley, J. M. and D. C. Handscomb. Monte Carlo Methods. London: Methuen & Co., Ltd., 1964.

[10] Ichbiah, Jean D. et al. "Rationale for the Design of the Ada Programming Language." SIGPLAN Notices, Vol. 14, No. 6B (June 1979), pp. 12-12 - 12-15.

[11] Lin, Kwei-Jay, Swaminathan Natarajan, Jane W.-S. Liu, and Tim Krauskopf. "Concord: A System of Imprecise Computations." Proceedings of the 1987 IEEE COMPSAC, Japan. October 1987.

[12] Lin, Kwei-Jay, Swaminathan Natarajan, and Jane W.-S. Liu. "Imprecise Results: Utilizing Partial Computations in Real-Time Systems." Proceedings Real-Time Systems Symposium, 1-3 Dec. 1987, pp. 210-217.

[13] Liu, Jane W.-S. and Kwei-Jay Lin. "On Means to Provide Flexibility in Scheduling." Ada Letters, Vol. 8, No. 7 (1988), pp. 32-34.

[14] Miller, Bill. "Ada Powers the Hellfire Missile Program." Defense Computing, Vol. 1, No. 5 (September/October 1985), pp. 43-44.

[15] Pizer, Stephen M. Numerical Computing and Mathematical Analysis. Chicago: Science Research Associates, Inc., 1975.

[16] Pizer, Stephen M. with Victor L. Wallace. To Compute Numerically. Boston: Little, Brown and Company, 1983.

[17] Taha, Hamdy A. Operations Research. New York: Macmillan Publishing Co., Inc., 1982.

[18] Tennenbaum, Jeremy. The Military's Computing Crisis: The Search for a Solution. Stock Research Report, Salomon Brothers, Inc., 22 Sep. 1987.

[19] Toetenel, W. J. and J. van Katwijk. "Asynchronous Transfer of Control," Ada Letters, Vol. 8, No. 7 (1988), pp. 65-79.

[20] Turski, W. M. "On Programming by Iterations," IEEE Transactions on Software Engineering, Vol. SE-10, No. 2 (March 1984), pp. 175-178.

[21] U.S. Department of Defense. DoD Directive 3405.1. 2 Apr. 1987.

[22] U.S. Department of Defense. DoD Directive 3405.2. 30 Mar. 1987.

[23] U.S. Department of Defense. Reference Manual for the Ada Programming Language ANSI/MIL-STD-1815A-1983. Washington, D.C.: GPO, 17 Feb. 1983.

[24] Verdix Corporation. Verdix Ada Development System User's Manual. Version 5.