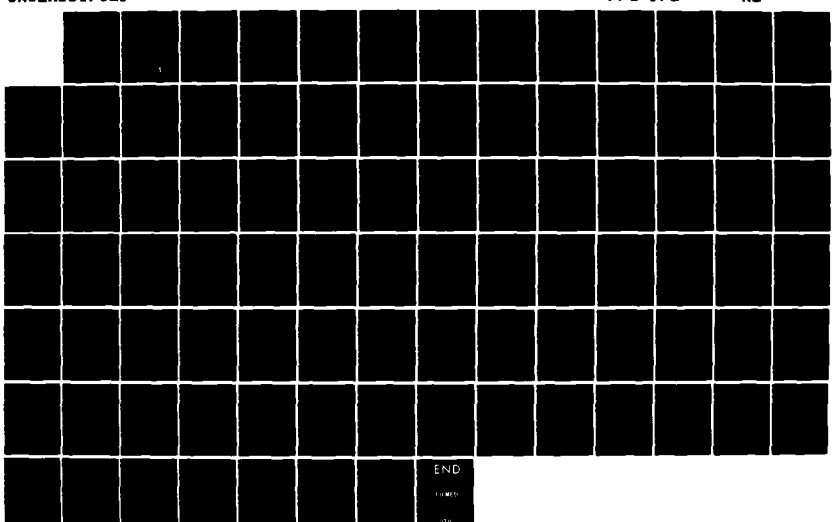
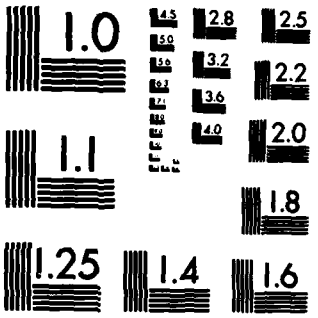
AD-A151 618 ADA (TRADEMARK) TASKING AND EXCEPTIONS: A FORMAL 1/1
DEFINITION(U) AIR FORCE INST OF TECH WRIGHT-PATTERSON
AFB OH D W GONZALEZ 1985 AFIT/CI/NR-85-28T

UNCLASSIFIED F/G 9/2 NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

UNCLASS

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER AFIT/CI/NR 85-28T	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE *(and Subtitle)* Ada Tasking and Exceptions: A Formal Definition		5. TYPE OF REPORT & PERIOD COVERED THESIS/DISSERTATION
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Dean W. Gonzalez		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS AFIT STUDENT AT: University of California, Los Angeles		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS AFIT/NR WPAFB OH 45433		12. REPORT DATE 1985
		13. NUMBER OF PAGES 85
MONITORING AGENCY NAME & ADDRESS*(if different from Controlling Office)*		15. SECURITY CLASS. *(of this report)* UNCLASS
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

DISTRIBUTION STATEMENT *(of this Report)*

ROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

STRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

SUPPLEMENTARY NOTES

APPROVED FOR PUBLIC RELEASE: IAW AFR 190-1

LYNN E. WOLAVER 28 Feb 85
Dean for Research and
Professional Development
AFIT, Wright-Patterson AFB OH

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

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

ATTACHED

DTIC
ELECTE
MAR 25 1985
S D

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASS

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

AD-A151 618

DTIC FILE COPY

85 03 11 054

28

Ada® Tasking and Exceptions:

A Formal Definition

Dean W. Gonzalez

Captain, USAF

1985

85 pages

Master of Science in Computer Science

University of California, Los Angeles

The formal language definition method used by Niklaus Wirth to describe the Euler programming language is applied to the Ada tasking and exception mechanisms. Packages are also included to the extent that they interact with tasks. A brief overview of each mechanism is given, accompanied by a detailed explanation of salient portions of the Euler method. The two phases of the definition, translation and execution, are detailed in the appendices followed by examples. Minutiae important to the design of a complementary sequential definition are detailed.

"Ada" is a trademark of the U. S. Department of Defense

AFIT RESEARCH ASSESSMENT

The purpose of this questionnaire is to ascertain the value and/or contribution of research accomplished by students or faculty of the Air Force Institute of Technology (AU). It would be greatly appreciated if you would complete the following questionnaire and return it to:

AFIT/NR
Wright-Patterson AFB OH 45433

RESEARCH TITLE: ADA Tasking and Exceptions: A Formal Definition

AUTHOR: GONZALEZ, DEAN W.

RESEARCH ASSESSMENT QUESTIONS:

1. Did this research contribute to a current Air Force project?

( ) a. YES ( ) b. NO

2. Do you believe this research topic is significant enough that it would have been researched (or contracted) by your organization or another agency if AFIT had not?

( ) a. YES ( ) b. NO

3. The benefits of AFIT research can often be expressed by the equivalent value that your agency achieved/received by virtue of AFIT performing the research. Can you estimate what this research would have cost if it had been accomplished under contract or if it had been done in-house in terms of manpower and/or dollars?

( ) a. MAN-YEARS _____ ( ) b. $_____

4. Often it is not possible to attach equivalent dollar values to research, although the results of the research may, in fact, be important. Whether or not you were able to establish an equivalent value for this research (3. above), what is your estimate of its significance?

( ) a. HIGHLY SIGNIFICANT ( ) b. SIGNIFICANT ( ) c. SLIGHTLY SIGNIFICANT ( ) d. OF NO SIGNIFICANCE

5. AFIT welcomes any further comments you may have on the above questions, or any additional details concerning the current application, future potential, or other value of this research. Please use the bottom part of this questionnaire for your statement(s).

NAME GRADE POSITION

ORGANIZATION LOCATION

STATEMENT(s):

28

UNIVERSITY OF CALIFORNIA

Los Angeles

Ada® Tasking and Exceptions:

A Formal Definition

A thesis submitted in partial satisfaction of the

requirement for the degree Master of Science

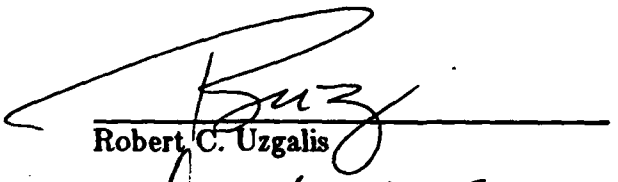
in Computer Science

by

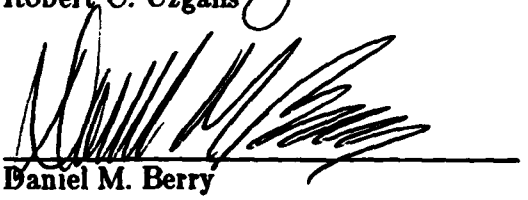
Dean W. Gonzalez

1985

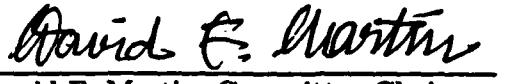
The thesis of Dean W. Gonzalez is approved.



Robert C. Uzgalis



Daniel M. Berry



David F. Martin, Committee Chair

University of California, Los Angeles

1985

TABLE OF CONTENTS

ABSTRACT OF THE THESIS

Ada® Tasking and Exceptions:

A Formal Definition

by

Dean W. Gonzalez

Master of Science in Computer Science

University of California, Los Angeles, 1985

Professor David F. Martin, Chair

The formal language definition method used by Niklaus Wirth to describe the Euler programming language is applied to the Ada tasking and exception mechanisms. Packages are also included to the extent that they interact with tasks. A brief overview of each mechanism is given, accompanied by a detailed explanation of salient portions of the Euler method. The two phases of the definition, translation and execution, are detailed in the appendices followed by examples. Minutiae important to the design of a complementary sequential definition are detailed.

"Ada" is a trademark of the U. S. Department of Defense

Introduction

This thesis presents a formal definition of the tasking and exception mechanisms in Ada[®], via the method used for the Euler programming language [Wir1][Wir2]. The definition is useful for individuals wishing a straightforward, precise characterization of these two difficult mechanisms. Programmers and system designers require such information which is not provided in the Ada standard.

The Euler description method is a two phase system using a translator and an abstract machine. The translator essentially traverses a parse tree using the postorder technique and generates a list of instructions for a virtual machine. Execution of these instructions creates the desired result. This approach retains complete control within the translated program.

This definition necessarily encompasses only part of the Ada language. Most capabilities associated with task interactions and all aspects of the exception mechanism are included. The formalization of task activation for tasks contained within packages and data access from within tasks and packages is given. Since the detailed semantics of subprograms and block statements is not part of this definition, the interaction between tasks and these units is not considered. The use of the entry family construct and representation specifications are not treated. Also the creation of task objects via declarations using explicit task types and via the evaluation of task allocators is not included. The pragma *priority* cannot apply since each task runs on its own processor and the pragma *shared* is not implemented.

"Ada" is a trademark of the U. S. Department of Defense

Previous Work

The Ada language has undergone several revisions, culminating in the latest and standard informal definition, MIL-STD-1815A [DoD]. Major modifications were made to the tasking mechanism of Preliminary Ada invalidating formal definitions created prior to the standard's release. Several pre-standard formal definitions exist. The most notable, perhaps, is the Inria definition which accompanied the release of Preliminary Ada [Inria]. It was an attempt at a denotational definition, but it fell short of the mark as it was never completed and did not include tasking. Another definition, using the Vienna Definition Method was made by Bjørner [Bjø1][Bjø2]. This approach uses a syntactically sugared λ-calculus, and a meta parallel processing mechanism to define tasking. An operational approach to the semantics of tasking and exceptions only, using labelled transition systems was made by Li and is very concise [Li].

The SEMANOL meta-programming language was used to create a multiprocessor definition of full Preliminary Ada [Belz]. Standard SEMANOL was extended to include Dijkstra's P and V operations and a 'co-compute' command which causes virtual simultaneous initiation of SEMANOL processes. These were sufficient to define a tasking model using a virtual multiprocessor environment.

Recent efforts model the Ada standard and consist mainly of a high-level supervisor or kernel that controls all tasks via subroutine calls or message passing. Riccardi has placed emphasis on task creation, activation, and termination [Ric]. He specifies how each of these mechanisms is implemented with subroutine calls to a runtime supervisor and notes that the remainder of the tasking requirements are specified in another report [Bak]. A set of ker-

nels, each managing a single processor in a networked system, communicate with tasks and each other in the message-based system of Weatherly [Wea]. Another method uses a set of software modules to model the essential features of Ada tasking and is implementation oriented to allow a bootstrap addition of concurrency to a sequential Ada environment [Lea]. Each of these designs approach a tasking model by using a higher-level set of software routines to control program units.

In contrast to all other approaches, Brindle et. al. [Bri] directly interpret a parse tree. They specify execution routines that are invoked for each node in the tree. Their system is a formalization of what they implemented in the extended interpreter for the Arcturus Programming Environment [Sta][Tay].

Exceptions

Execution of statement sequences occasionally gives rise to errant conditions. It is inconvenient to program checks for all conditions such as division by zero, index out of bounds, and various constraint errors in typical programming languages that provide only the conditional statement for detection. The exception mechanism in Ada allows us to "catch" these conditions by indicating that they require special attention. Exception names must be declared but the language also specifies predefined exception names that are not declared. Exception declarations occur in a declarative part by using the reserved word **exception**. One or more exception handlers may also be provided within the scope corresponding to the declarative part. Each handler associates an exception name or set of names with a sequence of statements. A handler using the reserved word **others** is used by all exception names that have no associated handler.

When an exception is raised, a search is made for an exception handler in the current scope. If no handler exists, program control is passed to the dynamic predecessor scope and the exception is raised in that scope. This is called propagation. Likewise, if the name of the exception raised and the universal matching name **others** is not specified in a handler the exception is propagated. The exception is thus propagated until an appropriate handler is found, the containing task is exited, or the main program is exited. Exit from a task causes the task to be completed while exit from the main program causes program execution to cease. If a handler is found, the sequence of statements associated with the exception name is executed and then the scope is exited. Notice that execution of the sequence that raised the exception cannot be resumed, nor, in the general case, retried. Therefore exceptions are mechanisms that allow programmers to gracefully exit scopes that may contain handlers. If an exception is raised within the handler, control is passed to the dynamic predecessor scope and the new exception is raised in this new scope.

Exceptions may be raised explicitly or implicitly. The **raise** statement when used with an argument raises the named exception. When used without an argument, which is only allowed within an exception handler, it causes the last exception raised to be propagated to the dynamic predecessor scope. Exceptions are raised implicitly by detection of predefined exceptional conditions by the run-time system. For example, they may be detected during a mathematical operation as in the case of division by zero, or during a data reference as in the case of array index out of bounds. These implicit exceptions have predefined names, are not declared, and programmers may use them in the same way that explicitly declared exception names are used.

Packages

Ada contains three types of program units: subprograms, packages, and tasks. Functions and procedures alone comprise the subprogram unit and should be familiar to the reader. Packages are physical collections of program units, object and type declarations, and other items that are declared within a package specification or package body. In this paper only a definition of the inclusion of tasks in packages and how packages access data is given. Data access is described in a separate section.

Packages consist of two parts, a *specification* that defines the package's interface to the containing program and a *body* that implements the actions the package performs. Each part can contain its own complete set of declarations. Tasks contained in package specifications must be created during elaboration of the declarative part of the corresponding package body. Such tasks must also be activated immediately after elaboration of that declarative part. This is also true of tasks declared in the package body's declarative part. Although packages are passive program units, package bodies may contain a sequence of statements which is executed as part of the elaboration of the body.

Tasks

The task program unit allows us to write concurrent algorithms since each task runs in parallel with all other active tasks. This concurrency is asynchronous but synchronization points, called entries, can be specified by the programmer. Tasks may also communicate by sharing global data but synchronization is not guaranteed.

Task units are composed of a *specification* and a *body*. Task specifications

define the parts of the unit visible to the containing program while the body specifies the actions a task performs during execution. The specifications and bodies are declared in the declarative part of any program unit. The declaration of a task specification may create an explicit task type or an anonymous task type. The explicit task type is used with other variable declarations to create task objects while the anonymous type is merely a side effect of task object definition using only the task specification. For example, the Ada sequence

```
declare
task type A_TASK is
  . . .
  . . .
end A_TASK;
task_one, task_two: A_TASK;
```

declares two task objects, task_one and task_two using the explicit task type, A_TASK. Declaration of one task object by using an anonymous task type is done as follows:

```
declare
task TASK_ONE is
  . . .
  . . .
end TASK_ONE;.
```

Task objects may also be created dynamically using access values (Ada pointers). The declaration,

```
declare
type A_TASK_POINTER is access A_TASK;
one_task, another_task: A_TASK_POINTER;
```

following the explicit task type declaration above allows us to create tasks at execution time by using an allocator which returns an access value. Therefore,

```
one_task ← new (A_TASK);
another_task ← new (A_TASK);
```

defines, creates, and activates two task objects at execution time.

The body of a task unit is associated with an unbounded number of task objects. The following body is used by both 'one_task' and 'another_task'.

```
task body A_TASK is
   . . .
   . . .
end A_TASK;
```

Task objects are created and activated only at execution time. Creation of a task object associates the object name with a task unit. Activation of a task object consists of elaboration of the body's declarations followed by execution of the code contained in the body. Those objects that are fully specified within a declarative part are created during elaboration of that part and are generally activated after elaboration of the containing declarative part. The only exception is that task objects declared in a package specification are activated after elaboration of the declarative part of the corresponding package body. Evaluation of allocators (denoted by the reserved word **new**) creates and activates task objects and returns an access value for the object.

After a task object is activated, it executes in asynchronous parallel with all other task objects that are activated. A method of controlling their execution is necessary if other program units are to interact with task objects. Ada implements the rendezvous concept whereby two independent tasks meet during execution at a specific point in their respective bodies and possibly exchange data. This meeting occurs at a specific entry point by one task calling an entry (the calling task) and one task accepting the entry (the called task). These actions are effected by the entry call statement and the accept statement, respectively.

Entry points are declared in the task specification by using the reserved

word **entry** followed by an entry name, an optional index range, and an optional formal parameter list. Entry points are specified by their entry name and the names may be overloaded in which case the formal parameter list is used for resolving the names. The optional index range allows an entry declaration to refer to an entire family of entries distinguished only by the actual index value.

Ada provides several variations of the entry call and accept statements. All are described well by [Booch] and here only the simplest form of each is examined. The form of the entry call statement includes only an entry name, an entry family (actual) index value and an optional actual parameter list. We will examine only single entries, distinguished by the fact that they have no index value. The accept statement contains the reserved word **accept**, an entry name, and entry family (actual) index value, the formal parameter list declared for this entry point and an optional body. Again the entry family index value will not be considered and only single entries are allowed.

At entry points, tasks meet. When a calling task arrives at an entry point, by executing an entry call statement, if no called task has previously arrived at the same entry point, the calling task is suspended. Similarly, when a called task arrives at an entry point, by beginning execution of an accept statement, if a calling task has not previously arrived at the same entry point, the called task is suspended. Called and calling tasks that meet respond as follows: **in** and **inout** formal parameters of the entry are bound to the actual parameters of the entry call statement, the calling task remains suspended while the body of the accept statement is executed, after completion of this body the accept statement is complete. **Out** and **inout** formal parameters have their values returned to the calling task and finally both

tasks continue asynchronously in parallel. It is important to note that an accept statement body may contain any kind of statement.

This simple mechanism is extended by the selective wait, conditional entry call, and timed entry call statements. The selective wait allows a called task to wait, possibly for a limited amount of time, for only one of a set of entry points for rendezvous. The conditional entry call statement makes a calling task issue a call for an entry point that allows an immediate rendezvous. The timed entry call insures that if a rendezvous cannot begin in a specified amount of time an alternative sequence of statements is executed.

Exceptional conditions may arise during the processing of a rendezvous. These conditions require that specific exceptions are raised in either the called or calling tasks. Exceptions that are not handled within a task body cause the task to be completed.

The master-dependent relationship of program units is defined as follows. Task bodies, subprogram bodies, block statements, and package bodies may create task object declarations and hence may be parent scopes. A task's master is its nearest containing parent scope that is not a package body and that task is a dependent of the master.

Finally, how tasks complete execution is important. Tasks complete normally by reaching the end of the sequence of statements comprising their body. A completed task is terminated only when all of its dependents are terminated or when those that are not terminated are waiting at an open terminate alternative of a selective wait statement. An abort statement causes a task to become *abnormal*. The abort statement may occur anywhere in a statement sequence and can abort any task whose name is visible at that point. Aborting a task causes all its dependent tasks to become abnormal

also. Abnormal tasks become completed if they are not in rendezvous, those that are in rendezvous are completed after completion of the rendezvous. Exceptions are raised in tasks that interact with abnormal tasks.

Details of the Definition

The Euler definition method consists of two phases which correspond to the traditional translation and execution phases, respectively, of the software life cycle. Phase I operates on a source file input of a particular programming language and creates the input to Phase II, in our case a set of tables and a list of machine instructions. Phase II "executes" the instruction list using an abstract machine architecture.

The translation phase is defined by a set of productions and a set of effects, one effect for each production (Appendices 3, 4, and 5). An implicit bottom-up parser which utilizes a two part stack is applied in this phase. One part of the parser stack, **S**, is used for reducing source language phrases and the other part is a value stack , **V**, used to hold semantic values which are in effect passed to other nodes in the parse tree. Reductions are performed by replacing a phrase identified as the right part of a production on the top of **S** by the nonterminal symbol on the left part of the production. If reduction cannot occur then a new token is shifted from the input onto the stack. Reduction of formal parameter specifications must shift the entire specification string onto V. Immediately prior to reduction the list of instructions in the associated effect is performed, after which reduction occurs. The pointer **i** used in the set of effects always points to the top element of **S** during reduction.

This phase can be simulated by traversing the parse tree using postorder

and simulating the instructions listed in the effect for the reduction applied at the desired node. After all the effects are applied, a list of instructions is present in the object code array **P**. These instructions are executed using Appendices 2 and 6 and the abstract machine structure described below.

Several data structures are used during Phase I to aid translation. They are listed in Appendix 1 under the Phase I only and Phases I and II sections. The Phase I only section contains structures for unique package name generation and name lookup. The **activation_list_stack** keeps a list of declared task names which must be included in an activate instruction after reduction of a declarative part. Package specifications that declare tasks cannot activate them. So an intermediary structure, **package_table**, is used to hold the names of those tasks; they become objects of the **activate** instruction in the corresponding package body.

Each declarative part allows us to declare tasks and packages. The visibility rules of Ada therefore permit us to create more than one task or package with identical names. Since in this definition there is a single task table and only one package table, unique names are generated for indices to these tables. Each reduction of declarative part must therefore allow for mapping a non-unique name string of a task or package in a particular scope to the unique name belonging to it. The unique names are generated by simple counters and the mapping is done using a stack of lists for tasks, **task_name_list_stack**, and one for packages, **pack_name_list_stack**. Upon entry to a declarative part that can contain packages and tasks a null list is placed on top of each stack and as packages and tasks are encountered their external and unique internal names are added to the appropriate stack's top element. Each element of the stack is a list of (non-unique_name,

unique_name) pairs. A function **assoc**, an extension of the LISP function *assoc*, is used to access names in the stack. The LISP function *assoc* takes a search key and a list of pairs and returns from the pair list the first pair whose first element is equal to the search key. The function **assoc** extends *assoc* by including searching through the stack to find the topmost list containing the search key (a non-unique name) and the result is the second element of the pair that would be returned by *assoc*. If no match is found then the name is not visible and an error occurs, halting processing.

The parallel Ada machine consists of an unbounded number of processors, one for each task in a program, each with its own set of data structures, and a single set of global data structures. Each processor follows the machine cycle of Appendix 2 and contains a set of compound structures and a set of simple registers. The most important global structures are the program array **P**, **task_table**, **master_table**, and **package_table**.

The program array at execution time contains the instructions created during translation. All processors reference the same array by an instruction pointer, **k**.

Since a number of tasks may be declared in a program, **task_table** is associated with the program and contains all information concerning the status of the program's tasks. Each task executes on its own processor. At times, a processor may modify the status of any task and to prevent unsynchronized interaction between separate processors, each record in **task_table** may be locked to prevent access during data modification. Each record in **task_table** records the name of its processor and keeps pointers to the program array for the task's declarative part and body part. The status of each task is recorded and if the task is suspended at any time, the reason for

suspension is also recorded. During rendezvous, the calling task's record keeps the name of the called task and vice versa. During execution of an accept statement body, another accept statement or entry call statement or any other Ada statement may be encountered. It is important to implement the rendezvous name field as a stack of such fields since one rendezvous may lead to another.

To model the rendezvous mechanism, each entry (only single entries are allowed, entry families are not implemented) of a task is denoted by a record belonging, through a linked list, to the task. This record keeps the name of the entry and the formal parameter string declared in the task specification. Also, a queue exists to contain the unique task names that are waiting for a rendezvous at that entry point.

Each task is associated with the number of dependent tasks it spawns since each master task is suspended until all of its dependent tasks are activated. The number of dependent tasks that are not yet terminated is recorded in the master task's record in **task_table** since a task may not terminate until all its dependents are terminated. Finally, a count of the number of dependents waiting at an open terminate alternative of a selective wait statement is kept since if all of a task's dependents are waiting thusly, they can be terminated if the master has completed its execution.

Master_table records all master-dependent relationships by using three fields and is indexed by unique task name. A task's master's name and its first dependent's name (if any dependents exist) are recorded. If a task has more than one dependent the sibling name field is used. A task therefore has at least one dependent if the dependent field contains a task name. The remainder of a task's dependents are those tasks named in the sibling field of

all of its dependents. Extension of this mechanism to accomodate masters that are subprograms and block statements is straightforward but not within the scope of this definition.

Only two aspects of packages are of interest here. They are the proper activation of tasks as discussed above and unique data access requirements as described below. **Package_table** is used only for these two purposes and is indexed by unique package name.

During the execution phase two unique global structures are required. First, there is a method of controlling whether a processor is running or not running. The active vector, **active_proc_vec**, contains a bit for each processor. If the bit is true the processor is running and if it is false the processor is halted. **Master_stack** records the name of each master as it is encountered (remember that only tasks are masters here). Thus the top element of the stack will always contain the name of the current code segment's master. Actually, use of the stack, since it only records tasks as masters, is merely a convenience for extension of this definition to incorporate other types of masters.

Each processor executes instructions from the program array by following the machine loop of Appendix 2 and has access to all Phase II storage locations. Specific to each processor are its stack **S**, the stack pointer **t**, a display **D**, a list **guard_list** of guards, and two lists of select statement alternatives, **accept_alts** and **delay_alts**. **S** serves four functions, as an expression stack, a scope mark stack for task scopes, accept statement scopes, and package scopes, an event mark stack for recording specific important events (discussed below), and a display scope stack. Display scopes are created on entry to task bodies, subprograms contained in packages, and accept statement bodies and

rely on interaction between a processor's display **D** and stack **S**. This is discussed further in the Data Access section. The three lists **guard_list**, **accept_alts**, and **delay_alts** along with the processor registers **term_alt** and **else_alt** record all data required to implement selective wait statements. The **guard_list** points to the code for each alternative of a selective wait. Each alternative is guaranteed to begin with at least the trivial guard, true, by the action of Phase I. Therefore it is possible to evaluate each guard and determine if the alternative is open or closed. Open alternatives are recorded in one of the four structures denoted above and after all alternatives are examined, the appropriate action is taken.

Specific registers belong to each processor. They record the processor's name, the name of the task it is running, an instruction pointer, an event mark pointer to record interesting events (e.g. encountering a selective wait statement), and a unique scope identifier generator whose use is outlined in the Data Access section. The mark pointer **mp** records dynamic scope environment marks and is important for proper exception propagation. Any processor can of course access the global data structures, but it can also access another processor's structures simply by subscripting the structure's name with the target processor's name. For example, $S_n[i_n]$ refers to the top element of processor n's run-time stack.

Processors, and hence tasks, communicate internally (i.e. as part of the run-time system) in two ways; they insert state information in **task_table**, both for their task and in a few cases for other tasks, and each processor has a mailbox to which message packets are sent. Messages are processed as part of the machine cycle loop. They are used solely for informing a task that it must process an exception implicitly raised by some other task due perhaps to

a rendezvous anomaly. Mailboxes and messages are used since no synchronization point is available for recognizing exceptional conditions which might be raised at any time.

The details of exception processing are less involved. At translation time **except** instructions are generated in each scope that may contain a handler for each declared and each predefined exception name. If a sequence of statements within a handler is specified for any of these names, a pointer is added to the instruction so that at run-time an exception handler reference is placed on **S**. When an exceptional condition is generated, the current scope's stack is searched for the named condition. If no occurrence is found in the scope and no handler exists for the universal matching name **others**, the scope is exited and control passes to its dynamic predecessor where the exception is again raised. As stated above, tasks that do not handle an exception are completed and main programs that do not handle an exception should likewise be completed. The latter action is not specified here because subprograms are outside of this definition's scope.

Ada requires that certain execution anomalies such as index out of bounds and division by zero generate exceptions using the predefined exception names. Obviously, detection of these conditions requires a mechanism embedded in the variable assignment function, for detecting constraint errors, the expression evaluation function for detecting division by zero, and the array index evaluation function for detecting index errors. These implicit mechanisms must parallel the one used in Appendix by **except_macro**

Special marks are used in this definition to identify the occurrence of various specific situations. They are indexed by **ep** and are chained with a single link. Each chain is separated by procedures, blocks, package bodies, and

accept statement bodies which use the **mp** pointer to identify dynamic scopes. The marks and the functions which create them are listed in Appendix 2. A brief explanation of the use for each mark is given below:

Exception Mark

When control is within an exception handler, this mark is created so exceptions produced therein are propagated outside the handler.

Selective_wait Mark

Several constructs have different actions when contained within a selective wait statement as noted by this mark.

Task Mark

Tasks must be completed if exceptions are not handled within the body; this mark is at the bottom of the processor's stack.

Timed Mark

Before a timed entry call is issued, the delay expression is evaluated and an entry in the delay list is made. This mark records the location of the delay expression before the timed entry call statement is entered.

Timed_call Mark

This mark is used to direct control upon entering a timed entry call statement.

A definition of the sequential mechanisms of Ada must include certain details if it is to complement this definition. These details are enumerated in Appendix 8.

Data Access

Each task requires access to a set of global variables, the visible data. Chirica et. al. [Chir] modified the Euler machine to use the *display* method of addressing and they further adapted the model to apply in the multiprocessor environment of parallel Euler. A *display* is a list of elements representing visible scopes. Each element contains a processor name, run-time stack index, and a unique scope id and specifies a visible set of data. Each processor is referenced by its name through the *display*.

In parallel Euler, each parallel clause contains no declarations and therefore the *display* of each new processor is simply a copy of its parent's *display*. In the Ada tasking model each task can have declarations so the parent's *display* must be supplemented by an additional element. That element points to a task mark which contains the task's local variable storage list along with a unique scope identifier, and is the first element on the task's execution stack. Data references are made in the identical manner of the DPEM using the @ instruction which creates a reference including a processor name, a mark index, a unique scope identifier, and a multilevel ordinal number, uniquely specifying the data element.

Packages may be collections of subprograms, so when a particular subprogram within a package is invoked it has access to all data in the textual ancestors of the package as well as the package's data and the subprogram's local data. This access is denoted by the *display* of the package (contained in **package_table**) augmented at run-time to include the subprogram's data space. Only the process of creating the package's *display* is defined, in Appendix 4.

Accept bodies require an augmented *display* that contains a binding of

actual parameters to formal parameters. This process is accomplished by the **accept_rendezvous** instruction which creates an **accept_body_mark** on the stack **S**, containing a unique scope identifier and the parameter storage list. A copy scheme is used for passing parameters. Actual parameters are contained in a list as the top element of the calling task's stack, upon issuance of the entry call. Upon acceptance of the rendezvous, all parameters of mode **in** and **inout** are dereferenced, if necessary, and a copy of the accessed data is inserted in the actual parameter storage list. This is also done for all parameters whose type is an access type. Parameters whose mode is **out** are represented in the storage list by an empty field. Upon completion of the rendezvous, all parameters of mode **inout** and **out** are copied back. The accept body mark requires two fields to implement this scheme, a copy of the actual parameter list and a list of formal parameter modes.

Examples

Included in this thesis are two examples, in Appendix 7, which illustrate important portions of this definition. Each example is a task body that contains various combinations of task units, exception handlers, and package units. The examples are presumed to exist directly within an Ada program unit or block statement. Therefore, the list of abstract machine instructions listed with the examples, since they correspond to a task body, must be executed on some preexisting processor acting within a given scope. That is, the outermost task body of each example must have been activated. The first example illustrates interaction between tasks using each type of entry call statement and each type of accept statement. The second example demonstrates packages that define a task unit, and use of the exception mechanism.

While these examples are not intended to demonstrate all (nor even most) of the possible interactions between tasks, packages, and exception handlers they do illustrate important features of the execution of programs containing these mechanisms.

Conclusion

The parallel Euler definition method allows the creation of an operational definition of a concurrent programming language free from constraints imposed by process schedulers. By assuming the existence of an unbounded number of processors creation of a definition in the ideal spirit of concurrency is possible. A definition of a major subset of Ada tasking features and their interaction with exception handling has been presented in this style. This definition includes a set of Ada mechanisms; exceptions, tasking, and packages, comparable to the scope of previous definitions. Unlike some previous definitions this paper presents a completely self-contained unit where control of a mechanism always resides within the program unit containing the use of that mechanism. That is, task units maintain control within themselves and rendezvous statements also retain control within themselves. The contribution of this work therefore is the application of a well-known formal definition method to the description of complex Ada mechanisms. This contribution results in an abstract machine approach of program translation and execution that can be used as a guide for implementation.

The creation of task objects via declarations using explicit task types and via the evaluation of task allocators is not included in this definition. These actions are more appropriately handled by a definition of the entire object creation mechanism which must copy parts of this paper, particularly con-

cerning the activation of tasks. The detailed semantics of subprograms and block statements is beyond the scope of this definition so the interaction between tasks and these units also is not included. The definition of interaction is straightforward, as it is an extension of the interaction between tasks and packages and can be added to this paper if a semantics of subprograms and block statements is available. Finally, the semantics of entry families is not defined. The formalization of entry family semantics requires the dynamic association of entry call queues with actual index values provided in entry call statements and accept statements and is beyond the scope of this definition.

APPENDIX 1
Data Structures

Phase I only:

```
activation_list_stack: stack of              -- known task names
                list of integer;
my_pack_name,
  my_task_name: integer;                     -- unique names
pack_name_string,
  task_name_string: string;                  -- non-unique names
unique_pack_name: integer init(0);           -- unique name generator
pack_name_list_stack: stack of               -- all visible package names
          list of (string,                   -- non-unique name
                integer);                    -- unique name

accept_name_stack: stack of string;          -- used to check entry name
                                             -- ending an accept statement

exception_list: list of string;              -- list of visible exception names

work_list: list of string;                   -- temporary

N: array[1..maxint] of list of ();           -- symbol table has arbitrary lists

m, n, r, s, t:  integer init(0);             -- indices for N
```

Phases I and II:

```
entry_record: record of                -- one record for each entry
  entry_name:   string,                -- our entry name
  formal_part:  string,                -- the declared textual string
  call_queue:   list of string,        -- list of task names
                                       -- waiting for rendezvous
  next_entry:   ptr,
end record;

master_record: record of
  master:     integer,
  child:      integer,       -- if more than one dependent,
  sibling:    integer,       -- sibling link is used
end record;
```

```
package_record: record of
  display: list of
    (integer,                          -- processor name
     integer,                          -- mark index (into S)
     integer) init (),                 -- unique scope id
  has_body: boolean init false,
end record;

task_record: record of
  lock:          boolean          -- allow or disallow access
                 init (false),    -- to entire record
  entry_ptr:     ptr,             -- ptr to first entry
  proc_name:     integer,         -- processor for this task
  status:        (not_activated, activating, deactivate, suspended,
                 activated, abnormal, complete, terminated)
                 init (not_activated),
  suspended_at:  string,          -- one of: timed,
                                  --   (call, {entry_ptr}, {task_name})
                                  --   (accept, {entry_ptr})
                                  --   select, delay
  rendezvous_with: stack of string,    -- task we are rendezvousing with
  body_code:     integer,              -- pointer to our code
  decl_part_code: integer,             -- pointer to code for elaboration
  task_type:     boolean,              -- are we ?
  dep_count:     integer init (0),     -- count of dependents not terminated
  activating_count: integer,           -- dependents not activated yet
  wait_count: integer init(0),
     -- count of dependents waiting on open term alt in a select statement
end record;

master_table:  array [(task, block, subprogram, library_package),
                 1..maxint] of master_record;
      -- indexed by master type and name

task_table: array [1..maxint] of task_record;
      -- indexed by unique task name

package_table: array[1..maxint] of package record;
      -- indexed by unique package name

unique_task_name: integer init (0);    -- unique name generator

task_name_list_stack: stack of         -- all visible task names
                 list of
                 (string,              -- non-unique name
                  integer);            -- unique name

P: array[1..maxint] of list of ();     -- arbitrary lists for translated program

k: integer init (0);                   -- index for P
```

Phase II only:

```
active_proc_vec: array[1..maxint] of boolean;    -- one bit for each proc
                                                 -- false → proc must sleep
                                                 -- true → proc must run

master_stack: stack of    -- top is master to code being executed
              integer;    -- only tasks are masters herein
```

Phase II only: (one of the following for each processor)

```
guard_list: list of integer;    -- list of guards to be processed

accept_alts: list of            -- open accept alternatives
        accept_alt_type;

accept_alt_type: list of
           (string,        -- entry name
            string,        -- formal part
            integer,       -- pointer to parallel_continue
            integer);      -- code for body of accept

delay_alts: list of          -- open delay alts
        delay_alt_type;

delay_alt_type: type list of
              (integer,      -- delay value
               integer);     -- code for this alt

D: list of                   -- display
     (integer,               -- processor name
      integer,               -- mark index (into S)
      integer);              -- unique scope id
```

Phase II only: (processor registers)

```
my_task_name: integer;    -- task we are running

ep:   integer init (0);   -- event pointer
                          -- used to keep track of interesting things 'mp'
                          -- doesn't know about

exception_reg: string;    -- most recent exception raised

term_alt: boolean;        -- open terminate alt?

else_alt: integer;        -- if non-zero, code pointer for else part

k,                        -- instruction pointer
scope_id,                 -- scope id reg
proc_name,                -- name of this processor
t: integer init(0);       -- temporary register
```

APPENDIX 2
Run Time Model

Machine cycle loop:

```
cycle:  k ← k+1
T:      obey rule designated by P[k][1]
        do while length(mailbox) ≠ 0
          if hd(mailbox)[1] = 'raise' then
            exception_reg ← hd(mailbox)[2]
            mailbox ← tl(mailbox)
            except_macro
          endif
        enddo
        goto cycle
```

General routines:
-- Each use of these routines is denoted by boldface.

↓ dereference operator

assoc(elem, list_stack)
  return the *cdr* of the first pair in the list nearest the top of **list_stack** which has a *car* of **elem**.

clock_time()
  return the current value of the system clock

dep_count_macro(task_name)
  Decrement by one the dep_count of the master of **task_name**. If the master's dep_count becomes zero and master has completed execution, terminate master.

deps_of(task_name)
  return the number of dependents of **task_name** not terminated

Error
  active_proc_vec[proc_name] ← false -- halt errant processor

find_entry_ptr(entry_name)
  using knowledge of visible tasks, number and types of actual parameters on the run-time stack, return a pointer to the appropriate entry

find_task_name(entry_name)
  using knowledge of visible tasks, number and types of actual parameters on the run-time stack, return the name of the task containing the called entry

lock(task_name)
 do while task_table[**task_name**].lock -- loop until unlocked
 enddo
 task_table[task_name].lock ← true

min(delay_list)
  return an element of **delay_list** whose time element is at least as small as that of every other element.

new_processor
  acquire a new processor, call it t
  $i_t \leftarrow id_t \leftarrow 0$

parm_modes(parm_string)
  arm_string is a formal parameter specification string. From it, determine and return a list of terms where each may be in, inout, or out corresponding by position to the formal parameter's mode. Parameters whose type is an access type are associated with accessin for the mode in and accessout for the modes inout and out.

pop(stack)
  remove the top element of **stack**.

push(stack, elem)
  place **elem** on top of **stack**.

send_message(task_name, action, action_parm)
  proc ← task_table[**task_name**].proc_name
  $mailbox_{proc} \leftarrow mailbox_{proc}$ & (**action**, **action_parm**)

sleep(proc_name)
  active_proc_vec[**proc_name**] ← false

top(stack_name)
  return the top value on **stack_name**

truncate(list)
  remove the last element from **list**

unlock(task_name)
 task_table[**task_name**].lock ← false

```
wake(proc_name)
  if active_proc_vec[proc_name] then
    -- proc is running
    do while active_proc_vec[proc_name]        -- wait till it sleeps
    enddo
  endif
  active_proc_vec[proc_name] ← true            -- make the proc run
```

Marks and types:

Mark:	Parameter list:
exception	(**exception**, ep link)
selective_wait	(**selective_wait**, ep link)
timed	(**timed**, instruction pointer, ep link)
timed_call	(**timed_call**, task name, entry pointer, instruction pointer, ep link)

Type:	Assignment function:
accept	accept_body_mark(scope id, parameter storage list, actual parameter copy list, formal parameter mode list, mp, ep)
data reference	dataref(processor name, mark index, scope id, multilevel ordinal number)
exception handler	ehref(name string, instruction pointer)
package	package_mark(scope id, local variable storage list, mp, return address, ep)
task	task_mark(scope id, local variable storage list)

Type:	Recognition function:
accept	isaccept(element)
data reference	isref(element)
exception handler	iseh(element)
package	ispackage(element)
task	istask(element)

APPENDIX 3
Exception Rules and Effects

Exceptions: Syntax

E1) declarative_part → decl_list

E2) decl_list → decl decl_list

E3) decl_list → ϵ

E4) decl → exception_decl

E5) exception_decl → exception_list : **exception** ;

E6) exception_list → exception_list , id

E7) exception_list → id

E8) raise_statement → **raise**

E9) raise_statement → **raise** exception_name

E10) exception_option → exception_option_head exception_handler_list

E11) exception_option → ϵ

E12) exception_option_head → **exception**

E13) exception_handler_list → exception_handler_list
exception_handler_head
sequence_of_statements

E14) exception_handler_list → exception_handler_head
sequence_of_statements

E15) exception_handler_head → **when** exception_choice_list =>

E16) exception_choice_list → exception_choice_list |
exception_choice

E17) exception_choice_list → exception_choice

E18) exception_choice → id

E19) exception_choice → **others**

E20) decl → task_declaration

E21) decl → package_declaration

E22) decl → task_body

E23) decl → package_body

Exceptions: Effects

```
E1) -- put an empty field on exception list to mark this scope
    exception_list ← exception_list & ''
    -- make a reference instruction for each predefined exception
    N[n ← n+1] ← ('except', 'others', k ← k+1)
    P[k]       ← ('except', 'others', Ω)
    N[n ← n+1] ← ('except', 'constraint_error', k ← k+1)
    P[k]       ← ('except', 'constraint_error', Ω)
    N[n ← n+1] ← ('except', 'numeric_error', k ← k+1)
    P[k]       ← ('except', 'numeric_error', Ω)
    N[n ← n+1] ← ('except', 'program_error', k ← k+1)
    P[k]       ← ('except', 'program_error', Ω)
    N[n ← n+1] ← ('except', 'storage_error', k ← k+1)
    P[k]       ← ('except', 'storage_error', Ω)
    N[n ← n+1] ← ('except', 'tasking_error', k ← k+1)
    P[k]       ← ('except', 'tasking_error', Ω)
    -- make a reference for each exception declared in an outer scope,
    -- that is still visible
   work_list ← exception_list
    L1: if hd(work_list) = '' then work_list ← tl(work_list) endif
        if work_list = () then goto L2; endif
        if hd(work_list) = '' then goto L1 endif          -- '' is scope marker
        N[n ← n+1] ← ('except', hd(work_list), k ← k+1)
        P[k] ← ('except', hd(work_list), Ω)
        work_list ← tl(work_list)
        goto L1
    L2:

E2) Λ

E3) Λ

E4) Λ

E5) Λ

E6) -- remember where exception name is
    N[n ← n+1] ← ('except', V[i-1], k ← k+1)
    P[k] ← ('except', V[i-1], Ω)
    exception_list ← exception_list & V[i]
```

```
E7)  -- make a reference instruction for each declared exception
     N[n ← n+1] ← ('except', V[i], k ← k+1)
     P[k]       ← ('except', V[i], Ω)
     exception_list ← exception_list & V[i]

E8)  P[k ← k+1] ← 'draise'

E9)  P[k ← k+1] ← ('raise', V[i])

E10) t ← n
     L1: if t ≤ m then Error; endif        -- got outside of scope
       if N[t][1] = 'jump' then goto L2
       t ← t-1; goto L1
     L2: s ← N[t][2]                       -- chase jump fixup chain
     L3: if s = Ω then goto L4
       r ← P[s][2]
       P[s][2] ← k+1                       -- insert instruction address
       s ← r
       goto L3
     L4: n ← t-1                           -- pop N

E11) Λ

E12) N[n ← n+1] ← ('jump', k ← k+1)
     P[k]       ← ('jump', Ω)

E13, 14) t ← n
     L1: if t ≤ m then Error; endif        -- got outside of scope
       if N[t][1] = 'jump' then goto L2
       t ← t-1; goto L1
     L2: P[k ← k+1] ← ('jump', N[t][2])
                                           -- insert backward pointer
       N[t][2] ← k

E15) s ← V[i-1]                            -- v gets a list
     L1: t ← n                             -- t walks through N
       if s = () then goto L4
       r ← hd(s)
       s ← tl(s)
     L2: if t ≤ m then Error; endif        -- got outside of scope
       if N[t][1] = 'except' and N[t][2] = r then goto L3
       t ← t-1; goto L2
     L3: if P[N[t][3]][3] ≠ Ω then Error; endif   -- first time here?
       P[N[t][3]][3] ← k + 1
       goto L1
     L4:

E16) V[i] ← V[i-2] & V[i]                  -- V[i] may be 'others'

E17) V[i] ← (V[i])                         -- make an element into a list

E18) Λ
```

E19) V[i] ← 'others'

E20) Λ

E21) Λ

E22) Λ

E23) Λ

APPENDIX 4
Package Rules and Effects

Packages: Syntax

P1) package_declaration → package_specification ;

P2) package_specification → package_spec_head declarative_part
end package_name_option

P3) package_spec_head → **package** id **is**

P4) package_name_option → *package*_simple_name

P5) package_name_option → ϵ

P6) package_body → package_body_head package_body_option **end**
*package*_simple_name ;

P7) package_body_head → package_body_header package_decl_option

P8) package_body_header → **package body** *package*_simple_name
is

P9) package_decl_option → decl_list

P10) package_decl_option → ϵ

P11) package_body_option → **begin** sequence_of_statements except_option

P12) package_body_option → ϵ

Packages: Effects

P1) Λ

P2) package_table[my_pack_name].act_list ← **pop**(activation_list_stack)

P3) my_pack_name ← unique_pack_name ← unique_pack_name + 1
pack_name_string ← V[i-1]
push(pack_name_list_stack, **pop**(pack_name_list_stack) &
(pack_name_string, my_pack_name))

P4) if V[i] ≠ pack_name_string then **Error**; endif

P5) Λ

P6) P[k ← k+1] ← 'package_end'
t ← n
L1: if t < m then **Error**; endif -- got outside of scope
if N[t][1] = package then goto L2; endif
t ← t-1
goto L1
L2: P[N[t][2]][3] ← k+1 -- make package_begin point to end
n ← t-1 -- pop N
-- remove exception names we declared from exception_list
L3: if hd(exception_list) = '' then goto L4
exception_list ← tl(exception_list)
goto L3
L4: exception_list ← tl(exception_list) -- remove our marker

P7) -- activate tasks in packages with no body
work_list ← pop(pack_name_list_stack)
L1: if work_list = () then goto L3; endif
if pack_table[hd(work_list)].has_body then goto L2; endif
-- here if package has no body
P[k ← k+1] ← ('activate', pack_table[hd(work_list)].act_list)
L2: work_list ← tl(work_list)
goto L1
L3:
P[k ← k+1] ← ('activate', **pop**(activation_list_stack) &
pack_table[my_pack_name].act_list)
-- activate tasks declared in package body and package specification

P8) pack_name_string ← V[i-1]
my_pack_name ← **assoc**(V[i-1], pack_name_list_stack)
pack_table[my_pack_name].has_body ← true
-- begin elaboration of package body
P[k ← k+1] ← ('package_begin', my_pack_name)
N[n ← n+1] ← ('package', k)
P[k ← k+1] ← ('except', Ω, 0) -- top of handler ref list
-- make a reference instruction for each predefined exception
N[n ← n+1] ← ('except', 'others', k ← k+1)
P[k ← k+1] ← ('except', 'others', Ω)

```
N[n ← n+1] ← ('except', 'constraint_error', k ← k+1)
P[k]       ← ('except', 'constraint_error', Ω)
N[n ← n+1] ← ('except', 'numeric_error', k ← k+1)
P[k]       ← ('except', 'numeric_error', Ω)
N[n ← n+1] ← ('except', 'program_error', k ← k+1)
P[k]       ← ('except', 'program_error', Ω)
N[n ← n+1] ← ('except', 'storage_error', k ← k+1)
P[k]       ← ('except', 'storage_error', Ω)
N[n ← n+1] ← ('except', 'tasking_error', k ← k+1)
P[k]       ← ('except', 'tasking_error', Ω)
-- now mark this new scope in several places
push(task_name_list_stack, ())
push(pack_name_list_stack, ())
exception_list ← exception_list & ''
```

P9) Λ

P10) Λ

P11) Λ

P12) Λ

APPENDIX 5
Tasking Rules and Effects

Task specification and body: Syntax

T1) task_declaration → task_specification ;

T2) task_specification → **task** id spec_option

T3) task_specification → **task type** id spec_option

T4) spec_option → spec_option_head entry_decl_repeat rep_clause
end name_option

T5) spec_option → ϵ

T6) spec_option_head → **is**

T7) entry_decl_repeat → entry_declaration entry_decl_repeat

T8) entry_decl_repeat → ϵ

T9) rep_clause → ϵ -- rep_clauses are not used

T10) name_option → *task*_simple_name

T11) name_option → ϵ

T12) task_body → task_body_head task_body_decl **begin**
sequence_of_statements exception_option
end name_option ;

T13) task_body_head → **task body** *task*_simple_name

T14) task_body_decl → decl_head task_decl_option

T15) decl_head → **is**

T16) task_decl_option → declarative_part

T17) task_decl_option → ϵ

Task specification and body: Effects

T1) Λ

```
T2)  my_task_name ← unique_task_name ← unique_task_name + 1
     task_table[my_task_name].entry_ptr ← old_entry   -- point to entry list
     if task_name_string ≠ '' and task_name_string ≠ V[i-1] then
       Error;          -- check name on end of spec
     endif
     task_table[my_task_name].task_type ← false
     push(task_name_list_stack, pop(task_name_list_stack) &
                          (task_name_string, my_task_name))

T3)  my_task_name ← unique_task_name ← unique_task_name + 1
     task_name_string ← V[i-1]
     task_table[my_task_name].task_type ← true
     push(task_name_list_stack, pop(task_name_list_stack) &
                          (task_name_string, my_task_name))

T4)  Λ

T5)  Λ

T6)  old_entry ← null
     new_entry ← new(entry_record)     -- allocate space for first one

T7)  Λ

T8)  dispose (new_entry)        -- throw away last one allocated
     if old_entry ≠ null then
       old_entry.next_entry ← null       -- terminate list
     endif

T9)  Λ

T10)  task_name_string ← V[i]            -- check the name later

T11)  task_name_string ← ''              -- don't try and check name

T12)  P[k ← k+1] ← 'complete'         -- task is complete
      -- check name at end of body
      if task_name_string ≠ '' and task_name_string ≠ V[i-7] then
        Error
      endif
      -- insert pointer to jump around this body during elaboration of parent
      t ← n
      L1:  if t ≤ m then Error; endif        -- got outside of scope
           if N[t][1] = 'jump' then goto L2; endif
           t ← t-1
           goto L1
```

```
    L2:  P[N[t][2]][2] ← k+1
    n ← t-1                                  -- pop N
    -- remove exception names we declared from exception_list
    L3: if hd(exception_list) = '' then goto L4
        exception_list ← tl(exception_list)
        goto L3
    L4: exception_list ← tl(exception_list)          -- remove our marker

T13)  my_task_name ← assoc(V[i], task_name_list_stack)
    P[k ← k+1] ← ('jump', Ω)          -- task bodies aren't elaborated
    N[n ← n+1] ← ('jump', k)            -- until they are activated
    task_table[my_task_name].decl_part_code ← k
    push(activation_list_stack, pop(activation_list_stack) & my_task_name)
    V[i-2] ← V[i]                     -- pass up name string

T14)  -- activate tasks in packages with no body
    work_list ← pop(pack_name_list_stack)
    L1:  if work_list = () then goto L3; endif
        if pack_table[hd(work_list)].has_body then goto L2; endif
        -- here if package has no body
        P[k ← k+1] ← ('activate', pack_table[hd(work_list)].act_list)
    L2:  work_list ← tl(work_list)
        goto L1
    L3:
    -- activate all of this task's children
    P[k ← k+1] ← ('activate', pop(activation_list_stack))
    -- code to elaborate declarations has been made
    task_table[my_task_name].body_code ← k

T15)  push(activation_list_stack, ())      -- more declarations
    P[k ← k+1] ← ('except', Ω, 0)  -- end of reference list
    -- make a reference instruction for each predefined exception
    N[n ← n+1] ← ('except', 'others', k ← k+1)
    P[k]       ← ('except', 'others', Ω)
    N[n ← n+1] ← ('except', 'constraint_error', k ← k+1)
    P[k]       ← ('except', 'constraint_error', Ω)
    N[n ← n+1] ← ('except', 'numeric_error', k ← k+1)
    P[k]       ← ('except', 'numeric_error', Ω)
    N[n ← n+1] ← ('except', 'program_error', k ← k+1)
    P[k]       ← ('except', 'program_error', Ω)
    N[n ← n+1] ← ('except', 'storage_error', k ← k+1)
    P[k]       ← ('except', 'storage_error', Ω)
    N[n ← n+1] ← ('except', 'tasking_error', k ← k+1)
    P[k]       ← ('except', 'tasking_error', Ω)
    -- mark beginning of this scope in several places
    push(task_name_list_stack, ())
    push(pack_name_list_stack, ())
    exception_list ← exception_list & ''

T16)  Λ

T17)  Λ
```

Entries, entry calls, accept statements, delay statement: Syntax

T18) entry_declaration → **entry** id family_option formal_option ;

T19) family_option → ϵ -- families are not used

T20) formal_option → formal_part
-- formal parameter specification string is in V[i]

T21) formal_option → ϵ

T22) entry_call_statement → *entry*_name actual_option ;

T23) actual_option → (actual_part)

T24) actual_option → ϵ

T25) accept_statement → accept_head accept_body_option ;

T26) accept_head → **accept** *entry*_simple_name index_option
formal_option

T27) index_option → ϵ -- families are not used

T28) accept_body_option → **do** sequence_of_statements **end**
entry_name_option

T29) accept_body_option → ϵ

T30) entry_name_option → *entry*_simple_name

T31) entry_name_option → ϵ

T32) delay_statement → **delay** simple_expression

Entries, entry calls, accept statements, delay statement: Effects

T18) new_entry.entry_name ← V[i-3] -- make an entry descriptor
new_entry.formal_spec ← V[i-1] -- formal spec used to resolve
-- overloaded definitions
new_entry.call_queue ← () -- no calling tasks
old_entry ← new_entry -- remember last entry
new_entry ← old_entry.next_entry ← new(entry_record)
-- make another entry and link to it

T19) Λ

T20) V[i-2] ← V[i-1] -- pass up parm spec string

T21) V[i] ← '' -- make sure spec string is empty

T22) P[k ← k+1] ← ('call_rendezvous', V[i-2]) -- call an entry
P[k ← k+1] ← 'wake_up_check' -- check to see who woke us

T23) Λ -- code is made by actual_part
-- actual parm list is left on top of S

T24) Λ

T25) P[k ← k+1] ← 'parallel_continue'
-- rendezvous is complete
t ← n
L1: if t ≤ m then **Error**; endif
if N[t][1] = 'accept' then goto L2
t ← t-1; goto L1
L2: P[N[t][2]][4] ← k -- insert pointer to parallel_continue
n ← t-1 -- pop N stack

T26) **push**(accept_name_stack, V[i-2])
P[k ← k+1] ← ('accept_rendezvous', V[i-2], V[i], Ω)
N[n ← n+1] ← ('accept',k) -- remember where statement is

T27) Λ

T28) Λ -- body code has been made

T29) Λ

T30) if V[i] ≠ **pop**(accept_name_stack) then **Error**; endif

T31) Λ

T32) P[k ← k+1] ← 'delay' -- delay length will be on stack

Select statements: Syntax

T33) select_statement → selective_wait

T34) select_statement → conditional_entry_call

T35) select_statement → timed_entry_call

T36) selective_wait → select_head select_alternative select_alt_repeat
else_option select_tail

T37) select_head → **select**

T38) select_alternative → guard_option select_wait_alternative

T39) select_alt_repeat → **or** select_alternative select_alt_repeat

T40) select_alt_repeat → ϵ

T41) select_tail → **end select ;**

T42) else_option → else_head sequence_of_statements

T43) else_option → ϵ

T44) else_head → **else**

T45) guard_option → guard_head condition =>

T46) guard_option → ϵ

T47) guard_head → **when**

T48) select_wait_alternative → accept_statement sequence_option

T49) select_wait_alternative → delay_alternative

T50) select_wait_alternative → **terminate ;**

T51) delay_alternative → delay_statement sequence_option

T52) sequence_option → sequence_of_statements

T53) sequence_option → ϵ

T54) conditional_entry_call → select_head immediate_entry_call
sequence_option immediate_else_part
select_tail

T55) immediate_entry_call → *entry*_name actual_option ;

T56) immediate_else_part → immediate_else_head sequence_of_statements

T57) immediate_else_head → **else**

T58) timed_entry_call → select_head entry_call_statement
sequence_option delay_part_head
delay_alternative select_tail

T59) delay_part_head → **or**

Select statements: Effects

T33) Λ

T34) Λ

T35) Λ

T36) Λ

T37) N[n ← n+1] ← ('select_end',Ω)
-- initialize fixup chain
N[n ← n+1] ← ('timed',k ← k+1)
-- header for select
P[k] ← ('timed',Ω)
N[n ← n+1] ← ('guard',k ← k+1)
P[k] ← ('guard',())

T38) t ← n -- add to the jump chain
L1: if t ≤ m then **Error**; endif -- got outside of scope
if N[t][1] = 'select_end' then goto L2
t ← t-1; goto L1
L2: P[k ← k+1] ← ('jump',N[t][2])
-- insert backward pointer
N[t][2] ← k -- point to new instruction

T39) Λ

T40) Λ

T41) t ← n -- give proper pointer to all
-- jump instructions
L1: if t ≤ m then **Error**; endif -- got outside of scope
if N[t][1] = 'select_end' then goto L2
t ← t-1; goto L1
L2: s ← N[t][2] -- find the last jump
L3: if s = Ω goto L4 -- chase fixup chain
r ← P[s][2]
P[s][2] ← k+1 -- insert the pointer
s ← r
goto L3
L4: n ← t-1 -- pop N stack

```
T42)  t ← n                          -- add to the jump chain
      L1: if t ≤ m then Error; endif      -- got outside of scope
          if N[t][1] = 'select_end' then goto L2
          t ← t-1; goto L1
      L2: P[k ← k+1] ← ('jump',N[t][2])
                                     -- insert backward pointer
          N[t][2] ← k                -- point to new instruction

T43)  Λ

T44)  t ← n                          -- make an implicit guard
      L1:  if t ≤ m then Error; endif
           if N[t][1] = 'guard' then goto L2
           t ← t-1; goto L1
      L2:  P[N[t][2]][2] ← P[N[t][2]][2] & k+1
                                     -- add to guard list
           P[k ← k+1] ← 'else'

T45)  P[k ← k+1] ← 'eval_guard'   -- explicit guard present

T46)  t ← n                          -- make a trivial guard
      L1:  if t ≤ m then Error; endif
           if N[t][1] = 'guard' then goto L2
           t ← t-1; goto L1
      L2:  P[N[t][2]][2] ← P[N[t][2]][2] & k+1
                                     -- add to guard list
           P[k ← k+1] ← ('push,true')
           P[k ← k+1] ← 'eval_guard'

T47)  t ← n                          -- beginning of explicit guard
      L1:  if t ≤ m then Error; endif
           if N[t][1] = 'guard' then goto L2
           t ← t-1;  goto L1
      L2:  P[N[t][2]][2] ← P[N[t][2]][2] & k+1        -- add to guard list

T48)  Λ

T49)  Λ

T50)  P[k ← k+1] ← 'terminate'

T51)  Λ

T52)  Λ

T53)  Λ

T54)  Λ

T55)  P[k ← k+1] ← ('immediate_call_rendezvous', V[i-2], Ω)
      N[n ← n+1] ← ('immediate', k)
```

T56) A

```
T57) t ← n                          -- add jump instr to chain
    L1: if t ≤ m then Error; endif
        if N[t][1] = 'select_end' then goto L2
        t ← t-1; goto L1
    L2: P[k ← k+1] ← ('jump', N[t][2])
                                    -- insert backward ptr
        N[t][2] ← k                 -- point to new instr
        t ← n
    L3: if t ≤ m then Error; endif      -- fixup immediate_rendezvous
        if N[t][1] = 'immediate' then goto L4
        t ← t-1; goto L3
    L4: P[N[t][2]][3] ← k+1         -- insert pointer to else part
        n ← t-1                     -- pop N stack
```

T58) A

```
T59) t ← n                          -- add jump instr to chain
    L1: if t ≤ m then Error; endif
        if N[t][1] = 'select_end' then goto L2
        t ← t-1; goto L1
    L2: P[k ← k+1] ← ('jump', N[t][2])
                                    -- insert backward ptr
        N[t][2] ← k
        t ← n
    L3: if t ≤ m then Error; endif
        if N[t][1] = 'timed' then goto L4
                                    -- point to delay part
        t ← t-1; goto L3
    L4: P[N[t][2]][2] ← k+1
        n ← t-1                     -- pop N stack
```

Abort statement: Syntax

T60) abort_statement → **abort** *task*_name task_name_repeat ;

T61) task_name_repeat → , *task*_name task_name_repeat

T62) task_name_repeat → ϵ

Abort statement: Effects

T60) P[k ← k+1] ← ('abort', **assoc**(V[i-2], task_name_list_stack) & V[i-1])

T61) V[i-2] ← **assoc**(V[i-1], task_name_list_stack) & V[i]

T62) V[i] ← ()

APPENDIX 6
Machine Level Instructions

```
-- This appendix also contains macros (no procedures) that are
-- invoked by various instructions.  The instructions are denoted
-- by boldface.
-- Everything is in alphabetical order by instruction name and macro name.


abnormal_check

-- at synchronization points check if our task became abnormal
if task_table[my_task_name].status = abnormal then
  lock(my_task_name)
  task_table[my_task_name].status ← complete
  comp_check_macro(my_task_name)
  unlock(my_task_name)
  -- wait until all dependents are terminated
  do while task_table[my_task_name].dep_count ≠ 0
  enddo
  lock(my_task_name)
  task_table[my_task_name].status ← terminated
  unlock(my_task_name)
  dep_count_macro(my_task_name)
  sleep(proc_name)
endif


abort, task_name_list

locals
  name_list: list of integer
end locals
abnormal_check              -- synchronization point
name_list ← task_name_list
do while name_list ≠ ();
  abort_tree(hd(name_list), deps_of(hd(name_list)))
  name_list ← tl(name_list)
enddo


abort_tree(task_name, dep_list)
-- make task_name abnormal, abort each of its dependents, then
-- complete or terminate task_name
```

```
locals
  task_name: integer
  dep_list: list of integer
  entry_ptr: pointer
  called_task: integer
  queue, queue_head: list of integer
end locals
if task_table[task_name].status ≠ terminated then
  -- don't abort terminated tasks
  lock(task_name)
  task_table[task_name].status ← abnormal
  unlock(task_name)
  if dep_list ≠ () then
    task_table[task_name].status ← abnormal
    do while dep_list ≠ ()
      abort_tree(hd(dep_list), deps_of(hd(dep_list)))
      dep_list ← tl(dep_list)
    enddo
  endif
  -- check for suspended abnormal tasks
  lock(task_name)
  if task_table[task_name].status = suspended and
  (task_table[task_name].suspended_at = delay or
  task_table[task_name].suspended_at = timed or
  task_table[task_name].suspended_at = accept or
  task_table[task_name].suspended_at = select) then
    -- complete these abnormal tasks
    task_table[task_name].status ← complete
    comp_check_macro(task_name)
    if task_table[task_name].dep_count = 0 then
      task_table[task_name].status ← terminated
      unlock(task_name)
      dep_count_macro(task_name)
      lock(task_name)
    endif
  endif
  unlock(task_name)
  if task_table[task_name].status = suspended and
    (task_table[task_name].suspended_at[1] = call and
     task_table[task_name].rendezvous_with = '') then
    -- calling task not in rendezvous
    entry_ptr ← task_table[task_name].suspended_at[2]
    called_task ← task_table[task_name].suspended_at[3]
    lock(called_task)
    -- remove from entry queue
    queue ← entry_ptr.call_queue
    queue_head ← ()
    do while hd(queue) ≠ task_name
      queue_head ← queue_head & hd(queue)
      queue ← tl(queue)
    enddo
    queue ← queue_head & tl(queue)
```

```
    entry_ptr.call_queue ← queue
    unlock(called_task)
    lock(task_name)
    task_table[task_name].status ← complete
    comp_check_macro(task_name)
    unlock(task_name)
    if task_table[task_name].dep_count = 0 then
      lock(task_name)
      task_table[task_name].status ← terminated
      unlock(task_name)
      dep_count_macro(task_name)
    endif
  endif
  if task_table[task_name].status = not_activated then
    -- complete tasks that have not yet begun activation
    lock(task_name)
    task_table[task_name].status ← complete
    comp_check_macro(task_name)
    unlock(task_name)
    if task_table[task_name].dep_count = 0 then
      lock(task_name)
      task_table[task_name].status ← terminated
      unlock(task_name)
      dep_count_macro(task_name)
    endif
  endif
  -- maybe all dependents are now terminated
  if task_table[task_name].dep_count = 0 then
    lock(task_name)
    task_table[task_name].status ← terminated
    unlock(task_name)
    dep_count_macro(task_name)
    sleep(task_table[task_name].proc_name)
  endif
endif


accept_rendezvous, entry_name, formals, k-value
-- k-value points to our parallel_continue

locals
  entry_ptr: pointer
  called_task: integer
  proc: integer
  mode_list: list of string
  ref_list: list of ()
    -- arbitrary element types in list of actual parameters from calling task
end locals
abnormal_check                    -- synchronization point
if S[ep][1] = selective_wait then
    -- this is an open accept alternative
    accept_alts ← accept_alts & (entry_name, formals, k-value, k)
```

```
    if guard_list = () then
      selective_wait                    -- do selective wait thinking
    else
      k ← hd(guard_list)-1              -- transfer to next guard
      guard_list ← tl(guard_list);
    endif;
  else
    -- now find entry in our task with entry_name and formals
    entry_ptr ← task_table[my_task_name].entry_ptr;
    if entry_ptr = null then Error; endif;
    do while (entry_ptr.entry_name ≠ entry_name or
    entry_ptr.formal_part ≠ formals );
      entry_ptr ← entry_ptr.next_entry
      if entry_ptr = null then Error; endif;
    enddo;
    lock(my_task_name)
    if length(entry_ptr.call_queue) = 0 then
      task_table[my_task_name].status ← suspended;
      task_table[my_task_name].suspended_at ← ('accept', entry_ptr)
      unlock(my_task_name)
      sleep(proc_name)
      -- wait for a call_rendezvous
      lock(my_task_name)
      task_table[my_task_name].status ← callable;
    endif;
    unlock(my_task_name)

    -- now see if we are target of timed entry call
    calling_task ← hd(entry_ptr.call_queue)
    d_lock
    if task_table[calling_task].status = suspended and
       task_table[calling_task].suspended_at = timed then
      -- timed entry call rendezvous successful,
      -- point caller to wake_up_check
      proc ← task_table[calling_task].proc_name
      k_proc ← S_proc[i_proc][4]

    -- specify we are in a rendezvous, tell calling task who we are
    lock(calling_task)
    push(task_table[calling_task].rendezvous_with, my_task_name)
    unlock(calling_task)
    lock(my_task_name)
    push(task_table[my_task_name].rendezvous_with, calling_task)
    entry_ptr.call_queue ← tl(entry_ptr.call_queue);
    unlock(my_task_name)

    -- now copy actual parameters (from caller's stack)
    proc ← task_table[calling_task].proc_name
    S[i ← i+1] ← accept_body_mark(scope_id ← scope_id + 1, (), S_proc[i_proc],
                    parm_modes(entry_ptr.formal_part), mp, ep, k-value)
    mp ← i
    ep ← 0
```

```
   -- for mode markers in, inout, accessin, and accessout copy in actuals
   -- S[i][2] is parameter storage list of accept body
   mode_list ← S[i][4]
   ref_list ← S[i][3]
   for t ← 1 to length(mode_list)
     if mode_list[t] ≠ 'out' then
       if isref(ref_list[t]) then
         S[i][2] ← S[i][2] & ref_list[t]↓          -- deref it
       else
         S[i][2] ← S[i][2] & ref_list[t]           -- no deref
       endif
     else
       S[i][2] ← S[i][2] & ()                      -- just make space
     endif
   endfor
   D ← D & (proc_name, i, scope_id)        -- augment display in accept body
   -- control will now pass to accept statement body
endif;


activate, name_list

locals
  task_list, temp_list: list of integer
  master, next_sibling, task_name: integer
  done: boolean
end locals
abnormal_check                  -- synchronization point
temp_list ← name_list
task_list ← ()
push (master_stack, my_task_name)        -- only place masters are specified
do while temp_list ≠ ();                 -- don't activate term'd tasks
   task_name ← hd(temp_list)          -- remove them from the list
   temp_list ← tl(temp_list)
   if task_table[task_name].status ≠ terminated then
     task_list ← task_list & task_name
   endif
enddo
lock(my_task_name)
task_table[my_task_name].activating_count ← length(task_list)
unlock(my_task_name)
do while task_list ≠ ();                 -- activate each task
   task_name ← hd(task_list); task_list ← tl(task_list)
   proc ← new_processor          -- initialize a new processor
   task_table[task_name].proc_name ← proc
   proc_name_proc ← proc                 -- tell proc its name
   my_task_name_proc ← task_name         -- set up proc. to run
   k_proc ← task_table[task_name].decl_part_code
   master_table[task, task_name].master ← top(master_stack)
   master ← top(master_stack)
   -- make a task mark as first element in stack
   S_proc[i_proc ← i_proc+1] ← task_mark(scope_id_proc ← scope_id_proc+1,())
```

```
    -- augment display of parent and give to task
    D_proc ← D&(proc_name_proc, i_proc, scope_id_proc)

    -- put task in dependent list of master
    task_table[master].dep_count ← task_table[master].dep_count + 1
    master ← master_table[task, master].master;
    if master_table[task, my_task_name].child = () then
      -- task is the first child
      master_table[task, my_task_name].child ← task_name;
    else
      -- task is a sibling
      next_sibling ← master_table[task, my_task_name].child
      done ← false
      do until done
         if master_table[task, next_sibling].sibling = () then
           master_table[task, next_sibling].sibling ← task_name
           done ← true;
         else
           next_sibling ← master_table[task, next_sibling].sibling
         endif;
      enddo;
    endif;
    task_table[task_name].status ← activating
    wake(proc_name)
enddo;
-- now wait till all dependents are activated
do until task_table[my_task_name].activating_count = 0
enddo
if task_table[my_task_name].status = deactivate then
    -- a dependent of ours had problems, detected by exception process
    send_message(my_task_name, 'raise', 'tasking_error')
endif
-- tell master we are activated
master ← master_table[task, my_task_name].master
lock(master)
task_table[master].activating_count ←
                        task_table[master].activating_count - 1
unlock(master)
lock(my_task_name)
task_table[my_task_name].status ← callable
unlock(my_task_name)
```

```
call_rendezvous, name
-- actual parameters are a list on top of S

locals
  entry_ptr: pointer
  task_name: integer
end locals
abnormal_check                    -- synchronization point
entry_ptr ← find_entry_ptr(name)
task_name ← find_task_name(name)

lock(task_name)
if task_table[task_name].status ≠ complete and
 task_table[task_name].status ≠ terminated and
 task_table[task_name].status ≠ abnormal then
  if S[ep][1] = timed then
    -- timed entry call
    unlock(task_name)
    t ← S[ep][2]
    -- make a mark to do the call, k points to wake_up_check
    S[ep] ← (timed_call, task_name, entry_ptr, k, S[ep][3])
    k ← t-1       -- go evaluate delay amount
  else
    -- normal call
    if length(entry_ptr.call_queue) = 0 and
      task_table[task_name].status = suspended and
      task_table[task_name].suspended_at = ('accept', entry_ptr) then
        -- called task was waiting for us
        entry_ptr.call_queue ← entry_ptr.call_queue & my_task_name
        unlock(task_name)
        wake(task_table[task_name].proc_name)
    else
      entry_ptr.call_queue ← entry_ptr.call_queue & my_task_name
                          -- put us on the call queue
      unlock(task_name)
    endif
    lock(my_task_name)
    task_table[my_task_name].status ← suspended
    task_table[my_task_name].suspended_at ← ('call', entry_ptr, task_name)
    unlock(my_task_name)
    sleep(proc_name)
    -- called task will wake us
    lock(my_task_name)
    task_table[my_task_name].status ← callable
    unlock(my_task_name)
  endif
else
  -- called task has completed
  unlock(task_name)
  send_message(my_task_name, 'raise', 'tasking_error')
endif
```

complete

```
lock(my_task_name)
task_table[my_task_name].status ← complete
comp_check_macro(my_task_name)
unlock(my_task_name)
-- wait until all dependents are terminated
do while task_table[my_task_name].dep_count ≠ 0
enddo;
lock(my_task_name)
task_table[my_task_name].status ← terminated
unlock(my_task_name)
dep_count_macro(my_task_name)
sleep(proc_name)


comp_check_macro(task)
-- check each entry of task for queued calls

locals
  entry_ptr: pointer
end locals
entry_ptr ← task_table[task].entry_ptr
do while entry_ptr ≠ null;
    do while entry_ptr.call_queue ≠ ()
      send_message(hd(entry_ptr.call_queue), 'raise', 'tasking_error')
      entry_ptr.call_queue ← tl(entry_ptr.call_queue)
    enddo
    entry_ptr ← entry_ptr.next_entry
enddo
```

delay

```
locals
  queue, queue_head: list of integer
  target_time: time
end locals
abnormal_check                    -- synchronization point
if S[ep][1] = selective_wait then
    delay_alts ← delay_alts & (pop(S), k)    -- open delay alt
    if guard_list = () then
      selective_wait                  -- do selective wait thinking
    else
      k ← hd(guard_list)-1            -- do the next guard
      guard_list ← tl(guard_list);
    endif
else
  if S[ep][1] = timed_call then
    lock(S[ep][2])
    -- issue the call
    S[ep][3].call_queue ← S[ep][3].call_queue & my_task_name
```

```
    unlock(S[ep][2])
    -- wait for rendezvous or until time is up
    do while task_table[S[ep][2]].rendezvous_with ≠ my_task_name
     if clock_time() ≥ target_time then
       -- time is up!
       lock(S[ep][2])
       queue_head ← ()
       queue ← S[ep][3].call_queue
       do while hd(queue) ≠ my_task_name
         queue_head ← queue_head & hd(queue)
         queue ← tl(queue)
       enddo
       S[ep][3].call_queue ← queue_head & tl(queue)
       unlock(S[ep][2])
     endif
    enddo
    -- if rendezvous was accepted, called task will point our k to
    -- wake_up_check, else we fall through after this delay
  else
    -- normal delay statement
    target_time ← clock_time
    if top(S) > 0 then target_time ← target_time + pop(S)
    else pop(S)
    endif
    do while clock_time() < target_time
    enddo
  endif
endif


draise

  except_macro        -- this is easy


else

else_alt ← k                        -- accept alts. have else part
selective_wait                        -- this was the last in a set of alts
```

eval_guard

```
if S[i] = false then
   t ← pop(S)                -- this alternative is closed
   if guard_list = () then
     selective_wait          -- no more guards, do selective wait thinking
   else
     k ← hd(guard_list)      -- set up to do another guard
     guard_list ← tl(guard_list)
   endif
else
   t ← pop(S)                -- alt is open, just pop exp. stack
endif
```

except

```
-- create a handler reference
if P[k][3] ≠ Ω then
   S[i ← i+1] ← ehref (P[k][2], P[k][3])
endif
```

except_macro

```
-- exceptions raised in an activating task causes the task to complete
-- and an exception is raised in the master
if task_table[my_task_name].status = activating then
  lock(my_task_name)
  task_table[my_task_name].status ← complete
  comp_check_macro(my_task_name)
  unlock(my_task_name)
  lock(master_table[task, my_task_name].master)
  task_table[master_table[task, my_task_name].master].status ←
                                          deactivate
  unlock(master_table[task, my_task_name].master)
  -- now wait until all of our dependents are terminated
  do while task_table[my_task_name].dep_count ≠ 0
  enddo
  lock(my_task_name)
  task_table[my_task_name].status ← terminated
  unlock(my_task_name)
  dep_count_macro(my_task_name)
  sleep(proc_name)
endif

   if S[ep][1] = exception then goto L2; endif     -- exception is in a handler
L1:  -- go ahead and start looking for a handler
   if iseh S[i] and (S[i][1] = exception_reg or S[i][1] = 'others')
     then goto L3; endif    -- found proper handler reference?
   if iseh S[i] and S[i][1] = Ω then goto L2; endif       -- more references?
   i ← i-1; goto L1        -- go check another reference
```

```
L2:  -- propagate to dynamic predecessor
  if istask(S[mp]) then goto L4; endif
   if isaccept(S[mp]) then goto L5; endif
   if ispackage(S[mp]) then goto L6; endif
   k ← S[mp][5]          -- transfer control to pred.
   t ← S[mp][2]          -- save dynamic link
   ep ← S[mp][6]         -- old event pointer
   S[mp] ← S[i]          -- save top of stack
   i ← mp                -- pop part of stack
   mp ← t                -- point to predecessor's predecessor
   goto L1               -- check new level
L3:  -- transfer control to the exception handler
   k ← S[i][2]-1
   S[i ← i+1] ← (exception, ep)    -- note control is in a handler
   ep ← i
   goto L7
L4:  -- tried to propagate exception out of task, complete the task
   lock(my_task_name)
   task_table[my_task_name].status ← 'completed'
   comp_check_macro        -- see if any tasks are waiting for us
   unlock(my_task_name)
   -- wait until all dependents are terminated
   do while task_table[my_task_name].dep_count ≠ 0
   enddo;
   comp_check_macro        -- see if any tasks are waiting for us
   lock(my_task_name)
   task_table[my_task_name].status ← terminated
   unlock(my_task_name)
   dep_count_macro(my_task_name)
   sleep(proc_name)
L5:  -- exception in an accept body not handled within an inner frame
   send_message(task_table[my_task_name].rendezvous_with,
            'raise', exception_reg)
   send_message(my_task_name,'raise',exception_reg)
   k ← S[mp][6]-1          -- complete accept statement
                           -- by executing parallel_continue
L6:  -- propagate from elaboration of package body
   k ← S[mp][4]            -- get return address
   ep ← S[mp][5]           -- restore ep of the pred.
   i ← mp                  -- pop S
   mp ← S[mp][3]           -- nearest mark is now pred.'s
   goto L1                 -- try the exception again
L7:


guard, k-value_list

if k-value_list ≠ () then
    -- guards only exist within a selective wait
    accept_alts ← ()                -- set up for checking alternatives
    delay_alts ← ()
    term_alt ← false
```

```
    else_alt ← 0
    guard_list ← tl(k-value_list)     -- process first guard, drop through
    S[i ← i+1] ← (selective_wait, ep)
    ep ← i
endif


immediate_call_rendezvous, entry_name, k-value
-- Ada conditional entry call statement

locals
  entry_ptr: pointer
  called_task: integer
end locals
entry_ptr ← find_entry_ptr(entry_name)
called_task ← find_task_name(entry_name)
done ← false
lock(called_task)
if task_table[called_task].status ≠ completed and
 task_table[called_task].status ≠ terminated and
 task_table[called_task].status ≠ abnormal then
  if entry_ptr.call_queue = () and
   task_table[called_task].status = suspended and
   task_table[called_task].suspended_at = ('accept', entry_ptr) then
      entry_ptr.call_queue ← entry_ptr.call_queue & my_task_name
      unlock(called_task)
      lock(my_task_name)
      task_table[my_task_name].status ← suspended
      task_table[my_task_name].suspended_at ←
                          ('call', entry_ptr, called_task)
      unlock(my_task_name)
      -- start the rendezvous
      wake(task_table[called_task].proc_name)
      sleep(proc_name)
      lock(my_task_name)
      task_table[my_task_name].status ← callable
      unlock(my_task_name)
  else
      unlock(called_task)
      k ← k-value - 1    -- can't immediately rendezvous, do the else
  endif
else
  -- called task has completed its execution
  unlock(called_task)
  send_message(my_task_name, 'raise', 'tasking_error')
endif


immediate_rendezvous returns boolean
-- part of Ada selective wait statement

locals
```

```
  return_val, done: boolean
  work_list: list of accept_alt_type
  entry_ptr: pointer
  calling_task: integer
  proc: integer
  mode_list: list of string
  ref_list: list of ()
    -- arbitrary element types in list of actual parameters from calling task
end locals
return_val ← false
done ← false
work_list ← accept_alts          -- list of (entry_name, formals, loc, loc)
do while (not done and work_list ≠ ())
   -- find an entry pointer in our task
   entry_ptr ← task_table[my_task_name].entry_ptr
   do while (entry_ptr.entry_name ≠ hd(work_list)[1] or
   entry_ptr.formal_part ≠ hd(work_list)[2];
     entry_ptr ← entry_ptr.next_entry
     if entry_ptr = null then Error; endif;     -- no pointer to entry
   enddo
   if entry_ptr.call_queue ≠ () then
     calling_task ← hd(entry_ptr.call_queue)
     lock(calling_task)
     push(task_table[calling_task].rendezvous_with, my_task_name)
     unlock(calling_task)
     lock(my_task_name)
     push(task_table[my_task_name].rendezvous_with, calling_task)
     entry_ptr.call_queue ← tl(entry_ptr.call_queue)
     unlock(my_task_name)

     -- now copy actual parameters (from caller's stack)
     proc ← task_table[calling_task].proc_name
     S[i ← i+1] ← accept_body_mark(scope_id ← scope_id + 1, (),
       S_proc[i_proc], parm_modes(entry_ptr.formal_part),
       mp, ep, hd(work_list)-1)
     mp ← i
     ep ← 0

     -- for mode markers in, inout, accessin, and accessout copy in actuals
     mode_list ← S[i][4]
     ref_list ← S[i][3]
     for t ← 1 to length(mode_list)
       if mode_list[t] ≠ 'out' then
         if isref(ref_list[t]) then
           S[i][2] ← S[i][2] & ref_list[t]↓          -- deref it
         else
           S[i][2] ← S[i][2] & ref_list[t]           -- no deref
         endif
       else
         S[i][2] ← S[i][2] & ()                      -- just make space
       endif
     endfor
```

```
      D ← D & (proc_name, i, scope_id)        -- augment display in accept body
      k ← hd(work_list)[4]        -- run body of the accept
      done ← true
      return_val ← true
    else
      work_list ← tl(work_list)
  endif
enddo
return(return_val)
```

jump, k-value

```
k ← k-value - 1                  -- transfer control
if S[ep][1] = selective_wait then
  i ← ep-1                       -- pop the stack since guard inst.
  ep ← S[ep][2]
endif
```

package_begin, unique_pack_name, return_address

```
-- create a package display descriptor, during elaboration
pack_table[unique_pack_name].display ← D        -- D is of our parent

-- make a mark for the locals, augment display
S[i ← i+1] ← package_mark(scope_id ← scope_id + 1, (),
                          mp, return_address-1, ep)
mp ← i
ep ← 0
D ← D & (proc_name, i, scope_id)
```

package_end

```
mp ← S[mp][3]                    -- point to enclosing scope
ep ← S[mp][5]                    -- restore ep
t ← D[length(D)][2]
D ← truncate(D)                  -- remove package scope
i ← t                            -- pop S
-- execution (elaboration of containing declaration part) now continues
```

parallel_continue

```
locals
  mode_list: list of string
  parm_list, ref_list: list of ()
    -- arbitrary element types in list of actual parameters from calling task
end locals
abnormal_check                   -- synchronization point
-- copy back out and inout mode parameters
```

```
mode_list ← S[D[length(D)](4)                -- formal modes
ref_list ← S[D[length(D)](3)                 -- actual parms copy
parm_list ← S[D[length(D)](2)                -- parms used in body
for t ← 1 to length(mode_list)
  if (mode_list(t) = inout or mode_list(t) = accessout or mode_list(t) = out) then
    if not isref(ref_list(t)) then Error; endif
    ref_list(t)↓ ← parm_list(t)              -- deref and copy
  endif
endfor
D ← truncate(D)                              -- remove latest scope
i ← mp - 1                                   -- pop stack since accept mark
ep ← S[mp][6]
mp ← S[mp][5]

-- rendezvous is now complete
lock(my_task_name)
calling_task ← pop(task_table[my_task_name].rendezvous_with)
unlock(my_task_name)
lock(calling_task)
pop(task_table[calling_task].rendezvous_with)
unlock(calling_task)
wake(task_table[calling_task].proc_name)  -- continue in parallel


push,value

S[i ← i+1] ← value


raise

   exception_reg ← P[k][2]        -- save exception name
   except_macro


selective_wait

locals
  min_delay: delay_alt_type
  target_time: time
  done: boolean
  master: integer
end locals
abnormal_check                               -- synchronization point
if accept_alts = () and else_alt = 0 then
   -- all alternatives closed, no else part
   send_message(my_task_name, 'raise', 'program_error')
else
   if not immediate_rendezvous then   -- if true, rendezvous is set up
     if else_alt ≠ 0 then
       k ← else_alt                  -- do the else part
     else
```

```
lock(my_task_name)
task_table[my_task_name].status ← suspended
task_table[my_task_name].suspended_at ← 'select'
unlock(my_task_name)
if delay_alts ≠ () then   -- do a delay alternative
  min_delay ← min(delay_alts)
  target_time ← clock_time()
  if car(min_delay) > 0 then
    target_time ← target_time + car(min_delay)
  endif
  done ← false
  do until done                -- do a rendezvous or timeout
    if clock_time() ≥ target_time then
      done ← true
      k ← cdr(min_delay)
    else
      done ← immediate_rendezvous
    endif
  enddo
else
  if term_alt then
    master ← master_table[task, my_task_name].master
    lock(master)
    task_table[master].wait_count ← task_table[master].wait_count + 1
    unlock(master)
    done ← false
    do while not done
      if immediate_rendezvous then
        -- call of an entry has been accepted
        done ← true
        lock(master)
        task_table[master].wait_count ← task_table[master].wait_count - 1
        unlock(master)
      else
        if task_table[master].status = complete and
         task_table[master].dep_count =
         task_table[master].wait_count then
          -- master is completed and all deps are waiting too
          lock(my_task_name)
          task_table[my_task_name].status ← terminated
          unlock(my_task_name)
          sleep(proc_name)
        endif
      endif
    enddo
  else
    -- wait for rendezvous
    do until immediate_rendezvous
    enddo
    lock(my_task_name)
    task_table[my_task_name].status ← callable
   unlock(my_task_name)
```

```
        endif
      endif
      lock(my_task_name)
      task_table[my_task_name].status ← callable
      unlock(my_task_name)
    endif
  endif
endif
```

terminate

```
term_alt ← true                    -- open terminate alternative
if guard_list = () then
   selective_wait                  -- end of alternative list
else
   k ← hd(guard_list)              -- examine next alternative
   guard_list ← tl(guard_list)
endif
```

timed, delay_exp_ptr

```
-- later we'll need to know where delay expression is located
if delay_exp_ptr ≠ Ω then
  S[i ← i+1] ← (timed, delay_exp_ptr, ep)
  ep ← i
endif
```

wake_up_check

```
abnormal_check                     -- synchronization point
if S[ep](1) = timed then
  -- timed entry call succeeded
  i ← ep - 1                       -- pop stack since timed mark
  ep ← S[ep](3)
else
  -- normal rendezvous, no timed entry call
  t ← pop(S)                       -- remove actual parms
endif
```

APPENDIX 7
Examples

These examples contain four elements (1) a portion of Ada code which is assumed to exist within a declaration part; (2) a list of actions occurring during Phase I. In this list, token strings denote when that token is shifted and pushed onto V while production rule names denote a reduction and an effect to be applied; (3) the list of instructions produced by Phase I; (4) a detailed explanation of how the abstract machine code makes the example achieve the desired result is presented.

Example 1:

```
1.  task body master is

2.    task first is
3.      entry A (x: in integer);
4.    end first;
5.    task body first is
6.    begin
7.      accept A (x: in integer) do      -- simple accept
8.        delay 10;
9.      end;
10.   end first;

11.   task second is
12.     entry B (y: in integer);
13.     entry C (z: in integer);
14.   end second;
15.   task body second is
16.   begin
17.     A(1);
18.     select                          -- selective wait with delay alt.
19.       when {boolean expression} =>
20.         accept B(y: in integer) do
21.           {sequence 1}
22.         end B;
```

```
23.         {sequence 2}
24.       or
25.         delay 5;
26.         {sequence 3}
27.     end select;
28.     accept B (y: in integer);        -- simple accept
29.     select                           -- selective wait with else part
30.       accept B(y: in integer) do
31.         {sequence 4}
32.       end B;
33.       or
34.         accept C(z: in integer) do
35.           {sequence 5}
36.         end C;
37.       else
38.         {sequence 6}
39.     end select;
40.     select                           -- selective wait with terminate alt.
41.       accept B(y: in integer);
42.       or
43.         terminate;
44.     end select;
45.   end second;

46.   task third is
47.   end third;

48.   task body third is
49.     task fourth is
50.     end fourth;
51.     task body fourth is
52.     begin
53.       delay 15;
54.     end fourth;
55.   begin
56.     select                           -- conditional entry call
57.       B(2);
58.       else
59.         B(3);
60.     end select;
61.     B(4);                            -- entry call
62.     select                           -- timed entry call
63.       C(5);
64.       or
65.         delay 5;
66.     end select;
67.   end third;
68. begin
69.   null;
70. end master;
```

Action sequence:
master, T13, T15, first, T6, A, T19, (x: in integer), T20, T18, T8 T7, T9, first, T10, T4, T2, T1, E20,
first, T13, T15, T17, T14, A, T27, (x: in integer), T20, T26, 10, T32, T31, T28, T25, E11, first, T10, T12, E22,
second, T6, B, T19, (y: in integer), T20, T18, C, T19, (z: in integer), T20, T18, T8, T7, T7, T9, second, T10, T4, T2, T1, E20,
second, T13, T15, T16, T14, A, 1, T23, T22, T37, T47, {boolean expression}, T45, B, T27, (y: in integer), T20, T26, {sequence 1}, B, T30, T28, T25, {sequence 2}, T52, T48, T38, T46, 5, T32, {sequence 3}, T51, T49, T38, T40, T39, T43, T41, T36, T33, B, T27, (y: in integer), T20, T26, T29, T25, T37, T46, B, T27, (y: in integer), T20, T26,
{sequence 4}, B, T30, T28, T25, T53, T48, T38, T46, C, T27, {z: in integer}, T20, T26, {sequence 5}, C, T30, T28, T25, T53, T48, T38, T40, T39, T44, {sequence 6}, T42, T41, T36, T33, T37, T46, B, T19, (y: in integer), T20, T26, T29, T25, T53, T48, T38, T46, T50, T38,
T40, T39, T43, T41, T36, T33, E11, second, T10, T12, E22,
third, T6, T8, T9, third, T10, T4, T2, T1, E20,
third, T13, T15, fourth, T6, T8, T9, fourth, T4, T2, T1, E20,
fourth, T13, T15, T17, T14, 15, T32, E11, fourth, T12, E22, E3, E2, E1, T16, T14,
T37, B, 2, T23, T55, T53, T57, B, 3, T23, T22, T56, T41, T54, T34,
B, 4, T23, T22,
T27, C, 5, T23, T22, T53, T59, 5, T32, T53, T51, T41, T58, T35, E11, third, T10, T12, E22, E3, E2, E2, E2, E2, E2, E1, T16, T14, E11, master, T10, T12

Final contents of program array, **P**:

1. jump, 124
2. except, Ω, 0 -- decl code for master
3. except, others, Ω
4. except, constraint_error, Ω
5. except, numeric_error, Ω
6. except, program_error, Ω
7. except, storage_error, Ω
8. except, tasking_error, Ω
9. jump, 23
10. except, Ω, 0
11. except, others, Ω -- decl code for first
12. except, constraint_error, Ω
13. except, numeric_error, Ω
14. except, program_error, Ω
15. except, storage_error, Ω
16. except, tasking_error, Ω
17. activate, ()
18. accept_rendezvous, A, (x: in integer), 21 -- body for first
19. push, 10
20. delay
21. parallel_continue
22. complete

```
23. jump, 81
24. except, Ω, 0
25. except, others, Ω                    -- decl code for second
26. except, constraint_error, Ω
27. except, numeric_error, Ω
28. except, program_error, Ω
29. except, storage_error, Ω
30. except, tasking_error, Ω
31. activate, ()
32. push, 1                              -- body for second
33. call_rendezvous, A                   -- entry call
34. wake_up_check
35. timed, Ω                  -- selective wait with delay alt
36. guard, (37, 44)
37. {code for boolean expression}
38. eval_guard
39. accept_rendezvous, B, (y: in integer), 41
40. {code for sequence 1}
41. parallel_continue
42. {code for sequence 2}
43. jump, 50
44. push, true
45. eval_guard
46. push, 5
47. delay
48. {code for sequence 3}
49. jump, 50
50. accept_rendezvous, B, (y: in integer), 51    -- accept statement
51. parallel_continue
52. timed, Ω                  -- selective wait with else part
53. guard, (54, 60, 66)
54. push, true
55. eval_guard
56. accept_rendezvous, B, (y: in integer), 58
57. {code for sequence 4}
58. parallel_continue
59. jump, 69
60. push, true
61. eval_guard
62. accept_rendezvous, C, (z: in integer), 64
63. {code for sequence 5}
64. parallel_continue
65. jump, 69
66. else
67. {code for sequence 6}
68. jump, 69
69. timed, Ω                  -- selective wait with term alt
70. guard, (71, 76)
71. push, true
72. eval_guard
73. accept_rendezvous, B, (y: in integer), 74
74. parallel_continue
```

```
75. jump, 80
76. push, true
77. eval_guard
78. terminate
79. jump, 80
80. complete
81. jump, 122
82. except, Ω, 0                        -- decl code for third
83. except, others, Ω
84. except, constraint_error, Ω
85. except, numeric_error, Ω
86. except, program_error, Ω
87. except, storage_error, Ω
88. except, tasking_error, Ω
89. jump, 101
90. except, Ω, 0                        -- decl code for fourth
91. except, others, Ω
92. except, constraint_error, Ω
93. except, numeric_error, Ω
94. except, program_error, Ω
95. except, storage_error, Ω
96. except, tasking_error, Ω
97. activate, ()
98. push, 15                            -- body for fourth
99. delay
100. complete
101. activate, (5)
102. timed, Ω                           -- body for third
103. guard, ()
104. push, 2
105. immediate_call_rendezvous, B, 107
106. jump, 110
107. push, 3
108. call_rendezvous, B
109. wake_up_check
110. push, 4
111. call_rendezvous, B
112. wake_up_check
113. timed, 119
114. guard, ()
115. push, 5
116. call_rendezvous, C
117. wake_up_check
118. jump, 121
119. push, 5
120. delay
121. complete
122. activate, (2, 3, 4)
123. complete                           -- body for master
124. {code for the next declaration in containing scope}
```

Explanation:

This example demonstrates the activation and elaboration of task units and the interaction of tasks via each kind of entry call statement and each kind of accept statement. Task body, *master*, is presumed to exist within some program unit's declarative part. The elaboration of that declarative part cannot elaborate *master* and executes the **jump** instruction at 1 which causes elaboration to proceed at the next declaration. At some later time activation of *master* causes its eleboration, using a different processor, to begin at instruction 2. The elaboration of *master* continues until instruction 123 is reached, when execution of the sequence of statements of the task body, statement 69, begins. Statements 68-70 generate only the **complete** instruction at 123. This instruction completes *master* and enters a wait loop until all of *master*'s dependent tasks are terminated, then *master* terminates. The elaboration of all program unit declarative parts begins by placing an exception handler reference list end marker on the execution stack **S**, done for *master* by the **except** instruction at 2. This marker is used during exception processing to signify that if a handler reference has not yet been found it cannot be found within this scope and propagation results. Following this instruction, each predefined exception name, including **others**, is associated with a handler via the except instruction, 3-8. Since a backpatch method is used for code generation, we cannot exclude these instructions even though no handlers exist, so the instruction pointer field of each **except** instruction remains Ω. That Ω field precludes creation of a reference for the specified name on **S**. Elaboration is now complete since the task specifications and bodies for *first*, *second*, and *third* cannot yet be elaborated. Control passes from instruction 9, to 23, to 81, to 122. The **activate** instruction now begins

activation of *first*, *second*, and *third* whose unique names are 2, 3, and 4 respectively, by acquiring three unused processors and assigning each task to one. Activation of each task on these processors begins with elaboration of its declarative part, at instructions 10, 24, and 82 for tasks *first*, *second*, and *third* respectively. *Master* does not continue until each of these has completed its activation, denoted by completion of their **activate** instruction. Note that every task body contains the **activate** instruction even if it has no dependent tasks.

The bodies of *first* and *second* have empty declaration parts and no exception handlers so, as in *master*, all that occurs is that instructions 10 and 24 place exception handler reference list end markers on *first*'s and *second*'s respective execution stacks. Also, *third* does not elaborate the body of *fourth* at this time and only an exception handler reference list end marker is placed on **S**. The instructions 11-16, 26-30, and 83-88 have no effect and the **activate** instructions at 17 and 31 merely inform *master* that two of its dependents are activated. Since *first* and *second* have no dependents they now begin execution at instructions 18 and 32 respectively. *Third* however, begins activation of *fourth* at instruction 101 and waits there until *fourth* is activated. Thus, *master* awaits the activation of *third* which awaits the activation of *fourth*. *Fourth*'s elaboration places an exception handler reference list end marker on its execution stack, *third* is informed that *fourth* has completed its activation by the **activate** instruction at 97, and then *fourth* begins its execution at 98. Now *third* has completed activation and begins execution at instruction 102 and *master* also begins execution, at instruction 123. Since *master*'s body contains only the null statement (statement 69) execution of the body is immediately finished, *master* completes, and waits at

instruction 123 until each of its dependents terminates, then *master* terminates.

At this point in time, *first*, *second*, *third*, and *fourth* are executing in parallel. The body of *first* contains only one accept statement at 7. This corresponds to instruction 18 which suspends *first* until a statement calling entry A is executed by another task. Since *fourth* contains only a delay statement, statement 53, the corresponding abstract machine code instructions 98-99 effect this delay by waiting at instruction 99 until 15 seconds have passed. Then instruction 100 is executed, *fourth* is completed and terminated and *third*'s dependent count becomes zero indicating the termination of all its dependents. Since each task has its own processor, *fourth*'s processor has nothing more to do and after being put to sleep after *fourth*'s termination it may sleep for the life of the current program or may be reclaimed for use by another task. This definition does not describe how processors are acquired or disposed of. *Third* begins execution with a conditional entry call statement, statement 56. Since *second* is not waiting to accept a call for B the else part of the statement is executed, a simple entry call statement. This sequence corresponds to instructions 102-108. The **timed** instruction at 102 does not apply here, nor does the **guard** instruction at 103 since their second fields are Ω and () respectively. The evaluation of the actual parameter for statement 57 is done by instruction 104 and an attempt at an immediate rendezvous is made by the **immediate_call_rendezvous** instruction at 105. Since *second* is not waiting to accept a call to B, control passes to instruction 107 where an actual parameter is evaluated, the entry call to B is queued, and *third* waits for a rendezvous at B to take place. When that rendezvous is complete, *second* will awaken *third*. The reason why *second* has not reached

statement 18 is that it calls entry A, statement 17, by executing instructions 32-33. *Second* is suspended until the rendezvous is complete. The rendezvous includes the binding of the actual parameter, 1, to the formal parameter x as part of the **accept_rendezvous** instruction at 18 and the execution of the accept statement body, statement 8, instructions 19-20, which first delays for 10 seconds. After this delay, the **parallel_continue** instruction at 21 awakens *second* which continues in parallel with *first*. The **wake_up_check** instruction at 34 checks for particular rendezvous anomalies that may have occured which require special attention. *First* immediately completes and terminates since it has no dependents. *Second* is now the only executing task and reaches statement 18, a selective wait with a delay alternative. The delay part, statement 25, requires that if a call to B is not accepted within 5 seconds of reaching the selective wait, {sequence 3} at statement 26 is executed. Note the presence of a guard at statement 19. If the boolean expression contained in the guard evaluates to false, B can never be accepted, while if it evaluates to true, B is accepted immediately since a call from *third* is pending. This sequence is represented by instructions 35-48. Again the **timed** instruction at 35 does not apply. The **guard** instruction at 36 contains a list of pointers to the two guards to process. It also places a mark on the execution stack indicating entry into a selective wait statement. Open alternatives of the selective wait statement (those with an open guard) are determined by allowing control to pass from guard to guard until the guard list is exhausted. The boolean expression at instruction 37 is evaluated. If true, **eval_guard** at 38 allows control to pass to **accept_rendezvous** at 39 which creates an accept alternative descriptor in the global data structure accept_alts (see Appendix 1). Then control is passed, by

accept_rendezvous, to the trivial guard at 44. If the first guard is closed, **eval_guard** passes control to instruction 44 and no accept alternative is available. Existence of a delay alternative is recorded by the **delay** instruction at 46 in the global data structure delay_alts. Since no more guards exist, **delay** makes a determination as to which alternative is chosen. Let us assume that the accept alternative is available, therefore it will be chosen since *third* has already issued a call to it. As above for the rendezvous at A, *second* and *third* now rendezvous at B. Statement 21, instruction 40, executes during the rendezvous and after completion of the rendezvous, *second* awakens *third* and continues execution at statement 23, instruction 42, and *third* continues execution at instruction 109.

Second and *third* now rendezvous at B, statements 28 and 61, instructions 50 and 111, respectively but *second* contains no accept body to execute so they both continue since *second* immediately awakens *third*. *Second* then reaches another selective wait statement, denoted by the **guard** instruction at 53 with a nonempty list. Since each guard is open, i.e. instructions 54 and 60 evaluate to true and 66 requires no evaluation of a boolean expression (an implicit trivial guard of true), all alternatives are open. Thus three alternatives are entered into the global data base, an accept for B at instruction 56, an accept for C at instruction 62, and an else alternative at instruction 66. If a call to B or C cannot immediately be accepted (as determined by the else instruction) then the code for {sequence 6}, (statement 38) at instruction 67 is executed. We notice that *third* has reached a timed entry call statement, statement 62, denoted in the machine code by the **timed** instruction at 113 with a second field of 119. This causes evaluation of the delay amount, instruction 119, and then the **delay** instruction at 120 waits to determine if

the call to C, instruction 116, is accepted within that amount of time. Now, *second* is checking for an immediate call to C while *third* issues a call to C and cancels the call if it is not accepted within 5 seconds. It is clear that if *second* reaches this point first it will continue with its else part. However, if *third* arrives at this point first it will wait for at most 5 seconds within which time *second* may arrive and begin a rendezvous. If *second* and *third* rendezvous at C, the code for {sequence 5} (statement 35) at instruction 63 is executed and then they continue, *second* at instruction 64 and *third* at instruction 117. If they do not, *second* executes the sequence associated with the else alternative, statement 38, instruction 67, while *third* continues, after 5 seconds, with the empty sequence following the delay statement, which places it at instruction 121. **Third** completes and terminates since *fourth* has already terminated. *Second* executes one final selective wait statement, this time with a terminate alternative, statement 43 (instruction 78). The terminate alternative is selected immediately since *second*'s master, *master*, is complete and all of *master*'s dependents (i.e. *first* and *third*) are either terminated, or waiting on an open terminate alternative of a selective wait statement. Thus all processors have halted and execution is complete.

Example 2:

```
1.  task body master is

2.    package first is
3.      task second;
4.      task body second is
5.      begin
6.        delay 10;
7.      end second;
8.    end first;
9.    package body first is
10.     task third is
11.     end third;
12.     task body third is
13.       an_exception: exception;
14.     begin
15.       raise an_exception;
16.       exception
17.         when an_exception => null;
18.     end third;

19.   begin
20.     delay 20;
21.   end first;
22.   another_exception: exception;
23. begin
24.   abort second;
25.   exception
26.     when tasking_error => delay 10;
27. end master;
```

Action sequence:
master, T13, T15, first, P3, second, T5, T2, T1,
second, T13, T15, T17, T14, 10, T32, E11, second, T10, T12, first, P4, P2, P1, E21,
first, P8, third, T6, T8, T9, third, T4, T2, T1, E20,
third, T13, T15, an_exception, E7, E5, E4, E3, E2, T16, T14, an_exception, E9, E12, an_exception, E15, E14, E10, third, T12, E22, E3, E2, E2, P9, P7, 20, T32, E11, P11, first, P6, E23, another_exception, E7, E5, E3, E2, E2, E2, T16, T14, second, T62, T60, E12, tasking_error, E17, E15, 10, T32, E14, E10, master, T10, T12

Final contents of program array, **P**:

```
1. jump, 53
2. except, Ω, 0                          -- decl code for master
3. except, others, Ω
4. except, constraint_error, Ω
5. except, numeric_error, Ω
6. except, program_error, Ω
7. except, storage_error, Ω
8. except, tasking_error, 50
9. jump, 21
10. except, Ω, 0                         -- decl code for second
11. except, others, Ω
12. except, constraint_error, Ω
13. except, numeric_error, Ω
14. except, program_error, Ω
15. except, storage_error, Ω
16. except, tasking_error, Ω
17. activate, ()
18. push, 10
19. delay
20. complete
21. package_begin, 1, 46                 -- elaborate first
22. except, Ω, 0
23. except, others, Ω
24. except, constraint_error, Ω
25. except, numeric_error, Ω
26. except, program_error, Ω
27. except, storage_error, Ω
28. except, tasking_error, Ω
29. jump, 42
30. except, Ω, 0                         -- decl code for third
31. except, others, Ω
32. except, constraint_error, Ω
33. except, numeric_error, Ω
34. except, program_error, Ω
35. except, storage_error, Ω
36. except, tasking_error, Ω
37. except, an_exception, 41
38. activate, ()
39. raise, an_exception
40. jump, 41
41. complete
42. activate, (2, 3)
43. push, 20                             -- body for first
44. delay
45. package_end
46. except, another_exception, Ω         -- another decl for master
47. activate, ()
48. abort, 2                             -- body for master
49. jump, 52
50. push, 10
```

51. delay
52. complete
53. {code for the next declaration in containing scope}

Explanation:

This example demonstrates elaboration of tasks contained in packages and the effect of raising an exception in a scope where a handler is provided. Task body, *master*, is presumed to exist within some program unit's declarative part. The elaboration of that declarative part cannot elaborate *master* and executes the **jump** instruction at 1 which causes elaboration to proceed at the next declaration. At some later time, activation of *master* causes its elaboration, using a different processor, to begin at instruction 2. The elaboration of *master* continues until instruction 48 is reached, when execution of the sequence of statements of the task body, statement 24, begins. The elaboration of all program unit declarative parts begins by placing an exception handler reference list end marker on the execution stack **S**, done in *master* by instruction 2. This marker is used during exception processing to signify that if a handler reference has not yet been found it cannot be found within this scope and propagation results. Following this instruction, each predefined exception name, including **others**, is associated with a handler via the **except** instructions, 3-8. Since a backpatch method is used for code generation, we cannot exclude these instructions even if a handler does not exist and the instruction pointer fields is Ω, as in instructions 3-7. The reference instruction (instruction 8) for **tasking_error** has a non-Ω pointer field and creates a reference on **S** to that handler at instruction 50. The first declaration in *master* is a package specification. The fact that the specification contains a task declaration is noted during translation so that *second* can be activated during elaboration of the body of *first*, specifically at instruction 42,

referenced by the unique name 2. Execution of the **package_begin** instruction (21) creates a local data storage list on S for this package and denotes the instruction, in the third field of the instruction, used to abandon the package elaboration during exception propagation. Elaboration of the package body for *second* begins with the creation of an exception handler reference list end marker, instruction 22. Since the package body for *second* only declares a task, the next action of elaboration is the activation of *second* and *third*, instruction 42. Further elaboration suspends until these tasks have completed their activation. Activation of *second* and *third* consists of acquiring a processor for each task and starting the processors at location 10 for elaboration of *second* and 30 for elaboration of *third*. *Second* contains no exception handlers and no dependent tasks, so activation consists only of creating an exception handler reference list end marker, instruction 10, and reporting to *master* that activation is complete via the **activate** instruction at 11. *Third* does contain a handler for an_exception and in addition to the exception handler reference list end marker, instruction 30, a handler reference is placed on S for an_exception by the **except** instruction at 37. Then *third* also reports its activation to *master* via instruction 38 and now executes in parallel with *second* and the continued elaboration of the body of *first*. This elaboration continues with the statement following the declarative part of *first*, statement 20, instructions 43-44. This statement serves only to delay the exit from package *first*. While this delay is executing (*second* also delays at statement 6, instructions 18-19), and *third* raises the exception an_exception, at instruction 39. The **raise** instruction finds the handler reference placed on S by the **except** instruction at 37. Control is then passed to the handler which contains no instructions, in correspondence with null

statement 17, and *third* is immediately completed and terminated. Nothing else happens until the delay statement of *first* is complete, then elaboration of the declaration part of *master* continues by executing the **except** instruction at 42 which creates no handler reference on S since a handler does not exist for another_exception. Since *master* has no offspring tasks an **activate** instruction with an empty list is executed and then the body of *master* is entered. *Master* has only one statement in its body, statement 24, instruction 48. This **abort** instruction aborts *second* using its unique internal name, 2, only if *second* has not yet terminated, that is, if it is still suspended at instruction 19. The **jump** instruction at 49 is executed next by *master* and passes control around the handler for **tasking_error** to the **complete** instruction at 52 which completes and terminates *master*.

APPENDIX 8

Interface Control Details

1. Task objects may be defined using variables of an explicit task type. These must be entered into the task table and activation list of the master at elaboration time.

2. Use of values from entry or task attributes is part of the normal run time system. The information is obtained from the task table.

3. Subprogram and block marks must be extended to contain a sixth piece of data, the value of **ep** (the event mark pointer). Each time a mark is created and the value of **ep** is recorded **ep** must be reset to zero.

4. Each scope that can be a master must put its name on the master stack when the scope is entered and remove its name when the scope is exited. This is required since the parent of a task may not be its master (in particular, the package body parent). Names take the forms: (task, *task_name*), (block, *block_name*), (subprogram, *subprogram_name*), (library_package, *package_name*).

5. The master_table structure defined in appendix 1 must be filled for each scope that is a task master.

6. Note that this definition assumes only tasks are masters. If other scopes

are to be implemented as masters, each use of **master_table** must discriminate between them.

7. Reduction of declarative_part, basic_declarative_item, and later_declarative_item must check the has_body boolean for each package specification declared within and if it is false, assume an implicit body for the package. The package specifications declared will be listed on top of **pack_name_list_stack**.

8. Reduction of *declarative_part*, *basic_declarative_item*, and *later_declarative_item* must begin by pushing an empty list on **activation_list_stack** and both name list stacks and finish by removing the top element from each. Also, **task_name_string** and **pack_name_string** must be cleared and restored upon entry and exit respectively.

9. Each scope that is a parent of tasks must include the **activate** instruction immediately following declaration elaboration.

10. Each scope that may contain an exception handler must remove it's declared exception names from **exception_list**.

References

[Bak] Baker, T. P. and G. A. Riccardi, **A Runtime Supervisor to Support Ada Tasking, Part 2, Rendezvous and Delays**, Florida State University Ada Project Report 83-7.

[Belz] Belz, F. C., E. K. Blum, and D. Heimbigner, **A Multiprocessing Implementation-Oriented Formal Definition of Ada in SEMANOL**, SIGPLAN Notices, Vol. 15, No. 11, November, 1980, pp. 202-212.

[Bjø1] Bjørner, D. and O. N. Oest ed., *Towards a Formal Description of Ada*, **Lecture Notes in Computer Science**, No. 98, Springer-Verlag, 1980.

[Bjø2] Bjørner, Dines and Hans Henrick Lovengreen, **On a Formal Model of the Tasking Concept in Ada**, SIGPLAN Notices, Vol. 15, No. 11, November 1980, pp. 213-222.

[Booch] Booch, G.,*Software Engineering with Ada*, The Benjamin/Cummings Publishing Company, Inc., Menlo Park, California, 1983.

[Bri] Brindle, A. F., D. F. Martin, et. al., **A Model for the Run-Time Processing of Ada Tasking**, Aerospace Technical Report No. ATR-84(8233)-2.

[Chir] Chirica, L. M., D. F. Martin, et. al., **Two Parallel Euler Run Time Models: the Dangling Reference, Impostor Environment, and Label Problems**, UCLA Computer Science Department internal paper.

[DoD] *Ada Programming Language*, Department of Defense, Washington, D.C., ANSI/MIL-STD-1815A, 1983.

[Inria] *Formal Definition of the Ada Programming Language*, Inria Corp., France, November 1980.

[Lea] Leathrum, J. F., **Design of an Ada Run-Time System**, *Proceedings of the IEEE Computer Society 1984 Conference on Ada Applications and Environments*, 1984, pp. 4-13.

[Li] Li, Wei, **An Operational Semantics of Tasking and Exception Handling in Ada**, University of Edinburgh Department of Computer Science, December 1981.

[Ric] Riccardi, G. A. and T. P. Baker, **A Runtime Supervisor to Support Ada Task Activation, Execution and Termination (Preliminary Report)**, *Proceedings of the IEEE Computer Society 1984 Conference on Ada Applications and Environments*, 1984, pp. 14-22.

[Sta] Standish, T. A., and Taylor, R. N., **Arcturus: a Prototype Advanced Ada Programming Environment**, *SIGSOFT Software Engineering Notes: Proceedings, ACM SIGSOFT/SIGPLAN Symposium on Practical Software Development Environments*, Vol. 9, No. 3, 1984, pp. 57-64.

[Tay] Taylor, R. N. and Standish, T. A., **Steps to an Advanced Ada Programming Environment**, *Proceedings of the 7th International Conference on Software Engineering*, 1984, p. 116-125.

[Wea] Weatherly, Richard M., **A Message-Based Kernel to Support Ada Tasking**, *Proceedings of the IEEE Computer Society 1984 Conference on Ada Applications and Environments*, 1984, pp. 136-144.

[Wir1] Wirth, Niklaus and Helmut Weber, **Euler: A Generalization of Algol and its Formal Definition: Part 1**, *Communications of the ACM*, Vol.9, No. 1, January, 1966, pp. 13-25.

[Wir2] Wirth, Niklaus and Helmut Weber, **Euler: A Generalization of Algol and its Formal Definition: Part 2**, *Communications of the ACM*, Vol.9, No. 2, February, 1966, pp. 89-99.

Bibliography

Ada Programming Language, Department of Defense, Washington, D.C., ANSI/MIL-STD-1815A, 1983.

Booch, G.,*Software Engineering with Ada*, The Benjamin/Cummings Publishing Company, Inc., Menlo Park, California, 1983.

Chirica, L. M., D. F. Martin, et. al., **Two Parallel Euler Run Time Models: the Dangling Reference, Impostor Environment, and Label Problems**, UCLA Computer Science Department internal paper.

Wirth, Niklaus and Helmut Weber, **Euler: A Generalization of Algol and its Formal Definition: Part 1**, *Communications of the ACM*, Vol.9, No. 1, January, 1966, pp. 13-25.

Wirth, Niklaus and Helmut Weber, **Euler: A Generalization of Algol and its Formal Definition: Part 2**, *Communications of the ACM*, Vol.9, No. 2, February, 1966, pp. 89-99.

END

FILMED

4-85

DTIC