Task Dependence and Termination in Ada*

Laura K. Dillon
Department of Computer Science
University of California, Santa Barbara 93106

Keywords: Ada tasking, task termination, master/dependent relations, distributed termination, run-time storage management, tasking execution model

*Research partially supported by NSF grants CCR-8702905 and CCR-9014382 (with cooperation from DARPA).

Abstract

We use a contour model of Ada tasking to analyze the semantics of task dependence and termination. Our analysis clarifies the complex definitions and rules that determine when procedures and tasks terminate during execution of an Ada program and examines the implications of and possible motivation for the termination rules.

We show that the termination rules prevent the premature deallocation of data associated with the runtime-instances of scope units, but that the policy is unnecessarily conservative in this regard. For task instances created by invoking a storage allocator, we show that the conservative termination policy permits more efficient management of heap storage than a less conservative policy.

We also examine the impact of the termination rules in Ada on the synchronization of concurrent unit instances. The rules are shown to support both master-slave and client-server applications. In particular, we show how the rules governing distributed termination guarantee that the terminating instances can no longer affect the outcome of a computation.

The analysis is intended give programmers a better understanding of Ada tasking and help language designers assess the strengths and weaknesses of the termination model.

1 Introduction

The Ada programming language has seen increased use in applications with stringent reliability requirements as programmers have mastered the features it provides for abstraction and reuse. However, the tasking features of Ada still see relatively little use, especially in the kinds of critical applications for which tasking was designed. Tasking is difficult to understand and to learn to use effectively. There is little question that this difficulty contributes to the reluctance of programmers to use tasking in safety-critical applications.

Concurrency adds an extra dimension to the complexity of writing and understanding programs, and the extreme complexity of the semantics of Ada tasking compounds this problem. The Ada Language Reference Manual (ALRM) [25] and other natural language descriptions of tasking are subject to incompleteness and misinterpretation. Such descriptions provide little help in visualizing the effects of potential interactions among tasks in a program.

The semantics of task dependence and termination in Ada are particularly difficult to understand. Seemingly arbitrary details in the definitions of these concepts makes them hard to motivate and remember, providing a likely source of programming errors.

In [13] we describe a visual execution model for Ada tasking that is designed to help programmers attain a deeper understanding of the tasking semantics. Derived from Johnston's contour model for block-structured programming languages [20], it uses contour diagrams to illustrate the execution of a tasking program in a concrete pictorial fashion. In this paper, we use the model to analyze the semantics of task dependence and termination in Ada. The paper clarifies the termination rules defined in the ALRM and examines their implications and possible justification.

Section 2 introduces the contour model by way of an example. We use this example in Section 3 to illustrate the definitions and rules that govern the termination of Ada procedures and tasks. The next two sections examine issues relating to run-time management of storage: Section 4 considers the impact of Ada's termination rules on the management of stack storage and Section 5 considers their impact on the management of heap storage. We then turn to issues relating to synchronization of terminating procedures and tasks, showing how the rules provide for two common synchronization paradigms: Section 6 considers master-slave synchronization and Section 7 considers client-server synchronization. We discuss related work in Section 8 and give a summary and concluding remarks in Section 9. The appendix contains proofs of the theorems that are developed to characterize the implications of the termination rules.

2 Overview of the Contour Model

This section describes those features of the contour model of Ada tasking needed for the analysis of Ada's termination rules. A more comprehensive description of the model appears in [13]. For brevity, we consider only the interaction between procedures and tasks; problems arising from their interaction illustrate the main issues of concern for termination. This section assumes familiarity with the fundamentals of tasking in Ada.

The contour model represents an Ada program as a data structure, called an *algorithm*, and defines an abstract machine that interprets the algorithm, simulating execution of the program. An execution is described by a sequence of data structures, called *records of execution*, that represent instantaneous program states. Interpretation of an algorithm is relatively simple and intuitive, with concurrency modeled by interleaving parallel threads of control. Nondeterminacy within an Ada program is manifest by nondeterminacy in the interpretation process. From a given algorithm, the interpreter can produce different execution sequences corresponding to different interleavings of parallel threads of control and different choices for resolving nondeterminism within threads. We describe records of execution in some detail below. For brevity, however, we do not describe the representation of an Ada program as an algorithm, but talk in a loose sense of directly executing the program, as is typical for in-class use of the contour model to describe other programming languages [9],[10].

We use the following program fragment to illustrate the ideas presented in this section. Line numbers ($s1$, $s2$, etc.) are provided for ease of reference and are not part of the program. We show only those portions of the program that are affect termination. The comments at lines $s15$, $s22$, and $s32$ represent contour model instructions and are not part of the program.

```
s1   procedure MAIN is
s2     task type TTA;
s3     FIRST: TTA;
s4     procedure PROC_A is
s5       task type TTB;
s6       type OUTER is access TTB;
s7       procedure PROC_B is
s8         type INNER is access TTB;
s9         SECOND: INNER;
s10        THIRD: OUTER;
s11        FOURTH: TTB;
s12      begin -- PROC_B
s13        SECOND := new(TTB);
s14        THIRD := new(TTB);
s15        -- complete;
s16      end PROC_B
```

```
    task body TTB is
      ...
    end TTB;
  begin -- PROC_A
    PROC_B
    -- complete;
  end PROC_A

  task body TTA is
    ...
  begin -- TTA
    PROC_A;
    ...
  end TTA;
begin -- MAIN
  null;
  -- complete;
end MAIN;
```

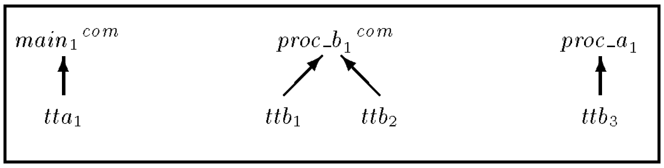
Figure 1 shows a record of execution for a state that could arise during execution of this program.[2] The *contour diagram* (large rectangle labeled $main_1$) contains a set of nested contours, shown as labeled rectangles, and one or more *processors*, shown as Π symbols. Contours represent run-time instances of the scope units in the program and processors represent threads of control, as described below. The *dependency diagram* (top rectangle) shows the direct masters of the task instances in a program state. Section 3 discusses the direct dependence relation and explains how the termination rules make use of this relation.

The contour diagram in a record of execution that represents a program state contains a contour for each instance of a scope unit that has been created and has not yet terminated. We label contours, for ease of reference, with unique names obtained from the names of the associated procedures and task types. For example, the contour $main_1$ in Figure 1 represents an instance of the procedure *MAIN* and the contours ttb_1, ttb_2 and ttb_3 represent instances of the task type *TTB*.

The upper left corner of a contour contains a *data array*, which shows the bindings of local identifiers to values. Figure 1 shows the values of return points, tasks, task access objects, and access types, but does not show the values of task types or procedures since they have no bearing on termination. Similarly, the local data for tasks of types *TTA* and *TTB* are left unspecified.

As customary in contour models, we regard every procedure instance as implicitly declaring a *return point*, called *%RET*. The protocol for transferring control to a procedure instance initializes the return point with a cell containing two components. The first component, called the *dynamic link*, designates the caller (the instance that invoked the procedure) and the second designates the

[2] In general, a record of execution contains other special-purpose diagrams that show additional information required to manage task instances, such as the status of entry queues and the conditions on which task instances are blocked. We do not show these diagrams as this information is not relevant to the discussion below.



direct dependences

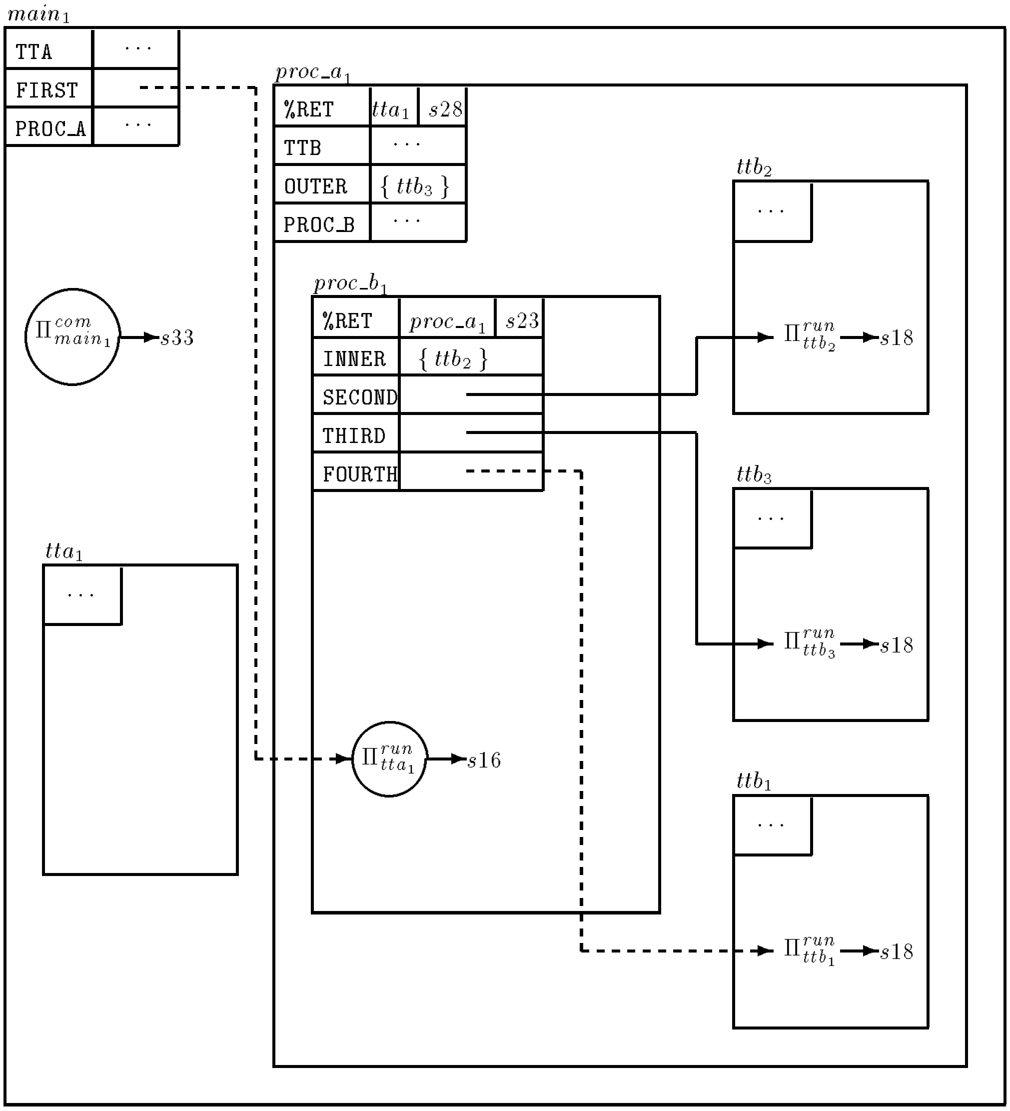


Figure 1: A Record of Execution

instruction to be executed on return from the procedure instance. Thus, for example, the return point in the procedure instance $proc\_b_1$ shows that, upon its termination, $proc\_b_1$ will return control to $proc\_a_1$ at instruction $s23$.

Elaborating a task declaration creates a thread of control for a new task instance and binds the identifier named in the declaration to the new thread. We represent a task value by a dashed arrow to the thread of control, as shown by the bindings for *FIRST* and *FOURTH*.

Access values are pointers, or references, to other objects. A generic *new* operation in Ada provides storage allocators for all access types. The *new* operation for a task access type allocates a new task instance of the designated task type and returns a reference (access value) to the new task instance. We represent a task access value by a solid arrow to the thread for the designated task instance, as shown by the bindings for *SECOND* and *THIRD*.

The binding for an access type indicates the *collection* of task instances designated by objects of the access type, as illustrated by the bindings for *OUTER* and *INNER*. Collections are discussed in Section 5.

A thread of control (processor) is created for each of the task instances spawned during execution of an Ada program and for the root instance of the top-level procedure, which we regard as defining a root task type. The subscript on a thread of control identifies the associated task instance and the superscript assigns it a state. We distinguish three states for modeling the semantics of termination in Ada.[3]

- *Run*ning: Control is in the body of the task instance or in some procedure instance that the thread has invoked.

- *Com*pleted: Control is after the body of the task instance, but the task instance has not terminated.

- *Term*inated: The task instance has terminated.

The program state shown in Figure 1 contains five threads of control. The thread for $main_1$ is completed and the threads for tta_1, ttb_1, ttb_2 and ttb_3 are running.

The horizontal arrow to the right of a thread, called the *instruction pointer*, defines the thread's locus of control. By convention, (the processor symbol for) a thread is nested directly in (the contour of) the instance that has control of the thread. Thus, a thread's instruction pointer designates the instruction, in the body of the instance that directly encloses the thread, that the thread is about to

[3] Additional states are required to model creation and abortion of tasks.

execute. For instance, in Figure 1, control of the thread tta_1 is at statement $s16$ in the body of the procedure instance $proc\_b_1$. A thread of control is *blocked* if its next instruction cannot be executed until a concurrent thread performs some action, such as termination of a dependent. We place a circle around a thread to signify that it is blocked. A thread of control is *ready* if it is neither blocked nor terminated. On each machine cycle the contour model interpreter nondeterministically selects a ready thread of control, fetches the instruction designated by the thread's instruction pointer, increments the instruction pointer, and then executes the instruction.

The conventions described above permit the record of execution in Figure 1 to be interpreted as follows. The initial thread $main_1$ has elaborated the declarative part of the root task instance, executed its body, and completed. Completion of the task instance is signified by the superscript on $main_1$ in the dependency diagram. Completion of the task instance also completes the thread $main_1$, as shown by the superscript on the thread. The task instance tta_1 was spawned during elaboration of the declaration for *FIRST*. Its thread tta_1 has elaborated the task instance's declarative part and invoked *PROC_A*, thereby creating the procedure instance $proc\_a_1$; the thread has then proceeded to enter $proc\_a_1$, elaborate its declarative part and invoke *PROC_B*, thereby creating the procedure instance $proc\_b_1$; finally, the thread tta_1 has entered $proc\_b_1$, elaborated its declarative part, executed its body and completed $proc\_b_1$. Completion of a procedure instance does not affect the state of the thread that executes it, so that the thread tta_1 is running, but blocked. Execution of $proc\_b_1$ spawned three task instances of type *TTB*. The task instance ttb_1 was spawned by the elaboration of the declaration for *FOURTH* in instruction $s11$, and ttb_2 and ttb_3 were spawned by the evaluation of the allocator expressions in instructions $s13$ and $s14$, respectively. These later three threads have all elaborated the declarative parts of the associated task instances and are in their bodies. The threads $main_1$ and tta_1 are blocked, so that the interpreter will execute one of the threads ttb_1, ttb_2, or ttb_3 on the next machine cycle.

For the discussion of Ada's termination rules, we note that the dynamic links in the unit instances created by a thread of control connect them into a chain, called the *dynamic chain*. The dynamic chain for a thread of control starts at the instance that has control of the thread and enumerates the unit instances that the thread is executing, in reverse invocation order, with the associated task instance at the base of the chain.[4] In the program state described by Figure 1, for example, the dynamic chain of the thread tta_1 is $\langle proc\_b_1, proc\_a_1, tta_1\rangle$. The dynamic chains of the remaining

[4] Spawning a task creates the thread for a new task instance, but the task instance is created and executed by the new thread, not the spawning thread. Thus, a task instance belongs to the dynamic chain of its own thread of control.

threads contain only the base task instances.

The justification for important aspects of Ada's termination policy makes use of relationships between the referencing environments of the unit instances in a program state. Contour models use nesting to show these relationships. A new instance of a scope unit is nested directly in the instance that determines the new instance's global referencing environment. In the case of a new task instance, the global referencing environment is determined by the instance that contains the binding for the task type, and in the case of a new procedure instance, by the instance that contains the binding for the procedure. In Figure 1, for example, the task instance tta_1 and the procedure instance $proc\_a_1$ are nested directly in the task instance $main_1$, which contains the bindings for *TTA* and for *PROC_A*. Similarly, $proc\_b_1$, ttb_1, ttb_2 and ttb_3 are nested directly in $proc\_a_1$, which contains the bindings for *PROC_B* and *TTB*.

Adherence to this convention ensures that the referencing environments of concurrent instances can be determined visually, according to the manner in which they are nested. The referencing environment of a unit instance consists of all bindings in the *static chain* that begins with the instance itself and enumerates enclosing instances in order of decreasing nesting depth. A thread of control inherits the referencing environment (static chain) of the instance that directly encloses the thread and that therefore has control of the thread. A reference to a simple identifier designates the closest binding for the identifier (the first binding for the identifier found by searching instances on the static chain). More generally, an identifier can be qualified with the name of the declaring scope unit to designate a binding in a specific instance on the static chain. All bindings in the instances that enclose a thread are therefore said to be *visible* to the thread; an identifier is visible if a binding for the identifier is visible. A value is *accessible* if it is bound to a visible identifier or if it is designated by an accessible access object. In Figure 1, for example, the thread tta_1 can access the thread ttb_2 by dereferencing *SECOND*. However, the local data belonging to the task instance tta_1 is not accessible to any thread of control; in particular, the thread tta_1 cannot access the local data belonging to its own task instance since the thread is in $proc\_b_1$ and data in the task instance tta_1 is not accessible in $proc\_b_1$.

3 Termination in Ada

Terminating a procedure instance deallocates its contour and returns control of the thread to the caller, which is next on the thread's dynamic chain. Terminating a task instance deallocates its

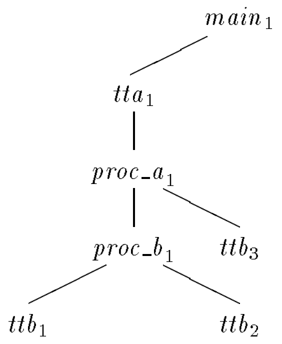


Figure 2: The Hierarchy of Masters and Dependents

contour and makes the thread permanently ineligible for execution.

The rules that determine when instances terminate make use of the *master* and *dependent* relations, whose definitions distinguish task instances by how they are spawned. The program above illustrates the two methods for spawning task instances: A task instance is said to be *elaborated* if it is spawned by elaborating a task declaration and *allocated* if it is spawned by invoking a task allocator. In either case, we refer to the instance that spawns a new task instance as the *parent* of the new instance.[5] In the program state in Figure 1, for example, $proc\_b_1$ is the parent of ttb_1, ttb_2 and ttb_3.

A *direct master* is defined for every task instance in a program state, except the root instance of the top level procedure. If a task instance is elaborated, the direct master is the parent. If a task instance is allocated, the direct master is the instance that contains the binding for the associated access type, which could be the parent or could be an instance of a global scope unit. In Figure 1, for example, $proc\_b_1$ is both the parent and direct master of ttb_2. However, $proc\_a_1$, which contains the binding for *OUTER*, is the direct master of ttb_3.

The direct master relation determines the direct dependences in a program state. All other dependences are indirect. The ALRM defines indirect dependences between a task instance and all instances that follow one of its masters on the dynamic chain of the thread that executes the master and between a task instance and all masters of its masters.

These definitions define a hierarchy of masters and dependents for each program state. The hierarchy is rooted in the root task instance of the top level procedure, and organized so that

[5] This terminology is not standard. Brindle and Taylor [12] call the instance that spawns a new task instance the parent frame, and call the task instance associated with the thread that created the parent frame the parent task. Baker and Riccardi [5] use parent to refer to this parent task. In [7], the task instance executing the direct master is called the parent.

dependents descend from their masters. Figure 2 shows the hierarchy of masters and dependents for the program state depicted in Figure 1. Vertical lines represent dynamic links and slanted lines represent direct dependences. The *chain of masters* for a task instance is obtained by walking up the hierarchy of masters and dependents, beginning at the direct master and ending at the root task instance. This gives the following chains of masters for the task instances in the program state shown in Figure 1.

$main_1$: $\langle\ \rangle$

tta_1: $\langle main_1 \rangle$

ttb_1, ttb_2: $\langle proc\_b_1, proc\_a_1, tta_1, main_1 \rangle$

ttb_3: $\langle proc\_a_1, tta_1, main_1 \rangle$

The ALRM provides two rules to govern the termination of instances created during execution of an Ada program. The first rule pertains to instances that have completed. An instance completes when it finishes executing the statements in its body. Thus, a completed instance no longer requires a thread of control.

Rule 1 *A unit instance terminates when it is completed and all of its dependents are terminated.*

In Figure 1, for example, the superscript attached to $proc\_b_1$ in the dependency diagram signifies that $proc\_b_1$ has completed. However, according to the above rule, it cannot terminate until ttb_1 and ttb_2 are terminated. Thus, the thread in $proc\_b_1$ blocks, preventing its contour from being deallocated.

The second termination rule determines when executing a *select statement* can result in termination. A select statement allows an instance to selectively accept calls for *rendezvous* on *entries* associated with another task instance. (Such rendezvous are the primary means by which concurrent instances synchronize and communicate during execution of an Ada program.) So-called *selective wait alternatives* in a select statement specify the entries for which calls are to be accepted. *Guards* are used to classify alternatives as either *open* or *closed*, giving the task instance some control over the selection of alternatives. A select statement may also contain a *terminate alternative*. If present, a terminate alternative permits the *distributed termination* of all task instances that depend on a master, provided that the master is completed and that control of each nonterminated dependent of the master has reached a select statement with an open terminate alternative; we say that a master is *asleep* if it meets these conditions and that the (non-terminated) dependents of the master are ready to select terminate alternatives. Thus, the second termination rule can be summarized as follows.

Rule 2 *A task instance terminates when its execution reaches a select statement with an open terminate alternative and some master of the task instance is asleep.*

We provide an example in Section 7 that illustrates the application of this rule.

Terminating an instance of a procedure ends its lifetime. However, the lifetime of a task instance may extend past its termination. This is necessary, because concurrent instances can interrogate attributes associated with a task instance and invoke its entries after it has terminated. The ALRM does not explicitly stipulate when the lifetime of a task instance ends and all information pertaining to the task instance can be deallocated.

The next section examines how these rules affect traditional stack-based methods for managing the dynamic chains of threads of control during execution of a program. Subsequent sections consider issues relating to the management of allocator storage and to synchronization and distributed termination of concurrent instances.

4 Reclaiming Stack Storage

Block structured programming languages allow relatively efficient memory management schemes in which the dynamic chains of threads of control are maintained as run-time stacks. This is made possible by the last-in-first-out manner in which a thread enters and leaves instances. However, when multiple threads share referencing environments, an instance may finish executing before other instances are finished accessing its data. In the program state in Figure 1, for example, the task instance $main_1$ has completed and no longer requires a thread of control. However, the remaining threads can access $main_1$'s local data, which will be deallocated when $main_1$ terminates. The termination rules must guarantee that an instance does not terminate as long as any concurrent instances can access its data.

The following theorem shows that the only threads that can access a task instance's local data are the thread for the instance and threads for dependents of the task instance. If u and v denote unit instances, we write $u \subset v$ to signify that u is nested in v.

Theorem 1 *Consider a program state S containing unit instances u and v. Let t denote the task instance associated with the thread of control that is executing u. If $u \subset v$ then either* *(i) v comes after u on the dynamic chain of the thread for t or*

(ii) t depends on v

The proof Theorem 1 makes use of the rules for adjoining new instances to a contour diagram and inducts on the number of instances created and the number of instances terminated during an execution leading to S. Details can be found in the appendix.

Theorem 1 guarantees that, if an instance meets the conditions of Rule 1, then the thread executing the instance is the only thread that can access the instance's local data. Thus, the completed instance can be safely deallocated. If an instance meets the conditions of Rule 2, then Theorem 1 guarantees that those threads that can access the instance's local data are all executing dependents of the sleeping master and, therefore, are all ready to select terminate alternatives. This means that all dependents of the sleeping master can be terminated without compromising the referencing environments of other unit instances.

Thus, the termination policy in Ada ensures that the local data of a unit instance is not deallocated prematurely simply because the instance no longer requires a thread of control. However, were this the only concern, then Ada's termination policy would be overly conservative. For instance, in the program state in Figure 1, termination of the procedure instance $proc\_b_1$ must await termination of the task instances ttb_1 and ttb_2. However, the visibility and scope rules guarantee that the threads for these task instances cannot access $proc\_b_1$'s local data in this or any subsequent state produced by execution of the program, so that there is no need to delay termination of $proc\_b_1$.

This situation arises because the direct master of a task instance does not determine its global referencing environment. Defining the direct master of a task instance to be the unit instance that declares the associated task type would result in a less conservative termination policy. In the program state in Figure 1, for example, this modification would make $proc\_a_1$ the direct master of all task instances created from the binding for *TTB*. The procedure instance $proc\_b_1$ could then terminate and return control to $proc\_a_1$ while ttb_1 and ttb_2 are running. Modifying the definition of the direct master relation in this manner does not affect the validity of Theorem 1. The less conservative termination policy, therefore, would also guarantee that a unit instance's local data is not deallocated if concurrent instances can access its data. For example, termination of $proc\_a_1$ would await termination of the task instances ttb_1, ttb_2, and ttb_3, which can access $proc\_a_1$'s local data.

5 Reclaiming Allocator Storage

An important consideration that arises in the case of allocated task instances is how their creation and termination affects the management of allocator storage. We show in this section that, in the case of an allocated task instance, the conservative definition in the ALRM for the direct master facilitates the reclamation of allocator storage.

In general, the evaluation of an allocator expression creates a new value of some *designated type* and returns an access value designating the new value. The access value can be assigned to global variables and passed to and from subprograms as parameters. Thus, unlike a value that is created by elaborating a declaration, an allocated value can remain accessible long after its parent terminates. Allocated values, therefore, cannot be placed on the run-time stacks of the task instances, but must be maintained in a separate storage area using a more sophisticated management scheme, such as a heap.

The problems associated with reclaiming values that are dynamically allocated during execution of a unit instance are well known. Ideally, an implementation should reclaim allocated values as soon as they become inaccessible. However, the run-time overhead associated with such garbage collection schemes can be very high. At the other extreme, not reclaiming allocated values that are inaccessible wastes storage. The severity of this problem depends on the particular program and the extent to which it makes use of allocated data; for many programs the additional storage demands incurred by a nonreclamation strategy are unacceptable [3].

Ada permits a relatively efficient scheme for reclaiming allocated values through the use of *collections*. In Ada, each access type has an associated collection, which consists of the values designated by values of the access type. In our contour model of Ada tasking, we show the collection associated with an access type as its value. Thus, in Figure 1, we represent the value for $OUTER$ by the collection $\{ ttb_3 \}$ and the value for $INNER$ by the collection $\{ ttb_2 \}$.

The ALRM states that a language implementation can reclaim the values in the collection associated with an access type upon terminating the unit instance that declares the access type. This policy makes perfectly good sense from the standpoint of accessibility of the values in the collection, as shown by the following theorem, which is an easy consequence of the typing rules in Ada.

Theorem 2 *Consider a program state containing a unit instance v that declares an access type. Let c denote the collection that is bound to the access type. Values in c can be accessed in an instance u only if $u \subseteq v$.*

When the access type declaration hypothesized in Theorem 2 is for a task access type, the values in c are allocated task instances and v is their direct master. Thus, if a task instance is allocated, then it can only be accessed from within its direct master.

However, when reclaiming allocated task instances, accessibility is not the only issue that must be considered. A task instance does not have to be accessible to affect the outcome of a computation. A task instance should only be reclaimed if it is either terminated or ready to select a terminate alternative. Ada's conservative termination policy ensures that, upon terminating a unit instance v, all task instances in the collections bound to access types declared by v are either terminated or ready to select open terminate alternatives. The less conservative termination policy discussed in Section 4 does not provide this guarantee. For example, in the program state shown in Figure 1, the less conservative termination policy would allow $proc\_b_1$ to terminate while ttb_2, which belongs to the collection bound to *INNER*, is still running. Ada's termination policy does not permit $proc\_b_1$ to terminate until ttb_2 is either terminated or ready to select a terminate alternative. An implementation can reclaim ttb_2 at this point.

However, if collections were associated with the designated type of an access value, rather than the access type, and reclaimed upon terminating the instance that declares the designated type, then the problem of reclaiming nonterminated task instances would not arise using the less conservative termination policy. An allocated task instance would be reclaimed upon terminating the unit instance that declares the designated type. Thus, defining the direct master of a task instance to be the instance that elaborates the task type declaration and associating collections with designated types would result in a less conservative termination policy and still permit efficient memory management.

The main tradeoff, in terms of memory management and potential parallelism, is that, in general, the less conservative termination policy increases concurrency at the expense of possibly delaying the reclamation of allocated objects. For allocated task instances, this delay might be warranted in order to increase parallelism. However, most allocated values are passive and, unlike allocated task instances, cannot affect the outcome of a computation once they become inaccessible; no purpose would be served by delaying the reclamation of more general allocated objects.

6 Synchronizing termination of 'slave' tasks

The impact of Ada's termination policy on the run-time management of storage primary affects the efficiency of language implementations. Because the termination policy determines the manner in which instances created by different threads of control synchronize their termination, the termination policy also affects the ease of writing programs. Concurrent programming is used extensively in two important types of applications: master-slave applications and client-server applications. This section considers the extent to which Ada's termination rules support the synchronization required in master-slave applications.

A master-slave application is one in which a parent spawns one or more 'slave' task instances to perform specific sub-tasks on behalf of the parent. It is common, in this context, to refer to the parent as the 'master'; however, we will continue to use the parent terminology since this use of the term 'master' is not consistent with the definition in the ALRM if the slave is allocated, rather than elaborated. In master-slave applications, the parent should not terminate until all the slaves that it spawned have terminated, as, logically, the parent is not finished until the slaves are. Master-slave applications are easily programmed in Ada using elaborated task instances to implement slave tasks, as illustrated by the following program fragment.

```
procedure PROD_CONS is
  type ITEM is ...
  I1, I2:  ITEM;
  task type PROD1;
  task type PROD2;
  procedure PRODUCE is
    P1:  PROD1;
    P2:  PROD2;
  begin -- PRODUCE
    null;
  end PRODUCE;
  task body PROD1 is
    ...  -- "produces" I1
  end PROD1;
  task body PROD2 is
    ...  -- "produces" I2
  end PROD1;
  procedure CHECK is
    ...  -- "checks" I1 and I2
    ...  -- may spawn new PROD1 and PROD2 tasks
  end CHECK;
  procedure CONSUME is
    ...  -- "consumes" I1 and I2
  end CONSUME;
```

```
begin -- PROD_CONS
  PRODUCE;
  CHECK;
  CONSUME;
end PROD_CONS;
```

In this example, the root task instance $prod\_cons_1$ spawns an instance $produce_1$ of *PRODUCE*, which spawns two slave task instances $prod1_1$ and $prod2_1$. These slaves concurrently produce values for *I1* and *I2*. An instance $check_1$ of *CHECK* examines the values of *I1* and *I2*, regenerating them if necessary. Finally, the values are consumed by an instance $consume_1$ of *CONSUME*. Clearly, $produce_1$ must not terminate until its slaves have terminated. Ada's termination Rule 1 provides the required synchronization, in this case, because $produce_1$ is both the direct master and parent of the slave task instances.

It is more difficult to achieve the required synchronization using the less conservative termination policy discussed in Sections 4 and 5, which redefines the direct master of an elaborated task instance to be the unit instance that declares the task type. In the above example, this modification would make $prod\_cons_1$ the direct master of $prod1_1$ and $prod2_1$, allowing $produce_1$ to terminate and $check_1$ to execute concurrently with $prod1_1$ and $prod2_1$. The program would therefore permit erroneous race conditions in the use of the shared variables *I1* and *I2*. In this example, the declarations for *PROD1* and *PROD2* cannot be placed in the declarative part of *PRODUCE* in order to provide the required synchronization, as then the declarations would not be visible in *CHECK*. Thus, either the procedures *PRODUCE* and *CHECK* would both have to declare appropriate producer task types or rendezvous would have to be used to explicitly synchronize the unit instances in their use of the shared variables.

A similar problem arises using Ada's termination rules if slave task instances are allocated, rather than elaborated. The termination rules provide the synchronization required in master-slave applications only if the slave's parent is also its direct master. Therefore, the access type associated with a slave that is allocated must be declared in the parent. For example, the following version of *PRODUCE* has essentially the same effect as the original version, except that it serializes the activation of the slave task instances.

```
procedure PRODUCE is
  type PACC1 is access PROD1;
  type PACC2 is access PROD2;
  P1:  PACC1;
  P2:  PACC2;
begin -- PRODUCE
  P1 := new PROD1;
  P2 := new PROD2;
end PRODUCE;
```

However, because the placement of declarations also determines their scope, visibility requirements can conflict with synchronization requirements. In the above *PROD_CONS* program, for instance, *CHECK* can also spawn allocated instances of types *PROD1* and *PROD2*. In order that the declarations for the access types be visible in both *PRODUCE* and *CONSUME*, a programmer is likely to put the declarations in *PROD_CONS*. However, this would not give the required synchronization and would result in erroneous race conditions.

In the case of an allocated task instance, it is necessary to distinguish the notions of parent and master. In Ada, an elaborated task is essentially a local constant that is declared in the parent. An elaborated task instance is therefore naturally viewed as constituting part of the parent. However, task access objects can be modified in the manner of all access objects. This means, for example, that a procedure can spawn an allocated task instance and pass the access value designating the new task instance to a caller or assign the access value to a global data object. In such situations, the creation, but not the execution, of the task instance is to be considered part of the execution of the procedure. Ada's definition for the direct master of an allocated task instance makes such synchronization possible, as would the less conservative definition discussed in Sections 4 and 5.

7 Distributed Termination

This section discusses the implementation of client-server applications in Ada, which require a different pattern of synchronization than master-slave applications. We show how Ada's termination policy supports the distributed termination of a group of task instances implementing a 'server' and its 'clients'.

In Ada, a server task is typically implemented using an infinite loop statement containing a selective wait statement with a terminate alternative. This permits the server to repeatedly accept calls from clients, as long as some client might require its services, and permits the server to terminate when its services are no longer needed. Ada's termination Rule 2 provides for such distributed termination of concurrent task instances. To understand how, it is necessary to understand the implications of the condition in Rule 2 that some master be asleep.

The following theorem gives necessary conditions for an instance u to invoke an entry of a task instance t.

Theorem 3 *Consider a program state containing a task instance t and a unit instance u. Let m denote the direct master of t and let t' denote the task instance associated with the thread of control executing u. If t is accessible in u, then either*

(i) m comes after u on the dynamic chain of the thread for t' or

(ii) t' depends on m

Thus, a task instance can be accessed only by the threads that execute its master and its master's dependents.[6]

We discuss the proof of Theorem 3 below. However, first we use it to prove that, if a program state contains a sleeping master, then no thread is at an entry call statement for an entry belonging to any of the sleeping master's dependents.

Theorem 4 *Consider a program state containing a task instance t with a master s that is asleep. If the thread for a task instance t' can access t, then either*

(i) the thread is blocked waiting to terminate s or

(ii) the thread is ready to select an open terminate alternative

Proof. Let m denote the direct master of t. By Theorem 3, either m is on the dynamic chain of the thread for t' or t' depends on m. If t' depends on m, then it also depends on s. As s is asleep and t' is not terminated, we have (ii) in this case. If m is on the dynamic chain of the thread for t' and m has completed, then $m = s$ and we have (i). Finally, if m is on the dynamic chain of the thread for t' and m has not completed, then no master from m through t' on the chain of masters for t is completed. Thus, s comes after t' on the chain of masters for t. In this final case, therefore, we again conclude that t' depends on s, giving (ii). ∎

[6] Theorem 3 depends on the restriction to procedures and tasks. In particular, we must preclude functions. A function can return a task instance that it elaborates to a caller. In this case the caller can call the task's entries, although both the task instance and its master are necessarily terminated. However, this is regarded as a pathological case and of no practical use; it has been recommended that this case be considered erroneous because of the negative impact that it has on the implementation of storage recovery for terminated tasks [17].

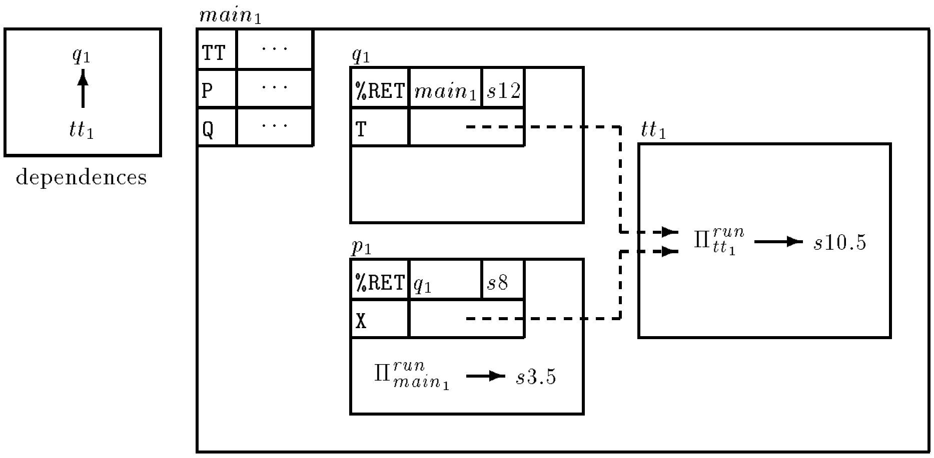


Figure 3: A Task Instance that is Accessible From Outside its Master

Theorem 4 implies that no thread can invoke the entries of a server task instance if some master of the server is asleep. Thus, the server can be terminated.

Theorem 3 follows easily from Theorems 1 and 2 in case the hypothesized task instance t is allocated. However, if the task instance is elaborated, then Theorems 1 and 2 do not apply, as an elaborated task instance can be accessed from outside of its direct master. Consider, for example, the following program fragment.

```
s1   procedure MAIN is
s2     task type TT;
s3     procedure P(X: in TT) is ...  end;
s4     procedure Q is
s5       T: TT;
s6     begin
s7       P(T);
s8       ...
s9     end Q;
s10    task body TT is ...  end TT;
s11  begin
s12    Q;
s13  end MAIN;
```

The record of execution in Figure 3 shows a program state in which the task instance tt_1 is accessible in the procedure instance p_1, although p_1 is not nested in the procedure instance q_1, which is tt_1's direct master. This example is representative, in the sense that an elaborated task instance is accessible from outside its master only if it is passed as a parameter to a nonenclosing unit instance. Theorem 3 is proved, in the case that t is elaborated, by induction on the number of unit instances that have received t as a parameter. A proof appears in the appendix.

The following program fragment shows why Rule 2 does not permit an instance to select an open terminate alternative as soon as some master has completed, but requires that the instance wait until all dependents of the completed master are either terminated or ready to select terminate alternatives.

```
procedure MAIN is
  task type USR;
  task type SVR is
    entry E;
  end SVR
  U1, U2:  USR;
  S: SVR;
  task body SVR is
  begin
    loop
      select
        accept E; ...
      or
        terminate;
      end select;
    end loop;
  end SVR;
  task body USR is ...  end USR;
begin
  null;
end MAIN;
```

The record of execution in Figure 4 shows a program state in which the master $main_1$ of the task instance svr_1 has completed and execution of svr_1 has reached an open terminate alternative. The instruction pointer for svr_1 references the two open alternatives that svr_1 can selectively execute. If either usr_1 or usr_2 invokes svr_1's entry E, then svr_1 must select the accept alternative. Thus, it would be premature for svr_1 to select the terminate alternative at this point. However, if usr_1 and usr_2 terminate or reach open terminate alternatives, all the dependent task instances can terminate.

Finally, we note that Rule 2 allows the sleeping master to be an indirect master in case execution of the direct master reaches an open terminate alternative. This is illustrated by the following program fragment.

```
s1   procedure MAIN is
s2     task type USR;
s3     task type SVR_A is
s4       entry E_A;
s5     end SVR_A;
s6     U1, U2:  USR;
s7     S_A: SVR_A;
s8     task body USR is ...  end USR;
s9     task body SVR_A is
s10      task type SVR_B is
s11        entry E_B;
s12      end SVR_B;
s13      S_B: SVR_B;
s14      task body SVR_B is
s15      begin
s16        loop
s17          select
s18            accept E_B; ...
s19          or
s20            terminate;
s21          end select;
s22        end loop;
s23      end SVR_B;
s24    begin    -- SVR_A
s25      loop
s26        select
s27          accept E_A;
s28        or
s29          terminate;
s30        end select;
s31        if ...  then S_B.E_B end if;
s32      end loop;
s33    end SVR_A;
s34  begin
s35    null;
s36  end MAIN;
```

Figure 5 shows a program state that leads to distributed termination. The state shown in the figure has a single successor, which is produced when the thread svr_b_1 executes the select statement at $s17$. The terminate alternative $s20$ is open and $main_1$, an indirect master of svr_b_1, is asleep. Therefore, the dependents svr_a_1 and svr_b_1 are sequenced on to their terminate alternatives on the next machine cycle. The instance svr_b_1 then terminates, awakening svr_a_1, which then also terminates. When svr_a_1 terminates, it deallocates the thread for its direct dependent svr_b_1 and awakens $main_1$. Finally, $main_1$ terminates, deallocating the remaining threads and ending the computation.

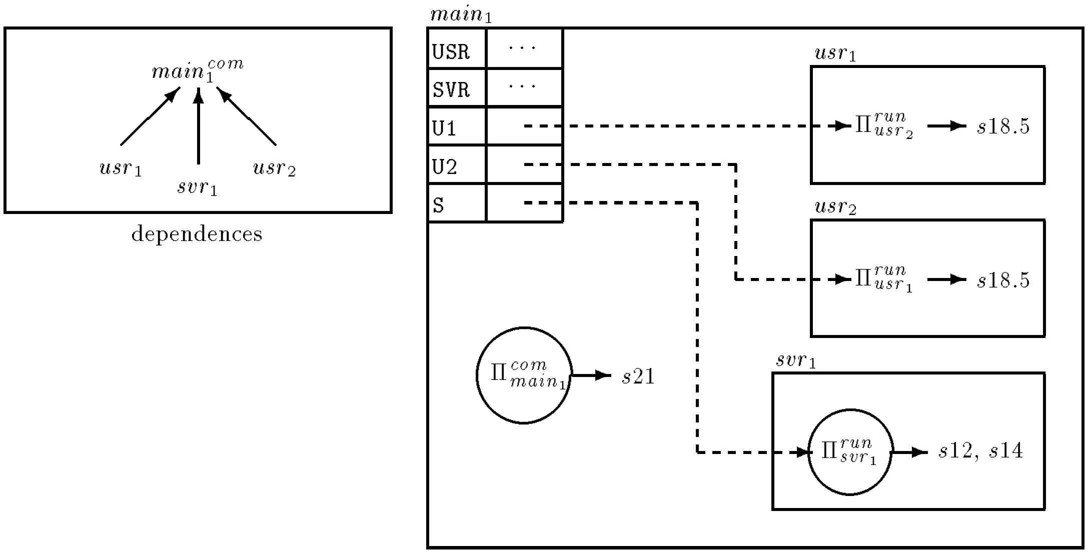


Figure 4: The Distributed Termination Criteria Must Take into Account All Dependents of the Completed Master

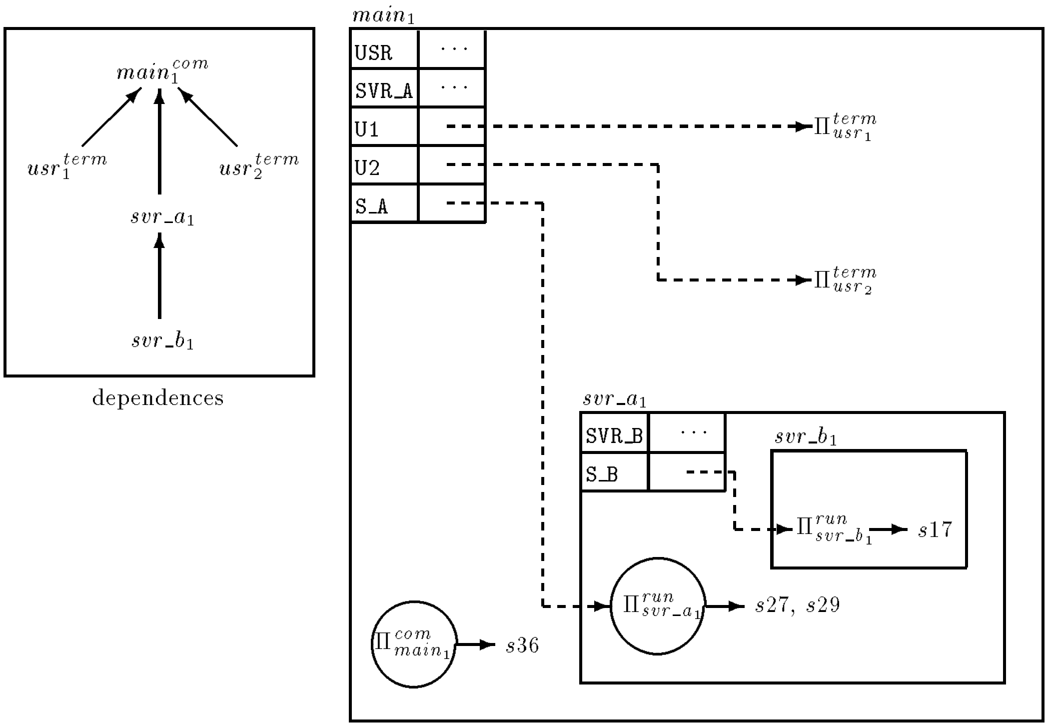


Figure 5: State Leading to Distributed Termination

We note that if the direct master m of a task instance t is terminated, then Theorem 3 implies that the task instances associated with the threads of control that can access t all depend on m. As all dependents of m are terminated, this means that t is no longer accessible; no thread can interrogate its attributes or invoke its entries. The lifetime of a task instance ends, therefore, with the termination of its direct master. We represent this in our contour model by deallocating the thread for a task instance upon terminating its direct master.

8 Related Work

The ALRM and its clarifications [1] provide the definitive references for all Ada language features. A variety of Ada language text books [6], [15] describe the language in a more tutorial format, and several specialize in Ada tasking [16],[23].

Ichbiah, et. at. [18] discuss the rationale behind the design of the Green language, which was the precursor to Ada. However, the differences between tasking in Green and tasking in Ada are substantial. A rationale for the design of Ada was developed after it became both an ANSI and military standard [19]. The goal of the rationale is to impart the overall philosophy of the language design and to explain the motivation behind key features of the language. The rationale mentions the need to protect the integrity of referencing environments as motivation for Ada's termination policy.

Outstanding problem areas in Ada relating to tasking and its usability in real-time applications are discussed in [2]. These and problems identified in other studies resulted in the Ada 9X requirements [4]. Requirements address the need to reclaim heap storage, in general, and space used for control blocks of tasks created by an allocator, in particular; they also address problems of access to a task outside its direct master, among many other problems. A rationale for the design of Ada 9X, which uses essentially the same termination model as Ada 83, appears in [7]. This report does not discuss the rationale behind the definitions of the dependence relations or of the termination rules. Issues relating to task dependence and termination are discussed in [5], which describes a run-time supervisor for Ada tasking.

Johnston's contour model provides an operational model for block-structured programming languages that has been used extensively in teaching applications. The model is the basis for a textbook on programming language structures [22], and has been used to illustrate the semantics of parameter passing in Ada [11] and multitasking in PL/1 [9]. An automated version of the model supplies a visual semantics for ALGOL programs [24]. A contour model of Ada tasking very similar to ours is described in [8]. The model in [8] is used both to clarify the semantics of the concurrency contructs in Ada and also to evaluate their complexity.

The execution model outlined in [12] is designed to support debugging of Ada tasking programs and to provide for display of the dynamic execution status of concurrent programs. The debugger for Ada tasking described in [14] provides a graphical abstraction for rendezvous that helps the user visualize the effects of interprocess synchronization and communicatation.

A compiler/interpreter for Ada tasking, called Small-Ada, has been found to be effective in teaching concurrency constructs [21]. Small-Ada scrolls the source code of tasks as they execute, permitting the user to observe the execution of a concurrent program. User-selectable tasking options give the user control over the speed of execution and the scheduling of ready tasks. An automated debugger helps the user identify and correct certain classes of anomalies.

9 Conclusion

We have used a contour model of Ada tasking to clarify the rules that govern the termination of procedures and tasks in Ada and to explore the implications of the termination rules. Our analysis has shown that Ada's termination policy ensures that referencing environments are not deallocated prematurely, but that the policy is unnecessarily conservative in this regard. For task instances created by invoking a storage allocator, we have shown that the conservative termination policy permits more efficient reclamation of allocator storage than a less conservative policy. We have also considered how Ada's termination policy supports the synchronization required in master-slave and client-server applications. In particular, we show that the rules that govern distributed termination in Ada ensure that a server selects a terminate alternative only if its services can no longer be required.

One conclusion of our analysis is that conflicting concerns relating, on the one hand, to run-time management of storage and, on the other, to synchronization of terminating instances have resulted in overly complex rules, whose implications are not easily demonstrated nor understood. One apparent source of confusion is the dual role that the placement of task declarations and task access type declarations plays in the semantics. In Ada, as in all block-structured languages, the placement of a declaration determines its scope. However, in the case of a task declaration or a task access type declaration, the placement of the declaration also determines how concurrent instances synchronize their termination. This violates a basic principle of programming language design, namely, that the same language mechanism should not be used to express largely independent concerns. A proposal that decouples these concerns would help provide a cleaner termination model.

Programmers need to understand the intricacies of Ada tasking if they are to make effective use of it when writing programs. Likewise, language designers should understand the justification for the tasking semantics and its implications, if they are to take advantage of the strengths of tasking while avoiding its pitfalls. The clarification and analysis of the semantics of task dependence and termination presented above will give programmers and language designers a better understanding of tasking in Ada.

References

[1] ACM SIG Ada. *The Approved Ada Language Commentaries*, Spring 1989. Grebyn Corporation, Vienna, VA.

[2] ACM SIG Ada. *Proc. Third Inter. Workshop on Real-Time Ada Issues*, Farmington, PA, June 1990. Also published as *Ada Letters*, X(4), Spring 1990.

[3] Ada 9X Project Office, Software Engineering Institute, Carnegie Mellon University, Pittsburgh. *Ada 9X Requirements*, Dec. 1990.

[4] Ada 9X Project Office, Software Engineering Institute, Carnegie Mellon University, Pittsburgh. *Ada 9X Requirements Rationale*, May 1991.

[5] T. Baker and G. A. Riccardi. Ada tasking: From semantics to efficient implementation. *IEEE Software*, pages 34–46, Mar. 1985.

[6] J. P. Barnes. *Programming in Ada*. Addison-Wesley, 1982.

[7] J. P. Barnes. Introducing Ada 9X. *Ada Letters*, XIII(6):61–132, November/December 1993.

[8] B. Belkhouche and L. M. Lawrence. A semantic model of Ada tasking. *Journal of Pascal, Ada & Modula-2*, pages 29–41, March/April 1990.

[9] D. M. Berry. Notes on PL/1. Unpublished lecture notes, University of California, Los Angeles, 1975.

[10] D. M. Berry. Notes on the contour model. Unpublished lecture notes, University of California, Los Angeles, 1975.

[11] D. M. Berry. Treatment of Ada procedures: Aliasing, anomalies, and erroneousness. Technical report, Unisys Corporation, Santa Monica, CA, Feb. 1988.

[12] A. E. Brindle, R. N. Taylor, and D. F. Martin. A debugger for Ada tasking. *IEEE Trans. Software Engineering*, 15(3):293–304, Mar. 1989.

[13] L. K. Dillon. A visual execution model for Ada tasking. *ACM Trans. Software Engineering and Methodology*, 2(4):311–345, Oct. 1993.

[14] M. B. Feldman and M. L. Moran. Validating a demonstration tool for graphics-assisted debugging of Ada concurrent programs. *IEEE Trans. Software Engineering*, 15(3):305–313, Mar. 1989.

[15] N. Gehani. *Ada, an Advanced Introduction*. Prentice-Hall, Englewood Cliffs, NJ, 1983.

[16] N. Gehani. *Ada: Concurrent Programming*. Prentice-Hall, Englewood Cliffs, NJ, 1984.

[17] J. Goodenough. Real-time tasking semantics working group: Recovering storage for terminated tasks. In ACM SIG Ada [2], page 46.

[18] J. Ichbiah. Rationale for the design of the Ada programming language. *SIGPLAN Notices*, 14(6), 1979.

[19] J. Ichbiah, J. Barnes, R. Firth, and M. Woodger. *Rationale for the Design of the Ada Programming Language*. Cambridge University Press, 1991.

[20] J. B. Johnston. The contour model of block structured processes. In *Proc. ACM Symp. Data Structures in Programming Languages*, pages 35–82, Feb. 1971. SIGPLAN Notices.

[21] A. V. Lopes, M. B. Feldman, and R. S. Heller. A controlled experiment with software for teaching Ada tasking. In *Proc. Tri-Ada 93*, Seattle, WA, Sept. 1993. ACM SIGAda.

[22] E. I. Organick, A. I. Forsythe, and R. P. Plummer. *Programming Language Structures*. Academic Press, New York, second edition, 1978.

[23] K. Shumate. *Understanding Concurrency in Ada*. McGraw-Hill, 1988.

[24] J. W. Thomas and E. I. Organick. Visible semantics for programming languages. In *Proc. 1974 ACM National Conf.*, pages 416–421, 1974.

[25] United States Department of Defense, Washington, D.C. *Reference Manual for the Ada Programming Language*, Jan. 1983. ANSI/MIL-STD-1815A.

A Appendix

Proof of Theorem 1. Let u and v denote unit instances in a program state S, and t denote the task instance associated with the thread of control executing u. Let $(*)$ denote the property:
If $u \subset v$ then either

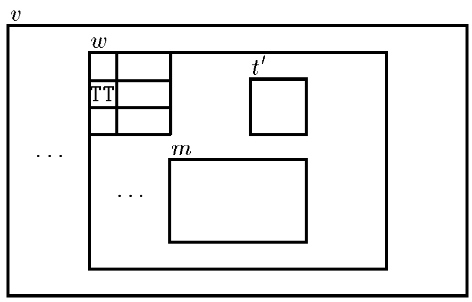
(i) v comes after u on the dynamic chain of the thread for t or

(ii) t depends on v

Clearly, $(*)$ holds vacuously if S consists of a single root task instance. Moreover, terminating an instance that has control of a thread preserves $(*)$ trivially. We show that creating a new unit instance also preserves $(*)$. The claim then follows by induction.

Suppose that execution has produced a state S satisfying $(*)$ and that a new task instance t' is created on the next machine cycle, producing a state S'. Let u and v denote instances in S' such that $u \subset v$. If $u \neq t'$, then the induction hypothesis implies that one of (i) or (ii) holds. If $u = t'$, then $t' \subset v$. We show that, in this later case, t' depends on v, and so (ii) holds.

Let TT denote the identifier used in the program for referring to the type of t', w denote the unit instance that contains the binding for TT visible from the instance that spawned t', and m denote the direct master of t'. Then the contour for w directly encloses the contour for t', and so the hypothesis $t' \subset v$ implies $w \subseteq v$. Moreover, the binding for TT in w is visible from m, and so that $m \subseteq w$. This gives the following picture:



Now, if $m = v$, then t' depends (directly) on v. On the other hand, if $m \subset v$, then the induction hypothesis implies that either v comes after m on the dynamic chain of the thread executing m (and also executing v) or that the task instance associated with this thread depends on v. In either case, we conclude that t depends (indirectly) on v.

This proves that creating a new task instance preserves $(*)$. The proof that creating a new procedure instance preserves $(*)$ is similar. ∎

Proof of Theorem 3. If t is allocated, then Theorem 3 follows easily from Theorems 1 and 2. We assume, therefore, that t is elaborated.

Let T denote the identifier in the binding used for accessing t from u and let v denote the unit instance that contains the binding. Of course, $u \subseteq v$, and so Theorem 1 implies that either

(1a) v comes after u on the dynamic chain of the thread for t' or

(2a) t' depends on v

Now, for each unit instance w in the hypothesized program state, we denote by $A(w)$ the chain that begins with w and enumerates w's ancestors in the order induced by the hierarchy of masters and dependents. We then induct on the number n of unit instances in $A(u)$ that received t as an actual parameter.

For the base case, we note that if $n = 0$, then T does not denote a formal parameter. Thus, v spawned t by elaborating a task declaration. In this case, therefore, v is the direct master of t and the theorem follows easily from (1a) and (2a).

For the induction step, we take $n > 0$. If T does not denote a formal parameter, then the argument given for the base case applies. We assume, therefore, that T denotes a formal parameter, in which case v must be an instance of a subprogram. Let t'' denote the task instance associated with the thread that created v and let w denote the unit instance that invoked v.

Now, (1a) and (2a) imply that v belongs to $A(u)$. Clearly, then, w also belongs to $A(u)$, and $A(w)$ is the tail of $A(u)$ that begins with w. Applying the induction hypothesis to w, we conclude that either

(1b) m comes after w on the dynamic chain of the thread for t'' or

(2b) t'' depends on m

The above observations produce four cases. In the case that (1a) and (1b) hold, we conclude (i) of the theorem, and in the other three cases, we conclude (ii). ∎