Task Termination and Ada 95

A.J. Wellings and A. Burns
Real-Time Systems Research Group
Department of Computer Science
University of York, U.K.
{andy,burns}@minster.york.ac.uk

O. Pazy
48 Beeri St., Tel-Aviv 64233, Israel
pazy@world.std.com

September 9, 1996

Abstract

Ada 83 removed from the programmer the burden of coding potentially complex termination conditions between clients and servers by introducing an 'or terminate' option to the select statement. With the introduction of indirect communication (emphasised by the provision of protected objects in Ada 95), it is no longer straightforward to obtain program termination. This paper illustrates the problem and suggests that adding a terminate alternative to an entry call might solve the problem. The advantages and disadvantages of the approach are discussed.

Keywords: termination, Ada 95

1 Introduction

Although there were many perceived difficulties with the Ada 83 (U.S. Department of Defense, 1983) tasking model, one of its benefits was that it provided a simple mechanism for application programmers to specify termination conditions. With the introduction of indirect communication (emphasised by the provision of protected objects in Ada 95(Intermetrics, 1995)), it is no longer straightforward to obtain program termination.

This paper firstly considers the problem of termination when asynchronous communication is introduced. It then considers the extent to which termination in Ada 95 can be supported by changing the language so that entries can be called with an "or terminate" option. Finally we present our conclusions.

2 The Basic Problem

The motivation for having a terminate alternative on the Ada select statement was to provide a simple mechanism with which to terminate server tasks. For example, consider the following Ada program:

```
task type Producer;

task Consumer is
  entry Next(...);
end Consumer;
```

```
task body Producer is
begin
  loop
    ...
    Consumer.Next(...);
    exit when ...;
   end loop;
end Producer;

task body Consumer is
begin
  loop
    select
      accept Next(...) do
        ...
      end;
    or terminate;
    end select;
  end loop;
end Consumer;

P1, P2 : Producer;
```

This program will terminate irrespective of how many **Producers** are created. The **Consumer** task simply indicates that it wishes to terminate when there are no more tasks requiring its services. Without the terminate option, it would be necessary to program the **Consumer**'s termination explicitly.

With the above approach, the **Producer** has to wait for the **Consumer** to service its request. An alternative paradigm is where the **Producer** simply issues its request and then continues. The classical solution to this problem is to introduce a bounded buffer between the tasks. To do this in Ada 83 required the introduction of a buffer task and modifications to the **Producer** and **Consumer**.

```
task type Producer;
task Consumer;

task Buffer is
  entry Get(...);
  entry Put(...);
end Buffer;

task body Producer is
begin
  loop
    ...
    Buffer.Put(...);
    exit when ...;
  end loop;
end Producer;

task body Consumer is
begin
  loop
    ...
    Buffer.Get(...);
    ...
  end loop;
end Consumer;


task body Buffer is
  Full : Boolean := False;
  Empty : Boolean := True;
  ...
begin
  loop
    select
      when not Full =>
        accept Put(...) do
          ...
          Full := True;
          Empty := False;
        end;
    or
      when not Empty =>
        accept Get(...) do
          ...
          Full := False;
          Empty := True;
        end;
    or terminate;
    end select;
  end loop;
end Buffer;

P1, P2 : Producer;
```

However, this solution introduces a termination problem. When the **Producer** tasks finish, the **Consumer** task is left waiting on a closed entry of the **Buffer** task. Note also that the **Buffer** task will not terminate because the **Consumer** task has visibility of the buffer and consequently can call **Get**. The system is thus deadlocked. We are, therefore, forced to consider ways to terminate the consumer.

In Ada 95, the Buffer task would be replaced by a protected object. However, the problem for the consumer remains.

Programming termination in concurrent systems is non-trivial. Tokens must be passed from producers to consumers to indicate that a producer is about to terminate (this is the approach that is often taken with occam programs (Burns, 1988), for example). In Ada, each client could register with the server, and the server could keep track of all its registered clients. Clients would also de-register when they have finished with the server or when they terminate (using Ada's finalisation facility). Servers can then terminate when all their clients no longer need their services. However, it was just these types of *ad hoc* algorithms that Ada 83 was trying to avoid with its termination option on the select statement.

3 Termination in Ada Revisited

In this section we reconsider termination in Ada 95. In particular, we address whether termination can be supported by a change in the language definition.

3.1 Entry Call with a Termination Option

An alternative approach to requiring the user to program termination is to require that the language be extended to provide automatic termination. One way to provide this capability is to add a terminate option to the entry call facility.

Consider again the simple producer/consumer program. Consumers would be structured as follows:

```
task type Consumer;

task body Consumer is
begin
  loop
    select
      Buffer.Get(...);
      -- consume
    or terminate; -- NOT VALID Ada
    end select;
  end loop;
end Consumer;
```

The terminate option would be mutually exclusive with the delay, else and then abort options.

There are two cases to consider:

1. the Buffer object is a task
2. the Buffer object is a protected object

In both of these cases, a task waiting on an entry call with the terminate option will terminate if and only if all tasks which have the same master as the target protected (or task) object are either terminated or similarly waiting on an entry call with a terminate option. If one of these tasks is waiting on a different object than the first one considered, then the same termination check has to be performed for that object too. If the call with terminate is done from within an abortable part (in some dynamically-nested level), then the calling task is assumed to be "non-terminable" at that point.

Of course, producer/client tasks can also terminate in the normal way when they have finished their allotted work (by reaching the end of their body).

3.2 Semantic Changes/Difficulties

The proposal made in this section raises some semantic issues that have to be studied further. Below, we enumerate those topics that immediately come to mind with our initial thoughts about them.

Interaction with user-defined finalisation

Before all tasks waiting on an entry call with "terminate" can terminate, the construct that declares the protected object (or an access to the protected type) must itself be completed or have issued an entry call with "terminate". However, after a construct is completed, it must finalize. This finalisation step may include user-defined finalisation routines which may issue further entry calls on the

task or the protected object since the latter's entries are still visible at this point. Such entry calls will violate the termination condition which has already been determined; this is, of course, unacceptable. Note, however, that finalization routines must be at the library-level and in most cases they will not see the relevant object.

Extending the notion of masters

In order to properly define the set of "all possible tasks", the notion of masters will have to be used (mainly because access values designating protected objects can be passed around in a partition). Currently, this concept is tightly coupled to tasks and their active nature. It is also an essential part of the language model, but still quite subtle. More work is needed to determine if extending the concept of a master is feasible.

Interaction with Asynchronous Transfers of Control

For this feature to work, the termination condition defined above, must be relatively stable. That is, in order to avoid the need for a complicated protocol, there should be a limited set of well-defined events that can cause the calling task to leave the "call-with-terminate" state. As we have discussed above, two such events are currently recognised: one is when the corresponding entry becomes open (and then the entry call completes and the task continues execution). The other is when the termination condition is reached causing the task to terminate. Note that both of these cases are detected state changes in the protected object itself (that is, when its lock is held). They are not triggered asynchronously from the outside. As a consequence, the algorithm that is required to commit to termination is fairly straight-forward.

However, an entry call (with terminate) issued by the client, may itself be dynamically nested within an outer abortable part. The code that makes the call may not even be aware of the fact that it is inside such an abortable construct[1]. If the abortable part aborts, thus cancelling the entry call, the calling task may continue executing on a different path. This will make the termination condition transient which will require a much more complex synchronisation protocol implementation. Note that the construct in question may be aborted by a totally unrelated task (i.e. a task that is not necessarily in the "possible tasks" set). It may also be triggered by the opening of an entry of an unrelated protected object. This problem does not exist for the selective-accept with terminate construct. The language does not allow accept statements to be nested (dynamically or statically) inside an abortable part. The motivation for this restriction was partly due to similar problems.

There does not seem to be a simple solution to this problem. The implementation cost may be quite significant.

Interaction with the requeue statement

Clearly, the semantic meaning of requeuing a task with and without the abort option needs to be defined when an entry has been called with a terminate option. Presumably, requeuing with abort should allow the task to terminate, whereas without abort would remove the termination option. However, further consideration is needed to determine whether any other interactions exist.

3.3 Usability Issues

In addition to semantic problems introduced above, the result of adding a terminate option to an entry call means that:

- some programs which previously would deadlock would now terminate if the entry is called with a termination option, as illustrated below:

[1]Here, we are only concerned with abortable constructs which are not the entire task. No special problem is introduced if the calling task is aborted as a whole.

```
task A is
  entry One;
end A;


task B;


task body A is
begin
  ...
  select
      when False =>
        accept One do
          ...
        end;
  or
    terminate;
  end select;
  ...
end A;


task body B is
begin
  ...
  select
    A.One;
  or
    terminate; -- NOT VALID Ada
  end select;
  ...
end A;
```

- a task which previously had Tasking_Error raised, would now wait for termination if it called a completed task with the terminate alternative specified, as illustrated below:

```
task A is
  entry One;
end A;


task B;


task body A is
begin
  ...
  if False then
    accept One do
      ...
    end;
  else
    null;
  end if;
  ...
end A;


task body B is
begin
  ...
  select
    A.One;
  or
    terminate; -- NOT VALID Ada
  end select;
  ...
end A;
```

Finally, consider what would happen with the following code:

```
task A is
  entry One;
end A;


task B;


task body A is
begin
  ...
  select
    accept One do
      ...
    end;
  or
    terminate;
  end select;
  ...
end A;


task body B is
begin
  ...
  select
    A.One;
  or
```

```
    terminate; -- NOT VALID Ada
  end select;
  ...
end A;
```

In this situation, the rendezvous would occur and no termination would take place. However, if the accept statement was guarded and the guard evaluated to false, then termination would take place (if all other tasks are terminated or waiting at a select statement with a terminate option open).

4 Conclusion

In this paper we have revisited task termination in Ada 95. We conclude that termination of tasks involved in asynchronous communication is more difficult than for those involved in synchronous communication.

One potential solution to this problem is to add a terminate alternative to the entry call facility. Although this initially appears to be an attractive solution, there are semantic problems which need to be considered further (in particular, the interactions with finalisation and asynchronous transfer of control). Moreover, the addition of such a feature to the language is likely to add a significant cost to the implementation. A similar mechanism already exists for the selective-accept construct, but there it seems more justified since a task is inherently a more heavy-weight construct. A protected object is intended to serve as a light-weight and efficient mechanism, and therefore, every small addition to its implementation cost is much more noticeable. Also, there is a danger that cost will be incurred even if the feature is not used by a program.

References

Burns, A. (1988). *Programming in occam 2*, Addison Wesley.

Intermetrics (1995). Ada 95 reference manual, *ANSI/ISO/IEC-8652:1995*, Intermetrics.

U.S. Department of Defense (1983). Reference manual for the Ada programming language, *ANSI/MIL-STD 1815 A*, U.S. Department of Defense.