Integrating OOP and Tasking – The missing requeue*

A.J. Wellings and A. Burns
Department of Computer Science
University of York, UK
{andy,burns}@cs.york.ac.uk

Abstract

Ada 2005 has provided limited integration between its object-oriented programming and tasking facilities. The compromise has been to allow tasks and protected types to support interfaces but not inheritance. Whilst the language supports timed/conditional entry calls and the select-then-abort statement using interfaces, it does not allow the target of a requeue statement to be determined via an interface. This paper argues that this design decision needlessly limits the expressive power of the language. A motivating example is used to illustrate the case, and the semantics of the proposed language extension are discussed.

1 Introduction

One of the topics that has been covered by past workshops has been the integration of Ada 95 object-oriented programming features with the Ada tasking model [1, 2, 5]. Full integration would add significant complexity to the language (for example, see the suggested models [4, 8, 7, 6, 3]), and proposals for language changes did not receive widespread support. Ada 2005 has added Java-like interfaces to the OOP model and has used this opportunity to provide limited integration. The compromise is to allow task and protected types to implement some types of interfaces but still not to be tagged types. Hence inheritance is not supported but run-time dispatching is.

A side effect of the Ada 2005 approach is that it is now possible to make timed (or conditional) entry calls or use the select-then-abort statement whose target entries are based on normal Ada procedures (if they implement part of a limited, synchronized, protected or task interface). The language defines that when such a call is issued it is deemed to happen immediately and hence no timeout or wait occurs, and in the case of the select-then-abort statement, no abortable code is started. Whilst these side effects might seem strange, it is a consequence of the decision taken that interfaces that have some synchronization implied should be identified explicitly. In this way, the programmer knows that calls are potentially suspending and can add timeouts if necessary. It is also necessary to more easily identify potentially suspending calls from within a protected action. In contrast, Java adopts the approach where interfaces have no synchronization constraints identified, even though the encapsulated methods can be implemented by synchronized methods. But then Java does not provide an avoidance model for condition synchronization; instead it uses conditional waits within a synchronized method (where timeouts can be added if needed).

Although Ada 2005 has gone a long way to integrating its access to entries via interfaces, it has shied away from allowing a requeue operation via an interface. The following is taken from the minutes of the ARG Paris meeting (Feb. 2005) under a discussion on AI-397:

> "Steve Baird asks if you can requeue on a synchronized procedure. No, we don 't allow that, because the synchronized procedure doesn't necessarily denote an entry. And what such a requeue would do if the procedure wasn't an entry is rather unclear."

This paper suggests that this decision should be revisited. In section 2, we briefly review the Ada 2005 OOP and tasking integration, and examine the general role of requeue. In section 3, we present a motivating example for why requeue should be allowed to occur via a synchronized (or task or protected) interface. Then, in section 4, we define the semantics of such an operation. Finally, we draw our conclusions.

2 Tasking, OOP and Requeue

An interface is a type that has no state. It is essentially an abstract tagged typed with null components. All primitive operations on an interface type must be abstract or null.

*This work has been undertaken within the context of the EU ARTIST2 project.

Although the inspiration for Ada interfaces comes from the Java language, they cannot be used in exactly the same way as Java interfaces but should be thought of as abstract types. In particular, it is not possible to define an arbitrary subprogram that takes as a parameter an interface (as can be done in Java). The equivalent in Ada 2005 is to define the parameter type as a *class-wide type* or *class-wide access type* rooted at the Ada interface. Hence, Ada requires you to distinguish between a classwide type and a specific type. This has the advantage of allowing differentiation between dispatching calls and non-dispatching calls at the source level.

Formally, a tagged type can be derived from zero or one parent tagged type plus zero of more interface types. If a non-interface tagged type is present it must be the first in the list. Hence interfaces themselves can be derived from other interfaces.

2.1 Limited interfaces

In Ada, a limited type is a type that does not support the assignment statement and has no predefined equality operator. Interfaces can be limited; they are, however, not limited by default. A non-limited interface has a predefined equality operator available, whereas a limited interface does not.

A new interface can be composed of a mixture of limited and non limited interfaces but if any one of them is non limited then the resulting interface must be specified as **non limited**. This is because it must allow the equality and assignment operations implied by the non-limited interface.

Similar rules apply to derived types that implement one or more interfaces. The resulting type must be non limited if any of its associated interfaces are non limited.

2.2 Synchronized, protected and task interfaces

In order to provide integration between Ada's concurrency and OOP models, Ada 2005 provides three further forms of limited interfaces: *synchronized*, *protected* and *task*.

The key idea of a synchronized interface is that there is some implied synchronization between the task that calls an operation from an interface and the object that implements the interface.

Synchronization in Ada is achieved via two main mechanisms: the rendezvous and the protected action (call of an entry or protected subprogram). Hence, a task type or a protected type can implement a synchronized interface. The rendezvous provides control-oriented synchronization and the protected type provides data-oriented synchronization. Where the programmer is not concerned with the type of synchronization, a synchronized interface is the appropriate abstraction. For situations where the programmer requires control-oriented (or data-oriented) synchronization, task interfaces (or protected interfaces) should be used explicitly.

In summary:

- All task and protected interfaces are also synchronized interfaces, and all synchronized interfaces are also limited interfaces.
- A synchronized interface can be implemented by either a task or protected type.
- A protected interface can **only** be implemented by a protected type.
- A task interface can **only** be implemented by a task type.
- The controlling (dispatching) parameter in a synchronized (or protected or task) interface must be the first parameter. Furthermore, if the operation is to be implemented as an entry (either protected or task) then it should be an "out", an "in out" parameter or an "access" parameter.

There is a hierarchy in the types of limited interfaces: limited comes before synchronized, which comes before task/protected. Task and protected interfaces have equal position in the hierarchy. Two or more interfaces can be composed as long as the resulting interface is not before any of the component interfaces in the hierarchy. Hence a limited interface *cannot* be composed from a synchronized, protected or task interface. A synchronized interface *cannot* be composed from a task or protected interface; and a task or protected interface *cannot* be composed from each other.

Calling operations on objects that implement limited interfaces

Operations on interfaces that are defined as limited, synchronized, protected or task may be implemented directly by a protected or task type. The following should be noted.

1. A call to any operation on an object that implements a synchronized (or protected or task) interface may be blocked until the right conditions are in place for that operation to be executed. If the operation is implemented by a protected procedure or function then this blocking will be bounded. If the operation is implemented by a task or a protected entry then the blocking may be indefinite. Consequently, Ada 2005 allows a call to an operation defined by a synchronized (or protected or task) interface to be placed in a "timed or conditional entry call" statement or in a "select then abort" statement.

2. A task or protected type can also implement a limited interface. Consequently, call to objects that implement limited interfaces may also block! Ada 2005, therefore, allows them to be placed in a timed or conditional entry call statement or in a select-then-abort (ATC) statement.

3. Any call to an operation on an object that implements a limited (or synchronized or protected or task) interface that dispatches to a non-entry call, is deemed to have been "accepted" immediately and, therefore, can never time-out.

2.3 Tasks and interfaces

A task type in Ada 2005 can be declared to "implement" zero, one or more combinations of limited, synchronized and task interfaces. Task interfaces allow programmers to specify interfaces that must be implemented by tasks. Hence they allow the programmers greater control over expressing their intentions. However, as already mentioned, it is not possible for tasks to be derived from other tasks. Obtaining a semantic model for this is challenging given an already-existing language.

2.4 Protected types and interfaces

Protected interfaces are interfaces that can only be implemented by protected types. Protected interfaces should be used when

- the programmer wants data-oriented rather than control oriented synchronization,
- there is a level of indirection between the tasks needing the synchronization, or
- the programmer wishes to ensure that the required synchronization is implemented by a passive synchronisation agent rather than an active one.

2.5 Synchronized interfaces

Task and protected interfaces are the correct abstractions to use when the programmer wishes to commit to a particular implementation strategy. In some circumstances, this may not be appropriate. For example, there are various communication paradigms that all have at their heart some form of buffer. They, therefore, all have buffer-like operations in common. Some programs will use these paradigms and will not care whether the implementation uses a mailbox, a link or whatever. Some will require a task in the implementations, others will just need a protected object. Synchronized interfaces allow the programmer to defer the commitment to a particular paradigm and its implementation approach.

2.6 Requeue

Ada allows requeues between task entries and protected object entries. A requeue can be to the same entry, to another entry in the same unit, or to another unit altogether. Requeues from task entries to protected object entries (and vice versa) are allowed. However, the main use of requeue is to send the calling task to a different entry of the same unit from which the requeue was executed. It is important to appreciate that requeue is not a simple call. If procedure P calls procedure Q, then, after Q has finished, control is passed back to P. But if entry X requeues on entry Y, then control is not passed back to X. After Y has completed, control passes back to the object that called X. Hence, when an entry or accept body executes a requeue, that body is "completed, finalised and left" (ARM, Section 9.5.4).

One consequence of this is that when a requeue is from one protected object to another then mutual exclusion on the original object is given up once the task is queued. Other tasks waiting to enter the first object will be able to do so. However, a requeue to the same protected object will retain the mutual exclusion lock (if the target entry is open).

Finally, when requeuing from one protected object to another it is important to understand that while the call is being evaluated a mutual exclusion lock is held on both protected objects. It is therefore a bounded error to execute an external requeue request back to the requesting object (it would inevitably lead to deadlock). Moreover, the programmer cannot assume, when a requeue has taken place from one protected object to another, that any tasks released in the original object will execute before the entry in the destination object. Consider the following contrived example that consists of two protected objects.

```
protected Original is
  entry One;
  entry Two;
private
  Go : Boolean := False;
end Original;

protected Target is
  entry One;
end Target;
```

In the body of this, a call to `Original.One` is requeued to `Target.One`:

```
protected body Original is
  entry One when True is
  begin
    Go := True;
    requeue Target.One;
  end One;

  entry Two when Go is
  begin
    Go := False;
    Put_Line("Original Two executed");
  end Two;
end Original;
```

```
protected body Target is
  entry one when True is
  begin
    Put_Line("Target One executed");
  end One;
end Target;
```

Now consider a case where a task (A) is already queued on the `Original.Two` entry when another tasks (B) calls `Original.One`. As the barrier on `Original.One` is open, the entry is executed and `Go` flag is set to true, and the call is requeued on `Target.One`. First the lock on the `Target` protected object is obtained, and then the barrier is tested. If the barrier on `Target.One` was false, task B needs to be queued and then the `Original.One` entry needs to be finalized which involves task B executing the entry `Original.Two` on behalf on task A. Hence, the lock on the `Target` protected object is maintained until this finalization code has been executed; task B is then placed on the entry queue. If the barrier on `Target.One` is True (as it is in this case), task B executes the `Target.One` entry code and then finalizes that entry. It then finalizes the `Original.One` entry and executes the `Target.One` entry code and then finalizes that entry. Hence in the above example, the output "Target One executed" appears before "Original Two executed".

3 A Motivating Example

Although requeuing to the same entity represents the normal use of requeue, there are situations in which the full generality of this language feature is useful.

Consider the situation in which resources are controlled by a hierarchy of objects. For example, a network router might have a choice of three communication lines on which to forward messages: Line A is the preferred route, but if it becomes overloaded Line B can be used; if this also becomes overloaded Line C can be used. Each line is controlled by a server task; it is an active entity as it has housekeeping operations to perform. A protected unit acts as an interface to the router; it decides which of the three channels should be used and then uses requeue to pass the request to the appropriate server. The structure of the solution is given in the program fragment is given below:

```
type Line_Id is (Line_A, Line_B, Line_C);

type Line_Status is array (Line_Id) of Boolean;

task type Line_Controller(Id : Line_Id) is
  entry Request(...);
end Line_Controller;

protected Router is
  entry Send(...);
  procedure Overloaded(Line : Line_Id);
  procedure Clear(Line : Line_Id);
private
  Ok : Line_Status := (others => True);
end Router;

La : Line_Controller(Line_A);
Lb : Line_Controller(Line_B);
Lc : Line_Controller(Line_C);

task body Line_Controller is
 ...
begin
  loop
    select
      accept Request(...) do
        -- service request
      end Request;
    or
      terminate;
    end select;
    -- housekeeping including possibly
    Router.Overloaded(Id);
    -- or
    Router.Clear(Id);
  end loop;
end Line_Controller;

protected body Router is
  entry Send(...) when Ok(Line_A) or
                       Ok(Line_B) or Ok(Line_C) is
  begin
    if Ok(Line_A) then
      requeue La.Request with abort;
    elsif Ok(Line_B) then
      requeue Lb.Request with abort;
    else
      requeue Lc.Request with abort;
    end if;
  end Send;

  procedure Overloaded(Line : Line_Id) is
  begin
    Ok(Line) := False;
  end Overloaded;

  procedure Clear(Line : Line_Id) is
  begin
    Ok(Line) := True;
  end Clear;
end Router;
```

Now consider the generalization of code to allow a router to support *any* type of network interface controller. First, it is necessary to define the interface that all line controllers must support:

```
package Network_Interface is
  type Line_Interface is task interface;

  procedure Request(LI: in out Line_Interface)
            is abstract;

  type Any_Line_Interface is access all
            Line_Interface'Class;
end Network_Interface;
```

Any line controller can be defined, as long as it supports this interface:

```
type Line_Id is (Line_A, Line_B, Line_C);

type Line_Status is array (Line_Id) of Boolean;
```

```
task type Line_Controller(Id : Line_Id) is new
     Line_Interface with
  overriding entry Request(...);
end Line_Controller;
```

Now a general purpose router can be defined that will work with any line controller

```
protected type Router(LA: Any_Line_Interface;
                      LB: Any_Line_Interface;
                      LC: Any_Line_Interface) is
  entry Send(...);
  procedure Overloaded(Line : Line_Id);
  procedure Clear(Line : Line_Id);
private
  Ok : Line_Status := (others => True);
end Router;
```

However, the body of this protected type causes a problem:

```
protected body Router is
  entry Send(...) when Ok(Line_A) or
                       Ok(Line_B) or Ok(Line_C) is
  begin
    if Ok(Line_A) then
      requeue LA.Request with abort; -- ** NOT LEGAL
    elsif Ok(Line_B) then
      requeue LB.Request with abort; -- ** NOT LEGAL
    else
      requeue LC.Request with abort; -- ** NOT LEGAL
    end if;
  end Send;
  -- as before
end Router;
```

What this example shows is two things:

1. Requeing from one task/protected type to another is an integral part of the Ada model and one that gives it great power and flexibility.

2. Requeuing via interfaces is a necessary facility if interfaces are to be fully integrated with tasks and protected types.

4 Proposed Revised Requeue Semantics

Section 3 has illustrated the need to allow the use of requeue with interfaces. In this section we consider the semantics for the situation where the target requeue is not an entry call. There are three possible situations.

1. The target is a function inside or outside a protected object – this is an error condition that can be caught at compile time; there are no circumstances whereby a function in an interface can be implemented by an entry. This is similar to the illegal case where a function call is used as the target of a timed/conditional entry call or as a triggering event in a select-then-abort statement.

2. The target is a procedure inside a protected object – this is similar to the case where the target of a timed/condition entry call or select-then-abort triggering event turns out to be a procedure that has been implemented inside of a protected type implementing the associated interface. The corresponding interpretation would be to view the procedure as an entry with a "when True" barrier. Hence, a new protected action is started for the target protected object and the procedure is executed immediately(ARM, Section 9.5.4 par 11).

3. The target is a regular procedure (i.e. outside a protected object/task) – this is similar to the case where the target of a timed/condition entry call or select-then-abort triggering event turns out to be a procedure outside of the protected object/tasks implementing the associated interface. As this would succeed in this case, so should it succeed in the requeue case. However, to execute the procedure immediately would mean that the procedure would be executed at the ceiling priority of the original protected type (or the priority of the original rendezvous). This would seem to be wrong. Hence, the current protected action continues and completes before the procedure is called.

Return again to the example given in Section 2.6, but this time suppose that the `Target` is defined as:

```
type Target is synchronized interface;
procedure One(T: in out Target)
            is abstract;
type Any_Target is access all
            Target'Class;
```

and the `Original` protected object is now a type parameterized by the target.

```
protected Original(Target : Any_Target) is
  entry one;
  entry two;
private
  Go : Boolean := False;
end Original;

protected Target is
  entry one;
end Target;


protected body Original is
  entry One when True is
  begin
    Go := True;
    requeue Target.one;
  end One;

  entry Two when Go is
  begin
    Go := False;
    Put_Line("Original Two executed");
  end Two;
end Original;
```

Assuming that the target subprogram again outputs the string "Target One executed", then if the actually target

`One` turns out to be an open entry or a procedure then once again the output would be "Target One executed" followed by "Original Two executed". However, if the actual target `One` turns out to be a procedure outside of a protected type, the output would be "Original Two executed" followed by "Target One executed". These two cases would need to be recognised by the run-time system. It is unclear whether this would impose a significant implementation burden.

Alternative approach

As an alternative to the proposal given above, it could be argued that attempting to requeue to a non-entry is an error condition and a run-time exception (`Program_Error`) should be raised. As the Ada 2005 designers chose not to do this for the timed/condition entry call and select-then-abort cases, we proposed, for consistency, the same model.

Of course, where the programmer knows that the intended target is an entry, ideally there should be a mechanism to indicate this (perhaps by a pragma). This would allow the compiler to check, thereby avoiding the run-time error. If this approach were adopted, the semantics of timed/-conditional entry calls and the select-then-abort statement might need to be revisited to make the operations illegal on procedure calls again. However, given that limited interfaces can also be implemented by tasks and protected objects, adding such a pragma would, in effect, make them synchronized.

5 Conclusions

This paper has argued that it should be possible to execute a requeue statement (both with and without abort) via a limited, synchronized, protected or task interface. The proposed semantics are consistent with those that are currently defined for conditional and timed entry calls, and for the use of interfaces with the select then abort statement. Indeed, given the current semantics there are obvious counterparts for the requeue case. However, an implementation must perform the finalization of a called protected entry in a different order if the target is not encapsulated within a protected object. The actual overhead that this incurs are implementation-dependent.

6 Acknowledgements

The authors wish to thank Tucker Taft for a comment on an early draft of this paper and for helping us refine the proposed semantics given in Section 4.

References

[1] J. de la Puente. Session summary: Object-oriented programming and real-time. In *Proceedings of IRTAW8, Ada Letters*, pages 11–15, 1997.

[2] J.A. de la Puente. New language features and other language issues. In *Proceedings of IRTAW9, Ada Letters, Vol XIX(2)*, pages 19–20, 1999.

[3] R. Garcia and A. Strohmeier. Experience report on the implementation of EPTs for GNAT. In *Proceedings of IRTAW11, Ada Letters, Vol XX1I(4)*, pages 22–17, 2002.

[4] O. P. Kiddle and A. J. Wellings. Extended protected types. In *Proceedings of ACM SIGAda Annual International Conference (SIGAda 98)*, pages 229–239, November 1998.

[5] J.L. Tokar. Tasking and object orientation. In *Proceedings of IRTAW10, Ada Letters, Vol XX1(1)*, pages 9–10, 2001.

[6] A. J. Wellings, B. Johnson, B. Sanden, J. Kienzle, T. Wolf, and S. Michell. Integrating object-oriented programming and protected types in Ada 95. *ACM TOPLAS*, 22(3):506–539, 2000.

[7] A. J. Wellings, B. Johnson, B. Sanden, J. Kienzle, T. Wolf, and S. Michell. Object-oriented programming and protected objects in Ada 95. *Reliable Software Technologies - Ada-Europe 2000, Lecture Notes in Computer Science*, 1845:16–28, 2000.

[8] A. J. Wellings, B. Johnson, B. Sanden, J. Kienzle, T. Wolf, and S. Michell. Extensible protected types: Proposal status. In *Proceedings of IRTAW10, Ada Letters, Vol XX1(1)*, pages 105–110, 2001.