Session Summary: Language and Distribution Issues

Chairs: Tullio Vardanega and Michael González-Harbour

Rapporteur: Luís Miguel Pinho

1 Introduction

The goal of the session was to consider a set of additions and changes to the language arising from the accepted position papers, but still not consensual. The session lasted a full day, with Tullio Vardanega and Michael González-Harbour in charge, respectively, of the Morning and Afternoon periods.

At the beginning of the day, Tullio started by presenting the outline of the session, with an initial list of issues to discuss but also noting that the agenda was to be regarded as fairly open and that new or returning issues could easily be integrated.

The anticipated topics (and proponents) for the session were:

- Non-preemptive scheduling and the use of Ravenscar and sporadic tasks with EDF, proposed by Rod White;
- Named memory (storage) pools, non-blocking delays for hardware interfacing and parallel release of barriers, proposed by Luke Wong, Stephen Michell and Brad Moore;
- Generalising EDF support and user-defined clocks, by Andy Wellings and Alan Burns;
- Execution-time accounting of interrupts handlers, a topic with two position papers, one by Mario Aldea Rivas and Michael González-Harbour, and another by Kristoffer Nyborg Gregertsen and Amund Skavhaug;
- The Real-Time Transaction model, by Héctor Pérez Tijero, J. Javier Gutiérrez and Michael González-Harbour.

Other issues were also discussed during the session, either because they were continued from the previous day, or because they were considered to be related:

- Non-Uniform Memory Access architectures, by Andy Wellings, Abdul H. Malik, Neil Audsley and Alan Burns;
- The Ravenscar profile and Multiprocessors, by José Ruiz;
- The Real-Time Framework, by Jorge Real and Alfons Crespo.

The session was highly dynamic, with several rounds of discussion. In order to increase the readability of this report, the successive rounds of discussion on individual topics are collated in a single presentation, instead of actually following the chronological flow of the session.

2 Discussion

2.1 Non-preemptive scheduling

This topic started with a presentation by Rod White, proposing extensions to the Ada real-time features, mainly motivated by experience on practical industrial use of (some of the) new capabilities of Ada 2005 [1].

Rod's objectives were also to make the new features easy to use, without disrupting the current ones, introducing more transparency, as complexity or obscurity would impair industrial adoption.

In particular, Rod introduced three topics for discussion [1]:

1. A more complete and transparent model for the control of dispatching points in non-preemptive scheduling;
2. The use of the Ravenscar profile in conjunction with EDF dispatching;
3. The treatment of sporadic tasks under EDF dispatching.

In the first topic, Rod noted that the current mechanism for tasks to yield the processor in the non-preemptive model is to perform a non-blocking delay, such as a delay into the past (`delay until Clock_First`), which, although giving a correct execution, does not provide a clear understanding.

Furthermore, when a task relinquishes the processor, it is placed at the tail of the ready queue for its priority. Therefore, this command cannot be executed inside a protected subprogram. Rod then proposed a new package for handling non-preemptive scheduling:

```
package Ada.Dispatching.Non_Preemptive is
   procedure Yield_To_Higher;
   procedure Yield; -- Bounded error if executed within a protected operation
end Ada.Dispatching.Non_Preemptive;
```

`Yield` would replace the non-blocking delay as the mechanism for relinquishing the processor. `Yield_To_Higher` would put the task at the head of its priority queue, therefore only allowing higher priority tasks to execute. This would allow it to be performed inside protected subprograms.

Alan Burns then noted that `Yield` could have a different behaviour inside a protected subprogram, deferring the `Yield` operation to when the task left the protected object, instead of giving a bounded error.

In addition to the new package, Rod also proposed a new pragma (`Cooperate_On_Priority_Change`), that would allow the runtime to implicitly perform `Yield_To_Higher` operations anytime a task's priority was changed (such when entering or leaving a protected object). This would permit a more precise control on the responsiveness of the system.

In a second round of discussion, the workshop came back to this issue, and a proposal was made to include `Yield` and `Yield_To_Higher` in the language. The actual wording would be worked offline, but the general model was considered OK. A straw vote was taken, and the proposal was accepted with 19 votes in favour and 3 abstentions.

As for the `Cooperate_On_Priority_Change` pragma, there was a consensus that this was a new scheduling model and that as such it needed further thoughts. The proposal was thus withdrawn.

2.2 Ravenscar and EDF

On a different topic, Rod noted that EDF allows higher levels of processor utilisation, thus proposed its use within the Ravenscar profile, by introducing a second parameter to the pragma `Profile`:

```
pragma Profile (Ravenscar, EDF_Across_Priorities);
```

that would default to current behaviour (`FIFO_Within_Priorities`), if not specified. This would also allow for other scheduling strategies (such as non-preemptive) to be used together with Ravenscar.

It was not clear to Rod if this would be a variant of Ravenscar, or a new profile. Creating a new profile could be heavy for a simple change in the dispatching policy. Furthermore, Ravenscar is well established in industry – a new name would not be regarded as highly. Therefore, Rod's proposal was to maintain the Ravenscar brand, although probably with some "EDF tag".

At this point Tullio Vardanega asked whether the proposal also considered priority bands or only a model where all priorities had the same dispatching policy. Rod confirmed that his idea was the latter. Michael González-Harbour then noted that by allowing the use of priority bands would permit to mix critical and non critical tasks in the same system, thus with higher levels of flexibility. Nevertheless, Rod answered that for simplicity they would not implement priority bands in their Ravenscar kernel. Rod put forward that the Ravenscar kernel is used in non-critical parts of the system, for efficiency reasons, where EDF could be usefully employed.

In a second round of discussion, Joyce Tokar presented a model of “overriding” the dispatching policy in Ravenscar. Nevertheless, it was noted that all of the mechanisms which are used for EDF control would need to be analysed considering the Ravenscar restrictions (e.g. changing deadlines). The group considered that further work was required to think this proposal through and recommended this to be a subject for the next workshop.

2.3 Sporadic tasks under EDF

In another topic, also presented by Rod, Ada’s implementation of EDF dispatching assumes that tasks are periodic, and are scheduled using the clock. However, in embedded systems there are cases where periodic events in nature are released by an external event, and not by the language clock. And, for this case, there is no equivalent to the procedure `Delay_Until_And_Set_Deadline`, thus it is not possible to set the deadline of sporadic tasks when they are suspended.

Solutions to this problem are possible with the current features of the language, based on protected objects (examples of this are presented in [1]). However, a more efficient solution, and equivalent to the periodic `Delay_Until_And_Set_Deadline` mechanism, would be an extension to the Suspension_Object, through a child package:

```
package Ada.Synchronous_Task_Control.EDF is
   procedure Suspend_Until_True_And_Set_Deadline (S : in out Suspension_Object;
                                                   D : in Ada.Real_Time.Time_Span);
end Ada.Synchronous_Task_Control.EDF;
```

At this point, Andy Wellings raised the issue that user defined clocks (which would be discussed later on) could be a potential solution to this problem. However, it was not clear how that would be accomplished.

A vote was taken later in the session, and the proposal received 8 votes in favor and 12 abstentions. In the Workshop’s tradition, abstentions are taken as "informed non-opposition", thus this result was a sufficient basis to promote an AI on the topic. Alan then accepted to prepare an AI to support the proposal.

Another issue was raised with the recurrent use of the “*Suspend/Delay until true and set something*”, since in the previous day the same model was proposed for setting processor affinities. This was regarded as not scalable, as the number of attributes to set when suspending is increased. Although no conclusion was reached on the issue, it was considered something that should be further analyzed.

2.4 Non-blocking delays for hardware interfacing

After Rod’s presentation, the session moved on to Stephen Michell, who presented for consideration in the session [2,3]:

- Non-blocking delays for hardware interfacing
- Named memory (storage) pools
- Parallel release of barriers

In the first item, Steve presented the problem when a task needs to access hardware, but requires it to be ready after some “settle” time. The solution for the task to suspend itself is not possible inside a protected object (potentially suspending operations are not permitted), but if the task spin-waits it consumes CPU time undesirably.

The proposal was then to allow for some kind of delay, inside a protected subprogram. A consequence would be that objects would have to implement actual locks, which is not a problem in multiprocessor systems, where priority-based locks are no longer effective.

Alan Burns then put forward that the same effect could be currently obtained by setting a timer, and requeuing to a private entry. Upon expiration of the timer, the task would be released and could access the hardware. After some consideration, this was accepted as an already existent solution, and the proposal was withdrawn.

2.5 Named memory (storage) pools

Steve also presented a proposal to give the programmer more control on the specification of storage pools. In order to be able to specify and interface with different types of memory, a new API was proposed, where the memory model became similar to direct file I/O [3].

The proposed API would allow to create, read and write memory, but with a more precise control of the characteristics of the underlying memory type (for instance read only). The dynamic addition of memory to the pool was also analysed but considered more complicated to implement.

Andy Wellings also presented his view that many times it is necessary to be able to specify the actual address of storage pools. An attribute `'Address` could simply solve the problem. Steve noted that it would not be enough, as special memory may require explicit reads or writes to be performed.

Andy also added that in Non Uniform Memory Access (NUMA) architectures the same problem exists, since it is necessary to separate and identify the heap of each processor. There was some discussion on this issue, but no conclusion was drawn.

2.6 Parallel release of barriers

Another proposal by Steve was to allow the parallel release of several tasks in the same event, which would be useful in a multiprocessor environment. Currently, Ada defines that only the task at the head of an entry queue is released, which means that the release of multiple tasks must be sequential. Also, suspension objects only allow one suspended task.

Steve then proposed to add parallel release capabilities to protected entries and suspension objects. The first would be provided by a `pragma Barrier_Entry` which would identify an entry as a barrier. Furthermore, to allow for parallel execution inside the entry, it would not be allowed to change the protected state.

The latter would be supported by adding a new type `Group_Suspension_Object` to `Ada.Synchronous_Control`, which would allow one to specify a maximum count of suspended tasks. This solution would be simpler, but less flexible. Protected objects are general programming constructs, therefore a solution for protected entries must be much more generic, thus complex.

A note was then made that whatever the solution, it should not break if implemented in a sequential environment (e.g. uniprocessor systems). The resulting behaviour would need to be the same.

Ed Schonberg presented his view that, in order to avoid confusion with regular entries, this mechanism would need its own syntax. This was generally agreed upon by participants, but two different perspectives were put forward: either to add a barrier condition to functions, or to create “entry functions”. The advantage of the last is that the specification of the protected object would clearly indicate the parallel release. The existence of a barrier condition in the function would only be known within the protected body.

Later in the session, and after some work during the break, Steve proposed an API for the suspension barrier and a model for “entry functions”.

For the suspension barrier, the new proposal was not to mix barriers with suspension objects, and thus implement a child package instead:

```
package Ada.Synchronous_Control.Suspension_Group is

   type Group_Suspension_Object_Status is record
      Last_Released : Boolean;
      Count         : Positive);

   type Group_Suspension_Object(Count : Positive) is limited private;
        -- count of 1 => unlimited blocking count
        -- and explicit release
        -- suspended tasks can always be released by
        -- call to Set_True

   procedure Set_True (S : in out Group_Suspension_Object);

   procedure Set_False (S : in out Group_Suspension_Object);

   procedure Current_Status(S : Group_Suspension_Object)
      return Group_Suspension_Object_Status;

   procedure Suspend_Until_True(S : in out Group_Suspension_Object;
                                Unique : out Boolean);

end Ada.Synchronous_Control.Suspension_Group;
```

It was generally agreed that this could be accepted, but it was anyhow decided to defer the issue to the next day for further iteration.

Concerning the protected object model, the proposal was to support special “entry functions”, where the last released task would need to set the barrier back to false. This means that this special task would need to change the state of the protected object, which could impact the other tasks executing in parallel inside the entry. The solution was to define a `pragma Modifiable_State`, which permits to specify what state of the object could be changed by the last released task. All other tasks would not be able to read it.

```
protected type PT is
   ...
   entry function Barrier( A, B, C : Integer ) return Integer;
   pragma Barrier_Entry(Entry_Name => Barrier, Count => N);
   procedure Release;

private
   State : Local_State;
   Go_Now : Boolean;
   Total_Count : Integer;
   pragma Modifiable_State(Total_Count);
end PT;

protected body PT is
   entry function Barrier( A, B, C : Integer ) return Integer
      increments Total_Count when Go_Now is
   begin
      . . .
       if PT'Last_Released then
          Total_Count := 0;
       end if;
   end Barrier;
   procedure Release is
   begin
      if Total_Count = N then
         Go_Now := True;
      or else Now then
         Go_Now := True;
      end if;
   end release;
end PT;
```

Ed Schonberg noted the drawback of the model where one particular task (the last released one) has a special role, as it must be explicitly handled by the programmer inside the entry code. Some proposals were then made to include this special code inside the *when* clause of the entry, or it to be handled by the releasing task (by requeuing and waiting for the last task to leave the entry function).

A question was asked whether the ARG would accept a change of syntax, as this was the first proposal within the workshop with that requirement. It was considered possible by the members of the ARG present in the meeting, as this was a localized change.

The general feeling was that, although interesting, the model of how to control the resetting of the barrier after all task were released was still not yet mature enough.

The workshop came back to this issue in a third round of discussion, with a set of different proposals made by Andy Wellings. In the first proposal, and considering that the guard is not re-evaluated after each task release (tasks are all released simultaneously), it was up to the releaser to wait to clean the object state:

```
entry Go is
begin
  Data_Available  := True;
  if Parallel'Count > 0 then requeue Clean_Up2;
  else requeue Clean_Up1;
  end if;
end;

entry function Parallel when Data_Available and  Parallel'Count = 10 is
begin
  return Data;
end;

entry Clean_Up1 when Parallel'Count >  0 is
begin
  requeue Clean_Up2;
end;

entry Clean_Up2 when Parallel'Count = 0 is
begin
end;
```

A problem existed however, if a new task calls `function parallel` while the other task is executing inside. That needed further consideration.

The second approach was to add a special `entry'completion` procedure to be executed after all tasks are released:

```
entry Go is
begin
  Data_Available  := True;
end;

entry function Parallel when Data_Available  and  Parallel'Count = 10 is
begin
  return Data;
end;

when Parallel'Completion  procedure Clean_Up  is
begin
end;
```

The third proposal was to create a special finalize block that could be executed only once and that could modify the state of the object:

```
entry Go is
begin
  Data_Available  := True;
end;
```

```
entry function Parallel when Data_Available and  Parallel'Count = 10 is
begin
  return Data;
finally
  -- can update state
  -- only executed once
end;
```

No decision was eventually made, and the issue was deferred.

2.7 User-defined clocks

The workshop then addressed a proposal [4] from Andy Wellings and Alan Burns to revisit user-defined clocks, something that was considered for Ada 95 but that was not included in the language. Examples exist of a variety of different clocks in embedded systems (for example, a GPS clock), and Ada currently allows for implementations to define other clocks and make them available to applications. Nevertheless, the proposal was to define a root type for time, where all time types derive.

The proposal was updated to reflect not only the advances in the Ada language (particularly in the Object-Oriented model), but also to simplify the model, where applications do not manage the delay queues, which are left to the runtime. There were still some open issues, such as the necessity for user-defined clocks to call back the runtime, for example to handle time discontinuities (jumping forward or back to the past), or the relationship between user-defined and calendar time.

There was some discussion in this issue, but no decision was taken at the workshop.

2.8 Generalising EDF support

The next proposal, also by Andy Wellings and Alan Burns [5], was to integrate into Ada the support to user-defined scheduling, leveraging on the fact that the Preemption Level Control Protocol introduced in Ada 2005 to support EDF can be used with several scheduling algorithms.

For this, a new mechanism would be required that permits to specify a task attribute on which to base dispatching decisions and to also control dispatching within a priority level. It would also be necessary to define new dispatching points, which would include the point where the value of a task's dispatching attribute was changed. Andy also presented some open issues, namely if other scheduling schemes other than EDF are useful, and if there are other resource sharing protocols that could be supported by Ada to allow for a wide range of scheduling schemes. There was some discussion on this issue, but no decision was taken in the workshop.

2.9 Non-Uniform Memory Access architectures

Andy Wellings presented the issue of supporting in Ada the NUMA (Non-Uniform Memory Access) multi-processor architectures. The accompanying paper [6] argues that the programming model should allow for a more visible mapping of the architecture at the programming level. Ada abstracts programmers away from the low-level architecture of the hardware, which although appropriate for SMP, does not permit to use NUMA architectures predictably and efficiently. There were a few comments on this issue, but it was not discussed further in the Workshop.

2.10 Execution-time accounting of interrupts handlers

Two independent proposals were submitted to the workshop on the issue of execution-time accounting of interrupts handlers. Both proposals noted that the current model is implementation defined but the usual approach is to charge the execution time of interrupts into the currently running task, which does not provide for accurate accounting. This is even more important as the introduction of timing events causes programmers to shift code from tasks to this low overhead mechanism, which is accounted as an interrupt.

However, the two approaches differed in the way the actual accounting was done.

In the proposal by Mario Aldea Rivas and Michael González-Harbour [7], the interrupts execution-time is accounted in a global "conceptual" task. An API is also provided for applications to monitor the time executed in interrupts. Therefore, applications not only have a more accurate measure of tasks' execution time but can also measure the time spent in interrupts.

The proposal by Kristoffer Gregertsen and Amund Skavhaug [8] defines a "pseudo" task for each interrupt priority, with execution time accounting done per priority.

After some discussion, consensus was formed that the separation of execution-time accounting was necessary, although that it would be more appropriate that accounting was performed per `Interrupt_ID`.

In a later round of discussion, an issue was raised that a simple solution would be to introduce an implementation advice that tasks should not be charged for the execution time of interrupts. Then, this would become a runtime quality issue. The proposal to introduce such implementation advice was approved with 16 votes in favour and 3 abstentions.

A second vote was made to decide if the workshop would propose a model, in case the implementation supported execution-time monitoring of interrupts. There were 9 votes in favour, 1 opposed and 11 abstentions. It was then decided that both Mario Aldea and Kristoffer Gregertsen would work on an API to be analyzed on the following day.

2.11 Ravenscar and multiprocessor issues

One of the issues which had been closed in the previous day was the integration of multiprocessor in the Ravenscar profile, and whether the profile should specify one particular multiprocessor scheduling model. There was a proposal by José Ruiz [9] for Ravenscar to determine that tasks would be statically allocated to processors, with a Ravenscar partition being a single scheduling domain as sanctioned in the previous session.

Doubts were raised however, particularly regarding whether task migration would be allowed, and whether the scheduling would be local to each processor or global to the domain. In particular, task migration under user control would allow more flexibility and efficiency. However, concerns were voiced that task migration could impact the certification of Ravenscar-based systems.

As for local versus global scheduling, it was argued that Ravenscar would need a local scheduling approach, with one ready queue per processor. This would impact the definition of scheduling domains agreed earlier, since in that definition, scheduling was global within the same domain.

Another issue with global scheduling and task migration was the impact on the locks of protected objects. In this model, all locks would have to be actual, and no longer priority based locks. The static allocation and local scheduling approach would allow for more efficient implementations.

From the discussion, a list of possible models was introduced:

1. Tasks' fixed allocation, affinities specified by the user
2. Tasks' fixed allocation, affinities specified by the user, and allowing task migration
3. Tasks' fixed allocation, affinities specified by the runtime
4. Global scheduling, no fixed allocation

It was also considered that the model could be a mix, with some tasks fixed, while other were scheduled globally.

During the discussion, there were doubts as to whether all models were feasible, and whether the Ravenscar profile should specify a model at all. Multiprocessor scheduling is still fairly open area, with new models and algorithms every day, so maybe Ravenscar should be silent and afford implementations and programmers more flexibility. However, by doing that, different implementations could choose different models, something that the Ada standard tries to avoid.

Considering all of this, consensus was reached that the profile should not specify a model, but the workshop would recommend that Ravenscar is implemented together with option 1 above (tasks' fixed allocation, affinities specified by the user). This could be done through an implementation advice in the standard.

2.12 The Real-Time Framework

The Real-Time Framework had been proposed in the previous workshop, in 2008, and intended to provide a library of real-time utilities which could be used by application developers. In this workshop, Jorge Real proposed the integration of support for mode changes into the framework [10].

The proposal was based on the original implementation made available by Andy Wellings in 2008, which was also updated to work with a new version of the Ada compiler (at the time of the original framework, some Ada 2005 features were still not available in compilers). During the workshop Jorge also made available the updated framework code.

In the proposal, mode management is based on a synchronized interface, thus integrating the model with the object-oriented model of Ada. During the presentation of the mode manager, it was noted that in the implemented model, the mode manager depends on a user defined type (`List_Of_Modes`) and hence it is not independent from the application. It was then considered that a different model should be analysed.

Afterwards the discussion went on to consider whether the real-time utilities framework should be pursued for standardisation (for instance through a secondary standard) to make it more visible. A proposal was put forward to make it available in some form of collaborative platform, and announcing it to the Ada community as a work in progress.

It was decided to continue the work in an informal collaboration, and anyone interested in working on it to contact Jorge Real.

2.13 The Real-Time Transaction Model

The presentation of the real-time transaction model [11] was made by J. Javier Gutiérrez. The goal is to integrate distributed real-time transactions within the Distributed Systems Annex of Ada, allowing for a separation of concerns between the scheduling of both processing nodes and network, and the application code.

There was some discussion on whether an attempt should be made to standardize a real-time distributed model (or make it available as a technical report), such that all implementations would follow the same guidelines. The question was raised whether the integration of distribution and real-time would be useful and worth the effort. The issue has been around for several workshops, and it was generally agreed that this should be further pursued.

The discussion then went into a specific model for the Ravenscar profile. This particular model is not compatible with the profile, but it would be possible to make a compatible version. It was thus decided that a Ravenscar version should be proposed, with the intention to put forward a technical report. If the model was then accepted, it would be built also for full Ada.

3 Conclusions

The Language and Distribution session was mainly devoted to the discussion of the changes and additions to the language. An action list was permanently being built and updated reflecting the outcomes of the discussion, but for most of the issues actual decisions were deferred to the last, concluding, session of the workshop [12], where the final definition of the AIs to produce was completed.

References

[1] Rod White. *Providing Additional Real-Time Capability and Flexibility for Ada 2005*. This issue.

[2] Stephen Michell, Luke Wong, Brad Moore. *Realtime Paradigms Needed Post Ada 2005*. This issue.

[3] Luke Wong, Stephen Michell, Brad Moore. *Named Memory Pool for Ada*. This issue.

[4] Andy Wellings, Alan Burns. *User-Defined Clocks. Is it the right time now?* This issue.

[5] Andy Wellings, Alan Burns. *Generalizing the EDF Scheduling Support in Ada 2005*. This issue.

[6] Andy Wellings, Abdul H. Malik, Neil Audsley, Alan Burns. *Ada and cc-NUMA Architectures. What can be achieved with Ada 2005?* This issue.

[7] Mario Aldea Rivas, Michael González Harbour. *Execution time monitoring and interrupt handlers.* This issue.

[8] Kristoffer Nyborg Gregertsen, Amund Skavhaug. *Execution-time control for interrupt handling.* This issue.

[9] José Ruiz. *Towards a Ravenscar Extension for Multiprocessor Systems.* This issue.

[10] Jorge Real and Alfons Crespo. *Incorporating Operating Modes to an Ada Real-Time Framework.* This issue.

[11] Héctor Pérez Tijero, J. Javier Gutiérrez, Michael González Harbour. *Support for a real-time transactional model in distributed Ada*. This issue.

[12] Stephen Michell, Jorge Real. *Conclusions of the 14th International Real-Time Ada Workshop*. This issue.