Restricted Tasking Models

A. Burns and A.J. Wellings
Real-Time Systems Research Group
Department of Computer Science
University of York, UK

Abstract

High-integrity systems rarely make use of high-level language features such as Ada tasking. In this paper, simple language profiles (of Ada 95 concurrency features) are developed that are appropriate for various levels of integrity. A level-0 model (collection of Ada95 features) defines a minimal language profile and delivers deterministic (non-preemptive) behaviour. Scheduling is undertaken as part of the application and can thus be inspected and verified. Five other models are also presented that give different levels of expressive power. The motivation for this paper is to try and define models that would become de facto standards and that would be directly supported by kernel vendors and other tool suppliers.

1. Introduction

The Ada95 language revision has both increased the complexity of the tasking features and provided the means by which subsets (or profiles) of these features can be defined. To all of the Ada83 features (dynamic task creation, rendezvous, abort) has been added protected objects, ATC (asynchronous transfer of control), task attributes, finalisation, requeue, dynamic priorities and various low-level synchronisation mechanisms. Subsets are facilitated by pragma `Restrictions` that allows various aspects of the language to be limited in scope or removed from the programmer completely. The purpose of this paper is to define a number of language models (sets of language features) that form natural abstractions, and to determine the extent to which pragma `Restrictions` allows these models to be articulated.

Whilst the full language produces an extensive collection of programming aids, from which higher-level abstractions can be constructed [2], there are a number of motivations for defining restricted models:

- increasing efficiency by removing features with high overheads
- reduce non-determinancy for safety-critical applications
- simplify run-time kernel for high-integrity applications
- remove features that lack a formal underpinning
- remove features that inhibit effective timing analysis

Of course the necessary restrictions are not confined to the tasking model – but this paper only considers concurrency.

The different motivations for producing language subsets, plus the number of different features supported by the language could, theoretically, give rise to a large number of models. Fortunately, many of the motivations lead to similar language models, and certain features are naturally coupled. Hence it is possible to define a relatively small set of models. It is also important to distinguish between those models that lead to simplification of the kernel and those that merely reduce the semantic complexity of features and programs.

If the Real-Time Workshop could endorse such a set then this would have a major impact on vendors and could lead to the development of tailored kernels. Indeed, the experiences of vendors in implementing the full tasking model will be invaluable in defining the different models.

Whilst some models will be strict subsets of others, it is not appropriate to see all the models as fitting into a strict hierarchy. Nevertheless, this paper is organised so that the simplest models are presented first. In all six models are presented:

- Level-0 Model
- Time Triggered Model
- Event Triggered Model
- Synchronous Communication Model
- Full Language Model

- Two-Level Model

The aim of this paper is, however, to initiate a discussion at at the workshop.

2. Level-0 Model

Class A (or Class 1) software (as defined in safety standards such as DO-178B [4]) typically has a very restricted architecture. We shall assume that only periodic behaviours need to be supported. In order to reduce non-determinism and to increase the effectiveness of testing, non-preemptive, non-interruptible execution is required. Although non-preemption can reduce schedulability, it can be analysed and there are ways of improving its effectiveness.

The basic Level-0 model requires there to be a fixed set of tasks; when each task completes its non-preemptive execution, it suspends itself and thereby allows an application-level scheduling task to execute. This task picks out the next application task to run and resumes it. If no tasks are ready to execute, it busy-waits reading the system clock.

Although this is a very static approach, it has a number of advantages over the conventional use of a cyclic executive. The primary advantage is that it allows unrelated iteration rates for the tasks to be supported. It also provides a simple means of increasing schedulability by moving to cooperative scheduling (see section 2.2).

2.1. Language Features Employed

The Level-0 model uses the following features:

- library-level non-hierarchical tasks
- a static number of tasks
- the real-time clock defined in the Real Time Systems Annex – type `Time_Span` and `Clock` function
- pragma `Volatile` to ensure that shared data is used correctly
- necessary restrictions on sequential code to enable the prediction of worst-case execution times to be made (this is a general timing issue and will not be discussed further here)

It follows that protected objects, rendezvous, select statements, abort and ATC statements, delay and delay until statements and interrupts are not included in this profile.

Note that this very minimal set of features are outside the remit of the Real Time Systems Annex. However the added restrictions introduced by the Safety and Security Annex do give the necessary coverage:

```
No_Task_Hierarchy
No_Nested_Finalization
No_Abort_Statements
No_Termination_Alternatives
No_Task_Allocators
No_Implicit_Heap_Allocation
No_Dynamic_Priorities

Max_Select_Alternatives = 0
Max_Task_Entries = 0
Max_Protected_Entries = 0
Max_Storage_At_Blocking = ...
                  -- (some appropriate value)
Max_Asynchronous_Select_Nesting = 0
Max_Tasks = ... -- (a statically known value)

No_Protected_Types
No_Delay
```

Necessary scheduling is undertaken as part of the application code. Hence the run-time support needed is minimal: that necessary to support simple tasks (threads) and suspension. In particular, the ability for a task to suspend itself and for the application's scheduler (task) to resume others. This functionality could be obtained through the use of the following features defined in the Real-Time Systems Annex: `Asynchronous_Task_Control` or `Synchronous_Task_Control`. For example with `Asynchronous_Task_Control`, the following structure could be used. First some global arrays:

```
with Ada.Task_Identification;
use Ada.Task_Identification;
with Ada.Real_Time; use Ada.Real_Time;
package Scheduling_Data is
  N : constant := ... -- static number of tasks
  type Application_Ids is range 1..N;
  Next_Release : array(Application_Ids) of Time;
    -- holds time of next release of each task
  pragma Volatile_Components(Next_Release);
  Clients : array(Application_Ids) of Task_Id;
  pragma Volatile_Components(Clients);
end Scheduling_Data;
```

Each application task would take the following form:

```
with Scheduling_Data;
package Activity_One is
  pragma Elaborate_Body(Activity_One);
  Name : Scheduling_Data.Application_Ids := 2;
                              -- for example
end Activity_One;

with System;
with Ada.Real_Time; use Ada.Real_Time;
with Ada.Task_Identification;
use Ada.Task_Identification;
with Ada.Asynchronous_Task_Control;
use Ada.Asynchronous_Task_Control;
package body Activity_One is
  use Scheduling_Data;
  task The_Task is
    pragma Priority(System.Default_Priority);
  end The_Task;
  task body The_Task is
    Period : Time_Span := Milliseconds(30);
    Self : Task_Id;
  begin
    Self := Current_Task;
    Clients(Name) := Self;
    Next_Release(Name) := Clock + Period;
    loop
      Hold(Self);
      -- code of the task
      Next_Release(Name) := Next_Release(Name)
                          + Period;
    end loop;
  end The_Task;

end Activity_One;
```

Obviously each task must be given a unique place in the Next_Release array. The language's notion of priority is not used for scheduling the application tasks and so they can all be given the same priority. Indeed, at most one such task will ever be able to execute at any specific time.

The scheduler would therefore have the following form:

```
with System;
package Scheduler is
  task Sched is
    pragma Priority(System.Default_Priority-1);
      -- priority less than
      -- the application tasks
  end Sched;
end Scheduler;

with Scheduling_Data; use Scheduling_Data;
with Ada.Real_Time; use Ada.Real_Time;
with Ada.Task_Identification;
use Ada.Task_Identification;
with Ada.Asynchronous_Task_Control;
use Ada.Asynchronous_Task_Control;
package body Scheduler is
  task body Sched is
    Now : Time;
  begin
    loop
      Now := Clock;
      for Id in Application_Ids loop
        if Next_Release(Id) <= Now then
          Continue(Clients(Id));
          exit;
        end if;
      end loop;
    end loop;
  end Sched;
end Scheduler;
```

With no further kernel support, simple sporadic (event triggered) tasks can be accommodated. To protect against task overrun, an interface to a watchdog (interval) timer may be needed.

2.2. Cooperative Scheduling

One of the drawbacks to non-preemptive scheduling is the resulting reduction in schedulability. If a low priority task has a long execution then higher priority tasks will be blocked for this time before they can even start execution. With the level-0 model this problem can easily be addressed. If during the execution of its code a task does a speculative 'hold' i.e.

```
loop
  -- part A of the task
  Hold(Self);
  -- part B of the task
  Hold(Self);
  -- part C of the task
  Hold(Self);
  -- part D of the task
  Next_Release(Name) := Next_Release(Name)
                      + Period;
  Hold(Self);
end loop;
```

then the scheduler will be invoked at each call of Hold. If a higher priority task should now run (as its 'time' is due) then a task switch will occur. Alternatively, if this is not the case then the scheduler will resume the same task (as its Next_Release time is still in the past). The addition of 'hold' points can be done after the code has been written and will not undermine the integrity of the code (this can be compared with the process of splitting code up into small procedures so that they fit into a cyclic executive).

Data shared between non-preempted tasks does not need to be given mutual exclusion protection. Care must be taken with cooperative scheduling to ensure that a 'hold' call is not made during an atomic sequence.

2.3. Timing Analysis

This form of non-preemption and cooperative scheduling can be analysed using standard response time analysis [1]. All kernel (run-time) overheads must, of course, be accounted for. This includes the overhead involved in executing the 'hold' operation (even when no task switch occurs). An overview of the form of timing analysis available to Level-0 systems is given in an Appendix.

3. Time Triggered Model

Although the use of cooperative scheduling can reduce the impact of blocking, for some applications (with requirements for very responsive behaviour) preemption is needed. Preemption also protects a task from the over-run of lower priority tasks (without the need for a watch dog timer). Systems that require tasks with different levels of integrity to execute on the same processor will also need preemption. The key to this model is that all activities are periodic (i.e. triggered by the real-time clock).

Preemptive behaviour could be programmed with a more extensive use of the suspension facilities and interrupt handlers, but this would now be duplicating what the Real-Time

Systems Annex defines as ‘standard’ behaviour. Hence, it would seem more appropriate to specify this model in terms of the allowable restrictions defined in the Annex.

3.1. Language Features Required

Using the restricted tasking provision, defined in the Real-Time Systems Annex, the following language model can be specified:

```
No_Task_Hierarchy
No_Nested_Finalization
No_Abort_Statements
No_Termination_Alternatives
No_Task_Allocators
No_Implicit_Heap_Allocation
No_Dynamic_Priorities
No_Asynchronous_Control

Max_Select_Alternatives = 0
Max_Task_Entries = 0
Max_Protected_Entries = 0
Max_Storage_At_Blocking = ...
                  -- (some appropriate value)
Max_Asynchronous_Select_Nesting = 0
Max_Tasks = ... -- (a statically known value)
```

Application tasks use ‘delay until’ to program periodic activity and protected objects for communication (that is, no rendezvous). Protected entries are not used.

3.2. Timing Analysis

Again standard timing analysis for preemptive priority based scheduling (with ceiling priorities for protected objects) is applicable to this model. The kernel operations for manipulating the notional delay queue need to be modeled and then integrated into the analysis of the application tasks. Such integration has been undertaken for Ada kernels before.

4. Event Triggered Model

The alternative to time triggered activities is to see all tasks as being event triggered. To support this requires protected objects with entries. It is also convenient to allow interrupts in this model but to remove the clocks. Hence the `No_Delay` restriction is added and the calendar packages are omitted. Also `Max_Protected_Entries` needs to changed from the restrictions articulated in the previous section, but it can still be given a static value [1].

A natural extension to the event triggered model is to combine it with the time triggered one to give a standard fixed priority scheduling model.

[1]It should be noted that pragma `Restrictions` cannot explicitly remove the use of requeue. Once `Max_Protected_Entries` has a non-zero value then requeue would be allowed; however, the event triggered model described here does not require requeue.

4.1. Timing Analysis

Event triggered systems can be analysed as long as there is a bound, within any time interval, on the number of events (from each possible source) that can occur. This bound usually takes the form of a minimum inter-arrival interval, but it can also be expressed as an actual bound (say, 3 in any 25ms).

5. Synchronous Communication Model

Formal models of concurrency (e.g. CSP and CCS) often require a synchronous communication model. The facilities provided by Ada83 allowed such a model to be defined. A synchronous communication model would have the following restrictions:

```
No_Task_Hierarchy
No_Nested_Finalization
No_Abort_Statements
No_Task_Allocators
No_Implicit_Heap_Allocation
No_Dynamic_Priorities
No_Asynchronous_Control

Max_Select_Alternatives = ...
    -- (a statically known value)
Max_Task_Entries = ...
    -- (a statically known value)
Max_Protected_Entries = ...
    -- (a statically known value)
Max_Storage_At_Blocking = ...
    -- (some appropriate value)
Max_Asynchronous_Select_Nesting = 0
Max_Tasks = ...
    -- (a statically known value)
```

Protected objects are included as they allow a certain class of passive process to be implemented more efficiently (than using a task).

5.1. Timing Analysis

Although formal analysis of such models is easier, timing analysis is made more complex by the synchronous interactions. The temporal behaviour of one task is now coupled to the behaviour of all tasks it communicates with (and the tasks they communicate with etc). A general scheduling approach (and timing method) does not really exist for this approach, although some recent attempts, with a restricted model, have been published[3].

6. Full Language Model

The inclusion of task hierarchies, ATC, abort and requeue add significantly to the kernel complexity and overheads. It would seem appropriate therefore to include them only with the model that supports all the tasking and Real-Time Systems Annex features.

7. Two-Level Model

The 'full model' implies the use of a set of tasks that share a common address space and are scheduled in a single uniform way. Beyond this model there is a need to support a separation between groups of tasks, and not to require these groups to work to the same scheduling policy. One motivation for this would be the support of different levels of software integrity on the same hardware resource.

Ada's partition facility is perhaps the only way of providing this structuring. A partition would then have its own language model, but the scheduling of the partitions themselves would need to be addressed. So, for example, a three partition system (running on the same processor) could be time-sliced with one partition supporting a level-0 application, one time triggered partition, and the other partition supporting a full language model. The implementation would, of course, need to give protection between the partitions (which is not strictly required by the language definition).

8. Conclusion

A useful activity for the Real-Time Ada Workshop is to define 'standard' language profiles that would subsequently lead to run-time systems being developed that give direct and effective support to these profiles. This paper has attempted to initiate such an activity by presenting a number of natural language models.

Acknowledgements

The authors would like to thanks Offer Pazy for useful comments on an earlier draft.

Appendix: Analysis for Level-0 Systems

There are a number of ways of analysing a task set that is scheduled according to the Level-0 approach. For simple schemes with a small number of tasks, it may be possible to layout a time-line for the system and inspect its behaviour. For more complex systems, different forms of analysis are necessary.

To give an example of the analysis that is available, consider a system in which all tasks are initially released at the same time; fortunately a system that meets its timing requirements when a critical instant occurs will always meet its requirements even when releases are phased [6]. For task i we have

$$R_i \;=\; C_i \;+\; B_i \;+\; I_i$$

Where R_i is the worst-case response time, C_i is the task's worst-case execution time, B_i is the worst-case blocking time (that is, the time that a lower priority task could be executing after the release of task i), and I_i is the interference that the task suffers from higher priority tasks.

Three other parameters are needed to give a complete description of the temporal behaviour of the system:

C_{hold} – the time needed to switch back to the scheduler task when a task executes `Hold`.

C_{resume} – the time needed to resume an application task when the scheduler calls `Continue`.

S_i – the computation time of the scheduler task, this can be parameterised by the task that will be released (as the higher the priority of the task, the shorter the time spent in the scheduler loop).

The easiest way of accounting for these system overheads is to allocate them to the task that causes them, i.e.

$$C_i \;:=\; C_i \;+\; m(C_{hold} \;+\; C_{resume} \;+\; S_i)$$

where m is the number of calls to 'hold' – if a pure non-preemptive scheme is used then m is 1.

The worst-case blocking time happens when the release of a task occurs just after the scheduler accessed the clock. The scheduler then finds and executes the longest non-preemptive section of a lower priority task:

$$B_i = \max_{j \in lp(i)} (\hat{C}_j)$$

where $lp(i)$ is the set of tasks with lower priority than task i, and $\hat{C}_j$ is the longest non-preemptive section of task j (including the associated overheads).

The interference time actually depends upon the response time of the task (if it is long enough, a high priority task may execute more than once before the task i can start and hence complete). Thus

$$I_i = \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j$$

where $hp(i)$ is the set of tasks with higher priority than task i.

The full equation is thus

$$R_i \;=\; C_i \;+\; \max_{k \in lp(i)} (C_k) \;+\; \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j \qquad (1)$$

This can be solved by forming a recurrence relation. The smallest non-zero value of R_i that satisfies equation(1) is the worst-case response time of the task. Define the sequence of values w_i^n as follows:

$$\begin{aligned} w_i^0 &= C_i \\ w_i^{n+1} &= C_i + \max_{k \in lp(i)} (C_k) + \sum_{j \in hp(i)} \left\lceil \frac{w_i^n}{T_j} \right\rceil C_j \end{aligned}$$

The sequence is clearly monotonically non-decreasing. If $w_i^{n+1} = w_i^n$ then this value represents the smallest non-negative solution to equation(1). That is $R_i = w_i^n$. Alternatively, if $w_i^{n+1} > D_i$ for some n then the task's worst-case response time is beyond its deadline and the task must be deemed unschedulable.

An improved formulation

An improved formulation comes from noting that, because of non-preemption, the amount of interference included in equation (1) can be reduced; once a task is executing then it will not be preempted by the release of a higher priority task unless it offers to suspend. In a pure non-preemption model, it is easy to calculate the worst-case time ($\tilde{R}_i$) for the commencement of the task's execution:

$$\tilde{R}_i = \Delta + \max_{k \in lp(i)} (C_k) + \sum_{j \in hp(i)} \left\lceil \frac{\tilde{R}_i}{T_j} \right\rceil C_j$$

where Δ is an arbitrary small value needed to ensure that the task has actually started.

This equation can again be solved by forming a recurrence relation. The true response time is then easily obtained:

$$R_i = \tilde{R}_i + C_i$$

With cooperative scheduling, let $\tilde{C}_i$ be defined to be the size of the task's last non-preemptive section. All that now needs to be guaranteed by the scheduling analysis is $(C_i - \tilde{C}_i + \Delta)$. The general analysis equations become:

$$\begin{aligned} \tilde{R}_i &= C_i - \tilde{C}_i + \Delta + \max_{k \in lp(k)} (\hat{C}_k) + \sum_{j \in hp(i)} \left\lceil \frac{\tilde{R}_i}{T_j} \right\rceil C_j \\ R_i &= \tilde{R}_i + \tilde{C}_i \end{aligned}$$

Although in general non-preemption reduces schedulability, it is possible to define task sets that can only be scheduled by the cooperative approach. It has been shown [5], theoretically, that when tasks have their priority raised during execution they become more schedulable. Making the last stage of a task's execution non-preemptive has the effect of raising its priority and hence increased schedulability may result.

References

[1] A. Burns. Preemptive priority based scheduling: An appropriate engineering approach. In S. Son, editor, *Advances in Real-Time Systems*, pages 225–248. Prentice-Hall, 1994.

[2] A. Burns and A. Wellings. Ada 95: An effective concurrent programming language. In A. Strohmeier, editor, *Proceedings of Reliable Software Technologies - Ada-Europe '96*, pages 58–77. Springer-Verlag Lecture Notes in Computer Science, Vol 1088, 1996.

[3] A. Burns and A. Wellings. Synchronous sessions and fixed priority scheduling. *Journal of Systens Architecture (to appear)*, 1997.

[4] *Software Considerations in Airborne Systems and Equipment Certification DO-178B/ED-12B.* RTCA, December 1992.

[5] M. Harbour, M. Klein, and J. Lehoczky. Fixed priority scheduling of periodic tasks with varying execution priority. In *Proceedings 12th IEEE Real-Time Systems Symposium*, 1991.

[6] C. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *JACM*, 20(1):46–61, 1973.