A Formal Model of the Ada Ravenscar Tasking Profile; Delay Until[*]

Kristina Lundqvist, Lars Asplund
Uppsala University, Information Technology
Dept. of Computer Systems
P.O. Box 325, S-751 05 Uppsala, Sweden
Kristina.Lundqvist@docs.uu.se, Lars.Asplund@docs.uu.se

April 30, 1999

Abstract

The definition of the Ravenscar Tasking Profile for Ada95 provides a definition of a tasking runtime system with deterministic behaviour and low enough complexity to permit a formal description and verification of the model. A complete run-time system is being modeled using the real-time model checker UPPAAL, and this work describes the handling of *delay until*. Since scheduling is not yet modelled a simple non-preemptive scheduler has been used when verifying the delay queue.

Keywords: Ada Tasking, Delay Queue, Ravenscar, Formal Methods, Run-Time System, UPPAAL, Timed Automata.

1 Introduction

High Integrity systems traditionally do not make use of high-level language features such as concurrency. This is in spite of the fact that such systems are inherently concurrent. The traditional approach has been to declare concurrency to be a system issue, and to develop alternative methods, such as cyclic executive approaches to solve concurrency issues. The net result has been the creation of non-cohesive systems that are very difficult to analyse, let alone attempt any formal proofs of correctness.

The static properties of the Ada language, the strong typing facilities, and the relatively few occurances of implementation-dependent and unspecified behaviour can help in the development of provably correct programs in Ada. Indeed the existence of SPARK [Bar97], AVA [Smi92] and Penelope [GMP90] for

[*] This work is funded by Swedish National Board for Industrial and Technical Development, Swedish Nuclear Power Inspection, and Swedish Defence Material Administration

Ada83 showed that this was possible, although the proof technologies and Ada83 itself made the provable subset somewhat limited. It is also common belief that these proof methodologies could be extended to Ada tasks if a sufficiently predictable and precise tasking subset was specified.

Ada95 introduce a number of new tasking-related capabilities that improve the specification of concurrent behaviour, and give explicit permission to restrict language features to improve performance, predictability or correctness. At the 8th International Real Time Ada Workshop, a concurrency model was developed called the Ravenscar Tasking Model [DB98] [WB97] which could make the verification of tasks in Ada a reality.

2 The Ravenscar Profile

The Ravenscar model eliminates nearly all of the Ada tasking constructs that contain implementation dependencies and unspecified behaviour. The specification of the Ravenscar tasking model significantly advances the way of proving concurrent programs in that one can also prove that the tasking runtime system correctly implements the model.

To date there has been one implementation, Raven [DB98], of the Ravenscar model. This implementation can be certified in conjunction with an application program, but there has not yet been any attempts to formally analyse either the kernel itself, or an application using this kernel. In Raven some emphasis is put on suspension objects, which fits very well into the Ravenscar model.

There has been some previous work done to formally verify different run-time systems e.g. [Tol95] [Hut94]. The development process described in [FW97] attempts to capture all of the temporal properties of a preemptive kernel, and show how an implementation of them can be developed. The formal development of the kernel is done for a simple real-time operating system, designed to support a restricted Ada 95 tasking model. The kernel is specified using the logic of the Prototype Verification System (PVS) [CO$^+$95], with the temporal properties of the system expressed using Real-Time Logic (RTL) [JM86] embedded in PVS. The work is based on a previous work on specifying real-time kernels [FW96], the used subset of Ada is similar to the Ravenscar profile.

2.1 Ravenscar Limitations

One of the major capabilities that the Ravenscar model adds to high integrity concurrent systems is a deterministic intertask communication and synchronisation capability. This capability is provided by a “limited” form of Protected Objects (PO). The Ravenscar Profile limits Protected Objects and tasking in several ways that dramatically simplify PO behaviour, and in conjunction with restrictions on tasks, virtually all of the non-deterministic behaviour in a Ada Run Time System (RTS) are eliminated.

Besides the two main features, limited forms of tasks and POs, in the Ravenscar Profile, there are the *delay until*, support for user defined interrupts and a preemptive scheduler using a priority ceiling protocol as per Annex D (Real-Time Systems) [ARM95].

2.2 A Ravenscar-compliant Ada Run Time System

The restrictions imposed by the Ravenscar profile on an Ada Run Time System, gives a simplified kernel. The computational model that the kernel supports is made up of a fixed set of concurrently executing tasks, which can communicate asynchronously by means of Protected Objects. Each task should be structured as an infinite loop within which is a single invocation event. This is either a call to *delay until* or a call to a protected entry. Periodic tasks can be implemented by using the *delay until*, allowing a task to suspend itself until a fixed point in the future.

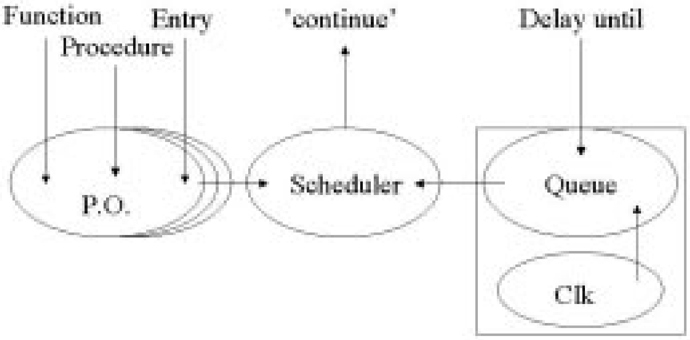


Figure 1: Control-flows between Protected Objects, the Delay Queue and the Scheduler.

There are five basic kernel operations, corresponding to the four operations that the application can invoke (i.e. call to a protected procedure, protected entry, protected function, or a delay until operation). The fifth kernel operation is response to external interrupts. Figure 1 shows some of the control flows between Protected Objects, the delay queue and the scheduler. For all four application operations there is a decision taken by the scheduler which task to run, indicated as 'continue' in figure 1. The External Interrupts has not yet been modelled and are not included in figure 1. The three calls to the PO, and a formal model of a PO is described in [LAM99], and the *delay until* is described in this paper.

2.3 The Mana-project

The Mana-project is aimed at developing and modeling a complete Ravenscar-compliant Ada Run-Time System (RTS) using formal development methodologies. The tool used for proving the correctness of the RTS by formal description and formal verification is UPPAAL [LPY97]. UPPAAL is still under development, and the Mana-project is working close with its development team.

One important factor for choosing UPPAAL is that UPPAAL is based on the theory of Timed Automata (TA), and these timed automata allow lower and upper time limits to be specified for transitions in the state machines. This feature allow for modelling a complete systems with not only its state transitions but also the execution times in the application tasks. The lower limit can be taken from the Best Case Execution Time (BCET) and the upper limit from the Worst Case Execution Time (WCET) of the tasks.

Out of the basic kernel operations, a previous paper [LAM99] described and analysed the Protected Object. This paper describes a model of the delay queue. The model is complete, but in order to do the verification there is a need for a scheduler. The design of a fully prioritized scheduler that implements the priority ceiling and inheritance protocol is planned to follow this work. The tests in this paper are therefore performed with a very simple non-preemptive model of the scheduler.

3 The UPPAAL Tool Box

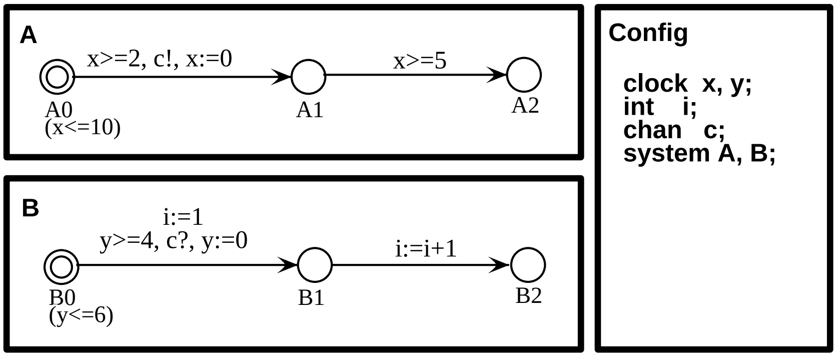
UPPAAL is a tool box for modelling [LPY97], simulation and verification of real-time systems, based on constraint-solving and on-the-fly techniques, developed jointly by Uppsala University and Aalborg University. It is appropriate for systems that can be modelled as a collection of non-deterministic processes with finite control structure and real-valued clocks, communicating through channels and (or) shared variables.

UPPAAL consists of three main parts: a description language, a simulator, and a model-checker. The description language is a non-deterministic guarded command language with data types. It serves as a modelling or design language to describe system behaviour as networks of timed automata extended with data variables. The simulator and the model-checker are designed for interactive and automated analysis of system behaviour by manipulating and solving constraints that represent the state-space of a system description. The simulator enables examination of possible dynamic executions of a system during early modelling or design stages and thus provides an inexpensive mean of fault detection prior to verification by the model-checker which covers the exhaustive dynamic behaviour of the system.

There has been some previous work concerning Ada and UPPAAL [Bjo95], where the relation between the Ada tasking model and the formal model timed automata is given, and also some guidelines for translating Ada tasking constructs into timed automata and vice versa.

3.1 Timed Automata and Uppaal

A timed automaton (TA) [AD90] is a finite automaton [HU79] extended with time by adding real-valued clocks to each automaton, and for each transition adding guards as enabling conditions. A system model in Uppaal consists of a collection of timed automata modelling the finite control structures of the system. In addition the model uses a finite set of clocks and integer variables.



```
clock  x, y;
int    i;
chan   c;
system A, B;
```

Figure 2: An example Uppaal model.

Consider the model in figure 2. The model consists of 2 components, A and B, with control nodes A1, A2, A3 and B1, B2, B3, and uses two clocks `x, y`, and one channel `c`. The edges of the automata are decorated with three types of optional labels;

- A guard, expressing a condition on the values of clocks and integer variables that must be satisfied in order for the edge to be taken. For example, the edge between A0 and A1 can only be taken when the value of the clock `x` is greater than or equals 2.

- A synchronisation action, which is performed when the edge is taken. In figure 2, the two processes may communicate via the channel `c`. When the edge is taken the action `c!` is performed thus insisting in synchronisation with B on the complementary action `c?`. That is for A to take the edge between A0 and A1, automaton B must simultaneously be able to take the edge from B0 to B1.

- A number of clock resets and assignments to integer variables. In figure 2, the clock `x` is reset to 0 when the transition between A0 and A1 is taken, similarly the integer variable `i` is increased by one whenever the transitions between nodes B1 and B2 is taken.

In addition to this, the control nodes may be decorated with invariants. Invariants are conditions expressing constraints on clock values in order for control to remain in a particular node. For example, in figure 2, control can only remain in A0 as long as the value of x is less than 10. The initial state of an automaton is indicated graphically by a double circled node.

To prevent a network of automata from delaying in a situation where two components are already able to synchronise, a channel may be declared as being *urgent*, i.e. as soon as it can be taken, no further delay is allowed before communication takes place.

In order to model atomicity of a transition sequence in an automaton, i.e. without time delay or interleaving of transitions in other automata, a state can be marked as being a *committed* state[1]. For example the atomicity of the action sequence Resume?Suspend?Q! in the task automaton, figure 6, is achieved by insisting that the committed locations in between must be left in zero time.

3.2 New Features in Uppaal98

There has been a number of improvements in Uppaal98 compared to the version of Uppaal used in our previous work [LAM99]. New features used in this paper are an array of integers (DQ), and a template automaton (T). Through instantiating a template automaton it is possible to get a variable number of automata. Finally there has been a major revision of the graphical interface, which makes the job of modelling complex systems easier.

4 Formalisation of a Delay Queue in Uppaal

We start with an overview of the components and their interaction via channels and shared variables. The modelling of the delay queue and its environment is separated into 4 different parts. The model of the queue (Queue), the system clock (Clock), a simplified model of a scheduler (Scheduler) and the tasks in the system (T). Figure 3 shows a flow-graph of the resulting model. Nodes represent automata, edges represent synchronisation channels, and clocks are shown in italics.

In the following subsections we give a detailed description of the clocks, variables and automata of the system.

4.1 Clocks

Two different clocks are used to model the system clock and the WCET for the task.

Clk Models the system time. Once every *Time_Unit* (i.e. when Clk==1) the value of variable Time will be increased by one, figure 5.

Tau Every task, figure 6, has its own Tau-clock which is used to model that task's WCET.

[1] A committed state is graphically indicated by a dot in the middle of the node.

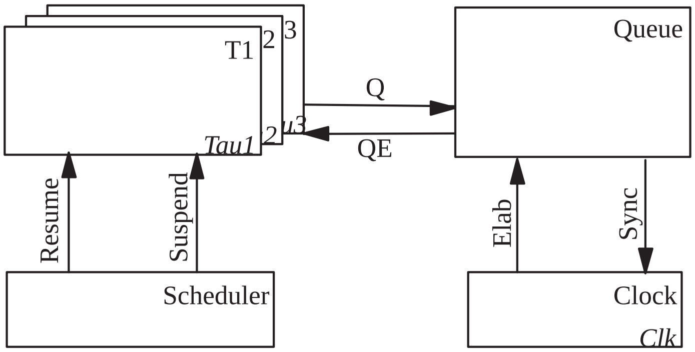


Figure 3: A Flow-Graph of the Delay Queue and its Environment.

4.2 Variables and Arrays

`DUntil` A task is blocked from further execution until the specified expiration time (`DUntil`) is reached.

`DQ` The array used to model the delay queue. Every task in the system has an unique place in this array where its *delay until* time is stored.

`Go` A variable used only to make sure that a task does not start executing before all system elaborations are done.

`i` Index number in the array `DQ`.

`j` When searching through the array `DQ`, this variable is used to keep track of the task with the shortest/closest *delay until* time.

`len` How many tasks are at the moment queued in the delay queue.

`max` A variable used to make sure that the array index never exceeds the array size. `max` has the value of one less than the maximum number of tasks in the system.

`Pid` Id of the queued task with the shortest *delay until* time.

`PidNo` Every task has a unique number (Process id number).

`PE` Variable used to guarantee that the correct task is released when a *delay until* time is reached.

`t` The shortest *delay until* time of all tasks at the moment queued.

`Time` The value of `Time` t represents the real time interval that start with t×Time_Unit and ends with $(t+1)$×Time_Unit. Time_Unit is the time taken by the clock `Clk` to get the value of 1.

4.3 System Model

To model and test the delay queue a system with four different automata is needed.

Scheduler The automaton for the simple non-preemptive scheduler, figure 4 has two channels `Resume` and `Suspend`. Both channels are urgent channels, which means that as soon as the processor is free, and there is a task ready to execute, the task will synchronise with the Scheduler on channel `Resume`. Only one task at the moment can execute. When a task has finished executing, it synchronises on channel `Suspend`.

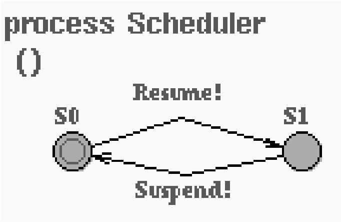


Figure 4: The Scheduler Automaton.

Clock The Clock automaton, figure 5, models the system clock and the variable `Time`, and also makes a number of initialisations/elaborations. The state `Stop` is only needed to make verifications stop, and has been chosen to be at `Time`==150.

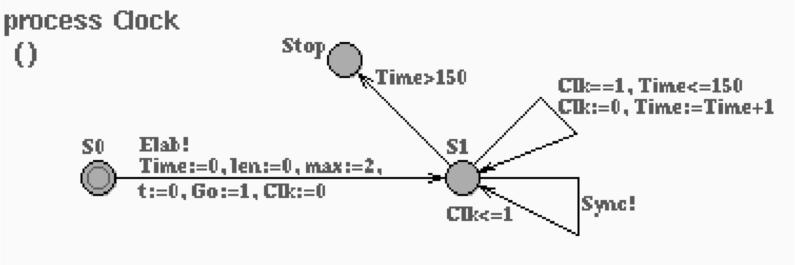


Figure 5: The Clock Automaton.

Task A suspended task will be suspended at node `S3`, figure 6. When the task is removed from the delay queue it gets suspended at node `S4` until the processor is free and the task will synchronise on channel `Resume` and stay at node `S5` for the time corresponding to that tasks WCET. As soon as the task has finished executing, it synchronises on `Sync`, calculates a new *delay until* time, gets queued, and suspends at node `S3`. To get an automaton the template automaton in figure 6, should be instantiated as e.g. T1 := T(Delay1, PidNo1, Temp1, 28, 1, 8, Tau1), which gives a task with id number 1, *delay until* 28, and a WCET of 8.

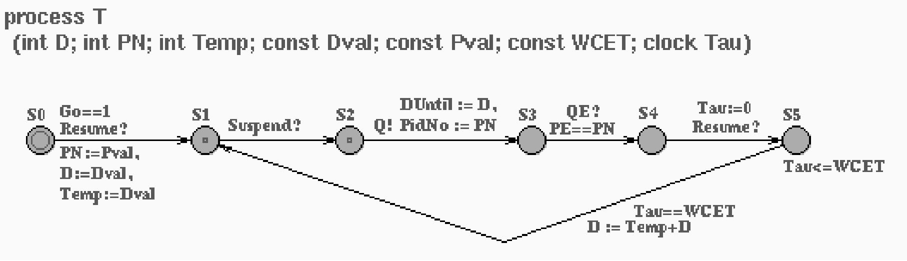


Figure 6: The Task Automaton.

Queue The `Queue` automaton in figure 7 can be partitioned into two parts. Nodes `S2` to `S4` handling queueing of tasks, and nodes `S5` to `S10` handling the releasing of tasks.

After elaboration, synchronising on channel `Elab` the automaton is waiting at node `S1` until a task makes *delay until.* The first task to make *delay until* will synchronise with `Q` (node Queue.S2) and set `t` to its *delay until* time, set `Pid` to its id number, and return to node `S1`. The next task to make a *delay until* will also synchronise with `Q`, this time ending up in state Queue.S4. If the new tasks *delay until* time is smaller than the active time (`t`), the active task will be queued, and the variables `t` and `Pid` will be assigned the new tasks `DUntil` and `PidNo`. If the new tasks *delay until* time is later than the active time, this new task will be queued and we return to state `S1` again.

As soon as the *delay until* of the active task agrees with the variable `Time` that task should be removed from the queue and the queued task with the shortest delay time should be found. If the active task is the only one in the queue the task is at node `S5` removed from the queue, and synchronisation with the task on channel `QE` takes place. If there are tasks queued, the active task will be removed, synchronise on `QE` at state `S6` and there after going through the array in order to find the task with the shortest delay time. When found, the variables `t` and `Pid` are instantiated and we are back at state `S1`. If there are more than one task with the same *delay until* time, they will all be removed from the queue at the same time (since all states `S5` to `S10` are *committed* states).

5 Validation and Verification of the Model

In this section we formalise informal requirements to test the modelled queue, and prove correctness using the symbolic model-checker of Uppaal.

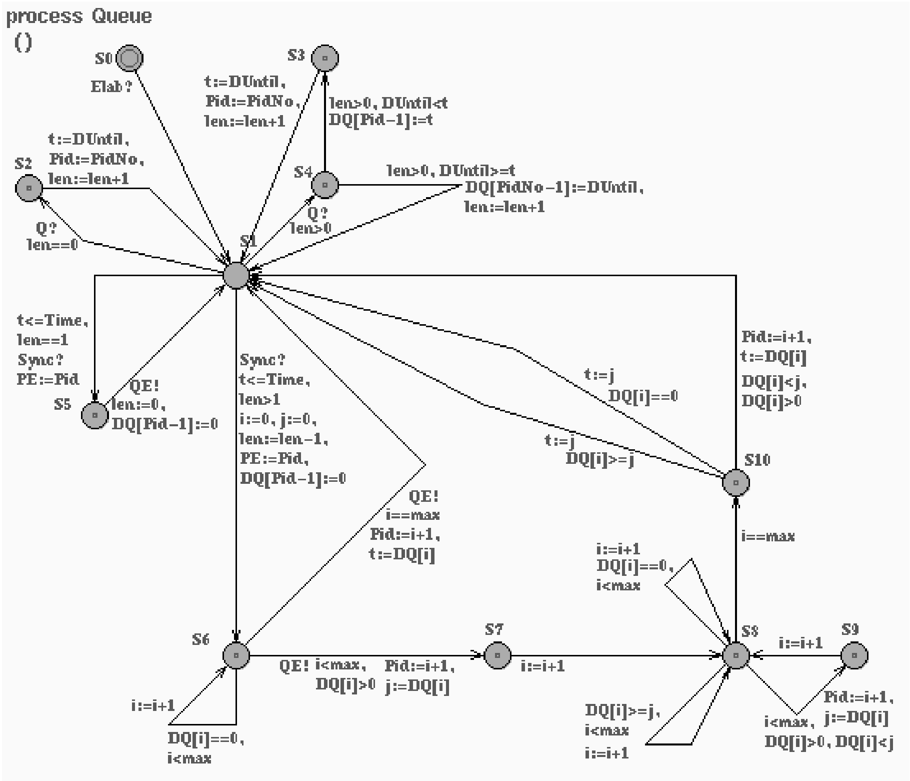


Figure 7: The Queue Automaton.

Uppaal is able to check for reachability properties [LPY97], in particular whether certain combinations of control-nodes and constraints on clock and data variables are reachable from an initial configuration. The properties that can be analysed are of the forms:

$$\varphi ::= \forall\Box\beta \mid \exists\Diamond\beta$$

$$\beta ::= a \mid \beta_1 \wedge \beta_2 \mid \neg\beta$$

Where a is an atomic formula being either an atomic clock/data constraint or a component location. Atomic clock/data constraints are either integer bounds on individual clock/data variables, or integer bounds on differences of two clock/data variables.

Intuitively, for $\forall\Box\beta$ to be satisfied all reachable states must satisfy β. Dually, for $\exists\Diamond\beta$ to be satisfied some reachable state must satisfy β.

The first property to show is that when a task `T`,figure 6, reaches node `S4` (i.e. when the task is removed from the queue and ready for rescheduling), the variable `Time` will always be larger than what the tasks *delay until* time was. This is formulated by the following three logical statements.

```
A[] (T1.S4 imply ((Time > Delay1) or (Time == Delay1)))
A[] (T2.S4 imply ((Time > Delay2) or (Time == Delay2)))
A[] (T3.S4 imply ((Time > Delay3) or (Time == Delay3)))
```

The second property is that never more than one task can execute at the same time, i.e. at most one of the tasks can be at state T.S5 at the same time. This is stated as follows;

```
A[] (T1.S5 imply not(T2.S5 or T3.S5))
A[] (T2.S5 imply not(T1.S5 or T3.S5))
A[] (T3.S5 imply not(T1.S5 or T2.S5))
```

5.1 Verification Results

The 6 properties mentioned above have all been verified using Uppaal and when the model checker is invoked it for all 6 properties returns

```
checked A[] (T1.S4 imply ((Time > Delay1) or (Time == Delay1)))
Requirement is satisfied
checked A[] (T2.S4 imply ((Time > Delay2) or (Time == Delay2)))
Requirement is satisfied
...
```

That is all six logical statements are satisfied.

5.2 Extended Test of the Model

There is no way of showing properties for a general number of tasks in Uppaal, and indeed not in any other approach based on model checking. We can not make a statement that the delay queue model is valid for any other number of tasks than the three used in the model above. As an experiment we have created a model with 10 tasks. The same kind of tests as above are made, and also this time all properties are satisfied.

Consider the schedule shown in figure 8. It shows three periodic tasks with *delay until* times of 25, 60, 70 and Worst Case Execution Times of 10, 14, and 22. The grey areas show where the tasks actually get to execute on the processor, the "white boxes" show where the tasks actually would have executed if the tasks were alone on the processor.

The shown schedule is the expected schedule for a perfect non-preemption scheduler, that follows the *delay until* times and the execution times. We intend to verify for all time steps that our model follows this expected schedule, but have done tests on only 6 different times (`Time`==45, 55, 72, 82, 90 and 112) where to check if our model manage to follow the given schedule. The letters `A, B, C, D, E, F` show where the tests are made. For example at time 45 (`A`), all three tasks are queued, i.e. in state T.S3. At time 72, task 1 is in the delay queue, task 2 is executing, and task three is ready to run, but waiting for the processor to be free. These tests are formulated as follows;

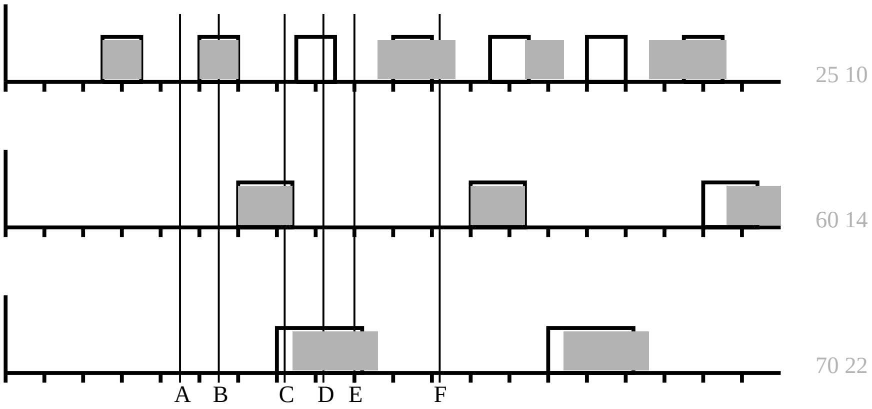


Figure 8: A Schedule for three tasks with *delay until* times 25, 60, 70, and WCET 10, 14, 22. The grey areas show where the tasks actually are executing

```
A[] ((Time==45) imply (T1.S3 and T2.S3 and T3.S3))
A[] ((Time==55) imply (T1.S5 and T2.S3 and T3.S3)))
A[] ((Time==72) imply (T1.S3 and T2.S5 and (T3.S3 or T3.S4)))
A[] ((Time==82) imply ((T1.S3 or T1.S4) and T2.S3 and T3.S5))
A[] ((Time==90) imply (T1.S5 and (T2.S3 or T2.S4) and T3.S3))
A[] ((Time==112) imply (T1.S5 and T2.S3 and T3.S3))
```

All six statements are satisfied.

6 Results and Future Work

To model a full Ada-95 Run-Time System using Formal Methods is not possible today. This and previous ref LAM works show that it is feasable to model the complete run-time system of the Ravenscar subset of Ada-95. In both ref LAM and this work there are no restrictions on the functionality of either the Protected Objects or the Delay Queue. Still the cpu-time to do the verifications only take roughly one second to compute on a pentium-class processor.

The complexity is thus low enough to encourage future extensions of the model. How this complexity will increase by adding more functionality is not known, but the goal of the Mana project is to have full functionality of a Ravenscar-compliant Run-Time System in conjunction with a moderate sized application program.

The next natural step to take is to make a complete model of a priority sensitive scheduler using the Immediate Ceiling Priority Inheritance Protocol.

References

[AD90] R. Alur, and D. Dill, "Automata for Modeling Real-Time Systems", Proceedings of the 17th International Colloquium on Automata, Languages and Programming, vol. 443, Springer-Verlag, 1990.

[ARM95] Intermetrics, "Ada95 Reference Manual", ISO/IEC-8652:1995, 1995.

[Bar97] J. Barnes, "High Integrity Ada - The SPARK Approach", Addison Wesley, ISBN 0-201-17517-7, 1997.

[Bjo95] L. Björnfot, "Ada and Timed Automata", In proc. Ada Europe, Frankfurt, Germany, LNCS 1031, pp. 389-405, Springer-Verlag, Oct 1995.

[Cha98] Pierre Chapront, "Ada+B The Formula for Safety Critical Software Development", Ada Europe, Uppsala, Sweden, LNCS 1411, pp. 14-18, Springer-Verlag, June 1998.

[CO$^+$95] J. Crow, S.Owre, J. Rushby, N. Shankar, and M. Srivas, "A tutorial introduction to PVS", WIFT'95: Workshop on Industrial-Strength Formal Specification Techniques, Boca Raton, Florida, April 1995.

[DB98] B. Dobbing and A. Burns, "The Ravenscar Tasking Profile for High Integrity Real-Time Programs", SIGAda'98, Nov 8-12, 1998.

[FW96] S. Fowler and A. Wellings, "Formal Analysis of a Real-Time Kernel Specification", FTRTFT'96, 1996

[FW97] S. Fowler and A. Wellings, "Formal Development of a Real-Time Kernel", 19th IEEE Real-Time Systems Symposium, Dec 1997.

[GMP90] David Guaspari, Carla Marceau, and Wolfgang Polak. Formal Verification of Ada Programs. IEEE Transactions on Software Engineering, vol. 16, no. 9, September 1990, pp. 1058-1075.

[HRG98] ISO/IEC PDTR 15942, Guidance on the Use of the Ada Programming Language in High Integrity Systems,

[HU79] J.E. Hopcroft and J.D. Ullman, "Introduction to Automata Theory, Languages and Computation", ISBN 0-201-02988-X, Addison-Wesley, 1979.

[Hut94] A. Hutcheon, "Safe Nucleus Formal Specification", Project Reference CI/GNSR/27: The Design and Development of Safety Kernel, Aug 1994.

[IRT99] To be published in Ada letters, spring 1999.

[JM86] F. Jahanian and A. K. Mok, "Safety analysis of timing properties in real-time systems", IEEE Transactions on Software Engineering, 12(9):890-904, Sept. 1986.

[LAM99] K. Lundqvist, L. Asplund, and S. Michell, “A Formal Model of the Ada Ravenscar Tasking Profile; Protected Object”, To appear in proc. Reliable Software Technologies - Ada-Europe, Santander, Spain, LNCS, Springer-Verlag, June 1999.

[LPY97] K.G. Larsen, P. Pettersson, and W. Yi, “UPPAAL in a Nutshell”, Int. Journal on Software Tools for Technology Transfer, Springer-Verlag, vol 1, number 1-2, pp. 134-152, Oct 1997.

[Smi92] M.K. Smith, The AVA Reference Manual: Derived from ANSI/MIL-STD-1815A-1983, Computational Logic Inc., Feb. 1992

[Tol95] R.M. Tol, “Formal Design of a Real-Time Operating System Kernel”, Ph.D. thesis, University of Groningen, 1995.

[WB97] A. Wellings and A. Burns, “Workshop Report”, The Eighth International Real-Time Ada Workshop (IRTAW8), Ada User Journal, vol 18, number 2, June 1997.