A Real-Time Kernel for Ravenscar

Gustaf Naeser
Department of Computer Science and Electronics
Mälardalen University
Sweden
Email: gustaf.naeser@mdh.se

***Abstract*— This report describes a Ravenscar compliant real-time kernel developed for and with use of formal modelling and methods. There kernel described is divided into application components, hardware components and kernel components. The components, their interfaces and operation, are described in this report.**

I. Introduction

The SafetyChip project defines a framework for development, monitoring and policing of formally verified hardware/software systems. The aim of the framework is to create an environment where the cost and complexity of using formal methods and formal verification can be reduced.

The framework consists of three kinds of components, the *application components* describing the software application, the *hardware components* describing, e.g., the processing resources available, and the *kernel components* which facilitate the application components' access to the hardware components.

The systems studied in the project have been embedded real-time systems written in Ada under the Ravenscar tasking profile [1]. The Ravenscar profile removes most run-time ambiguities of the Ada language by restricting its tasking. The choice of the Ada language and the tasking profile was encouraged by that setup being well suited for development of systems of the size and complexity most likely to be manageable by the SafetyChip framework.

The term system will be used to refer to a composition of all three kinds of components, and the final product of the development of an embedded system. Precise knowledge about the run-time operation of the hardware is required in order to accurately reason about and verify run-time properties of a system. This implies that intimate knowledge of the RTK and its operation is required. To reach that level of knowledge a Ravenscar kernel was designed and modelled.

The kernel described here is designed to support multiple processors (processing units). This support does lead to a more complex kernel but the additions are interesting since they allow the performance of a system to be improved without changes of the application.

The kernel can easily be extended and modified to gain more information about the systems behaviour. An example of this is given in the description of the Null Task, though it should be remembered that all extensions increase the resources needed during system verification. To illustrate the kind of properties that can be investigated, examples of general properties that can be interesting to a larger number of applications are described for some of the components.

To make it easier to compare to a predecessor kernel [4] the kernel is modelled in UPPAAL. Knowledge and understanding of UPPAAL's timed automata its notation is assumed or can be found in [2], [3].

II. Kernel Components

This section details the kernel components. The components are described with their interfaces and an UPPAAL model describing their behaviour is also given. Parts of the kernel, the Ready Queue and the Delay Queue, are described in [7] and [6] respectively.

Components communicate using interface functions. Data is passed using arguments (here called registers) that are identified in the models by their initial capital 'R'[1] and through global variables. For example, the register Rt is used to pass the identity of tasks, e.g., T_1. The registers used are Rt (task identities), Rp (task priorities), Ru (unit identity), Rd (release time for delays), Rpo (protected object identity), Rpy (protected object subroutine type) and Rpc (protected object subroutine identity). The global variables that the components can use are shown in Table I.

TABLE I
GLOBAL VARIABLES.

cnt_u	the number of processing units the kernel should use.
cnt_t	the number of tasks (note that cnt_i, below, is included in this number).
cnt_po	the number of protected objects.
cnt_i	the number of interrupts (and hence interrupt tasks).
time	the current system time.
STOPTIME	the system time at which the execution should be stopped.
barrier	an array which stores the values of the protected object entries' values.

[1]Note that components can have local data variables starting with a capital 'R' as well, e.g., the Ready Queue's local queue variables in Fig. 1.

In order for a value to be read from a register on a transition, that value must stored ahead of the read. There are two ways to ensure that a value is available. Either the register is written to on an earlier transition (and then left in the register) or is synchronisation occurs, writes by the sender are processed before reads of the receiver, and hence occur prior. Synchronisation is made over matching channels (channels with the same name) that are marked with an exclamation mark (chan!) for the sender and a question mark for the receiver (chan?). If a register is to be used to decide if a transition should be taken or not the value must be written by an earlier transition and can not be written on the same. The register assignments in the kernel are carefully designed so that the registers contain the required values.

An important task of the kernel components is to always make sure that the token for each application task's execution is kept alive. The token is passed from, e.g., the Ready Queue to the Delay Queue when a task is suspended after having requested a delay. The Delay Queue is from that point in time responsible for the task and must transfer the responsibility back to the Ready Queue in order for the task to execute.

A. The Ready Queue

The main task of the Ready Queue is to ensure that the processing units are loaded (running) the set of ready tasks of the highest priority. The queue should enforce a priority based scheduling where a first in first out (FIFO) order is maintained within each priority.

When a tasks becomes ready to run (runnable) and the Ready Queue is informed of this and inserts the task into the queue of runnable and running tasks. Tasks are inserted last within their priority to avoid starving tasks that have been runnable for a longer time. The Ready Queue keeps the execution token for any tasks in the queue of runnable tasks and it can actively change the running status of a running task by loading another task on the unit the task executes on, i.e., preempting it. A model of the Ready Queue is shown in Fig. 1. and the Ready Queue's interface functions are described in Table II and the variables and arrays it uses in Table III.

The Ravenscar tasking profile restricts the usage of dynamic priorities but the capability to dynamically change the priority of tasks is still required since the handling of access to protected objects require it. Ravenscar also restricts dynamic tasks by disallowing task termination and dynamic creation of tasks. For modelling reasons there is a create interface function which is called once by each task during system initiation.

The behaviour model of the Ready Queue can be divided into five components, each handling one of the input interface functions of Fig. II(a).

The initial and idle location of the ready queue in location n_0. To reduce state space during verification the temporary variable i should be zero when the Ready Queue is in this location.

TABLE II
READY QUEUE INTERFACE DESCRIPTION.

(a) The Ready Queues (input) interface, task control, which is used to modify the Ready Queue.

create(T_{id}, T^p)	create T_{id} with priority T^p (this does not add the task to the queue of runnable tasks).
runnable(T_{id})	add T_{id} to queue of runnable tasks.
suspend(T_{id})	remove T_{id} from queue.
changep(T_{id}, T^p)	change the priority of T_{id} to T^p.
unblock(T_{id})	put T_{id} last within its priority.

(b) The Ready Queues (output) interface containing functions used on other kernel components interfaces.

load(U_{id}, T_{id})	Unit control function to tell U_{id} to load T_{id}.
nopreempt(T_{id})	Unit control function to indicate that T_{id} continue to run.

TABLE III
VARIABLES USED BY THE READY QUEUE.

RQt	array holding an ordered queue of runnable and running tasks (identified using the task identities). The fist cnt_u positions in the array represent the right to a processing unit and tasks with their identities in those positions are executing.
RQp	array holding the tasks' priorities.
RQu	array holding the, if any, unit that tasks are running on.
RQuu	array with a position for each processing unit used to indicate units that are running.
used	variable holding the number of used units.
ready	variable holding the number of tasks running or ready to run.
i	temporary variable used while iterating.

The create interface function's behaviour is described on the transition $n_0 \longrightarrow n_0$ which leads from location n_0 and back in again. The action taken when the create interface is called (by observation of create?) is to store the priority sent using register Rp in the array used for storing priorities, RQp. The created tasks priority, sent in the Rt register, is used to index the position in which to store the priority.

The runnable array is kept in ordered so that the task with the highest priority is in the fist position of the array, position indexed by 0, and so that tasks of higher priority have lower position indexes than those with higher priority. Tasks of the same priority are ordered so that new tasks are inserted last within their priority, i.e., get higher indexes. The ready variable points to the leftmost unused position of the array, i.e., the position to the right of the tasks of the lowest priority. Tasks in the leftmost cnt_u (the number of processing units) positions should be

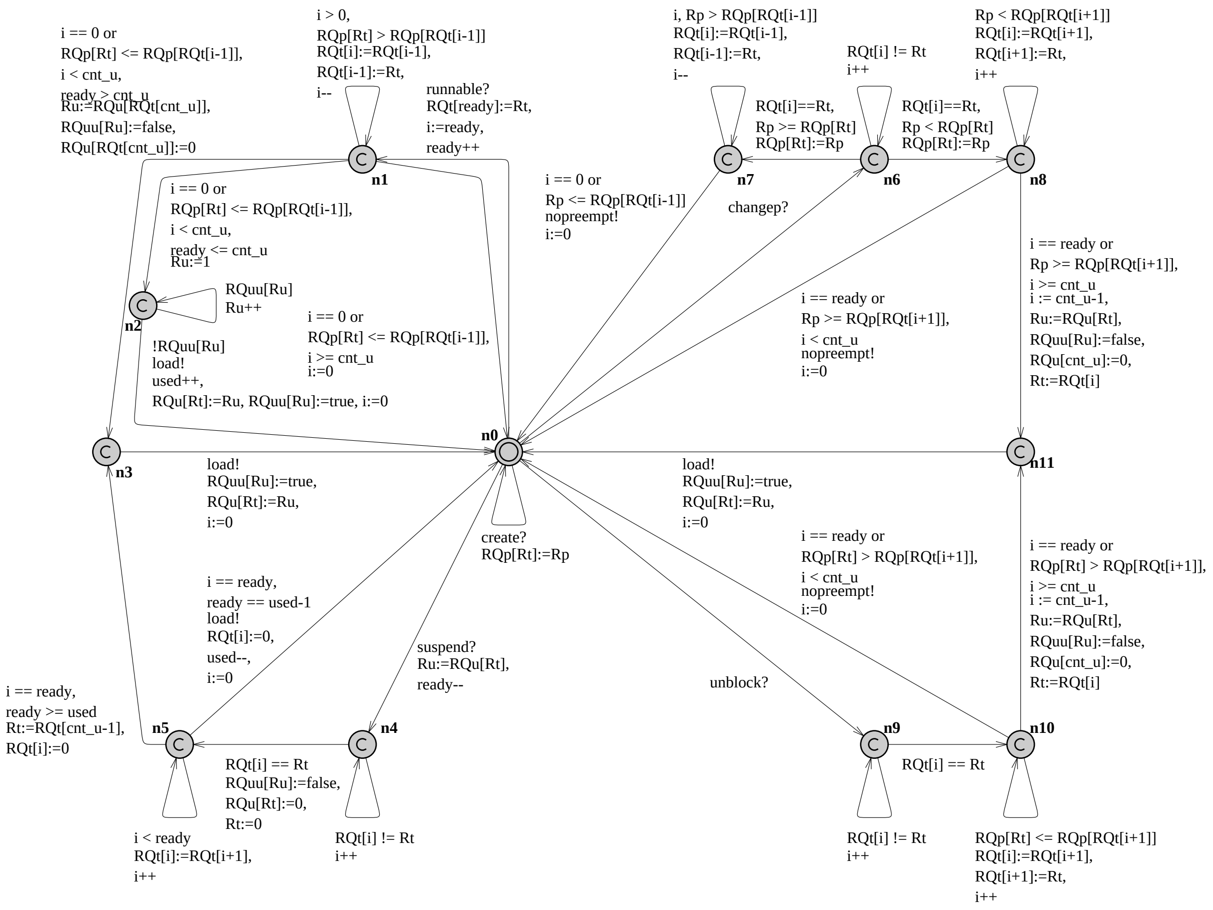

Fig. 1. The Ready Queue.

loaded into an unit and the positions will be referred to as running positions.

The runnable interface is called, $n_0 \longrightarrow n_1$ the transition stores the identity of the task made runnable, T_{Rt}, in the first unused position of the ready queue, RQt. The next action the Ready Queue takes is to sort the newly inserted task to its proper position which is carried out using the loop $n_1 \longrightarrow n_1$. The variable i is used to indicate the current position of T_i. The $n_1 \longrightarrow n_1$ loop will move the task ahead of tasks of lower priority ($RQp[Rt] > RQp[RQi[i-1]]$) but the task will not be moved further than to the first position of the array ($i > 0$). When T_1 has been moved to its correct position there are three possibilities; either T_1 did not reach a running position ($i >= cnt\_u$) in which case nothing more needs to be done and the transition $n_1 \longrightarrow n_0$ is taken or the task did reach a running position ($i < cnt\_u$) in which case one of three actions should be taken. If a runnable position was reached and this moved a task out of a running position, $T_{RQt[cnt\_u]}$, (which happens if $ready > cnt\_u$) the unit that task is running on is identified ($RQu[RQt[cnt\_u]]$) and unit information associated with that unit and the soon to be preempted task is updated to reflect that no task is running on the unit. Then T_{Rt} is loaded, $n_3 \longrightarrow n_0$ and unit information is updated to reflect this. If T_{Rt} after the n_1 loop reached a runnable position but this did not lead to that a task must be preempted ($ready <= cnt\_u$) the transition $n_1 \longrightarrow n_2$ is taken to n_2 where a loop is used to locate the free unit with the lowest identity. The T_{Rt} is loaded ($n_2 \longrightarrow n_0$) into that unit and information about unit usage is updated to reflect this.

The suspend interface is used for removing a task T_{Rt} from the ready queue. When suspending a task, $n_0 \longrightarrow n_4$, the unit that the task is running at ($RQu[Rt]$) is remembered in Ru. The loop $n_4 \longrightarrow n_4$ locates the array position of T_{Rt}. Then the unit information associated with the task is updated to reflect that it is not running and a Null Task is inserted in place of the task, $n_4 \longrightarrow n_5$. The loop $n_5 \longrightarrow n_5$ is used to move the Null Task, which has the lowest possible priority, to its proper place in the array. When the Null Task is in the correct position, either a task was moved into a running position ($ready >= used$) in which case that task is loaded ($n_5 \longrightarrow n_3 \longrightarrow n_0$) or no task was moved into a running position, in which case transition $n_5 \longrightarrow n_0$ is used to load the Null Task into the unit.

Transition $n_0 \longrightarrow n_6$ is used to serve a request to change the priority of a task T_{Rt}. The behaviour of this interface function is made easier by knowing that the priority can only be changed for running tasks. The loop $\mathsf{n}_6 \longrightarrow \mathsf{n}_6$ is used to locate the array position of the task. If the priority of T_{Rt} is raised, $Rp > RQp[Rt]$, the way units are loaded will not be affected and the transitions $\mathsf{n}_6 \longrightarrow \mathsf{n}_7 \longrightarrow \mathsf{n}_0$ will move the task to its new position in the array. The task is informed that it was not preempted by the call of the nopreempt interface function on $\mathsf{n}_7 \longrightarrow \mathsf{n}_0$. If, on the other hand, the task's priority is lowered, first $\mathsf{n}_6 \longrightarrow \mathsf{n}_8$ is taken and then the $\mathsf{n}_8 \longrightarrow \mathsf{n}_8$ loop moves the task to its new position. If the new position moved the task out of the running positions ($i >= cnt\_u$) the task that was moved into the running positions (which must be in position $RQt[cnt\_u - 1]$ is loaded onto the unit from which T_{Rt} should be preempted, $\mathsf{n}_8 \longrightarrow \mathsf{n}_{11} \longrightarrow \mathsf{n}_0$. If the priority was lowered but this did not move T_{Rt} out of the running positions, transition $\mathsf{n}_8 \longrightarrow \mathsf{n}_0$ will be used to inform the task that it was not preempted.

The unblock interface, which moves a task to the to the end within its priority starts with the $\mathsf{n}_0 \longrightarrow \mathsf{n}_{10}$ and its behaviour is similar to that expressed when a tasks priority is lowered.

B. The Delay Queue

The Delay Queue allows tasks to delay until a given time. If the release time is in the future the Delay Queue will take over the responsibility of the tasks execution and suspend it, but if the time is in the present or past the queue will inform the Ready Queue that the tasks should be scheduled last of tasks with the same priority.

The Delay Queue takes over the responsibility of the tasks execution when the task delays and must make it runnable at the release time to forward that responsibility to the Ready Queue. The Delay Queue interface functions are described in Table IV and the variables used are described in Table V.

One model of the Delay Queue is shown in Fig. 2. Different designs of the queue, exploring performance and size properties, are described in [6]. The queue shown here does, e.g., not maintain a FIFO release order of tasks with the same release time but instead releases tasks with lower identity prior to others with the same release time.

The queue in Fig. 2 uses the next variable to remember the position in DQd with the closes release time (next), if there are several with the same release time the one with the lowest identity is remembered. The transition $\mathsf{n}_0 \longrightarrow \mathsf{n}_1$ is used when the delay_until interface is accessed to delay a task T_{Rt}. There are three possible transitions back to the idle location n_0. The Ready Queue's unblock interface is accessed if the requested release time is in the past or present ($Rd <= time$). If there are no delayed tasks ($!delayed$) or the release time (Rd) of the of T_{Rt} is nearer than the nearest release time ($Rd < DQd[next]$) or the release time is the same but T_{Rt} has lower identity than the nearest task ($Rd == DQd[next]\ and\ R1 < next$), then T_{Rt} will be indicated as the next task to be released.

TABLE IV

Delay Queue interface description.

(a) The Delay Queues (input) interface.

delay_until(T_{id}, T^p, D)	delay take T_{id} until the release time, D, is reached. The task's priority, T^p, can be used by Delay Queues implementing prioritised release of tasks released at the same time.
tick	a time interval has passed and the Delay Queue should release tasks if their release times have been reached.

(b) The Delay Queues (output) interface containing functions used to guide the Ready Queue.

runnable(T_{id})	add T_{id} to queue of runnable tasks.
suspend(T_{id})	remove T_{id} from the Ready Queue.
unblock(T_{id})	put T_{id} last within its priority of tasks in the Ready Queue.

TABLE V

Variables used by the Delay Queue.

DQd	array holding the release times for tasks.
delayed	the number of delayed tasks.
next	the next task to be released, i.e., the task with the closest release time.
i	temporary variable used while iterating.

Otherwise, the release information for T_{Rt} will only be recorded.

No action is taken when a tick is observed and there are either no delayed tasks or the global time has not reached the closest release time, transition00. If on the other hand the next release time is reached the transition $\mathsf{n}_0 \longrightarrow \mathsf{n}_2$ is taken. The transition sets the registers for making the released task (T_{next}) runnable, which is done on $\mathsf{n}_2 \longrightarrow \mathsf{n}_3$, and removes delay information for it. Transition $\mathsf{n}_3 \longrightarrow \mathsf{n}_0$ is taken if there are no more delayed tasks. Otherwise, the loop $\mathsf{n}_3 \longrightarrow \mathsf{n}_3$ is used to locate the delayed task with the lowest index (identity) and then the $\mathsf{n}_3 \longrightarrow \mathsf{n}_4$ transition is taken. In location n_4 the closest release time is located. Should the release time of a task be equal to the global time, i.e., it should be released immediately, the transitions $\mathsf{n}_4 \longrightarrow \mathsf{n}_2 3$ is used to release it. Otherwise the task with the closest release time is located and its position stored in next before going back to the idle location.

C. The Protected Object Queue

Protected objects are used to enforce mutual exclusion of code and/or data. The protected objects described by Ada can have protected data and routines for accessing that data. The routines can be procedures, entries and functions. A protected procedure can both read and change data inside the object and must guarantee that

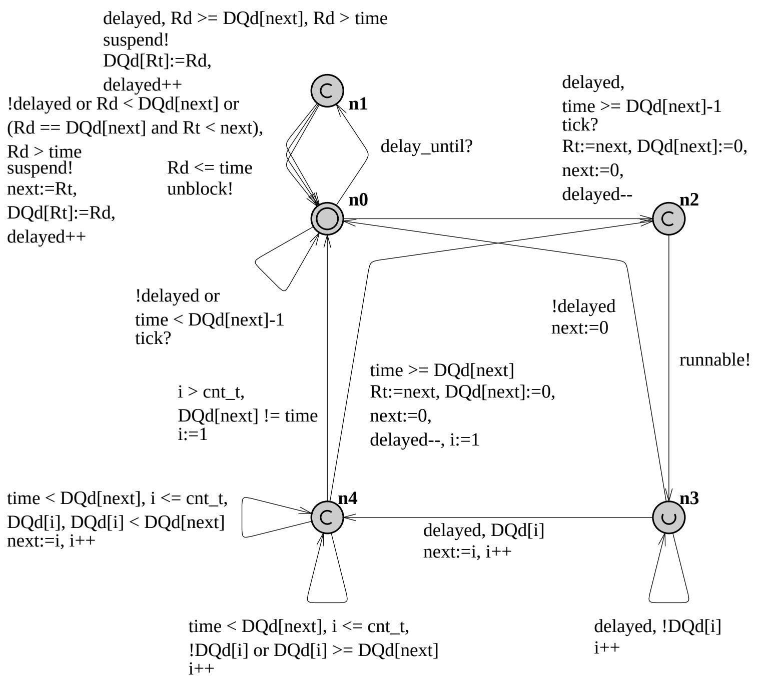

Fig. 2. The Delay Queue.

the task accessing the protected procedure has exclusive access to the object. When a task is executing a procedure the object is locked to other access enforcing the exclusive access. Protected entries work like procedures but have a guard which must be satisfied for the object to be entered. Protected functions can only be used to read data, not change. Since there is no need to lock other functions out of the object, several functions can be allowed to access the same object at a given time. When functions are accessing an object the object is locked to procedures since they might change the data.

The Ada standard requires that protected objects are accessed in a first in first out (FIFO) order, meaning that tasks should be arranged in a queue while waiting for access. The standard also gives precedence to entries and procedures over functions. The protected objects are accessed using a priority (ceiling) mechanism which raises the priority of the caller while it is accessing the protected object. To avoid deadlocks the priority used when executing inside an object must be higher than the highest priority of any task accessing the object. The raising of the priority is done in the task calling the protected object.

The Ravenscar profile restricts the behaviour of the objects by limiting the number of entries to one and the number of tasks waiting to access the entry to one. This effectively removes non-deterministic behaviour that can rise from the use of protected objects[2].

The queue uses several arrays to manage the applications protected objects. Each array contains information about all protected objects but for easy of description they will be described as if the queue managed access to one object only. The Protected Object Queue has two interface functions that tasks should use to access a protected object. The interface is described in Table VI and variables used by the queue are described in Table VII.

The SafetyChip's model of the Protected Object Queue is shown in Fig. 3.

Tasks call protected object routines using the call interface function. The Protected Objects Queue will then take control over the task's execution and suspend the task until access to the object can be granted. When a task is granted access to a subprogram the queue invokes the subprogram and hands over the task's execution to it using the aquire interface function. When the subprogram is finished it surrenders control back to the task using the endcall interface function and also informs the queue that the object is free again using the release interface function.

The functionality of the Protected Objects Queue has three main part, one for handling calls to protected procedures, n_4, one for calls to protected entries, n_3, and one

[2]It should be noted that the protected objects in a system with multiple processors (processing units) can have several tasks waiting to access protected procedures. This is however not possible in a single processor system (due to how protected objects are supposed to be managed by Ada). The Ravenscar profiles restriction of the number of tasks waiting for an entry might also have been intended for the protected procedures since they still in multi processor systems create situations similar to those the restriction removes.

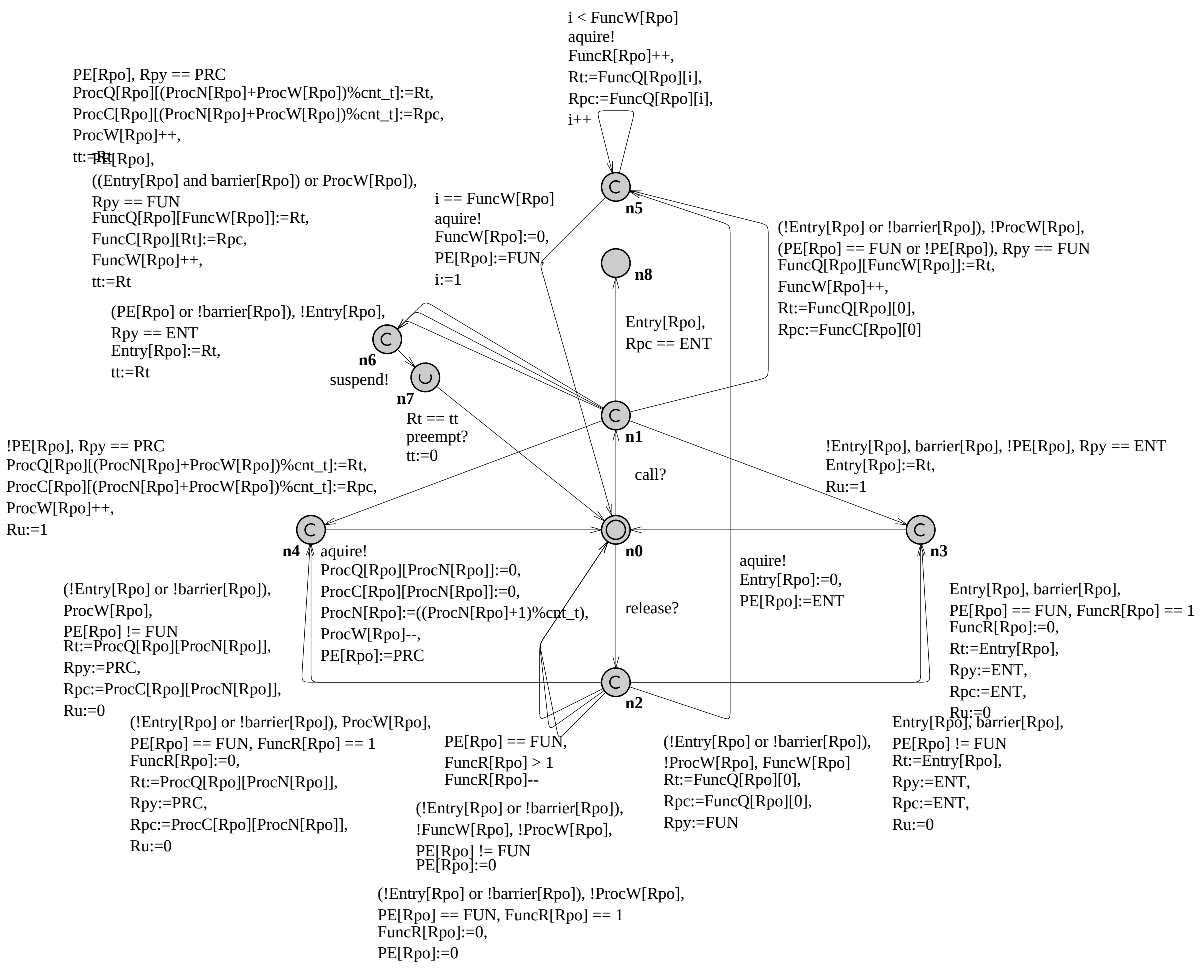
i < FuncW[Rpo]
aquire!
FuncR[Rpo]++,
Rt:=FuncQ[Rpo][i],
Rpc:=FuncQ[Rpo][i],
i++
PE[Rpo], Rpy == PRC
ProcQ[Rpo][(ProcN[Rpo]+ProcW[Rpo])%cnt_t]:=Rt,
ProcC[Rpo][(ProcN[Rpo]+ProcW[Rpo])%cnt_t]:=Rpc,
ProcW[Rpo]++,
tt:=Rt
PE[Rpo],
((Entry[Rpo] and barrier[Rpo]) or ProcW[Rpo]),
Rpy == FUN
FuncQ[Rpo][FuncW[Rpo]]:=Rt,
FuncC[Rpo][Rt]:=Rpc,
FuncW[Rpo]++,
tt:=Rt
i == FuncW[Rpo]
aquire!
FuncW[Rpo]:=0,
PE[Rpo]:=FUN,
i:=1
n5
n8
(!Entry[Rpo] or !barrier[Rpo]), !ProcW[Rpo],
(PE[Rpo] == FUN or !PE[Rpo]), Rpy == FUN
FuncQ[Rpo][FuncW[Rpo]]:=Rt,
FuncW[Rpo]++,
Rt:=FuncQ[Rpo][0],
Rpc:=FuncC[Rpo][0]
(PE[Rpo] or !barrier[Rpo]), !Entry[Rpo],
Rpy == ENT
Entry[Rpo]:=Rt,
tt:=Rt
n6
suspend!
n7
Entry[Rpo],
Rpc == ENT
Rt == tt
preempt?
tt:=0
n1
!PE[Rpo], Rpy == PRC
ProcQ[Rpo][(ProcN[Rpo]+ProcW[Rpo])%cnt_t]:=Rt,
ProcC[Rpo][(ProcN[Rpo]+ProcW[Rpo])%cnt_t]:=Rpc,
ProcW[Rpo]++,
Ru:=1
call?
!Entry[Rpo], barrier[Rpo], !PE[Rpo], Rpy == ENT
Entry[Rpo]:=Rt,
Ru:=1
n4
n0
n3
aquire!
ProcQ[Rpo][ProcN[Rpo]]:=0,
ProcC[Rpo][ProcN[Rpo]]:=0,
ProcN[Rpo]:=((ProcN[Rpo]+1)%cnt_t),
ProcW[Rpo]--,
PE[Rpo]:=PRC
aquire!
Entry[Rpo]:=0,
PE[Rpo]:=ENT
release?
(!Entry[Rpo] or !barrier[Rpo]),
ProcW[Rpo],
PE[Rpo] != FUN
Rt:=ProcQ[Rpo][ProcN[Rpo]],
Rpy:=PRC,
Rpc:=ProcC[Rpo][ProcN[Rpo]],
Ru:=0
Entry[Rpo], barrier[Rpo],
PE[Rpo] == FUN, FuncR[Rpo] == 1
FuncR[Rpo]:=0,
Rt:=Entry[Rpo],
Rpy:=ENT,
Rpc:=ENT,
Ru:=0
n2
(!Entry[Rpo] or !barrier[Rpo]), ProcW[Rpo],
PE[Rpo] == FUN, FuncR[Rpo] == 1
FuncR[Rpo]:=0,
Rt:=ProcQ[Rpo][ProcN[Rpo]],
Rpy:=PRC,
Rpc:=ProcC[Rpo][ProcN[Rpo]],
Ru:=0
PE[Rpo] == FUN,
FuncR[Rpo] > 1
FuncR[Rpo]--
(!Entry[Rpo] or !barrier[Rpo]),
!ProcW[Rpo], FuncW[Rpo]
Rt:=FuncQ[Rpo][0],
Rpc:=FuncQ[Rpo][0],
Rpy:=FUN
Entry[Rpo], barrier[Rpo],
PE[Rpo] != FUN
Rt:=Entry[Rpo],
Rpy:=ENT,
Rpc:=ENT,
Ru:=0
(!Entry[Rpo] or !barrier[Rpo]),
!FuncW[Rpo], !ProcW[Rpo],
PE[Rpo] != FUN
PE[Rpo]:=0
(!Entry[Rpo] or !barrier[Rpo]), !ProcW[Rpo],
PE[Rpo] == FUN, FuncR[Rpo] == 1
FuncR[Rpo]:=0,
PE[Rpo]:=0

Fig. 3. The Protected Object Queue.

for protected functions, n_5.

In the idle location, n_0, the Protected Objects Queue waits for calls to the call, $n_0 \longrightarrow n_1$, or release interfaces, $n_0 \longrightarrow n_2$.

In the location n_1 the three transitions $n_1 \longrightarrow n_6$ are used if the called protected object for some reason is not accessible by the caller. The three possible reasons are, for procedures ($Rpy == PRC$) that the requested object is already accessed ($PE[Rpo]$), for protected functions ($Rpy == PRC$) that there is a task waiting to access the objects entry which has a barrier that allows access ($(Entry[Rpo]\ and\ barrier[Rpo])$) or has tasks waiting for access to a protected procedure ($ProcW[Rpo]$), and for protected entries ($Rpy == ENT$) that the objects is already accessed or that the protected objects barrier does not allow access ($!Entry[Rpo]$). The entry case also checks that there is no task already queueing for the entry ($!Entry[Rpo]$). On the respective $n_1 \longrightarrow n_6$ transition the state of the called protected object Rpo is updated with information about the calling task. When location n_6 is reached, transitions $n_6 \longrightarrow n_7 \longrightarrow n_0$ will be used to suspend the task and preempt it (taking its execution token).

The location n_8 is used to indicate that, and can only be reached when, a task queues up for access to an entry for which a task is already queueing since this would violate the Ravenscar restriction. A property that can be investigated for a system is therefore that n_8 is never reached.

If a procedure was called and object can be accessed, transition $n_1 \longrightarrow n_4$ is used. In order to make it possible to reuse the acquiring of the object, $n_4 \longrightarrow n_0$, the transition $n_1 \longrightarrow n_4$ adds the procedure call to the queue of waiting calls. Transition $n_4 \longrightarrow n_0$ is used to aquire the object for the first task waiting in queue calling a procedure of the object.

The $n_1 \longrightarrow n_4 \longrightarrow n_0$ behaviour for procedures us replicated in $n_1 \longrightarrow n_3 \longrightarrow n_0$ for entry calls and in $n_1 \longrightarrow n_5 \longrightarrow n_0$ for function calls.

The release interface, from n_2, goes directly back to location n_0 if there is no queued call that can aquire the object. If a protected

If a protected entry should aquire the object (there is a task waiting, $Entry[Rpo]$, and the barrier allows

TABLE VI

Protected Object Queue interface description.

(a) The Protected Object Queue (task) interface which is used to by tasks to call protected objects.

call(T_{id}, PO_{id}, C, K)	Task T_{id} calls protected object $PO_{id}.C$ which is of kind K. K indicates if C is an entry, a procedure or a function.
endcall(T_{id})	Task T_{id} has finished accessing the protected object.

(b) The Protected Object Queue (internal) interface containing functions used internally to manage protected subprograms.

aquire(T_{id}, PO_{id}, C, U)	Function used when a call aquire the object PO_{id}'s subprograms (call) C. U indicated whether or not task T_{id} is running. The protected subprograms wait for this function to start accessing.
release(T_{id}, PO_{id})	Protected object PO_{id} is released.

access, $barrier[Rpo]$ and either the object was released by something else than a function, $PE[Rpo]! = FUN$ or it was released by the last function accessing the object ($PE[Rpo] == FUN, FuncR[Rpo] == 1$), the transition $n_2 \longrightarrow n_3$ will be used. The same behaviour is described for calls to protected procedures in $n_2 \longrightarrow n_4$ and for functions in $n_2 \longrightarrow n_5$.

The loop $n_5 \longrightarrow n_5$ is used to release all functions waiting for an object and the loop will be used after $n_2 \longrightarrow n_5$.

D. The Interrupt Queue

The Interrupt Queue is a component which contains a table of interrupt managers handling the interrupts. The interrupt mechanism uses suspended tasks that call protected objects when they are started. There is one suspended task for each interrupt that can be invoked which allows parallel interrupt processed.

The interface to the interrupts is described in Table VIII and variables used by the queue are described in Table IX.

When an interrupt, irq, is observed by the Interrupt Queue, $n_0 \longrightarrow n_1$, it makes the Interrupt Task of that interrupt runnable and disables (masks) the interrupt (so that at most one manager will be invoked for each interrupt, $interrupt[Rt] := true$), $n_1 \longrightarrow n_0$. When the task has finished its work it will inform of this by invoking the iack interface function, $n_0 \longrightarrow n_0$. Interrupts that are observed while their task is running ($interrupt[Rt]$) are ignored, $n_1 \longrightarrow n_0$.

The SafetyChip's model of the Protected Object Queue is shown in Fig. 4.

An example property that can be checked for the Interrupt Queue is that if interrupts are ignored, i.e., if the irq interface function ever called for an interrupt while the interrupt's manager is running. Ignored interrupts can indicate that the Interrupt Task is to slow in its processing.

TABLE VII

Variables used by the Protected Object Queue.

PE	the state the protected object. The values stored in the array indicate that the object is currently *0* not locked, ENT locked through an entry, FUN locked through function(s), and PRC locked through a procedure.
Entry	the identity of the task (Ravenscar restricts this to one task) waiting to access the entry of the protected object.
ProcW	the number of tasks waiting to access a protected procedure.
ProcQ	an array containing a list of the identities of the tasks waiting for access to protected procedures. The list is circular with ProcN pointing to the oldest call (first to be served).
Proc	an array holding the procedure to the corresponding task in ProcQ.
ProcN	a pointer to the next task to be released from ProcQ.
FuncW	the number of waiting function calls.
FuncQ	queue of tasks waiting for access to protected functions.
FuncC	an array holding the protected functions the tasks in FuncQ are calling.
FuncR	the number of functions currently inside a function call, i.e., accessing the protected object.
i	temporary variable used while iterating.
tt	temporary variable for holding a suspending task's identity.

TABLE VIII

Interrupt Queue interface description.

(a) The Interrupt Queue interface which is used to by the environment and the Interrupt Tasks.

irq(I_{id})	Called by the environment when interrupt with identity I_{id} occurs.
iack(I_{id})	Used by the Interrupt Task to reset the status of interrupt I_{id} so that the queue will observe the interrupt once again.

An Interrupt Task is described in Fig. 5. The task has the same priority as that of the protected object it calls, which ensures that the interrupt will be promptly handled. It can be noted that the task calls the Ready Queue's suspend interface function to suspend itself (and consumes the preempt. Normal tasks should not do this as it leaves themselves responsible for their execution which while preempted is not advisable since they do not have access to the processing resources required to change their state again. The manager can do this since it surrenders its execution control to the Interrupt Queue when calling its iack interface function.

III. Application Components

The application components are automatically transformed from the source code via temporal skeletons [5], [8]. There are two kinds of components, tasks and protected subprograms. The tasks are autonomous executives while

TABLE IX

Variables used by the Interrupt Queue.

interrupt	array containing the interrupts. The position corresponding to an interrupt is set to true when the interrupt has been observed and set to false when the interrupt has never been observed or when the Interrupt Task has finished its processing.

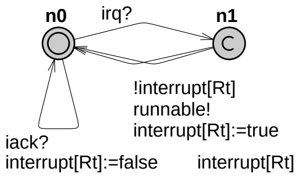

Fig. 4. The Interrupt Queue.

the protected subprograms wait dormant until accessed. The SafetyChip framework could do without the protected subprograms by inserting models for them into the tasks. However, inserting them would lead to larger task models which might reduce their readability.

The description here does not present any models for the application components as these are extremely application dependent.

A. Tasks

A description of the application tasks can be found in [5] which describes how temporal skeletons are transformed into backend Uppaal models.

All application tasks should start with creating themselves and then making themselves runnable. From this on they are free to call the interfaces of the kernel components to access their functionality. The interfaces available to tasks are described in Table X.

B. Protected Subprograms

The operation of protected subprograms is nearly that of application tasks but since they have no runnable identity of their own they need to be given one by a task.

Initially protected subprograms are waiting to be activated and they are activated by a call to the aquire interface function. The aquire instruction contains the identity of the task calling the subprogram, i.e., the identity which the subprogram should use while executing. The calling task can be running or preempted and the subprogram is supplied with this information when it is initiated. When the subprogram concludes its execution it should hand the tasks execution responsibility back to the task and then release the lock on the protected object. Then subprogram should then go back to waiting for another activation.

The special interface functions of the protected subprograms is described in Table XI. In addition to these the subprograms can use the interface available to the tasks.

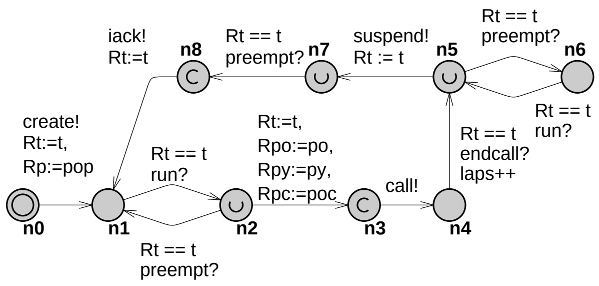

Fig. 5. The Interrupt Task.

TABLE X

Task interface description.

(a) The task (output) interface for making kernel calls.

create(T_{id}, T^p)	create T_{id} with priority T^p (this does not add the task to the queue of runnable tasks).
runnable(T_{id})	add T_{id} to queue of runnable tasks.
changep(T_{id}, T^p)	change the priority of T_{id} to T^p.
delay_until(T_{id}, D)	delay T_{id} until the release time, D, is reached.
call(T_{id}, PO_{id}, C, K)	Task T_{id} calls protected object $PO_{id}.C$ which is of kind K. K indicates if C is an entry, a procedure or a function.

(b) The task (input) interface containing functions used to indicate tasks' execution status.

run(T_{id})	used to indicate that T_{id} is running and is responsible for its own execution.
preempt(T_{id})	used to indicate that T_{id} has been preempted and the execution responsibility has been taken over by the Ready Queue.
nopreempt(T_{id})	used to indicate that T_{id} was not preempted, as expected, and the execution responsibility still lies with the task.
endcall(T_{id})	used to indicate that the task T_{id} has finished accessing a protected object and is again responsible for its own execution.

IV. Hardware Components

The hardware components describe the hardware environment the system uses. The environment consists of required and optional components. This section describes the required ones, the Clock and the Unit, and an optional one, an interrupt generator.

The main task of the hardware component models is to interact with the kernels external interface functions. This makes the model of the hardware only required during verification, and while the kernel components are present in the implementation, the hardware components are not implemented in the same way since the real components

TABLE XI

Protected subprograms interface description.

(a) The protected subprogram (input) interface.

aquire(T_{id}, PO_{id}, C, U)	Function used to activate the subprogram (call) C of protected object PO_{id}. U indicates whether or not task T_{id} is running.

(b) The protected subprograms (output) interface functions used to indicate completion of execution.

release(T_{id}, PO_{id})	Protected object PO_{id} is released.
endcall(T_{id})	Task T_{id} has finished accessing the protected object.

are present in the final system.

1) The Clock: The Clock describes a real-time clock generating tick events at a predetermined interval. Each tick represents that the interval has passed. For Ravenscar this is likely to be 1 ms which is the frequency the profile requires.

The interface functions of the Clock are described in Table XII and the variables that the Clock uses are described in Table XIII.

TABLE XII

Clock interface description.

(a) The Clock interface used to generate ticks to listening components.

tick	Used to inform other kernel components that time has passed.

TABLE XIII

Variables used by the Clock.

timer	Uppaal specific clock variable used to implement timed transitions. The variable is used to trigger the generation of a tick.
time	a counter of counting the number of ticks that has been generated, i.e., the time holds the current time.
STOPTIME	the time after which no more ticks will be generated. The Clock deadlocks when the STOPTIME is reached.

A model of the Clock is shown in Fig. 6.

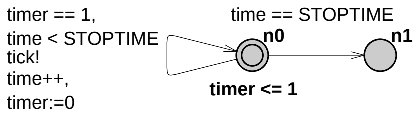

Fig. 6. The Clock.

The transition loop $n_0 \longrightarrow n_0$ uses Uppaal's timed transitions and can be taken as long as the STOPTIME has not been reached and there are no other transitions in the system that must be taken.

When the STOPTIME is reached the $n_0 \longrightarrow n_1$ will be taken which should ultimately deadlock the system.

A. The Unit (Processor)

The processing units are used to execute tasks. Units consist of two parts, the model of the actual processor and a model representing the Null Task, the idle task running on the processor when no application task is running on it, i.e., when the Unit is idle. There is one Null Task for each processor and the models of the tasks have been extended to provide utilisation data for the processor they are associated with.

The Units' interface of the is described in Table XIV and the variables they use in Table XV. The interface of the Null Tasks is the same as that of a normal task and the variables used are described in table Table XVI.

TABLE XIV

Unit interface description.

(a) The Unit (control) interface which is used to by the Ready Queue.

load(U_{id}, T_{id})	Unit U_{id} is instructed to load and execute task T_{id}.

(b) The Unit (tasking) interface containing functions used to inform tasks of their current status.

run(T_{id})	Used to inform a task that it is running.
preempt(T_{id})	Function used to inform a task that it has been preempted and is no longer running.

TABLE XV

Variables used by Units.

t	the identity of the task running on the unit.
tn	temporary variable for holding a task identity (used when preempting a task).

TABLE XVI

Variables used by Null Task.

id	the identity of the unit the task belongs to.
exectime	the accumulated time for which the Null Task has been loaded / running.
starttime	temporary variable to keep track of at which time the Null Task was started.

A model for the units is shown in Fig. 7, and for the Null Tasks in Fig. 8.

The unit waits in its initial location for a call to the load interface function. When the interface function is called to load task Rt, $n_0 \longrightarrow n_1$, the unit will preempt the currently running task, $n_1 \longrightarrow n_2$, and start Rt, $n_2 \longrightarrow n_0$.

Note that a unit's loading of a task, transition T_1, can be interrupted by order from the Ready Queue to load another task, T_2. As long as the kernel has not started T_1, i.e., represented by the unit using the run interface function, the new task T_2 can be loaded without needing to preempt task T_1.

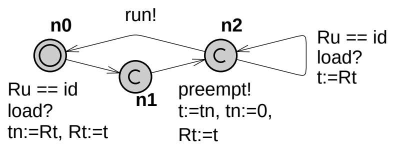

Fig. 7. The Unit (processor).

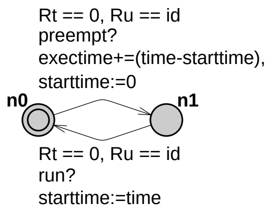

Fig. 8. The Null Task.

The need to keep track of execution time is not required in the Null Task and can, for verification or other reasons, be removed from the model. The tracking is an example of how the model can be extended to gain additional knowledge about the system's operation. The tracking is designed to record the time when the Null Task is loaded and to calculate for how long it was loaded and add this to an accumulative variable when unloaded.

V. Conclusion

This report describes a Ravenscar compliant real-time kernel. The kernel consists of a collection of interacting components. The components' interfaces are described and the behaviour of the individual components is presented using Uppaal's timed automata.

The kernel supports multiple processing units and dynamic priorities. The kernel is designed to be tailored for a specific system which makes it well suited for embedding. As an example of this, the number of tasks the application the kernel is to be used with can be set so that the kernel's size is minimised.

References

[1] A. Burns, B. Dobbing and G. Romanski, "The Ravenscar Tasking Profile for High Integrity Real-Time Programs", *Reliable Software Technologies — Ada-Europe 1998*, LNCS 1411, Springer–Verlag, 1998.

[2] G. Behrmann, A. David and K. G. Larsen, "A Tutorial on Uppaal", *Formal Methods for the Design of Real-Time Systems: 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems*, 2004.

[3] K. Larsen, P. Pettersson and W. Yi, "Uppaal in a Nutshell", *Int. Journal on Software Tools for Technology Transfer*, Springer–Verlag, 1997.

[4] K. Lundqvist and L. Asplund, "A Ravenscar-Compliant runtime kernel for safety critical systems", Real-Time Systems, 24(1), 2003.

[5] G. Naeser, "Transforming Timing Skeletons to Timed Automata", MRTC report ISSN 1404-3041 ISRN MDH-MRTC-187/2005-1-SE, Mälardalen Real-Time Research Centre, 2005.

[6] G. Naeser and J. Furunäs, "Evaluation of delay queues for a Ravenscar Hardware Kernel", MRTC report ISSN 1404-3041 ISRN MDH-MRTC-176/2005-1-SE, Mälardalen Real-Time Research Centre, 2005.

[7] G. Naeser and K. Lundqvist, "Component-Based Approach to Run-Time Kernel Specification and Verification", *ECRTS 05*, 2005.

[8] Gustaf Naeser, Kristina Lundqvist and Lars Asplund, "Temporal Skeletons for Verifying Time", *In Proceedings of SIGAda 2005*, ACM, 2005.