Transforming Temporal Skeletons to Timed Automata

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

***Abstract*— This paper presents a transformation from the intermediate temporal skeletons notation to timed automata, the notation used by the UPPAAL tool. The transformation includes only the transformation of the application, separate real-time kernel transformation is required.**

I. Introduction

Making formal models used for verification easy to read and understand should be mandatory. However, the detail needed in the actual verification often thwarts this desire. The approach taken in the SafetyChip framework is to use an easy-to-read intermediate model, a temporal skeleton, for model validation and the using further transformation from this model into the notation of the verification tool used for the formal verification. The UPPAAL tool is used for verification and design and the transformation from the temporal skeletons into the timed automata used by the UPPAAL tool is described.

The choice of using an intermediate notation was to open the possibility of strengthening the verification by using several verification tools. Allowing several backends can take advantage of enabling verification of the combined properties of the backend rather than limiting the verification to the properties one backend alone.

An predecessor to the SafetyChip kernel was modelled using UPPAAL by Lundqvist and Asplund [1]. To allow comparison of the two kernels the SafetyChip kernel uses the same backend.

The UPPAAL tool consists of three main modes: modelling, simulation and verification. Modelling is done in a graphical environment where timed automata can be drawn and decorated. The automata can then be transfered to the simulation and verification modes of the tool. The simulation allows execution of the automata to be visually shown and the verification part allows properties of the system to be formally verified.

The source code transformation into the notation of intermediate timing skeletons has been described in [5], this paper describes how the timing skeletons can be transformed into UPPAAL's timed automata.

II. Timing skeletons and interfaces to the RTK

The timing skeletons categorises application instructions as either passive or active instructions. Passive instructions only impact on run-time behaviour of a system comes from their consumption of computational resources while active instructions takes part in designing the temporal behaviour during run-time through communication with the RTK. Communication with the RTK is accomplished by using interface functions of the kernel components. A full description of the kernel and its interfaces can be found in [2]. A more detailed descriptions of the Ready Queue can be found in [4] and of the Delay Queue in [3].

A short description of the RTK's interface functions is shown in Table I.

An application task can call some of the kernel components' interface functions to instruct the kernel to carry operations like suspending a task, changing its priority, delaying it until a specific time, etc. Other interface functions are the means by which the kernel informs the tasks of what is happening, e.g., the kernel can inform a task that it is running when its execution is resumed after a time of suspension.

III. Backend model

The synchronisation available in UPPAAL is used for RTK calls, i.e., for affecting the temporal behaviour of the execution. The model can not take advantage of the clocks available in UPPAAL since their functionality does not meet the properties the RTK requires. Hence a clock variable, time, containing the current system time has to be used.

The rest of this section describes how different constructs of the temporal skeletons can be described using the timed automata of UPPAAL.

A. Sequential execution

Sequential execution is in the timed skeletons described using serial composition blocks. This serial composition is directly translated into the timed automata and is found in every automata, e.g., in Figure 2 where transition $n_0 \longrightarrow n_1$ is followed by transition $n_1 \longrightarrow n_2$. Serial composition can

TABLE I

RTK component interfaces functions accessible by tasks and protected subroutines. Functions set in *italic* are used by the RTK to control the application, whereas other functions are used to instruct the RTK.

(a) The Ready Queues interfaces used to manage the scheduling of application tasks.

create(T_{id}, T^p)	associate T^p with T_{id}
runnable(T_{id})	add T_{id} to queue
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
run(T_{id})	Used by the Ready Queue to inform T_{id} that it is running.[1]
preempt(T_{id})	Used by the Ready Queue to inform T_{id} that it has been preempted.[1]
nopreempt(T_{id})	Used to indicate that T_{id} should continue to run while it might have expected to be preempted.

(b) The Delay Queues (input) interface for delaying until given times.

delay_until(T_{id}, T^p, d)	Delay T_{id} until p at priority T^p.[2]

(c) The Protected Object Queue interface used by tasks and protected objects.

call(T_{id}, PO_{id}, C, K)	Used when T_{id} calls $PO_{id}.C$ which is of kind K.
aquire(PO_{id}, C, U, T_{id})	Start $PO_{id}.C$ for caller T_{id}. U is used to indicate if the task is running.
end_call(T_{id})	Informs task T_{id} that it leaves the protected object.
release(PO_{id})	Used to release PO_{id}.

[1] The preempt function belongs to the processing unit but has for reasons of keeping this presentation siple been moved to the Ready Queue. The relocation has no effect on the operational behaviour as it is seen in this presentation.
[2] The priority is used by Delay Queues implementing prioritised release of tasks released at the same time.

be used to serialise any kind of the skeletons' blocks regardless whether or not they are temporal.

B. Conditional execution

When blocks are separated by conditional expressions the construct shown in Figure 1 is used.

The transitions $n_0 \longrightarrow n_1$ and $n_0 \longrightarrow n_2$ should be attributed with guards, synchronisations and/or assignments derived from the source code. There can be any number of transitions $n_0 \longrightarrow n_i$ to create more choices.

C. Task initiation

A normal application task should announce its existence to the RTK at system initiation so that the RTK can insert it in its tables and Ready Queue. The RTK Ready Queue listens for a create signal which is sent on transition $n_0 \longrightarrow n_1$ in Figure 2. Directly following the create a normal application also informs the Ready Queue that the

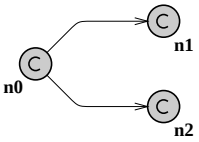

Fig. 1. Conditional execution is described using a location with two or more exiting transitions. The location is committed since execution must not linger in it.

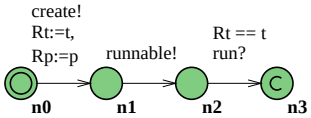

Fig. 2. The building block automaton for creating a task and make it runnable. Since location n_1 is committed the task identity need only be set at the $n_0 \longrightarrow n_1$.

task is ready to execute, i.e., that the task is runnable, which is made on $n_1 \longrightarrow n_2$.

Tasks are created with a set priority which is stored in the ready queue. Special tasks, like interrupt tasks, do not have the runnable transition, $n_1 \longrightarrow n_2$, in their initiation as they rely on external activation, e.g., by a an interrupt queue.

There is no need for a block which terminates a task since the Ravenscar profile does not allow task termination. However, a task which suspends and is not made ready by another entity will behave as if it was terminated. This behaviour is shown in the transitions $n_{15} \longrightarrow n_{16} \longrightarrow n_{17}$ in Figure 9.

D. Execution blocks

A snippet of code that have no RTK interaction and whose sole temporal effect is consumption of execution time is modelled as shown in Figure 3.

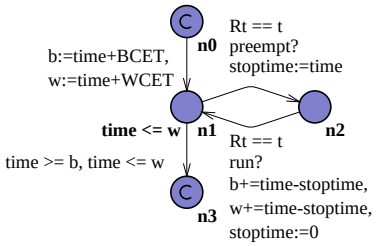

Fig. 3. The building block automaton for time consumption in an preemptive environment. If the task is preempted it will be suspended in the location n_2 until it is run again.

The transition leading into location n_0 initiates the time consumption by calculating the actual best and worst times with regard to the current system time which is stored in the global variable time. When at least b time has passed, i.e., the task has been processing that amount of time, the exiting transition $n_1 \longrightarrow n_3$ becomes enabled.

The exiting transition remains enabled until the time w is reached, at which time the transition leading to n_3 is forced.

If the task is preempted, $n_1 \longrightarrow n_2$, the time at which the preemption occurred will be recorded so that the execution times b and w can be updated correctly when the task starts running again, $n_2 \longrightarrow n_1$.

E. Delay until

Delay until a specific time is shown in Figure 4. The Ravenscar tasking profile only allows Ada's absolute `delay_until`, and not the relative `delay`. In the model we present here the could with little effort implement both constructs at little extra cost. Specifically, there would be no additional verification cost since the RTK Delay Queue component can work equally effectively with absolute and relative delays both.

When a task suspends there are two immediate possibilities. Either the release time, the time the task suspends until, *a*) is in the past or is on the present, or *b*) the it is in the future. If the release time has passed or is in the present the task should not be suspended and $n_1 \longrightarrow n_3$ will be used. Otherwise the task will be preempted and in location n_2 wait for its future activation.

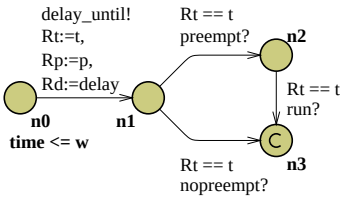

Fig. 4. The delay until construct. A delaying task is preempted if the release time is in the future, otherwise no preemption occurs and it can promptly continue its execution in location n_3.

F. Protected objects

Protected objects [6] are the mechanism used to achieve mutual exclusion in Ada. The objects are interesting since they are a source of non-determinism in Ravenscar tasking profile for systems with multiple processors.

A protected object has a data part and a code part. The data part contains the protected variables, and the code part contains subprograms used to access the data part. The code part can have functions, procedures and entries. Functions may not change the variables and are allowed to run concurrently with other functions. Accesses to the entries and procedures are however exclusive since they are allowed to modify data. In the Ravenscar profile the number of entries of a protected object is limited to one. Queueing is used to manage concurrent calls to procedures and entries. All restrictions made in the Ravenscar profile but for one can be checked with static analysis of source code. In order to verify if an application preserves the dynamic restriction (`Max_Entry_Queue_Depth => 1`), verification needs to be done using the execution times of the application or other means of analysis of the systems run-time behaviour.

The mechanism for mutual exclusion and handling of barriers is located in the Protected Object Queue of the RTK. Calls to protected objects are passed through that component and the component will make a task runnable when the object it calls is free and the task's turn to access the object comes up. The building blocks for accessing, calling, protected subroutines, bodies, is shown below.

1) Procedure and entry calls: The building block to call a procedure or entry of a protected object is shown in Figure 5. The sequence is started with the raising of the executing tasks priority. A change in priority can, if the priority is lowered, result in that the task is preempted, but since a task calling a protected object will always raise its priority, and since this never will preempt the task, there is only one transition leading from location n_0. However, there is no guarantee that the task will not be preempted one it has raised it priority and hence location n_2 allows preemption before the call to the protected object component it made, $n_2 \longrightarrow n_4 \longrightarrow n_5$.

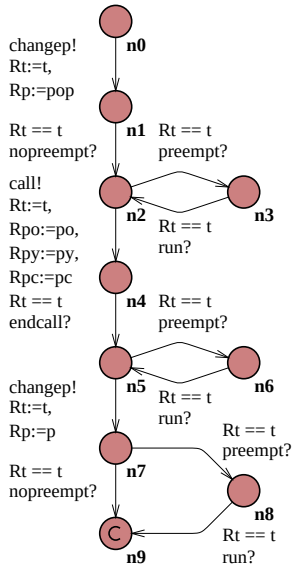

Fig. 5. The building block for calling a procedure or entry in a protected object. The task handles the priority changes prior to and after accessing the object.

2) Function calls: The building block for calling a function of a protected object is the same as that for calling procedures and entries, as above, with the exception that the function's code is inserted into the task rather than placed in a separate body automaton. The changes needed in Figure 5 is in location n_5 where the block for the protected function, described in Figure 6 should be inserted.

3) Body code: The body code for a protected procedure, entry or function is modelled as shown in Figure 6.

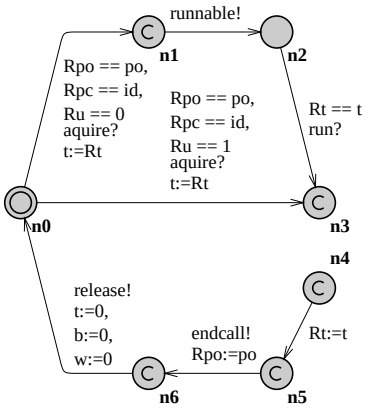

Fig. 6. The building block for the body of protected object code. The body code is started in location n_3 and ended in n_4.

A protected body consists of an aquire part, between n_0 and n_3, where the body code is invoked by the task, and a release part, between n_4 and n_0, where the protected call is done with its computation. The call specific code is inserted to form a transition chain $n_3 \longrightarrow n_4$.

IV. An Example Application

In this section an example application is presented. Timing skeletons and the timed automata for entities of the application is shown and explained. The automata below are patched so that they do not terminate and hence conform to the Ravenscar profile.

A. Protected Objects and the Application

The problem of racing conditions occurring when accessing protected objects will be illustrated in this section. The problem is interesting to this report since it cannot be non-pessimisticly verified without temporal information. The source code and outlines of the temporal skeletons for the application used in the discussion is shown in Figure 7.

We will in Section IV-C briefly show how to verify if tasks `Task2` and `Task3` can be calling the protected entry at the same time, thus violating the dynamic restriction of Ravenscar. We assume that all tasks, `Task1`, `Task2` and `Task3`, are started at the same time, zero, and that there are no other tasks in the system. We use call to the procedure `work(2, 3)` to indicate that the tasks executes for between 2 and 3 microseconds (BCET and WCET). The **`delay until`** in `Task2` is used to delay the release of the task until the system time reaches 10. We assume that the `Barrier` is initially false.

Even if all tasks have the same priority operation of the program will lead to that `Task1` sets the barrier to true and allows the other tasks to call the entry. `Task2` will either beat `Task1` in calling the protected object, and be suspended since the barrier is false, or reach the barrier when `Task1` has opened it. From looking at the code it is obvious that `Task2` will have left the protected objects entry before `Task3` will call it. However, this information

```
1  protected body Critical is
2     procedure Open is
3     begin
4        work(1, 1);
5        Barrier := True;
6     end Open;
7
8     entry Call when Barrier is
9     begin
10       work(2, 3);
11    end Call;
12 end Critical;
13
14 task body Task1 is
15 begin
16    Critical.Open;
17 end Task1;
18
19 task body Task2 is
20 begin
21    Critical.Call;
22 end Task2;
23
24 task body Task3 is
25 begin
26    delay until 10.0;
27    Critical.Call;
28 end Task3;
```

Fig. 7. A small system with a protected object and three tasks of which two contend for the access to the object. Outlines of the temporal skeletons are superimposed over the source code.

will not be obvious to a model without time. Such a model has no way in which the two calls to the procedure can be separated, i.e., the temporal behaviour the execution times add is needed for a non-pessimistic verification of this property.

B. Transformation into timed automata

We start by creating an automaton for the protected entry, `Call`, for the protected object, `Critical`. Any automaton we can construct with the building blocks presented above starts with the either a task initiation building block or with a block for protected object body code. The protected entry starts with the latter and within that block an execution block is used for the function call (`work`). The resulting automaton is shown in Figure 8.

The next automaton we create is that of `Task3`. `Task3` is started with a task initiation building block, followed by a delay block, a block to make a protected object entry call, and finally a block for suspending the automaton. There blocks are expanded to the timed automaton shown in Figure 9. The transitions between locations n_0 and n_2 are used for the task initiation, locations between n_3 and n_6 implements the delay, the call to the protected entry

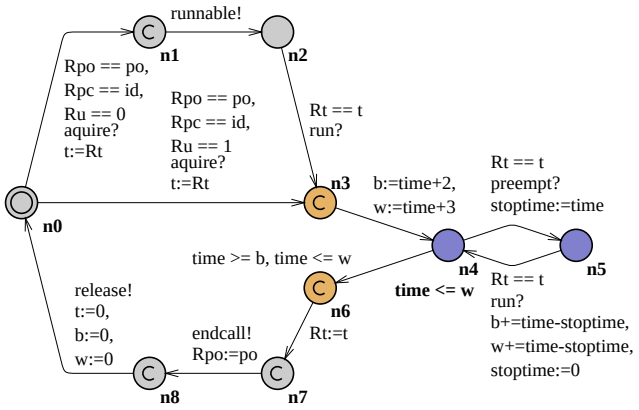

Fig. 8. The automaton for the protected entry `Critical.Call` of Figure 7.

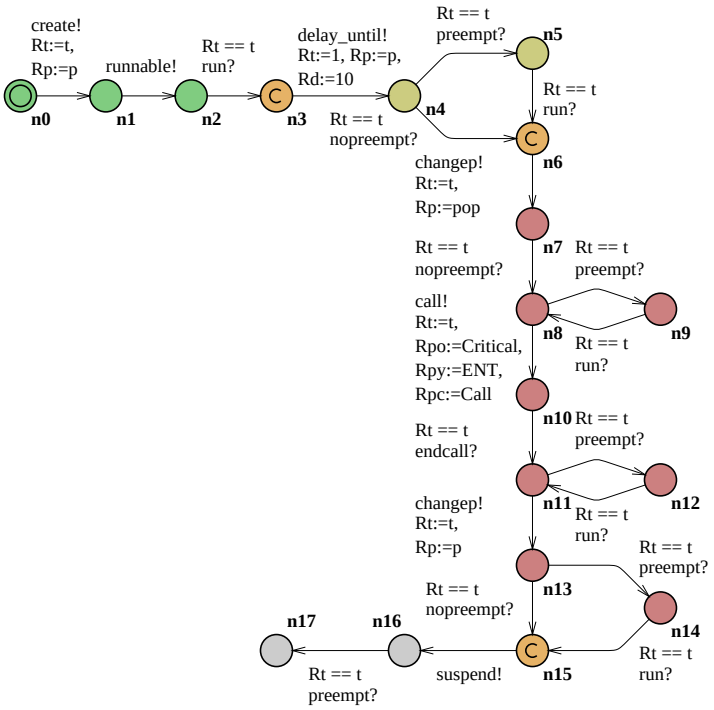

Fig. 9. The automaton for `Task3` of Figure 7.

is made by the ones between n_6 and n_{16}, and the final transitions, between locations n_{16} and n_{18}, suspends the task. Even in the final model the distinctive blocks are visible.

The other two tasks of Figure 7, `Task2` and `Task1` can be created by removing or changing blocks in `Task3`. `Task2` is created by fusing location n_3 with n_6 and removing all transitions in between, thus creating a automaton which does not make a delay. `Task1` looks like `Task2` but on transition $n_8 \longrightarrow n_{10}$ it calls `Critical.Open`, not `Critical.Call`.

The verification model for the application is composed of the protected object subroutines `Critical.Open` and `Critical.Call`, the tasks `Task1`, `Task2`, `Task3`, and automata for the RTK. The RTK automata needed model the Ready Queue, the Delay Queue, protected objects, processing units and their null tasks, and the system clock.

C. Verification

Interesting for the task at hand, i.e., the investigation if the two application tasks codewTask2 and codewTask3 can be queueing to call the entry at the same time, will be the property of the RTK automaton managing the protected object, the protected object queue. The queue is designed to enter an error location, n_8, if it detects that a task attempts to queue for an entry another task is already queueing for. Verification of the property in an application system can in Uppaal be made using the query $\exists\Diamond(PO.n8)$.

In general, the tool allows verification of *a*) reachability of states, *b*) global properties, like that certain conditions never occur during execution, and *c*) deadlocks. Transitions can be attributed with counters to gain detailed information about possible executions, e.g., the number of times a specific transition can or will be be used.

Other temporal properties that can be verified are, e.g.,

- if a specific building block can be, or always will be, reached before or after a given time
- how many times a specific task will do something, like being preempted, loop, etc.

V. Conclusions and Future Work

This report presents a set of building blocks that can be used to describe a tasking application. The report describes how temporal skeletons can be translated into the blocks in a straight forward way. Temporal properties of the skeleton can, together with a model of a real-time kernel, be formally verified and it is shown show how the highly elusive dynamic restriction of the Ravenscar tasking profile can be verified.

References

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

[2] G. Naeser, "A Real-Time Kernel for Ravenscar", MRTC report ISSN 1404-3041 ISRN MDH-MRTC-186/2005-1-SE, Mälardalen Real-Time Research Centre, 2005.

[3] 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.

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

[5] G. Naeser, K. Lundqvist and L. Asplund, "Temporal Skeletons for Verifying Time", *In Proceedings of SIGAda 2005*, ACM, 2005.

[6] A.J. Wellings, B. Johnson, B. Sanden, et al., "Integrating Object-Oriented Programming and Protected Objects in Ada 95", ACM Transactions on Programming Languages and Systems, 1999