TIMING CONSTRAINTS VALIDATION USING UPPAAL

Schedulability Analysis

Hongyan Sun

This paper presents an approach that formally models real-time tasks and scheduling strategies in terms of timed-automata, and then formalizes timing constraints of tasks into reachability properties, which, thus, can be validated by using model checking tool Uppaal. This approach is detailed through two pre-emptive priority-driven scheduling strategies: rate monotonic priority assignment and priority ceiling protocol.

1. Introduction

Real-time systems often have to satisfy *hard* real-time constraints, i.e. such a system consists of tasks have to be completed within strict time constraints. These time constrains are also called *deadlines*. If the deadlines are not satisfied, it can cause catastrophic consequences (Burns and Wellings, 1995).

Scheduling tasks so that real-time constraints are met is an important and active issue in real-time systems, and many scheduling algorithms and schedulability analyses have been established. However, the underlying computation model is usually restricted to a simple cyclic task model. Since most schedulability analyses identify only sufficient conditions, a given set of tasks may be schedulable even through they do not satisfy any of the known schedulability conditions (Burns, 1991). Schedulability analysis is an extremely hard problem, even when the execution times of all tasks are precisely known (Balarin *et al.*, 1998). In addition, most schedulability analysis methods are separated from the system (e.g. task, scheduler, and etc.) design, so that the schedulability analysis results may not reliable. This is much likely the case when more communications are involved in the system, e.g. in a distributed architecture (Kopetz, 1997), which the embedded systems often requires.

The original version of this chapter was revised: The copyright line was incorrect. This has been corrected. The Erratum to this chapter is available at DOI: 10.1007/978-0-387-35409-5_23

B. Kleinjohann (ed.), *Architecture and Design of Distributed Embedded Systems*

In this paper, the schedulability analysis is handled in a way that allows also validating the other system properties, e.g. *correctness*, while validating timing constraints, at the early stage of the system development. It formally models tasks and scheduling strategies using timed-automata (Alur and Dill, 1991), formalizes the timing constraints of tasks as reachability properties, and then uses model checking tool Uppaal (Larsen *et al.*, 1997) to verify the properties. This approach is discussed and illustrated through two pre-emptive priority-driven scheduling strategies: *rate monotonic priority assignment* (Liu and Layland, 1973) and *priority ceiling protocol* (Liu *et al.*, 1990).

Model checking tool Uppaal has been successfully applied to several time-critical systems, e.g. (Iversen *et al.*, 2000) and (Hune *et al.* 2000). It provides: a description language to describe or model system behaviors as networks of timed-automata; a model checker to check reachability properties by exploring the state-space; and a simulator to visualize the execution traces of a system.

The remainder of the paper is organized as follows: In the next section, it focuses on the *rate monotonic priority assignment* scheduling strategy and discusses how to model a task and a scheduler in terms of timed-automata, it shows how the timing constraints are formalized as a reachability property such that it can be verified using Uppaal. Section 3 focuses on the same issues but based on the *priority ceiling protocol.* Several sets of tasks are tested and the test results are respectively shown in both sections. The last section, section 4, presents some concluding remarks.

Acknowledgements

The author wishes to thank Associate Professor Hans Henrik Løvengreen and Associate Professor Hans Rischel at the Department of Information Technology, Technical University of Denmark, for their inspirations and valuable suggestions.

2. Rate Monotonic Priority Assignment

The *rate monotonic priority assignment* (or *rate monotonic* for short) strategy assigns a task priority according to the period of the task, such that the shorter the period, the higher the priority. It assumes that:

- Each task is periodic
- Each task has a deadline equal to their period
- Each task is independent
- Each task has a fixed worst-case computation time

2.1. Task Model

Figure 1 shows a task model in terms of timed-automata (supported by Uppaal), under the assumptions given above. Where, a task is modelled, once it initiates from the initial location **Init**, as an infinite loop that can be in any one of the locations:

Suspended, **Ready**, and **Running**. The **Suspended** location corresponds to a task that has not released yet. A task is in **Ready** when it can execute but its priority may be less than the current running task. A task is in **Running** when it has control of the CPU.

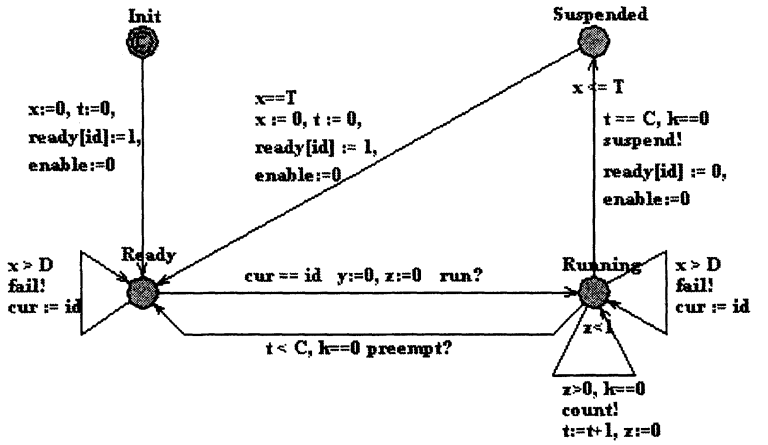

Figure 1. Task model

In Figure 1, T is the period of a task, D the deadline of a task, and C the worst-case computation time. x, y and z are clocks. Integer variable t is used as a counter to count how many CPU-time units have been used for a task, and it is introduced because of the syntax of Uppaal. For the same reason, the channel *count* and the integer variable k are introduced. A task is identified by its priority *id*. The priority is assigned, in this case, according to *rate monotonic priority assignment*.

Each task is an instantiation of the task model giving the values to a set of parameters (*id, T, C, D*).

A task releases itself by raising the flag *ready*. It synchronizes with the scheduler through communication channels *preempt*, *run* and *suspend*. The integer variable *cur* denotes the current running task, and the integer variable *enable* shows whether the current highest priority task is valid or not.

The *fail* channel is used to synchronize with a failure *Observer*, which is used for the verification purpose.

2.2. Scheduler

The *rate monotonic* scheduling is of the pre-emptive priority-driven scheduling. The scheduler is then modelled as show in Figure 2.

The model contains three locations, **Idle**, **Select**, and **Run**. Initially, when no task is ready, the scheduler is in the location **Idle**. It transits to the **Select** location once the highest priority task of the currently ready tasks is valid. It then selects that task

to run, i.e. it is in the location **Run** after the synchronization with the selected task over channel *run*.

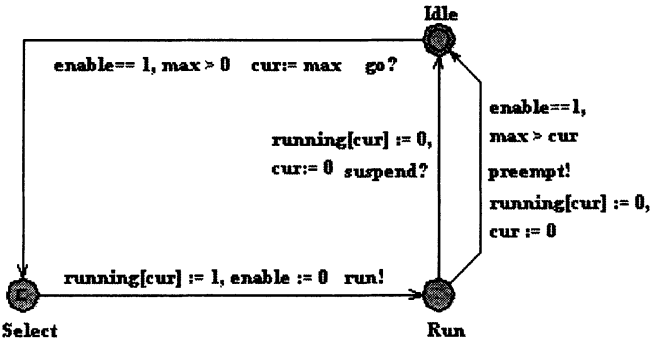

Figure 2. Scheduler model

In the location **Run**, if there is another higher priority task ready (*max* > *cur*), the scheduler will synchronize with the currently running task, which is then pre-empted, over channel *preempt*, and return to the **Idle** location in order to make a new selection. If the currently running task completes its computation, the scheduler will notice that through channel *suspend* and then returns to the **Idle** location.

The flag *running* indicates whether a task is running or not. The integer variable *max* denotes the highest priority task that is ready.

2.3. Timing Constraints Validation

In order to validate timing constraints, an *Observer* process is employed and it is modelled as show in Figure 3.

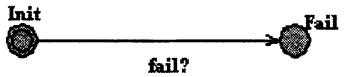

Figure 3. Observer model

The *Observer* synchronizes with the currently running or ready task over the *fail* channel and transits to the **Fail** location, once the deadline of a task is missed.

Thus, the timing constraints property becomes the reachability property. Formally, this reachability property $\wp$ is formalized in terms of Uppaal verification notions as:

$\wp$: *A*[] (not *Observer.Fail*)

It says that it is *always* the case that the Observer is *not* in the **Fail** location. This property can then be verified using Uppaal.

2.4. Tests

Three sets of tasks, shown in Table 1 to 3, are taken from (Burns and Wellings, 1995) as a comparison with the classical method. These three sets of tasks are verified against the property $\wp$ using the model checker of Uppaal. The dynamic behaviours of the system are examined using the simulator of Uppaal.

In the tables, *id* denotes the priority of a task, *T* denotes the period of a task, *C* the computation time and *D* the deadline.

Table 1. The first set of tasks

	Id	T	C	D
P1	3	30	10	30
P2	2	40	10	40
P3	1	50	12	50

Table 2. The second set of tasks

	Id	T	C	D
P1	3	16	4	16
P2	2	40	5	40
P3	1	80	32	80

Table 3. The third set of tasks

	Id	T	C	D
P1	3	20	10	20
P2	2	40	10	40
P3	1	80	40	80

The results show that for the first set of tasks, the property $\wp$ is not satisfied, i.e. a deadline is missed. For the last two sets of tasks, the property $\wp$ is satisfied, i.e. no deadline is missed.

3. Priority Ceiling Protocol

In the *rate monotonic* scheduling strategy, it is assumed that tasks are independent. When tasks require accessing data or synchronizing via protected shared resources, the potential for blocking is introduced, i.e. a task can be prevented from accessing a resource by another task that has already locked the same resource. Thus, it will introduce the uncontrolled *priority inversion* problem, i.e. the higher priority task is blocked by the lower priority task, which is not desirable.

Priority ceiling protocol is developed to make this problem more controllable, i.e. to reduce the worst-case task blocking time to at most the duration of execution of a single critical section of a lower priority task.

The basic idea of this protocol is to assign a *priority ceiling* to each semaphore that protects a critical section. The priority ceiling of each semaphore is equal to the highest priority task that may use this semaphore. A task can access a critical section only if its priority is higher than all priority ceilings of all the semaphores locked by other tasks, otherwise it is blocked and the blocking task will inherit its priority.

This scheduling strategy is modelled using four primary models, *task* model, *priority control* model, *scheduler* model, and *semaphore* model, which will be discussed in the following subsections.

3.1. Task Model

The task model is similar to the one in Figure 1, but it allows the different initiation time and dependent tasks, i.e. the task can access its critical sections when execution. Figure 4 shows only a part of the model, and the complete model is given at http://www.it.dtu.dk/~hs/DIPES.

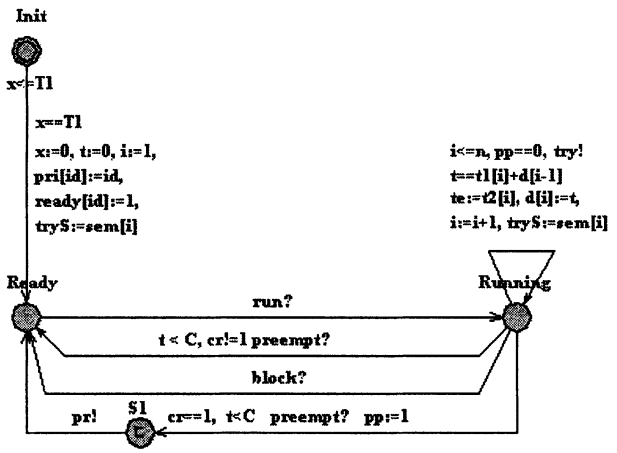

Figure 4. A part of the task model

When a task is in the location **Running**, it may try to access its critical section by synchronizing over channel *try* with the *priority control* process, once the instant of accessing the critical section is reached. If it has not high enough priority, it is blocked, which is informed by the scheduler over channel *block*, and then returns to the **Ready** location. The integer variable *cr* indicates whether the task is in the critical section or not, and *pp* indicates whether the task is accessing the critical section or not when it is preempted. If the task is preempted while in the critical section, it will at the same time inform the *priority control* process through the synchronization channel *pr*.

Each task is an instantiation of the task model giving the values to a set of parameters (*id, T, C, D, T1, n, t1, t2, sem*). Where, *id*, *T*, *C*, and *D* are respectively

the priority, the period, the computation time and the deadline of a task. *T1* denotes the initial release time of a task. The integer variable *n* indicates how many critical sections a task is going to access. Arrays *t1*, *t2*, and *sem* are of size *n* to provide the information about accessing these *n* critical sections. For $1 \leq i \leq n$, *t1*[*i*] gives the instant that the *i*th critical section will be accessed relative to *t1*[*i-1*]. By adding time delay *d*[*i*], it gives an absolute time. *t2*[*i*] gives the interval that the *i*th critical section executes, and *sem*[*i*] indicates which semaphore it will lock in order to access the *i*th critical section. The integer variable *i* indexes these arrays. The variable *pri* denotes the priority of a task. The original priority of a task is equal to its *id*. The integer variable *tryS* indicates which semaphore a task is going to lock.

3.2. Priority Control

The *priority control* process controls tasks such that a task should have a high enough priority in order to access its critical section according to the protocol. Figure 5 shows a part of the model. The complete model is given at http://www.it.dtu.dk/~hs/DIPES.

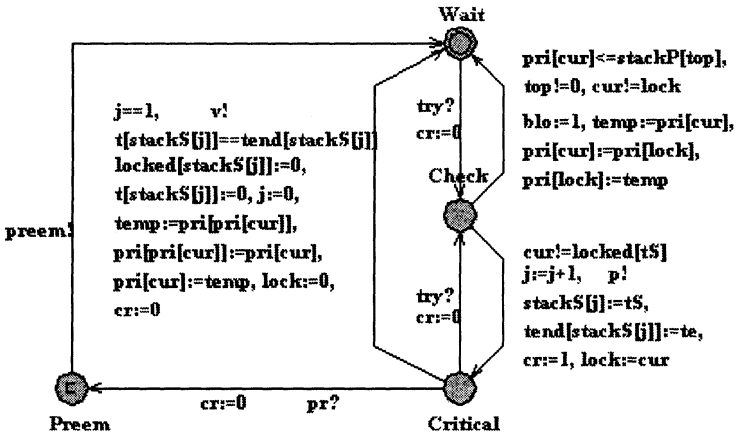

Figure 5. A part of the priority control model

In Figure 5, the model contains four locations, **Wait**, **Check**, **Critical**, and **Preem**. When the *priority control* process is in the location **Wait**, it will synchronize with the task process over channel *try* if that task wishes to access a critical section. It will then check if that task has a high enough priority in the location **Check**. It raises the flag *blo* if the task has not high enough priority, so that the task will be blocked and the blocking task will inherit the priority of the blocked task. This priority inheritance is done through three assignments: *temp*:=*pri*[*cur*], *pri*[*cur*]:=*pri*[*lock*], *pri*[*lock*]:=*temp*. Where, *pri*[*cur*] is the priority of the blocked task, *pri*[*lock*] is the priority of the blocking task. It allows the task to lock the semaphore protecting the desired critical section via the synchronization with the semaphore process over channel *p*, if the task has a high enough priority,

The *priority control* process is in the location **Critical** when the task is in the critical section. It transits to the **Preem** location if a higher priority task pre-empts

the current running task, and at the same time it informs the corresponding semaphore process over channel *preem*. It informs the corresponding semaphore process over channel *v* when the task completes its critical section, and the task resumes then its original priority. It allows a task to access a nested set of critical sections via the synchronization over channel *try* in the location **Critical**.

The integer variable *j* points to the top of stack *stackS*, which contains all the locked semaphores. For the semaphore *stackS*[*j*], *t*[*stackS*[*j*]] denotes how long its protected critical section has been executed, and *tend*[*stackS*[*j*]] denotes the time that critical section will be completed. *locked*[*stackS*[*j*]] indicates which task locks the semaphore.

3.3. Semaphore

A semaphore protecting a critical section is modelled as shown in Figure 6.

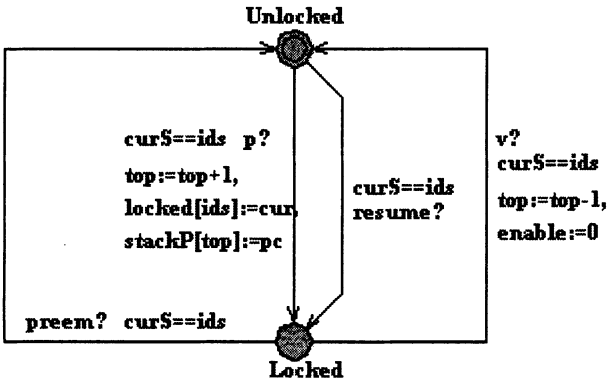

Figure 6. Semaphore model

Each semaphore process is an instantiation of the semaphore model giving the values to a set of parameters (*ids, pc*), where *ids* is the identifier of a semaphore, and *pc* is the priority ceiling assigned to a semaphore.

When a semaphore is not locked, it is in the location **Unlocked**. It transits to the location **Locked** if the *priority control* process allows a task to obtain the lock on it (synchronization either over *p* channel or *resume* channel). The *resume* channel is used when the task was pre-empted after it had the lock, and now it inherits a higher priority and resumes its lock. A stack *stackP* contains all the priority ceilings of the semaphores that are locked currently. The integer variable *top* points to the top of that stack.

In the location **Locked**, the semaphore will synchronized with the *priority control* process over channel *preem* if the task is pre-empted while in the critical section. It synchronizes with the *priority control* process over channel *v* when the task completes its critical section.

3.4. Scheduler

The scheduler model is similar to the one in Figure 2, except that it synchronizes with the currently running task process over *block* channel, in the location **Run**, when the task is blocked. Detailed model is given at http://www.it.dtu.dk/~hs/DIPES.

3.5. Timing Constraints Validation

The same *Observer* as shown in Figure 3 is employed to verify the property $\wp$ by using Uppaal.

3.6. Tests

Figure 7 shows two sets of tasks, which are verified against the property $\wp$ using model checker of Uppaal. The system behaviours are also simulated using Uppaal. The results show that both sets of tasks satisfy the timing constraints.

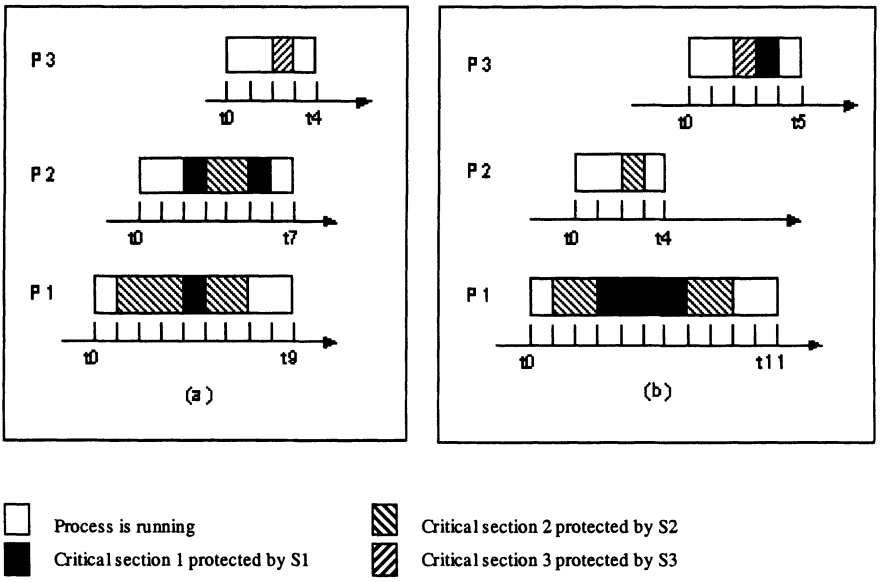

Figure 7. Two tested sets of tasks

In both case (a) and (b), task *P3* has the highest priority and *P1* has the lowest. *P1* initiates first and *P3* initiates the latest. The period of all the tasks is *20* time units. The deadline of *P3*, *P2*, and *P1* is respectively *15*, *18* and *20* time units.

In the case (a), both *P1* and *P2* will access critical section *1* and *2*. *P3* will access critical section *3*. *P1* will lock *S2* and enter critical section *2* after running one time unit. While in critical section *2* for three time units, it will lock *S1* and access critical section *1* for one time unit. After unlocking *S1*, it continues with critical section *2* for another two time units and then unlocks *S2*. Finally, after running another two time

units, it will finish one computation cycle. The computation time of *P1* is *9* time units.

Similarly, *P2* will first lock *S1* then *S2*, and unlock *S2* then *S1* as shown in the figure. The computation time of *P2* is *7* time units. *P3* will lock *S3* and access critical section 3 for one time unit and then unlock *S3*. Its computation time is *5* time units.

In the case (b), *P1* will access critical section *1* and *2*, *P2* critical section *2*, and *P3* critical section *1* and *3*. Again, *P1* will first lock *S2* then *S1*, and unlock *S1* then *S2*. *P2* will only lock *S2* for one time unit and then unlock *S2*. *P3* will lock *S3* first for one time unit, and after unlocking *S3* it will lock S1 for one time unit and then unlock *S1*. The computation time for *P1*, *P2*, and *P3* is respectively *11*, *4*, and *5* time units.

For both cases, the property $\wp$ is satisfied as the result of using the model checker of Uppaal.

4. Conclusion

This paper uses two scheduling strategies, *rate monotonic priority assignment* and *priority ceiling protocol*, as examples to discuss and illustrate a formal approach for schedulability analysis by means of model checking tool Uppaal. This approach allows also validating other properties of, e.g. *correctness*, while validating timing constraints of real-time tasks. In addition, the restrictions on the task model (as in most classical schedulability analyses) can be relaxed, e.g. it allows different deadlines and initial release instants as shown in the paper. It is possible to allow also different or concrete task models.

At the present stage of this work, the *correctness* of tasks and scheduling strategies is validated using the simulator to visualize traces of the system behaviours. It could be formalized as certain properties that can be verified using the model checker of Uppaal.

References

Alur, Rajeev and Dill, David (1991) "The Theory of Timed Automata", *Real-Time: Theory in Practice*, LNCS 600, 45-73.

Balarin, F., Lavagno, L., Murthy, P. and Sangiovanni-Vincentelli, A. (1998). "Scheduling for Embedded Real-Time Systems", *IEEE Design and Test Computers*, 15(1-3), 71-82.

Burns, Alan and Wellings, Andy (1995). *Real-Time Systems and Programming Languages*, Addison-Wesley.

Burns, Alan (1991). "Scheduling Hard Real-Time Systems: A Review", *Software Engineering Journal*, May, 116-128.

Kopetz, Hermann, (1997). *Real-Time Systems - Design Principles for Distributed Embedded Applications*, Kluwer Academic Publishers.

Larsen, Kim G., Pettersson, Paul and Wang, Yi (1997). "Uppaal in a Nutshell", *Journal on Software Tools for Technology Transfer*, 1(1-2), December, 134-152.

Liu, C. L. and Layland, J. W. (1973). "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment", *Journal of the Association for Computing Machinery*, 20(1), 46-61.

Liu, Sha, Rajkumar, Ragunathan and Lehoczky, John P. (1990). "Priority Inheritance Protocols: An Approach to Real-Time Synchronization", *IEEE Transactions on Computers*, 39(9), 1175-1185.

Hune, Thomas, Larsen, Kim G. and Pettersson, Paul (2000). "Guided Synthesis of Control Programs Using Uppaal", *Proc. of the IEEE ICDCS International workshop on Distributed Systems Verification and Validation*, April, E15-E22.

Iversen, Torsten K., Kristoffersen, Kaere J., Larsen, Kim. G., Laursen, Morten, Madsen, Rune G., Mortensen, Steffen K., Pettersson, Paul and Tomasen, Chris B. (2000). "Model-Checking Real-Time Control Programs – Verifying LEGO Mindstorms Systems Using Uppaal", *Proc. of 12th Euromicro Conference on Real-Time Systems*, June, 147-155.