Schedule Validation for Embedded Reactive Real-Time Systems

Felice Balarin
Cadence Berkeley Laboratories

Alberto Sangiovanni-Vincentelli
Department of EECS
University of California at Berkeley

Abstract

Task scheduling for reactive real time systems is a difficult problem due to tight constraints that the schedule must satisfy. A static priority scheme is proposed here that can be formally validated. The method is applicable both for preemptive and non-preemptive schedules and is conservative in the sense that a valid schedule may be declared invalid, but no invalid schedule may be declared valid. Experimental results show that the run time of our validation method is negligible with respect to other steps in system design process, and compares favorably with other methods of schedule validation.

1 Introduction

There is no universally accepted formal model for embedded systems, or even a universally accepted definition of which systems are considered embedded. In our approach, a system is considered embedded if it has the following characteristics:

reactive: We consider systems consisting of many tasks which are executed in reaction to some external events, or to some other tasks.

single-processor: We assume that all tasks are executed on the same processor. Although definitely not universal, this assumption is true of many embedded systems in practice, particularly for low-cost high-volume systems like consumer products.

DAC 97, Anaheim, California
(c) 1997 ACM 0-89791-920-3/97/06 ..$3.50

real-time: For the system to perform its function, the tasks must implement the correct functionality, *and* they have to be executed in a timely manner.

We address a question whether an embedded system performs its tasks in a timely manner. The answer to this question depends on timing properties of tasks and external events, as well as the scheduling policy: i.e. the set of rules determining which task to execute, if more than one task is ready to run. Consequently, we refer to this problem as *schedule validation*. Currently, the most used methods in answering this question are simulation and prototyping. These approaches have an obvious weakness that only a few of an infinite number of input patterns can be tried, and thus the correctness of the system can never be guaranteed.

In this paper, we propose a formal method based on a mathematical analysis of the schedule that is independent of input patterns in the test set. Ideally we would like to be able to show that the schedule is valid if and only if our method says so. However, (as detailed below) achieving this goal either implies complexity that is beyond reach with present day computers or ignoring some important characteristics of reactive real-time systems in the attempt to simplify the analysis. We settle for a conservative analysis, i.e., a method that guarantees that if the schedule passes the test then it is correct. On the other hand, if the schedule does not pass the test then the schedule may still be correct. In this way, simulation and prototyping on one side, and the formal method proposed in this paper are complementary: the former methods can be used to disprove the validity of the schedule, while the latter can be used to prove it.

We represent a system as a network of *tasks*, finite-state objects that execute asynchronously and communicate through production and consumption of *events*, but with functional information abstracted and timing information added.[1] We believe that our model is a useful timing abstraction of many discrete-event formalisms.

Our formalization of the correctness criterion arose from the case study in verification of embedded systems [2]. It was observed there that timing requirements can often be concisely expressed by the assumption that certain events

[1]This formal model is inspired by the model used in hardware/software co-design system POLIS [5].

are never dropped, i.e. that a produced event is always consumed before another event of the same kind is produced again. In this way, verification of timing and functionality can be cleanly separated, and the most suitable method can be applied to each.

This paper is organized as follows: We first present a formal model of computation in section 3. A method for validating preemptive schedules is presented in section 4, and extended to non-preemptive schedules in section 5. Experimental results are presented in section 6, and finally some conclusions and ideas for future work are given in section 7.

2 Related work

The schedule validation problem can be expressed as a standard verification problem in a formalism that combines timing information with finite-state systems. For example, Balarin *et al.* [3] formulated the schedule validation problem as a reachability problem for timed automata. This approach has the advantage of being exact (as opposed to conservative like the method proposed here), but it suffers from severe limitations due to its computation complexity: Balarin *et al.* report execution times for small systems that are several orders of magnitude larger than those obtained by the method proposed here.

A variety of models have been considered over past several decades by researchers in the real-time systems community. Most of them are extensions or modifications of the one introduced by Liu and Layland [7]. Audsley et. al. [1] proposed a schedule validation method for such systems. Harbour et. al. proposed a schedule validation method for a generalization of this model where every independent task can consist of a chain of subtasks [6].

The class of systems for which Audsley's method is applicable is strictly contained by the class of systems for which Harbour's method is applicable, which in turn is strictly contained by the class of systems for which the method in this paper is applicable. Only our method allows modeling components that can be executed in reaction to various (internal or external) events in their environment. This feature is crucial for realistic modeling of reactive real-time systems.

For systems for which all three methods are applicable, all three methods are the same. Similarly, for systems for which both Harbour's and our methods are applicable, they both compute the same result.

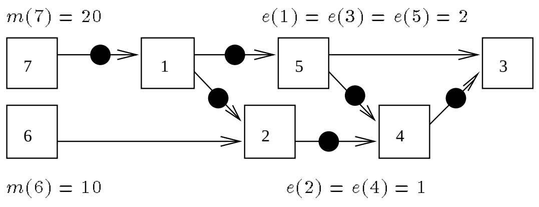
3 Model of Computation

We model a system as a collection of tasks with known priorities and execution times. Tasks are enabled either by external events or by execution of other tasks (internal events). Among the enabled tasks the one with the highest priority is executed. We assume that there exists some known minimum time between two occurrences of external events. A system is correct if certain critical events are never "dropped", i.e. if tasks enabled by these events are executed before another event of the same kind occurs.

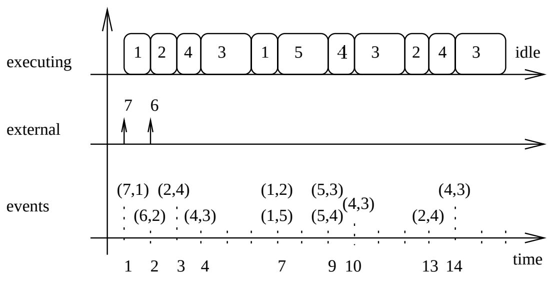
Formally, a *system* is a 6-tuple (T, e, U, m, E, C) where

- $T = \{1, 2, \ldots, n\}$ is a set of *internal task identifiers.* We assume that the identifier also indicates tasks priority and that larger identifier indicates higher priority. Since tasks are identified by their priorities, they must be unique. [2]

[2] Allowing multiple tasks to have the same priority would require only minor adjustments to our method, but it would obscure the notation without adding much substance, so we decided not to allow it.



a)



b)

Figure 1: A system (a), and its possible execution (b). Executions of external tasks are represented by upward arrows, while executions of internal tasks are represented by rounded boxes.

- $e : T \mapsto \mathbb{R}$ (where $\mathbb{R}$ is the set of positive real numbers) assigns to every internal task its *execution time.*
- U, such that $U \cap T = \emptyset$, is a set of unique *external task identifiers.* Intuitively, these tasks represent the environment. We are really only interested in occurrence of external events, but for the uniformity of presentation, it is convenient associate external events with tasks that generate them.
- $m : U \mapsto \mathbb{R}$ assigns to every external task *minimum time between two executions.*
- $E \subseteq (T \cup U) \times T$ is a set of *events.* Intuitively, $(i, j) \in E$ indicates that the execution of task i enables task j. Note that only internal tasks can be enabled in this way. We say that the graph with nodes $T \cup U$ and edges E is the *system graph.* We consider only system that have acyclic system graph. If i is an external task, then we say that (i, j) is an external event. Otherwise, we say that (i, j) is an internal event.
- $C \subseteq E$ is a set of *critical events.*

Consider, for example, a system in Figure 1a. It is a simplified version of shock absorber controller whose purpose is to set the hardness of vehicle's shock absorbers based on the inputs from various sensor (speed, vertical acceleration, steering angle, ...). In this simplified model, the system receives inputs from two external tasks: task 7 emits events indicating wheel motion (once every wheel revolution), and task 6 indicates the vertical acceleration. Tasks 1 and 5 compute the speed, tasks 2 and 4 check for error conditions, while task 3 chooses an appropriate setting based on this information. More precisely, task 1 does some initial processing of data and indicates to task 2 whether the vehicle is standing or moving. Task 2 uses this information and vertical acceleration data to check for errors (the vertical acceleration should be low while the vehicle is standing still). This information is passed to task 4 which also receives computed speed information from task 5, does some additional checking and makes the final determination whether the speed sensor is operating correctly. Six events are critical, as indicated by black dots. For example, missing a pulse from the wheel (event (7,1)) might cause a wrong speed calculation. On the other hand, it is important to keep track of current vertical acceleration, but missing an occasional reading does not compromise overall functionality.

An *execution* of a system is any sequence of timed events that satisfies the following:

- external tasks can execute at any time, as long as the time between two executions of any external task i is larger or equal than $m(i)$,
- immediately after the execution of task i, all the tasks j such that $(i,j) \in E$, are enabled, and task i is disabled,
- if a task i becomes enabled at some time t_1, then it will become disabled (i.e. it will terminate its execution) at time t_2 such that the total amount of time in the interval $[t_1, t_2]$ in which i has the highest priority among enabled tasks is $e(i)$.

An event (i,j) is said to be *dropped* if after i is executed, task i is executed again before task j is executed. An execution is *correct* if no critical events are dropped in it. A system is correct if all of its executions are.

A possible execution of the system in Figure 1a is shown in Figure 1b. We invite the reader to verify its correctness.

4 Validation of Preemptive Schedules

To show that a system is correct it suffices to show that for every critical event (i,j) the minimum time between two executions of i is larger than the maximum time between executions of i and j. Even though this may be too conservative (because the two may impose conflicting constraints on execution of other tasks), it is still not easy to compute. Therefore, we pursue the following approach:

1. We restrict our attention to a class of systems in which only external events can be dropped. The minimum time between two executions for these events is given by system's description.
2. We conservatively estimate the maximum time between executions of i and j.

In this paper, we only provide relevant results and intuition behind them. Formal proofs can be found in [4].

4.1 Analysis of internal tasks

The following proposition provides a simple sufficient condition to ensure no internal events can be dropped:

Proposition 1 *If $i < j$, then event (i,j) cannot be dropped.*

Indeed, event (i,j) enables task j, and since it has higher priority, it must be executed before i is executed again. We will generalize this reasoning to characterize a wider class of systems in which only external events can be dropped, but first we need some additional definitions.

A pair (F,N) of disjoint subsets of tasks is an *exclusive neighborhood* of some internal task i, if F and N satisfy the following conditions:

C1: $i \in F \cup N$,

C2: if k is in N, then all of its predecessors are in $F \cup N$ (i.e. $\forall j,k: \ ((k \in N) \wedge ((j,k) \in E)) \Longrightarrow (j \in F \cup N)$),

C3: every task in $F \cup N$ except i has a unique successor in N (i.e. $\forall k \in F \cup N - \{i\} \exists_1 j \in N: \ (k,j) \in E$), and i has no successors in N,

C4: $k < j$ for every $k \in F$ and every $j \in N$.

We say that F is the *frontier* and N is the *interior* of an exclusive neighborhood.

Intuitively, conditions **C1**–**C3** specify that an exclusive neighborhood is the portion of a system graph obtained by traversing backwards from i and cutting the traversal at frontier nodes. In addition, condition **C3** requires that portion of the system graph to be a tree. Finally, condition **C4** requires requires tasks at the frontier to have lower priorities than tasks in the interior. For example, in Figure 1, task 4 has an exclusive neighborhood with frontier $\{1,2\}$ and interior $\{4,5\}$.

The name "exclusive neighborhood" is chosen to suggest that at most one task can be enabled in that part of a system graph. An exclusive neighborhood does not always exist, but if it does, it can significantly simplify schedule validation, due to the following:

Theorem 1 *If $(i,j) \in E$, and (F,N) is an exclusive neighborhood of task i, such that:*

$$k < j \text{ for all } k \in F \ , \tag{1}$$

then (i,j) cannot be dropped.

Proposition 1 is just a simple corollary of Theorem 1, because $(\{i\}, \emptyset)$ is an exclusive neighborhood of i that satisfies (1).

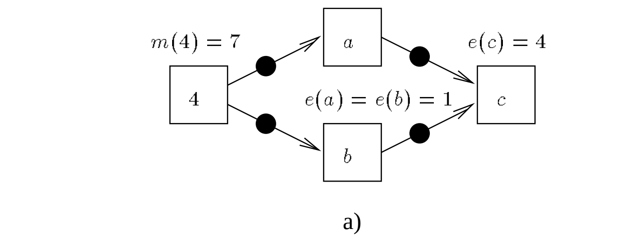
There are two possible applications of Theorem 1: analysis and synthesis. The analysis problem is to check whether some critical event can be dropped, given a system with assigned priorities. To check whether Theorem 1 applies to some event (i,j) it suffices to perform a backward breadth-first search of a system graph starting from i. The search is not continued beyond any task with probability less than j. If at any time some task is reached for the second time (violating **C3**), or an external task is reached (violating **C4**), then the search is terminated with failure. In this case, results are inconclusive: it may or may not be possible for (i,j) to be dropped. However, if the search is terminated because there are no more unexplored nodes with priority larger than j, then an exclusive neighborhood satisfying (1) is found, and we can conclude that (i,j) can never be dropped.

The synthesis problem is to assign priorities in a way that no critical internal events are dropped. There are many ways of doing it, but Proposition 1 suggests a priority assignment policy that is particularly easy to implement.

In the rest of this paper, we will consider only systems in which for every critical event $(i,j) \in C$ there exits an exclusive neighborhood satisfying (1). It is easy to check that the system in Figure 1 is such a system.

Unfortunately, this simplification in analysis comes at a price of optimality. Consider, for example, the system in Figure 2a. The system consists of one external task (4), there internal tasks (a,b,c), and all events are critical.

The only way to apply Theorem 1 to internal tasks is to assign to c a higher priority than to a and b, e.g. $a =$



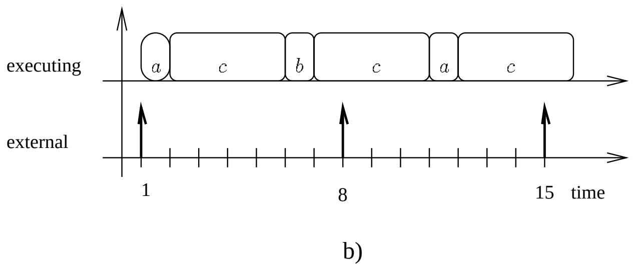


Figure 2: An example showing that having exclusive neighborhoods can be sub-optimal (a), and an error trace demonstrating it (b).

1, $b = 2$, $c = 3$ (the other case, $a = 2$, $b = 1$, $c = 3$, is symmetrical). However, in this case event $(4, b)$ can be dropped, as shown by the error trace in Figure 2b, where external task 4 executes at time 8 and then again at 15 without b executing in the meantime.

On the other hand, if c is assigned lower priority than a and b, e.g. $a = 3$, $b = 2$, $c = 1$, then Theorem 1 is no longer applicable, but no events are lost in this case. This follows from the simple observation that any execution of 4 when processor is idle, will be followed by executions of a, b and c (in that order), requiring the total of 6 time units. Since that is less than $m(4)$, it follows that the processor will always be idle when 4 is executed. Unfortunately, it is not known presently how to generalize this ad-hoc reasoning to arbitrary systems. Until such a generalization is discovered, we feel it is well justified to trade off optimality with predictability of behavior.

4.2 Analysis of external events

To check whether an external event (i, j) can be dropped, we need to check whether the execution of j can be delayed for more than $m(i)$ time units.

There are two possible sources of delay of execution of j: those due to tasks that are already enabled at the time i is executed, and those due to tasks enabled (possibly indirectly) by execution of some external tasks after j is enabled. Our analysis will proceed along the following lines:

1. Delays from different sources are computed independently and then summed, to get a bound on the actual worst case delay.
2. We limit the delay contribution from already enabled tasks by observing that in the worst case only a subset of tasks can be enabled (see (2)).
3. We limit the delay contribution from external task by observing that they can be executed only a limited number of times in a given time interval (see (3)).

In the rest of this section we will consider a typical execution of the system (sketched on Figure 3), where:

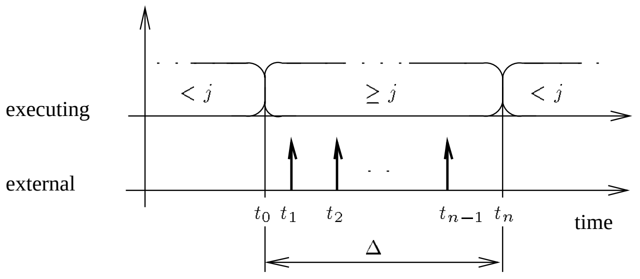


Figure 3: A typical execution.

- at time t_n task j completes the execution, and thus it is disabled,
- t_0 is the earliest time such that between t_0 and t_n no tasks with priority less than j are executed,
- in the interval $[t_0, t_n)$, external tasks can execute at any time.

Our goal is to bound $\Delta = t_n - t_0$. If we can show that Δ is less than $m(i)$, we would show that (i, j) cannot be dropped.

The first step in checking whether external events can be dropped is to compute so-called *partial loads.* A partial load $\delta(i, j)$ is the first time at which only tasks with priority lower than j are enabled, given that:

1. task i (which may be internal or external) finishes the execution at time 0,
2. no tasks other than i are enabled just before i finishes its execution,
3. no external tasks execute in the interval $[0, \delta(i, j)]$ (other than possibly i finishing its execution at 0).

In other words, $\delta(i, j)$ is the continuous work-load at priority j or higher caused by an execution of i. Obviously, partial loads can be determined by simulation. A more efficient of doing it (in time proportional to the square of size of $T \cup U$) is proposed in Section 4.3.

At time t_0 (in Figure 3), some internal task of priority less than j may be finishing its execution and enabling tasks of priority j or higher. In the worst case, such an event may cause the following work-load at priority j or higher:[3]

$$\max\{\delta(k, j) \mid k \in T,\ k < j\}\ . \qquad (2)$$

In a time interval of length Δ, some external task e can be executed[4], at most $\left\lceil \frac{\Delta}{m(e)} \right\rceil$ times, and every execution generates a work-load (at priority j or higher) of $\delta(k, j)$. Thus the total work-load at priority j or higher caused by external tasks in an interval of length Δ is bounded by:

$$\sum_{k \in U} \left\lceil \frac{\Delta}{m(k)} \right\rceil \delta(k, j)\ . \qquad (3)$$

Finally, since the work-load in any time interval where only tasks of priority j or higher execute, may be caused only

[3]We assume that sums and maxima over empty sets are zero, i.e. $\sum_{s \in S} s = \max\{s | s \in S\} = 0$ if S is empty.

[4]In this paper, we use $\lceil x \rceil$ to denote the smallest *positive* integer not smaller than x. Thus, according to this, slightly non-standard definition, $\lceil 0 \rceil = 1$.

by an execution of a task of priority less than j at the very beginning of the interval and by executions of external tasks during the interval, we have (from (2) and (3)):

$$\begin{aligned}\Delta &\leq \max\{\delta(k,j) \,|\, k \in T,\, k < j\} + \\ &\sum_{k\in U} \left\lceil \frac{\Delta}{m(k)} \right\rceil \delta(k,j) \ . \end{aligned} \quad (4)$$

It can be shown that true Δ must not only satisfy (4), but must also be smaller than the smallest positive Δ satisfying (4) with equality. Such a Δ can be computed by the following iteration computation:

$$\begin{aligned}\Delta_0 &= \max\{\delta(k,j) \,|\, k \in T,\, k < j\} \ , \\ \Delta_{l+1} &= \Delta_0 + \sum_{k\in U} \left\lceil \frac{\Delta_l}{m(k)} \right\rceil \delta(k,j) \quad l = 0,1,\ldots \ . \end{aligned} \quad (5)$$

The iteration will converge if:

$$\sum_{k\in U} \frac{\delta(k,j)}{m(k)} < 1 \ . \quad (6)$$

The following theorem summarizes the discussion above:

Theorem 2 *Let (6) be satisfied, and let Δ^* be the limit of iteration (5). If $\Delta^* < m(i)$ then (i,j) cannot be dropped.*

If $\Delta^* \geq m(i)$ or if (5) does not converge, the results are inconclusive: it may be the case that (i,j) can indeed be dropped, or it may be the case that (i,j) cannot be dropped, but our analysis is too conservative to prove it.

For example, to check whether event $(7,1)$ in Figure 1 can be dropped, we need to compute:

$$\begin{aligned}\Delta_0 &= 0 \ , & (7)\\ \Delta_{l+1} &= \left\lceil \frac{\Delta_l}{m(7)} \right\rceil \delta(7,1) + \left\lceil \frac{\Delta_m}{m(6)} \right\rceil \delta(6,1) \ . & (8)\end{aligned}$$

First, we need to determine (either by simulation or by the procedure described in the next section) that $\delta(7,1) = 11$ and $\delta(6,1) = 4$, and then we have that the iteration converges with $\Delta_2 = \Delta_3 = 19$. Since that is less than $m(7) = 20$, we conclude that $(7,1)$ cannot be dropped.

4.3 Computing partial loads

To compute partial loads, we first compute the function $\lambda : (T \cup U) \times T \mapsto \mathbb{R}$ such that $\lambda(i,n) = \delta(i,n)$ (where n in the highest priority task), and $\lambda(i,j) = \delta(i,j) - \delta(i,j+1)$ for all internal tasks $j < n$. In other words, $\lambda(i,j)$ is the additional work-load generated by an execution of i if the minimum priority task allowed to execute is j rather than $j+1$. It follows then easily that:

$$\delta(i,j) = \sum_{k\geq j} \lambda(i,k) \ .$$

It can be shown that λ can be characterized as follows:

$$\lambda(i,j) = \begin{cases} e(j) + \sum_{k>j} \lambda(j,k) & \text{if } (i,j) \in E \ , \\ \max\{\lambda(k,j) \,|\, (i,k) \in E,\, k > j\} & \text{otherwise.} \end{cases} \quad (9)$$

Notice, that to compute $\lambda(i,\cdot)$, we only need to know $\lambda(j,\cdot)$ for all successors j of i (i.e. for all j such that $(i,j) \in E$). Thus, (9) can be implemented with a single traversal of E in reverse topological order.

For example, for the system in Figure 1 (in a possible order of computation):

$\lambda(4,3) = 2$			
$\lambda(2,3) = 2$	$\lambda(2,4) = 1$		
$\lambda(5,3) = 2$	$\lambda(5,4) = 1$		
$\lambda(1,2) = 4$	$\lambda(1,3) = 2$	$\lambda(1,4) = 1$	$\lambda(1,5) = 2$
$\lambda(7,1) = 11$			
$\lambda(6,2) = 4$			

Note that task 3 contributes its execution time twice to the the total load of 11 in $\lambda(7,1)$. That is because there are two path from 1 to 3, one containing a task with priority lower than 3. Therefore, every execution of 1 will cause two executions of 3. From task 5 there are also two paths to task 3 but there are no lower priority tasks on them, so an execution of 5 will cause only a single execution of 3 (accounted by $\lambda(5,3)$). This observation can be used to assign priorities such that total workload is minimized. Unfortunately, this criterion might be in conflict with requirements of Theorem 1. It is still an open problem to find a priority assignment policy that balances these two conflicting requirement.

5 Validation of Non-Preemptive Schedules

Preemptive static priority scheduling has attracted a lot of attention due to its elegant theoretical properties and good service of urgent tasks. However, non-preemptive schemas are often used in practice due to significantly simpler implementation and lower execution time overhead. In this section we extend the analysis of section 4 to non-preemptive static priority scheduling. In this scheme, similarly to the preemptive case, if the processor is idle, then the task with the highest priority among enabled tasks is executed. However, once a task is selected for execution, it is run until completion, even is some higher priority tasks become enabled in the meanwhile.

First, note that Theorem 1 is still valid even for non-preemptive scheduling. Therefore we only need to check whether some external event (i,j) can be dropped. To do that, we consider a typical segment of the execution in which j can be enabled, but cannot be executed. Such a segment has the following characteristics:

- the segment begins at time t_0 with the execution of some task with priority no higher than j,
- all the other tasks executing in the segment have priorities higher than j,
- the segment ends at time t_n when task j starts executing,
- in the interval $[t_0, t_n)$, external tasks can execute only at any time.

Now, by reasoning similar to section 4, it can be shown that $\Delta = t_n - t_0$ must satisfy:

$$\begin{aligned}\Delta &\leq \max\{e(k) + \delta(k,j+1) \,|\, k \in T,\, k \leq j\} + \\ &\sum_{k\in U} \left\lceil \frac{\Delta}{m(k)} \right\rceil \delta(k,j+1) \ , \end{aligned} \quad (10)$$

where the first term accounts for the execution of the initial task in the interval and its successors, and the second term accounts for executions of external tasks.

Table 1: Schedule validation for PATHO

number of tasks	10	100	1000	2000	4000
run time [ms]	1	1	6	12	24

Again, the smallest Δ satisfying (10) with equality can be determined by straightforward iteration, which converges if (6) is satisfied.

Theorem 3 *Let (6) be satisfied, and let Δ^* be the smallest Δ satisfying (10) with equality. If $\Delta^* + e(j) < m(i)$ then (i, j) cannot be dropped.*

6 Experiments

We implemented the algorithms presented in this paper inside the *POLIS* HW/SW co-design system [5]. POLIS starts from a system specified as a network of interacting components, and provides an implementation as a set of software and/or hardware tasks. Software tasks are scheduled according to their priorities which are fixed. The user sets priorities and maximum times between execution of external events, while POLIS estimates the maximum execution time of each task. These estimates are then used in our algorithm to bound times between two executions of every task.

We tested our algorithm on two classes of examples: one from the automotive domain, the other from the reactive Real Time Operating System domain. The simpler example of the automotive domain is a dashboard controller which consists of 6 tasks (to calculate and display speed, fuel level, ...) which react to 13 external events. The schedule used was validated in less than 0.1s of CPU time of Sparc 10 workstation, a negligible time when compared to 8.6s it took for POLIS to estimate execution times of individual tasks (this time includes partially implementing the tasks, to the point that meaningful estimations can be made).

The larger example is a shock absorber controller [2] (of which the system in Figure 1 is a small subsystem). It has 48 tasks reacting to 11 external events. Our algorithm required 0.3s of CPU time, which was again negligible compared to 880s required to estimate execution times of individual tasks.

We also applied our algorithm to an existing model of the PATHO real-time operating system [3]. In this model, several tasks with known execution times share a processor according to a non-preemptive static priority schema. In [3] results are reported for systems with ten PATHO tasks that range from seconds to thousands of seconds (higher priorities tasks are easier to validate). The method presented in this paper validated all ten tasks in one milliseconds. Note that the speed of execution is so high that only for systems with thousands of task we could see the dependence of the run time to the number of tasks, and even then it is only linear (see Table 1).

To our knowledge, experiments reported in [3] are the only published results on using timed automata for schedule validation. Timed automata are interesting because they are representative of formalisms in which the schedule validation problem can be solved exactly, and for a wider class of systems than it is possible with the method proposed here. However, as PATHO schedule validation indicate, our approach is substantially more efficient.

7 Conclusions

We proposed a schedule validation method for reactive real-time systems scheduled by a static priority schema. We have developed versions of our method both for preemptive and non-preemptive scheduling. Our method is conservative in the sense that it might claim that a valid schedule is invalid, but it can never declare an invalid schedule to be valid. Initial experimental results has shown that the run time of our method is negligible compared both to other steps in the design process, and to schedule validation by different methods.

REFERENCES

[1] Neil C. Audsley, Alan Burns, M. Richardson, Ken W. Tindell, and Andy J. Wellings. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, pages 284–292, September 1993.

[2] Felice Balarin, Harry Hsieh, Attila Jurecska, Luciano Lavagno, and Alberto Sangiovanni-Vincentelli. Formal verification of embedded systems based on CFSM networks. In *Proceedings of the 33th ACM/IEEE Design Automation Conference*, pages 568–571, June 1996.

[3] Felice Balarin, Karl Petty, Alberto L. Sangiovanni-Vincentelli, and Pravin Varaiya. Formal verification of the PATHO real-time operating system. In *Proceedings of 33rd Conference on Decision and Control, CDC'94*, December 1994.

[4] Felice Balarin and Alberto Sangiovanni-Vincentelli. Schedule validation for embedded reactive real-time systems. Technical report, Cadence Berkeley Laboratories, October 1996.

[5] Massimiliano Chiodo, Paolo Giusto, Harry Hsieh, Attila Jurecska, Luciano Lavagno, and Alberto Sangiovanni-Vincentelli. A formal methodology for hardware/software codesign of embedded systems. *IEEE Micro*, August 1994.

[6] Michael Gonzalez Harbour, Mark H. Klein, and John Lehoczky. Timing analysis for fixed-priority scheduling of hard real-time systems. *IEEE Transaction on Software Engineering*, 20(1), January 1994.

[7] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-realtime environment. *Journal of the Association for Computing Machinery*, 20(1):46–61, January 1973.