Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

Schedulability of Herschel Revisited Using Statistical Model Checking *

Alexandre David[1], Kim G. Larsen[1], Axel Legay[1,2], Marius Mikučionis[1]

[1] Computer Science, Aalborg University, Denmark

[2] INRIA/IRISA, Rennes Cedex, France

The date of receipt and acceptance will be inserted by the editor

Abstract Schedulability analysis is a main concern for several embedded applications due to their safety-critical nature. The classical method of response time analysis provides an efficient technique used in industrial practice. However, the method is based on conservative assumptions related to execution and blocking times of tasks. Consequently, the method may falsely declare deadline violations that will never occur during execution. This paper is a continuation of previous work of the authors in applying extended timed automata model checking (using the tool UPPAAL) to obtain more exact schedulability analysis, here in the presence of non-deterministic computation times of tasks given by intervals [BCET,WCET]. Computation *intervals* with preemptive schedulers make the schedulability analysis of the resulting task model undecidable. Our contribution is to propose a combination of model checking techniques to obtain some guarantee on the (un)schedulability of the model even in the presence of undecidability.

Two methods are considered: symbolic model checking and statistical model checking. Since the model uses *stop-watches*, the reachability problem becomes undecidable so we are using an over-approximation technique. We can safely conclude that the system is schedulable for varying values of BCET. For the cases where deadlines are violated, we use polyhedra to try to confirm the witnesses. Our alternative method to confirm non-schedulability uses statistical model-checking (SMC) to generate counter-examples that are always realizable. Another use of the SMC technique is to do performance analysis on schedulable configurations to obtain, e.g., expected response times.

The methods are demonstrated on a complex satellite software system yielding new insights useful for the company.

* This paper is a special issue extension of [DLLM12]. The main differences are a more thorough description of the case-study and experiments, and the use of the polyhedra library APron. This work is partially supported by VKR Centre of Excellence MT-LAB, the Sino-Danish basic research center IDEA4CPS, EU ARTEMIS project MBAT, the regional CREATIVE project ESTASE, and the EU projects DANSE and DALI.

1 Introduction

Embedded systems involve the monitoring and control of complex physical processes using applications running on dedicated execution platforms in a resource constrained manner in terms of memory, processing power, bandwidth, energy consumption, or task response time.

Viewing the application as a collection of (interdependent) tasks various scheduling principles may be applied to coordinate the execution of tasks in order to ensure orderly and efficient usage of resources. Based on the physical process to be controlled, timing deadlines may be required for the individual tasks as well as the overall system. The challenge of schedulability analysis is now concerned with guaranteeing that the applied scheduling principle(s) ensure that the timing deadlines are met.

The classical method of response time analysis [JP86, Bur94] provides an efficient means for schedulability analysis used in industrial practice: by calculating safe upper bounds on the worst case response times (WCRT) of tasks (as solutions to simple recursive equations) schedulability may be concluded by a simple comparison with the deadlines of tasks. However, classical response time analysis is only applicable to restricted types of tasksets (e.g. periodic arrival patterns), and only applicable to applications executing on single processors. Moreover, the method is based on conservative assumptions related to execution and blocking times of tasks [BHK99]. Consequently, the method may falsely declare deadline violations that will never occur during execution.

Process algebraic approaches [BACC+98, SLC06] have resulted in many methods for specification and schedulability analysis of real-time systems. In particular, [BHK+04, BHM09] frameworks are based on timed automata and use Uppaal to find a schedule, show schedulability and then use Möbius to assess the performance of schedules in realistic settings. This paper demonstrates a combination of model-checking and stochastic simulation techniques for schedulability analysis using recent Uppaal SMC features on the same model extended with stochastic interpretation. In addition to providing quantitative information the case study is novel in showing complementary applications of proving and disproving schedulability by using symbolic and stochastic semantics of the same stop-watch automata model.

In our previous work [MLR+10], we extended the schedulability analysis framework of [DILS10] to obtain more exact schedulability results for a wider class of systems, e.g., task-sets with complex and interdependent arrival patterns of task, multiprocessor platforms, etc. We did so on the industrial case study of the Herschel-Planck dual satellite system. The system consists of two satellites sharing the same software architecture hence the analysis model is easily adaptable to both configurations. The control software of this system – developed by the Danish company Terma A/S – consists of 32 tasks executing on a single processor system with preemptive scheduler, and with access to shared resources managed by a combination of priority inheritance and priority ceiling protocols. The main result of this study was to conclude that the system was schedulable. In [DLLM12] we relaxed the strong assumption that the execution time of each task was the same as the worst-case execution time (WCET). Classical response time analysis [Bur94] considers that the task execution time is anywhere in between zero and worst case execution time (WCET), which is overly conservative and failed to conclude schedulability in one of the operating modes of the Herschel satellite, even though no deadline violations have been observed during testing.

In this work we were more precise and considered the execution times to be between the best and worst execution times [BCET, WCET], however we had no data about actual BCETs, hence BCETs were treated as a system parameter to find a schedulability threshold. In practice one would model fixed BCETs, attempt to prove schedulability and then attempt to find a concrete counter example if schedulability analysis failed using our techniques. The main result was to conclude schedulability only for larger BCET values and confirm that the more conservative range is indeed not schedulable, which means that the system could be proven schedulable if tasks had larger BCET values, i.e. were not too fast. Statistical model-checking (SMC) was used to generate counter examples for cases leading to deadline violations. In this work, we give a more complete description of the case, in particular the automata and the tasks. We expose the real complexity of the model. In addition, a complementary symbolic technique with polyhedra can be used to prove that symbolic traces obtained by Uppaal are realizable in some cases (even though the exploration is over-approximate).

The interesting technical points of our case are that we deal with non-deterministic computation times and dependencies between tasks, thus schedulability analysis of the resulting task model is undecidable [FKPY07]. Our technique uses stop-watches to obtain a guarantee on schedulability using over-approximation. If a deadline violation is found during the over-approximate analysis, then nothing can be concluded because the detected run can be spurious. Hence we use symbolic model-checking to assert schedulability and statistical model-checking to find concrete error traces in case of non-schedulability. Moreover, we use a polyhedra library to confirm symbolic traces obtained with our over-approximate exploration.

SMC Approach. The core idea of SMC [YS02, SVA04, HLMP04, YS06, KZH+09, RP09, LDB10] is to generate stochastic simulations of the system and verify whether they satisfy a given property. The results are then used by statistical algorithms in order to compute among others an estimate of the probability for the system to satisfy the property. Such estimate is correct up to some confidence that can be parameterized by the user. Several SMC algorithms that exploit a stochastic semantics for timed automata have recently been implemented in Uppaal [DLL+11a, DLL+11b, BDL+12]. In this case we assign a uniform probability distribution for the task executions times from an interval [BCET, WCET] which ensures that the SMC algorithm will try a fair distribution of different task execution times from that interval and provide conclusive evidence that the system run is actually realizable if a deadline violation is found. The SMC analysis is under-approximate in a sense that we cannot conclude safety if the error is not found, because the probability of finding one might be very low and we are out of luck.

The results obtained show that these techniques complement each other for either proving safety or finding error traces. In particular for Herschel, as illustrated in the summary Table 1: when $\frac{\mathrm{BCET}}{\mathrm{WCET}} \geq 90\%$ symbolic model-checking confirms schedulability, whereas statistical model-checking disproves schedulability for $\frac{\mathrm{BCET}}{\mathrm{WCET}} \leq 73\%$. For $\frac{\mathrm{BCET}}{\mathrm{WCET}} \in (73\%, 90\%)$ our experiments are inconclusive. In addition, we show how to obtain more informative performance analysis using SMC: e.g., expected response times when the system is schedulable as well as estimation of the probability of deadline violation when the system is not schedulable.

The paper is divided into four sections: Section 2 describes the model architecture, Section 3 explains the symbolic analysis for proving schedulability, Section 4

$f=\frac{\text{BCET}}{\text{WCET}}$:	0-73%	74-86%	87-89%	90-100%
Symbolic MC:	maybe	maybe	n/a	**Safe**
Statistical MC:	**Unsafe**	maybe		maybe

Table 1: Summary of schedulability of Herschel concluded using symbolic and statistical model checking.

shows the statistical analysis for obtaining performance measurements and disproving schedulability. Each section contains two parts: the first demonstrates the technique on a simple example and the second describes the interesting extra details from an industrial case study.

2 Modeling

This section introduces the modeling framework that will be used through the rest of the paper. We start by presenting the generic features of the framework through a running example. Then, we briefly indicate the key additions in modeling the Herschel-Planck application, leaving details to be found in [MLR+10].

Consider the state machine for tasks in common operating system shown in Fig. 1. Initially the task is in state *Idle* and it can be released into *Ready*. The task becomes *Running* when a processor is assigned to it and it starts executing. The task may be preempted back into state *Ready* if a higher priority task becomes ready and gets assigned the processor instead. During execution the task may become *Blocked* if it requests a resource and the resource is not available at that time, thus releasing the processor to other tasks. The task returns to state *Ready* once the requested resource becomes available.

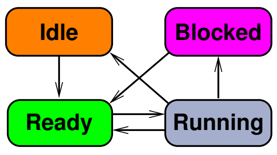


Figure 1: Typical state machine for tasks.

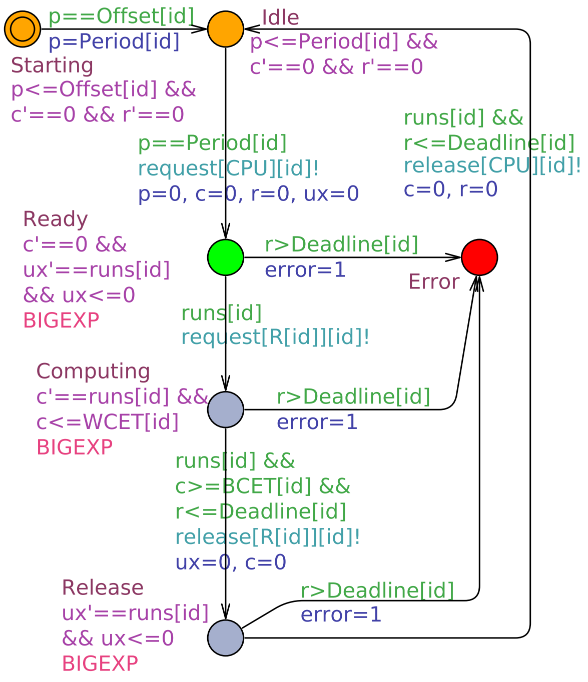
We consider a *Running Example*, that builds on instances from a library of three types of processes represented with timed automata templates in UPPAAL: (1) preemptive CPU scheduler, (2) resource schedulers that can use either priority inheritance, or priority ceiling protocols, and (3) periodically schedulable tasks. In what follows, we use broadcast channels in entire model which means that the sender cannot be blocked and the receiver can ignore it if it is not ready to receive it. Derivative notation like x'==e specifies whether the stop-watch x is running, where e is an expression evaluating to either 0 or 1. In UPPAAL a stop-watch is an ordinary clock that is stopped by setting its rate to zero. By default all clocks are considered running, i.e. the derivative is 1.

For simplicity, we assume periodic tasks arriving with *period* Period[id], with initial *offset* Offset[id], requesting a resource R[id], executing for at least *best case execution time* BCET[id] and at most *worst case execution time* WCET[id] and hopefully finishing before the *deadline* Deadline[id], where id is a task identifier. The wall-clock time duration between task arrival and finishing is called *task response time*: it includes the CPU execution time as well as any preemption or blocking time. The task safety is evaluated by the worst case scenario: comparing the *worst case response time* (WCRT) against the deadline of the task.

Figure 2a shows the UPPAAL declaration of the above mentioned parameters for three tasks (number of tasks is encoded with constant NRTASK) and one resource (number of resources is encoded with constant NRRES). The task_t type declaration says that task identifiers are integers ranging from 1 to NRTASK. Similarly the res_t type declares resource identifier range from 1 to NRRES. Parameters Period, Offset, WCET, BCET, Deadline and R are represented with integer arrays, one element per each task. Figure 2b shows a periodic task template in UPPAAL timed automata syntax: circles denote locations which have names (colored in dark red) and invariant expressions (magenta), and edges are decorated with guards (green), synchronizations (cyan) and updates (blue) in that order. The task starts in Starting location, waits for the initial offset to elapse and then moves to Idle location. The task arrival is marked by a transition to a Ready location which signals the CPU scheduler that it is ready for execution. Then location Computing corresponds to busy computing while holding the resource (might as well be blocked if the resource is not available) and Release is about releasing the resources and finishing. The periodicity of a task is controlled by constraints on a local clock p: the task can move from Idle to Ready only when p==Period[id] and then p is reset to zero to mark the beginning of a new period. Upon arrival to Ready, other clocks are also reset to zero: c starts counting the execution time, r measures response time and ux is used to force the task progress. The invariant on location Ready says that the task execution clock c does not progress (c'==0) and it cannot stay longer than zero time units (ux<=0) unless it is not running (ux'==runs[id]). The task also cannot progress to location Computing unless the CPU is assigned to it (runs[id] becomes true). When the CPU is assigned, the task will be forced to urgently request a resource and move on to Computing, where the computation time (valuation of c) increases only when it is marked as running (runs[id] is true). The task can stay in Computing for at most worst case execution time (c<=WCET[id]), cannot leave before best case execution time (c>=BCET[id]), but can be preempted by setting runs[id] to 0. If the resource is not granted then the resource scheduler is responsible for blocking the task from using the CPU. If the response time clock is below deadline (p<=Deadline[id]) then the task can move on to Release and similarly complete to Idle. Notice that the task competes for resources in locations Ready, Computing and Release, and it is forced to move to Error location if the response time exceeds dead-

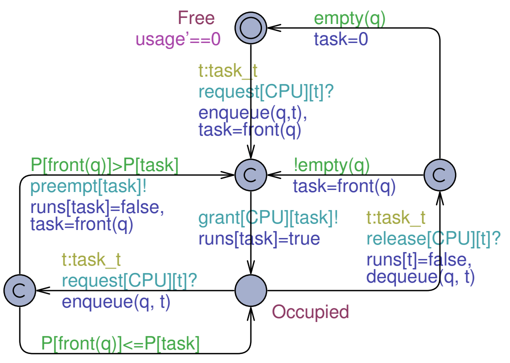
```
const int NRTASK = 3; // # of tasks
const int NRRES = 1; // # of resources
typedef int [1, NRTASK] task_t;
typedef int [1, NRRES] res_t;
const int f=80; // fraction of WCET, in %
int Period[task_t]   = { 100,  100, 100 };
int Offset[task_t]   = {  20,    0,  10 };
int WCET[task_t]     = {  15,   25,  40 };
int BCET[task_t]     = { WCET[1]*f/100,
    WCET[2]*f/100, WCET[3]*f/100 };
int Deadline[task_t] = {  20,   40,  70 };
res_t  R[task_t]     = {   1,    1,   1 };
int P[task_t] = { 3, 2, 1 }; // priorities
bool runs[task_t] = { 0, 0, 0 }; // is running?
bool error = false; // global variable
```

(a) Parameter declarations.

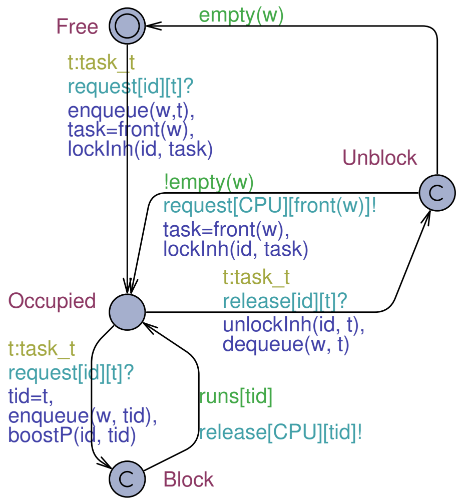


(b) Stopwatch automaton template.

Figure 2: Task model.



(a) Preemptive CPU scheduler.



(b) Resource using priority inheritance.

Figure 3: Schedulers for active and passive resources.

line (r>Deadline[id]). The exponential rate BIGEXP is a large number (10^6) used to minimize the (stochastic) delay when response time is beyond the deadline and force the simulation to terminate early upon such error.

The CPU scheduler is equipped with a task queue q sorted by task priorities P[t], where t is a task identifier and task variable holding the currently running task identifier. Function front(q) always returns the highest priority task identifier in the q queue. Figure 3a shows a CPU template which alternates between Free and Occupied locations.

When a request[CPU][t] arrives, the requesting task t is put into the queue and the CPU is being rescheduled. This is done either by immediate grant[CPU][task] and marking that the task is running runs[task]=true, or via preemption of the currently running task of lower priority, or simply returning to Occupied if the highest priority task in the queue is not higher than currently running. When a release[CPU][t] arrives, the requesting task t is de-queued, marked as not running (runs[t]=false), and the CPU is granted to the next highest priority task in the queue (if the queue is not empty). We use Uppaal committed locations (encircled with C) for uninterrupted (atomic) transitions, thus Free and Occupied are the only locations where the time can pass. In addition, the scheduler is equipped with usage stop-watch: usage is stopped by invariant usage'==0 at location Free and is running with default rate of 1 in location Occupied, hence its valuation computes the CPU usage.

A resource scheduler shown in Fig. 3b is equipped with its own waiting queue w. It operates in a similar way as CPU scheduler, that is by alternating between Free and Occupied. The main difference is that a resource cannot be preempted once it is locked. The locking operations follow the priority inheritance protocol implemented in functions lockInh(res,task), unlockInh(res,task). Operation boostP(res,task) raises the

```
const int NOTASK = 33; // number of tasks
const int NORES = 7; // number of resources
typedef int [0, NOTASK] taskid_t; // task identifier type
typedef int [0, NORES-1] resid_t; // resource identifier type
taskid_t owner[resid_t]; // records the owner of a resource
/** default priority of a task is inverse to its ID: */
taskid_t defaultP(taskid_t task) {
    if (task==0) return 0;
    else return NOTASK-task+1;
}
/** Check if the resource is available: */
bool avail(resid_t res) { return (owner[res]==0); }
/** Boost the priority of resource owner: */
void boostP(res_t resource, task_t task) {
    if (P[owner[resource]] <= defaultP(task)) {
        P[owner[resource]] = defaultP(task)+1;
        sort(q); // sorts the queue by descending priorities
    }
}
/** Lock based on priority inheritance protocol: */
void lockInh(res_t resource, task_t task) {
    owner[resource] = task; // mark as occupied by the task
}
/** Unlock based on priority inheritance protocol: */
void unlockInh(res_t resource, task_t task) {
    owner[resource] = 0; // mark as released
    P[task] = defaultP(task); // return to default priority
}
```

Listing 1: Data and function declarations.

priority of the resource res owner to higher level than the requesting task. Listing 1 shows the listing of UPPAAL code implementing the priority inheritance protocol.

Similarly to modeling the priority inheritance protocol in Fig. 3b, we model also priority ceiling protocol by modifying the locking functions.

Herschel. In this paper, we consider a large, industrial case study: the schedulability analysis of the control software of the Herschel satellite. This case study[1], which seriously challenges the capabilities of UPPAAL, uses the same basic stop-watch modeling principles as in the *Running Example* described above. The actual system consists of two satellites (Herschel and Planck) with shared software architecture split into basic software (BSW) and application software (ASW) tasks. Each system can be run in nominal and event operation modes which are characterized by different sets of tasks and schedulability is checked in each mode separately. Table 2 summarizes the number of tasks involved in each satellite and mode configuration. The software provider Terma has succeeded in showing the schedulability of all configurations using Alan Burns' [Bur94] framework except Herschel in event mode where PrimaryFunctions task was exceeding its deadline. Interestingly, no deadline violations were observed during simulation and testing of the system and thus we chose this configuration to investigate further by adding more details with a hope of achieving finer analysis. The Herschel model consists of 32 tasks sharing 6 resources using two protocols: priority inheritance and priority ceiling. Among these tasks, 24 are periodic [2], while 8 are triggered in a sequence. Table 3 shows a complete matrix of resource sharing among tasks.

The system was simulated and the task performance was measured in order to obtain a detailed description of individual operations. Table 4 shows an example of a description of task `PrimaryFunctions` which consists of a sequence of procedures like Data processing, Guidance etc.. For example, Data processing took $20577\mu s$ of processor time in total. During execution, resource Icb_R was used for $LC = 1600\mu s$ (locking time) of which $LCS = 1200\mu s$ (locking time in suspension) the processor was released due to suspended wait (e.g. task waited for an on-board device to finish operation). From this description of tasks we identified five basic operations needed to encode the task details: LOCK – locks the specified resource, UNLOCK – unlocks the specified resource, COMPUTE – requests CPU and actively uses it for specified time, SUSPEND – releases CPU for specified amount of time and END – finishes the task. Listing 2 shows the sequence of basic operations for Primary Functions encoded into a C-like data structure which is

Table 2: Herschel-Planck task configurations.

System:	Herschel		Planck	
Mode:	nominal	event	nominal	event
ASW	5	8	5	8
BSW	24	24	19	19
Total:	29	32	24	27

Table 3: System wide resource competition: ASW tasks (in bold) use priority ceiling and BSW (in plain) use priority inheritance protocols.

Task \ Resource	Icb_R	Sgm_R	PmReq_R	Other_R
MainCycle		✓		
PrimaryFunctions	✓	✓	✓	
RCSControl				✓
Obt_P	✓			
StsMon_P	✓			
Sgm_P		✓		
Cmd_P	✓			
SecondaryFunc1		✓	✓	✓
SecondaryFunc2	✓			✓

Table 4: `PrimaryFunctions` task behavior.

Primary Functions	
- Data processing	**20577**
	Icb_R(LCS: **1200**, LC: **1600**)
- Guidance	**3440**
- AttitudeDetermination	**3751**
	Sgm_R(LCS: **121**, LC: **1218**)
- PerformExtraChecks	**42**
- SCM controller	**3479**
	PmReq_R(LCS: **1650**, LC: **3300**)
- Command RWL	**2752**

[1] http://people.cs.aau.dk/~marius/Terma/

[2] In the actual system 16 of them are sporadic, but we model them as periodic to limit non-determinism.

```
const ASWFlow_t PF_f = { // Primary Functions:
    { LOCK,   Icb_R, 0 },              // Data  processing
    { COMPUTE, CPU_R, 1600-1200 },// computing with Icb_R
    { SUSPEND, CPU_R, 1200 },     //    suspended  with  Icb_R
    { UNLOCK, Icb_R, 0 },          //
    { COMPUTE, CPU_R, 20577-(1600-1200) }, // comp. w/o Icb_R
    { COMPUTE, CPU_R, 3440 },  // Guidance
    { LOCK,   Sgm_R, 0 },              //  Attitude   determination
    { COMPUTE, CPU_R, 1218-121 }, // computing with Sgm_R
    { SUSPEND, CPU_R, 121 },      //    suspended  with  Sgm_R
    { UNLOCK, Sgm_R, 0 },          //
    { COMPUTE, CPU_R, 3751-(1218-121) },// computing w/o Sgm_R
    { COMPUTE, CPU_R, 42 },      // Perform  extra  checks
    { LOCK,   PmReq_R,0 },         // SCM  controller
    { COMPUTE, CPU_R, 3300-1650 },// computing with PmReq_R
    { SUSPEND, CPU_R, 1650 },     //    suspended  with  PmReq_R
    { UNLOCK, PmReq_R, 0 },       //
    { COMPUTE, CPU_R, 3479-(3300-1650) },// comp. w/o PmReq_R
    { COMPUTE, CPU_R, 2752 },  // Command RWL
    { END,    CPU_R, 0 }           //     finished
};
```

Listing 2: Encoded `PrimaryFunctions` task.

being interpreted by an automaton template. The model includes three task templates with small variations on how the task is started and how resources are shared: BSW tasks are started periodically and resources are locked using priority inheritance, `MainCycle` and `ASW` (shown in Fig. 4) use the priority ceiling protocol where `MainCycle` is released periodically and ASW tasks are triggered by `MainCycle`.

The task templates follow the same scheme as in Figure 2b, except that there are more locations responsible for the same abstract state and the resource automata are merged into the task. For example, the task ASW template from Fig. 4 has location Idle to represent state *Idle*, Ready and WaitForCPU represent *Ready*, Compute, tryLock, Suspended, Computing, Next and Release perform the basic operations and correspond to *Running*, and location Blocked corresponds to abstract state *Blocked*. The edges decorated with both synchronizations enqueue! and enqueue? are actually two almost identical edges with overlapping layout where the difference is that one edge is with output and the other with input broadcast synchronization. Having both equivalent edges with broadcast input and output ensures that all ready tasks are going to be released at once and the scheduler will pick the highest priority task at once without intermediate rescheduling. This trick effectively provides partial order reduction when time does not elapse, i.e. when several tasks are released at the very same time, and thus the overall behavior does not change (the scheduler would eventually pick the highest priority task and preempt the rest anyway), but a great number of intermediate states are avoided.

3 Symbolic Safety Analysis

In this section we apply the classical zone-based symbolic reachability engine of Uppaal to verify schedulability. As we are considering systems with preemptive scheduling our models will be using stop-watches as described in the framework of [DILS10]. With the addition of having non-deterministic computation times – i.e. computation

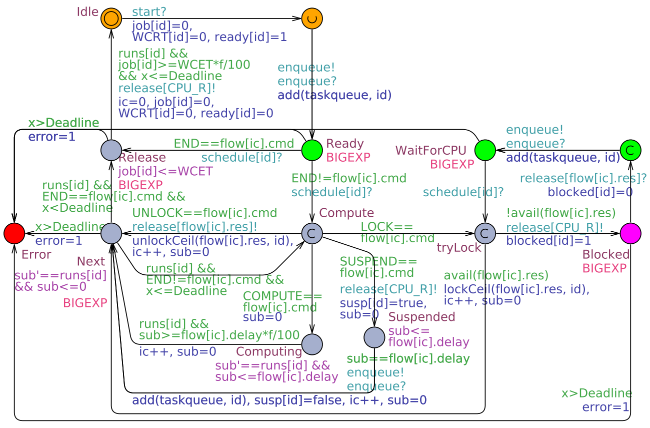


Figure 4: ASW task template executing a list of operations.

Table 5: Summary of schedulability of the *Running Example* example concluded using symbolic and statistical MC for varying sizes of computation time intervals.

$f = \frac{BCET}{WCET}$	0-79%	80-83%	84-100%
Symbolic MC:	maybe	maybe	**Safe**
Statistical MC:	**Unsafe**	maybe	maybe

intervals – as well as dependencies between task due to resources, schedulability of the resulting task model is undecidable as a consequence of the results of [FKPY07]. In our case, we extend the symbolic techniques using zones to work on stop-watches, the price being that the technique is now an over-approximation [CL00]. Our symbolic analysis is still useful to assert schedulability since it is a safety property. For cases where a system *may not* be schedulable we use our complementary statistical technique described in the next section.

Running Example As detailed in Section 2, our running example consists of three tasks with WCET times being $15, 25, 40$, deadlines $20, 40, 70$ and with one single resource shared by the three tasks. The query used in Uppaal to check for schedulability is:

```
A[]  !error
```

Here `error` is a Boolean being set whenever a task misses its deadline (see Fig. 2b). We are interested here in the results of the symbolic analysis shown in Table 5. For the execution time picked non-deterministically between 84% and 100%, our set of tasks is schedulable. However, if the execution time is less, it *may* be non-schedulable. The classical analysis focused on worst-case execution time does not apply here (undecidable as mentioned) and a shorter computation time for one task may cause another task to miss a deadline. For these cases, Uppaal returns symbolic counter-examples that may or may not be realizable. Here task `T(1)` may miss its deadline. For this simplified example, all the checks take less than $0.1s$ on a Core i7 2600.

In addition to schedulability, we may obtain upper bounds on the WCRTs but using the *sup* query of Uppaal. It is a special query that makes Uppaal generate the whole state-space and check the upper bounds of some clocks. We check the WCRTs of the three tasks with the query

```
sup:  T(1).r, T(2).r,  T(3).r
```

where `T(i).r` is a response time clock that starts growing when the task `T(i)` is released. The results fall into two classes: for computation intervals $[f\cdot\text{WCET}, \text{WCET}]$ with $f \geq 84\%$ the WCRTs are $20, 40, 70$ and for $f < 84\%$ the WCRTs are $55, 40, 70$, indicating the possibility of deadline violation for task `T(1)`.

Finally, using an additional stop-watch `usage`, which is only stopped when the CPU is free (and reset for each 2000 time-units) the query `sup: usage` returns the value 1600, providing 80% (= 1600/2000) as an upper bound of the CPU utilization.

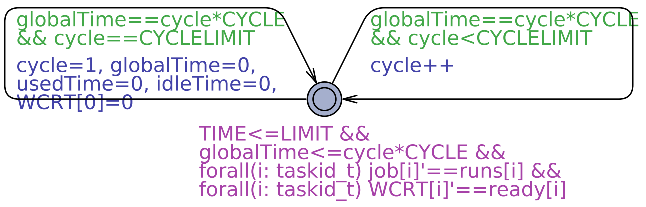
More Precise Results with Polyhedra. The symbolic techniques used in Uppaal rely on zones and they generate symbolic traces that may or may not be realizable when stop-watches are used. However, given such a trace, we use the APron polyhedra library [JM09] to re-execute the trace but using a polyhedra as the symbolic state instead of a zone. It is a matter of implementing the symbolic operations on a polyhedra instead of a zone. The result is that this technique may confirm that a trace is indeed realizable. Failure to confirm means that we still do not know if the state is reachable or not. What Uppaal does in this case is to resume the search in hope to find another trace that can be confirmed.

The new behavior of Uppaal is as follows: When a trace is requested with the polyhedra option, then the *maybe* verdict is changed into *satisfied* whenever the error state can be reached with the generated output trace[3].

When run on the simple example, all the traces can be confirmed on the first error found.

Herschel. Applying in a similar manner symbolic MC to the Herschel case seriously challenges the engine of Uppaal, due to the the explosion in the size of the symbolic state-space with the increase of the size of the computation time intervals. In fact, to avoid the analysis to run out of memory we have applied the so-called sweep-line method [CKM01]. The sweep-line method relies on a progress measure to release explored states and thus use verification memory more efficiently. Cyclic systems like ours do not have a notion of progress as they are supposed to run forever, nonetheless this heuristic proved to be crucial in managing the verification memory. For Herschel in particular, we exploit the

[3] This option is available since version 4.1.16 on Linux.



```
const int CYCLE = 250000; // one cycle duration
const int CYCLELIMIT = 2; // partition into two cycles
const int LIMIT = 16*250000; // explore up to 16 cycles
int cycle = 1;
progress { cycle; }
```

Figure 5: Progress measure based on cycles.

idea that most of the tasks settle down to Idle state in $250ms$ periods (see period specification in Table 7) and thus introduce a progress measure based on $250ms$ periods. Figure 5 shows an auxiliary automaton responsible for maintaining execution times of tasks, whose edges include periodic cycle increments. In our preliminary experiments we also limit our exploration with global invariant TIME<=LIMIT like in bounded model checking [BCCZ99], but otherwise the cycle variable traverses through all integers until CYCLELIMIT and then restarts from 1 again, thus allowing to explore an entire state space, while storing only a fraction of the state space mostly with the same (largest) progress number. The larger CYCLELIMIT the finer partitioning of the state space and hence larger memory savings, but memory savings come at the expense of performance: Uppaal may need to re-explore equivalent or similar states when the progress measure decreases (or drops to 1 in our case). The analysis can be made more precise by using smaller CYCLE values, which effectively splits the symbolic states into smaller zones at the expense of exploring more symbolic states.

Table 6 provides a summary of the effort spent in verifying the model. We started verification with model-time limited instances to get an impression of resources need to verify the model and once we gained enough confidence we increased the limit, thus the results are sorted by the model-time limit. The deterministic case of $f = 100\%$ is relatively cheap and even the unlimited case is verifiable within three hours. An important remark here is that the verification time correlates linearly with the limit and the unlimited case seems to correlate with 156 cycles, which is the least common multiple of all task periods. We managed to verify down to $f = 90\%$ where the verification time increased drastically to more than 6 days. Finally, for the case $f = 86\%$ the model-checker indicates a (possibly spurious) deadline violation after a little bit more than 4 days, which means that such a configuration is possibly unsafe. We also attempted BCET values in between 86 and 90% but the tool ran out of memory, hence no data available. The results from

Table 6: Verification statistics for different task execution time windows and exploration limits: the percentage denotes difference between WCET and BCET, limit is in terms of 250ms cycles (∞ stands for no limit, i.e. full exploration), states in millions, memory in MB, time in hours:minutes:seconds.

limit	$f = 100\%$			$f = 95\%$		
cycle	states	mem	time	states	mem	time
1	0.001	51.2	1.47	0.5	83.0	15:03
2	0.003	53.7	2.45	0.8	96.8	27:00
4	0.005	54.5	4.62	1.5	97.2	48:02
8	0.010	54.7	8.48	2.8	97.8	1:28:45
16	0.020	55.3	16.11	5.4	112.0	2:45:52
∞	0.196	58.8	2:39.64	52.7	553.9	27:05:07
limit	$f = 90\%$			$f = 86\%$		
cycle	states	mem	time	states	mem	time
1	1.5	124.1	1:22:43	3.3	186.9	6:39:47
2	2.4	139.7	2:09:15	5.3	198.7	9:14:59
4	4.4	138.3	3:48:40	9.2	274.6	14:12:57
8	9.1	156.5	8:38:42	18.2	364.6	28:35:32
16	17.8	176.0	16:42:05	35.4	520.4	44:06:57
∞	181.9	1682.2	147:23:25	**pos.unsafe**		99:07:56

Table 6 also shows that it takes more effort to prove schedulability when BCET values are lower, i.e. there is higher degree of non-determinism (larger state space) and hence the system is much less predictable if BCET values are not available.

Since symbolic MC proves that $f = 90\%$ case is safe, we also computed WCRT upper bounds. Table 7 compares the UPPAAL bounds on WCRTs with the bounds from classical response time analysis performed by Terma A/S. In particular Terma A/S found that `PrimaryF` task (#21) might violate its deadline even though this violation has never been observed neither in simulations nor in system deployment, whereas the bound provided by UPPAAL is still within the deadline, thus (re)confirming schedulability.

As the next section shows, it is difficult to hit error states with statistical model-checking. With model-checking, the verification time can take several days and depends on the number of cycles. We tried to confirm the error traces found by UPPAAL with polyhedra but we have not succeeded yet. What we did was to let UPPAAL generate a trace, try to confirm, resume the search in case of failure and repeat that every time an error state is hit. We stopped after several thousands of paths that could not be confirmed[4]. This does not mean that there is no trace. In fact our SMC technique finds such traces.

Table 7: Specification and worst-case response-times of individual tasks.

ID	Task	Specification		
		Period	WCET	Deadline
1	RTEMS_RTC	10.000	0.013	1.000
2	AswSync_SyncPulseIsr	250.000	0.070	1.000
3	Hk_SamplerIsr	125.000	0.070	1.000
4	SwCyc_CycStartIsr	250.000	0.200	1.000
5	SwCyc_CycEndIsr	250.000	0.100	1.000
6	Rt1553_Isr	15.625	0.070	1.000
7	Bc1553_Isr	20.000	0.070	1.000
8	Spw_Isr	39.000	0.070	2.000
9	Obdh_Isr	250.000	0.070	2.000
10	RtSdb_P_1	15.625	0.150	15.625
11	RtSdb_P_2	125.000	0.400	15.625
12	RtSdb_P_3	250.000	0.170	15.625
13	(no task, this ID is reserved for priority ceiling)			
14	**FdirEvents**	250.000	5.000	230.220
15	**NominalEvents_1**	250.000	0.720	230.220
16	**MainCycle**	250.000	0.400	230.220
17	HkSampler_P_2	125.000	0.500	62.500
18	HkSampler_P_1	250.000	6.000	62.500
19	Acb_P	250.000	6.000	50.000
20	IoCyc_P	250.000	3.000	50.000
21	**PrimaryF**	250.000	34.050	**59.600**
22	**RCSControlF**	250.000	4.070	239.600
23	Obt_P	1000.000	1.100	100.000
24	Hk_P	250.000	2.750	250.000
25	StsMon_P	250.000	3.300	125.000
26	TmGen_P	250.000	4.860	250.000
27	Sgm_P	250.000	4.020	250.000
28	TcRouter_P	250.000	0.500	250.000
29	Cmd_P	250.000	14.000	250.000
30	**NominalEvents_2**	250.000	1.780	230.220
31	**SecondaryF_1**	250.000	20.960	189.600
32	**SecondaryF_2**	250.000	39.690	230.220
33	Bkgnd_P	250.000	0.200	250.000

ID	WCRT			
	Terma	$f = 100\%$	$f = 95\%$	$f = 90\%$
1	0.050	0.013	0.013	0.013
2	0.120	0.083	0.083	0.083
3	0.120	0.070	0.070	0.070
4	0.320	0.103	0.103	0.103
5	0.220	0.113	0.113	0.113
6	0.290	0.173	0.173	0.173
7	0.360	0.243	0.243	0.243
8	0.430	0.313	0.313	0.313
9	0.500	0.383	0.383	0.383
10	4.330	0.533	0.533	0.533
11	4.870	0.933	0.933	0.933
12	5.110	1.103	1.103	1.103
13	(no task, this ID is reserved for priority ceiling)			
14	7.180	5.553	5.553	5.553
15	7.900	6.273	6.273	6.273
16	8.370	6.273	6.273	6.273
17	11.960	5.380	7.350	8.153
18	18.460	11.615	13.653	14.153
19	24.680	6.473	6.473	6.473
20	27.820	9.473	9.473	9.473
21	**65.47**	54.115	56.382	58.586
22	76.040	53.994	56.943	58.095
23	74.720	2.503	2.513	2.523
24	6.800	4.953	4.963	4.973
25	85.050	17.863	27.935	28.086
26	77.650	9.813	9.823	9.833
27	18.680	14.796	14.880	14.973
28	19.310	11.896	11.906	14.442
29	114.920	94.346	99.607	101.563
30	102.760	65.177	69.612	72.235
31	141.550	110.666	114.921	122.140
32	204.050	154.556	162.177	165.103
33	154.090	15.046	139.712	147.160

4 Statistical Analysis

In the previous section, we observed that symbolic MC can be used to conclude schedulability, but not to disprove it. This is reflected in the first line of Table 5 where

[4] This procedure decreases drastically the throughput of the exploration of the state space.

there is a wide range of values of f for which symbolic model-checking cannot conclude due to the potential presence of spurious counterexamples. In this section, we introduce statistical model-checking (SMC), a technique that we consider here to be the dual of symbolic model-checking. Namely, SMC can be used to disprove schedulability, but not to prove it.

Concretely, SMC is a simulation-based approach whose core objective is to estimate the probability for a system to satisfy a property by simulating and observing some of its executions, and then apply statistical algorithms to obtain the result. SMC is parameterized by two parameters: a *confidence interval size* on the estimate of the probability and a *confidence level* on the probability that the answer returned is correct. In terms of schedulability, SMC will thus be useful to generate concrete counterexample but cannot be used to conclude schedulability.

Several SMC algorithms have recently been implemented in UPPAAL [DLL+11b]. In this section, we will show how this implementation can be used not only to prove schedulability, but also to observe and reason on the execution of tasks. The latter will be done by exploiting the simulation engine and various information displayed by the GUI of the tool.

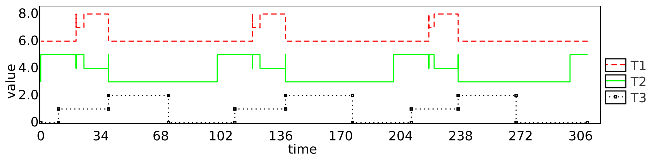
SMC relies on the assumption that the dynamics of the system is entirely stochastic. In [DLL+11a, DLL+11b], we have proposed a refined stochastic semantics for timed automata that associates probability distributions to both the time-delays spent in a given location as well as branching transitions. In this semantics timed automata components repeatedly race against each other, i.e. they independently and stochastically decide on their own how much to delay before outputting, with the "winner" being the component that chooses the minimum delay. Our stochastic schedulability model exploits the semantics of [DLL+11a,DLL+11b] as it assumes the execution time of the task to be picked uniformly in the interval $[f \cdot \mathrm{WCET}_i, \mathrm{WCET}_i]$.

In the rest of this section, we shall see how the SMC approach can be used to generate witness traces when concluding that the system is not schedulable with a probability greater than 0. We will also illustrate how the SMC engine of UPPAAL can evaluate the probability to reach a state violating a deadline. Finally, and not to be underestimated, we will show how the GUI of the UPPAAL tool can be exploited to give quantitative feedback to the user on, e.g., blocking time, CPU usage, distribution of response time.

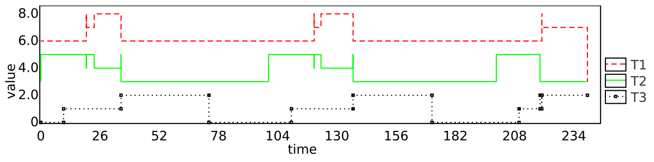
Running Example. Table 8 shows the query used to evaluate the probability of violating a deadline for runs bounded by 200 time units and the results for different values of f. We check only for cases when the symbolic model-checker reports that deadlines *may* be violated to generate a witness with SMC. The SMC technique gives results with certain levels of confidence and precision, i.e., the actual result is an interval. However, if the lower bound of the interval is greater than zero, this guarantees that the checker did find at least one witness. The case $f = 80\%$ is interesting because it seems to be a spurious result from the symbolic model-checker. In fact we can use cheaper hypothesis testing to show that the upper bound for the probability of finding an error is very low: the model-checker accepts the hypothesis Pr[<=200](<> error)<= 0.00001 with 99% confidence just in 25s, hence the error is very rare if possible at all. As summarized on line 2 of Table 1 SMC allows to confirm non-schedulability for $f \leq 79\%$ where the lower bound for the probability of finding an error is strictly greater than zero, i.e. there exist at least one concrete simulation run leading to an error.

Table 8: Probability of error estimation with 99% level of confidence: `Pr[<=200](<> error)`.

50%	70%	79%	80%
$[0.847, 0.858]$	$[0.604, 0.615]$	$[0.301, 0.312]$	$[0, 0.005]$



(a) Normal run using $f = 80$.



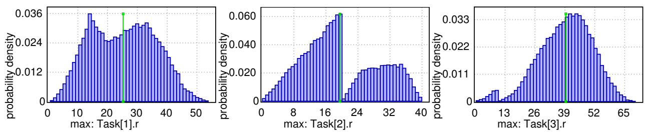
(b) Task 1 failed using $f = 79$.

Figure 6: Visualization of runs as a Gantt chart. The chart shows an encoding of the state with different weights corresponding to steps of different heights.

We can visualize traces (and inspect witnesses of deadline violation) by asking the checker to generate random simulation runs and visualize the value of a collection of expressions as a function of time in a Gantt chart. In addition, we can filter these runs and only retain those that reach some state, here the error state. This is done with the following query producing the plot in Fig. 6b:

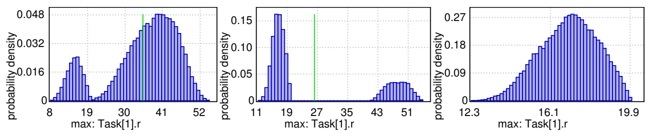
```
simulate 1000 [<=300] {
  (T(1).Ready+T(1).Computing+T(1).Release+runs[1]-2*T(1).Error)+6,
  (T(2).Ready+T(2).Computing+T(2).Release+runs[2]-2*T(2).Error)+3,
  (T(3).Ready+T(3).Computing+T(3).Release+runs[3]-2*T(3).Error)+0
} : 1 : error
```

If the filtering ("`:1:error`") is omitted, the plot contains all the runs, and for clarity just a single of them is displayed in Fig. 6a. As a result the plot encodes the task states (idle, ready, running or error) in the level of the curve. For example, Fig 6a shows that `T2` becomes ready and running starting from 0 time. At 10 time units



(a) Task `T1`. (b) Task `T2`. (c) Task `T3`.

Figure 7: Response time distributions for $f = 0\%$ show that lowest priority task `T3` is almost undisturbed, while `T1` and `T2` timings seems to be split into two parts implying that they are blocked and delayed by at least one task, moreover `T1` is far overdue past deadline at 20.



(a) 50% (not safe). (b) 79% (not safe). (c) 80% (seem OK).

Figure 8: Response time distributions for task `T1` show probability of unsafe cases (response times beyond 20) shrink and disappear when tweaking f from unsafe 50% to safe 80%, meaning that the system might be safe provided `T1` is not faster than 80% of its WCET.

task `T3` becomes ready, but is not running. Then at 20 time units task `T1` becomes ready and becomes running by preempting `T2` but then it immediately gives up the running status (due to resource blocking) and resumes by preemption when `T2` releases the resource. At this point `T2` is not finished yet and will be able to finish only when `T1` finishes and releases the CPU, hence there is a small spike just before going to the idle state. The lowest priority task `T3` has a chance to run and finish only when both `T1` and `T2` are done. Figure 6b is interpreted similarly, where task `T1` violates its deadline because `T3` managed to get the resource before `T1` and thus `T1` was blocked from finishing.

More insight on the behavior of the tasks is gained by estimating expected response times (maximums over a single run) using the queries:

```
E[<=200; 50000] (max: T(1).r)
E[<=200; 50000] (max: T(2).r)
E[<=200; 50000] (max: T(3).r)
```

The result is that the response time averages respectively: 16.96, 36.96 and 63.65 time units. In addition, the tool provides the probability densities over the maximum response times shown in Fig. 7. The plots show the effect of priority inversion on the higher priority tasks hindered by the lower priority task T3.

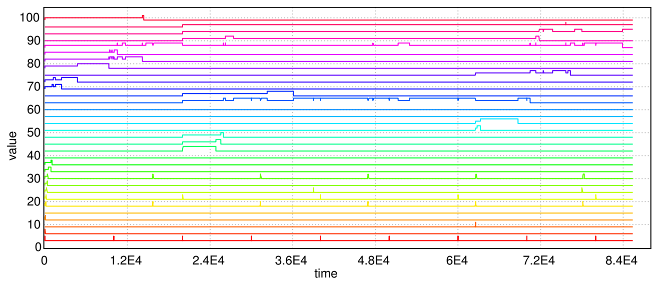
The response of `T1` goes beyond the deadline for $f = 0\%$, thus we evaluate the shapes of response time distributions for various f values in Fig. 8. Surprisingly there is a sharp contrast between $f = 79\%$ (unsafe for sure) and $f = 80\%$ which does not seem to exhibit the error and responds within 20 time units. This worst response time is more optimistic than the case $f = 83\%$ from symbolic analysis, which suggests that the symbolic analysis most probably is not exact for $f \in [80, 83]$. Figure 8a is an intermediate result between $f = 0\%$ (Fig. 7a) and $f = 79\%$ where the two seemingly normal "hills" are wide enough to meet each other, thus Fig. 7a is the result of two "hills": one from safe responses and the other slipped beyond a safety threshold but they are overlapping so tightly that this fact is hardly evident in Fig. 7a.

Table 9: Results of Herschel statistical model-checking.

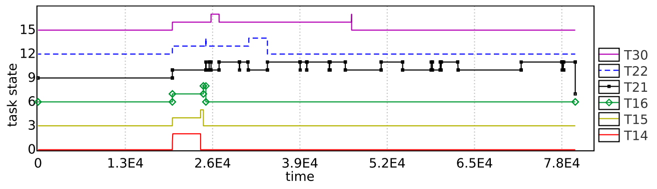
Limit	f	Total	Error traces		Earliest Error		Verification
cycles	%	runs, #	#	Prob.	cycle	offset	time
1	0	105967	1928	0.018194	0	79600.0	1:58:06
1	50	105967	753	0.007106	0	79600.0	2:00:52
1	60	105967	13	0.000123	0	79778.3	2:01:18
1	62	1036757	34	0.000033	0	79616.4	19:52:22
160	63	1060	177	0.166981	0	81531.6	2:47:03
160	64	1060	118	0.111321	1	79803.0	2:55:13
160	65	738	57	0.077236	3	79648.0	2:06:55
160	66	1060	60	0.056604	2	82504.0	2:62:44
160	67	1060	26	0.024528	1	79789.0	2:64:20
160	68	1060	3	0.002830	67	81000.0	2:67:08
640	69	1060	8	0.007547	114	80000.0	12:23:00
640	70	1060	3	0.002830	6	88070.0	12:30:49
1280	71	1060	2	0.001887	458	80000.0	25:19:35
640	71	7	1		21	80000	5:15
640	72	951	1		521	80000	11:04:26
1280	73	1734	1		1027	85000	40:46:05

Herschel. We apply this methodology to our more complex Herschel case-study to confirm deadline violations and to study performance.

Table 9 shows the results when we vary the execution times within the interval $[f \cdot \text{WCET}, \text{WCET}]$ and use the following query: Pr[<=160*250000](<> error). The table shows the probabilities in function of this factor f. We varied significance (α) and probability precision (ϵ) statistical parameters to achieve different number of runs. Since the model is complicated and search for counter examples is time consuming, the experiment resembles a manual search for the boundary while running multiple queries in parallel with different settings of statistical parameters and relaunching queries with more demanding settings if no error is found. The obtained counter examples are eventually sorted by the model parameter f. At first we limited the search to just one cycle of 250ms, but then at the point of $f = 62\%$ the errors are rarely found even with high confidence and many runs. Then we increased the limit which increased our chances of finding the errors, we were lucky to find some errors as early as in the first cycle. Most of the errors are found quite early (cases where $f < 68\%$), but for smaller time-windows it is much harder to find and the few found ones are quite far in the run. Eventually the search took more than a day to find only a few error instances for $f = 71\%$, hence



(a) A successful run with $f = 90$ (`PrimaryF` at level 63).



(b) Selected processes of a simulation run with $f = 50\%$, where `PrimaryF` (task `T21` at level 9) violates its deadline.

Figure 9: The first 85ms of Herschel model simulation run as displayed in UPPAAL SMC.

we stopped here. The new UPPAAL releases (since version 4.1.15) allow queries which stop exploration when a number of successful runs have been found. For example, the following would try 1000 runs, stop when the first error trace is found, and display trajectories of some diagnostic variables: `simulate 1000 [<=640*250000] {runs[21], ready[21]}:1: error`. The last three lines in Table 9 show some hard to find traces appended to older data (obtained without predicate in simulate query before UPPAAL version 4.1.15).

Similarly to Fig. 7, response times for the most stressed task `PrimaryF` are estimated by generating 2000 probabilistic runs limited to 156 cycles for the safe case of $f = 90\%$. The vast majority (1787) of instances responded before 51093.3 and the rest is distributed about evenly, which means that most runs have a good safe margin, and only rarely it is disturbed. The worst found response time was of 52851.2 which is significantly lower than bound of 58586.0 found by symbolic MC in Table 7. The computation for this model took 17.6 hours.

Figure 9a shows an overview chart of all 32 tasks interacting during the first 85ms. Each task can be identified by its base level 3*ID, thus `PrimaryF` with ID=21 is at 63. `PrimaryF` starts with an offset of 20ms and it has to finish before a deadline of 59.6ms. Under safe conditions of $f = 90\%$ `PrimaryF` finishes before $70500\mu s$ (Fig. 9a) but with $f = 50\%$ it fails at $79828.3\mu s$ (Fig. 9b). It seems that the overdue of `T21` is mostly caused by `T22` and `T30` lower priority tasks which in turn are delayed by high priority `T14` task.

5 Conclusion

In this paper, we have applied both symbolic MC and statistical MC to schedulability analysis. In particular, we have demonstrated that the complementary qualities of the two methods allow to conclusively confirm as well as disprove schedulability for a wide range of cases. This is an impressive result as the problem is known to be undecidable.

The experiments show that additional information such as BCET values is useful in reducing the complexity of symbolic model-checking while proving schedulability. Another important but counter-intuitive result is that the system can be schedulable with larger BCET values while non-schedulability has been shown for the same system except with smaller BCET values (or zero as in response time analysis). This means that developers have more possibilities to tweak the task model by changing priorities, resource sharing protocols, putting more task details as well as making the system more predictable by carefully padding execution times closer to WCET and hence guaranteeing schedulability.

In addition we have illustrated how the user can benefit from the UPPAAL features in plotting, observing and reasoning about task executions, and hence improving the modeling process. We also believe that the combination of symbolic MC and statistical MC will prove highly useful in analyzing systems with mixed critically, i.e. systems containing tasks with hard timing constraints as well as soft, where the timing constraints are permitted to be violated occasionally. In addition, we have presented an alternative symbolic technique using polyhedra that can confirm that some error trace are indeed realizable.

References

BACC+98. Hanene Ben-Abdallah, Jin-Young Choi, Duncan Clarke, Young Si Kim, Insup Lee, and Hong-Liang Xie. A process algebraic approach to the schedulability analysis of real-time systems. *Real-Time Systems*, 15:189–219, 1998. 10.1023/A:1008047130023.

BCCZ99. Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In W.Rance Cleaveland, editor, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1579 of *Lecture Notes in Computer Science*, pages 193–207. Springer Berlin Heidelberg, 1999.

BDL+12. Peter E. Bulychev, Alexandre David, Kim Guldstrand Larsen, Axel Legay, Marius Mikučionis, and Danny Bøgsted Poulsen. Checking and distributing statistical model checking. In *NASA Formal Methods*, volume 7226 of *Lecture Notes in Computer Science*, pages 449–463. Springer, 2012.

BHK99. Steven Bradley, William Henderson, and David Kendall. Using timed automata for response time analysis of distributed real-time systems. In *Systems, in 24th IFAC/IFIP Workshop on Real-Time Programming WRTP 99*, pages 143–148, 1999.

BHK$^+$04. H.C. Bohnenkamp, H. Hermanns, R. Klaren, A. Mader, and Y.S. Usenko. Synthesis and stochastic assessment of schedules for lacquer production. In *Quantitative Evaluation of Systems, 2004. QEST 2004. Proceedings. First International Conference on the*, pages 28 – 37, sept. 2004.

BHM09. Aske Brekling, Michael R. Hansen, and Jan Madsen. MoVES – a framework for modelling and verifying embedded systems. In *Microelectronics (ICM), 2009 International Conference on*, pages 149–152, dec. 2009.

Bur94. Alan Burns. *Principles of Real-Time Systems*, chapter Preemptive priority based scheduling: An appropriate engineering approach, page 225–248. Prentice Hall, 1994.

CKM01. Søren Christensen, Lars Kristensen, and Thomas Mailund. A Sweep-Line method for state space exploration. In *Tools and Algorithms for the Construction and Analysis of Systems*, TACAS 2001, pages 450–464, London, UK, 2001. Springer-Verlag.

CL00. Franck Cassez and Kim Guldstrand Larsen. The impressive power of stopwatches. In Catuscia Palamidessi, editor, *CONCUR*, volume 1877 of *Lecture Notes in Computer Science*, pages 138–152. Springer, 2000.

DILS10. Alexandre David, Jacob Illum, Kim G. Larsen, and Arne Skou. Model-based design for embedded systems. In Gabriela Nicolescu and Pieter J. Mosterman, editors, *Model-Based Design for Embedded Systems*, chapter Model-Based Framework for Schedulability Analysis Using UPPAAL 4.1, pages 93–119. CRC Press, 2010.

DLL$^+$11a. Alexandre David, Kim G. Larsen, Axel Legay, Marius Mikučionis, Danny Bøgsted Poulsen, Jonas Van Vliet, and Zheng Wang. Statistical model checking for networks of priced timed automata. In *FORMATS*, LNCS, pages 80–96. Springer, 2011.

DLL$^+$11b. Alexandre David, Kim G. Larsen, Axel Legay, Zheng Wang, and Marius Mikučionis. Time for real statistical model-checking: Statistical model-checking for real-time systems. In *CAV*, LNCS. Springer, 2011.

DLLM12. Alexandre David, Kim Guldstrand Larsen, Axel Legay, and Marius Mikučionis. Schedulability of Herschel-Planck revisited using statistical model checking. In *ISoLA (2)*, volume 7610 of *LNCS*, pages 293–307. Springer, 2012.

FKPY07. Elena Fersman, Pavel Krčál, Paul Pettersson, and Wang Yi. Task automata: Schedulability, decidability and undecidability. *Information and Computation*, 205(8):1149 – 1172, 2007.

HLMP04. Thomas Hérault, Richard Lassaigne, Frédéric Magniette, and Sylvain Peyronnet. Approximate probabilistic model checking. In Bernhard Steffen and Giorgio Levi, editors, *Verification, Model Checking, and Abstract Interpretation*, volume 2937 of *Lecture Notes in Computer Science*, pages 73–84. Springer Berlin Heidelberg, 2004.

JM09. Bertrand Jeannet and Antoine Miné. Apron: A library of numerical abstract domains for static analysis. In Ahmed Bouajjani and Oded Maler, editors, *Computer Aided Verification*, volume 5643 of *Lecture Notes in Computer Science*, pages 661–667. Springer Berlin Heidelberg, 2009.

JP86. Mathai Joseph and Paritosh K. Pandya. Finding response times in a real-time system. *Comput. J.*, 29(5):390–395, 1986.

KZH$^+$09. J-Pieter Katoen, I. S. Zapreev, E. Moritz Hahn, H. Hermanns, and D. N. Jansen. The ins and outs of the probabilistic model checker MRMC. In *Proc. of 6th Int. Conference on the Quantitative Evaluation of Systems (QEST)*, pages 167–176. IEEE Computer Society, 2009.

LDB10. Axel Legay, Benoît Delahaye, and Saddek Bensalem. Statistical model checking: An overview. In *RV*, volume 6418 of *Lecture Notes in Computer Science*, pages 122–135. Springer, 2010.

MLR$^+$10. Marius Mikučionis, Kim Guldstrand Larsen, Jacob Illum Rasmussen, Brian Nielsen, Arne Skou, Steen Ulrik Palm, Jan Storbank Pedersen, and Poul Hougaard. Schedulability analysis using Uppaal: Herschel-Planck case study. In Tiziana Margaria, editor, *ISoLA 2010 – 4th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation*, volume Lecture Notes in Computer Science. Springer, October 2010.

RP09. Diana Rabih and Nihal Pekergin. Statistical model checking using perfect simulation. In Zhiming Liu and Anders P. Ravn, editors, *Automated Technology for Verification and Analysis*, volume 5799 of *Lecture Notes in Computer Science*, pages 120–134. Springer Berlin Heidelberg, 2009.

SLC06. Oleg Sokolsky, Insup Lee, and Duncan Clarke. Schedulability analysis of AADL models. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, page 8 pp., april 2006.

SVA04. Koushik Sen, Mahesh Viswanathan, and Gul Agha. Statistical model checking of black-box probabilistic systems. In *CAV*, LNCS 3114, pages 202–215. Springer, 2004.

YS02. Håkan L.S. Younes and Reid G. Simmons. Probabilistic verification of discrete event systems using acceptance sampling. In Ed Brinksma and Kim Guldstrand Larsen, editors, *Computer Aided Verification*, volume 2404 of *Lecture Notes in Computer Science*, pages 223–235. Springer Berlin Heidelberg, 2002.

YS06. Håkan L. S. Younes and Reid G. Simmons. Statistical probabilistic model checking with a focus on time-bounded properties. *Inf. Comput.*, 204(9):1368–1409, 2006.