Schedulability Analysis of Herschel/Planck Software Using Uppaal

Marius Mikučionis and Kim G. Larsen and Brian Nielsen

Abstract This chapter shows how UPPAAL is applied in schedulability analysis of satellite attitude and orbit control software used in Herschel/Planck mission. Our method transforms the schedulability analysis into reachability analysis performed by UPPAAL. The chapter briefly describes the schedulability requirements and elaborates on the modeling framework designed to handle single processor hardware with a fixed priority preemptive scheduler, detailed task specifications, two resource sharing protocols and voluntary task suspension. The results include qualitative answers (whether the system is schedulable) as well as quantitative (response and blocking time estimates) which are comparable with classical response-time analysis.

Key words: schedulability analysis, timed automata, stop-watch automata, model-checking, verification

1 Introduction

The goal of schedulability analysis is to check whether all tasks finish before their deadline. Traditional approaches like [Burns(1994)] provide generic frameworks which assume worst-case scenario where consecutive response-times are calculated and compared with deadlines. Often, such conservative scenarios are never realized and thus negative results from such analysis may be too pessimistic. The idea is

Marius Mikučionis
Aalborg University, Aalborg, Denmark, e-mail: marius@cs.aau.dk

Kim G. Larsen
Aalborg University, Aalborg, Denmark, e-mail: kgl@cs.aau.dk

Brian Nielsen
Aalborg University, Aalborg, Denmark, e-mail: bnielsen@cs.aau.dk

to base the schedulability analysis on a system model with possibly more details, taking into account specifics of individual tasks. In particular this will allow a safe but far less pessimistic schedulability analysis to be settled using real-time model checking. Moreover, the model-based approach provides a self-contained visual representation of the system with formal, non-ambiguous interpretation, simulation and other possibilities for verification and validation.

Our model-based approach is motivated by and carried out on example applications in a case study of Herschel-Planck satellite system. Compared with classical response-time analysis our model-based approach is found to uniformly provide less pessimistic response-time estimates and allow to conclude schedulability of all tasks, in contrast to negative results obtained from the classical approach.

1.1 The Herschel-Planck Mission

The Herschel-Planck mission consists of two satellites: Herschel and Planck. The satellites have different scientific objectives and thus the sensor and actuator configurations differ, but both satellites share the same computational architecture. The architecture consists of a single processor, a real-time operating system (RTEMS), a basic software layer (BSW) and an application software (ASW).

The goal of the study is to show that ASW tasks and BSW tasks are schedulable on a single processor with no deadline violations. The tasks use preemptive fixed priority scheduler and a mixture of priority ceiling and priority inheritance protocols for resource sharing and extended deadlines (beyond period). In addition, some tasks need to interact with external hardware and effectively suspend their execution for a specified time. Due to suspension, this single-processor system has some similarity to multi-processor systems since parts of activities are executed elsewhere and the classical worst-case response-time analysis (applicable to single-processor systems) is pushed to its limits. One of the results of [Palm(2006)] is that one task may miss its deadline on Herschel (and thus the system is not schedulable) but this violation has never been observed in neither stress testing nor deployment.

Figure 1 shows the parameters which describe each periodic task: period defines how often the task is started, offset – how far into the cycle the task is started (released), deadline is measured from the instance when task is started and worst-case execution time within deadline.

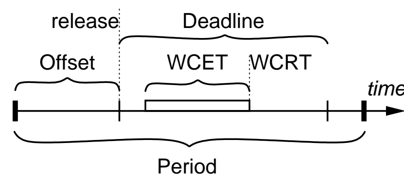


Fig. 1: Task time bounds.

Some tasks access shared resources and those are protected by semaphore locking to ensure exclusive usage. Sometimes tasks use resources repeatedly (locking and unlocking several times). When the resource semaphore is locked, a task may suspend its execution by calling hardware services and waiting for the hardware to finish thus temporarily releasing the processor for other tasks. The processor may be released multiple times during one semaphore lock. In response-time analysis, the processor utilisation is computed by dividing the sum of worst-case execution times by the duration of analysed time window.

Table 1 shows the description of the primary functions task from [Palm(2006)]. The task consists of six activities. Each activity is described by two numbers: CPU time / BSW service time (BSW service time is included in CPU time), followed by resource usage pattern if any. The resource usage is described by the following parameters:

LNS – total number of times the CPU has been released while the resource was locked (task suspension count).
LCS – total time the CPU has been released while the resource was locked (task suspension duration).
LC – total time the resource has been locked.
MaxLC – the longest time the resource has been locked.

For example "Data processing" takes $20577\mu s$ in total, from which it has locked the resource `Icb_R` for $1600\mu s$, and from which CPU has been released (execution suspended) for $1200\mu s$.

Table 1: The sequence of primary functions task from [Palm(2006)].

Primary Functions	
- Data processing	**20577**/2521
	Icb_R(LNS: 2, LCS: **1200**, LC: **1600**, MaxLC: 800)
- Guidance	**3440**/0
- Attitude determination	**3751**/1777
	Sgm_R(LNS: 5, LCS: **121**, LC: **1218**, MaxLC: 236)
- PerformExtraChecks	**42**/0
- SCM controller	**3479**/2096
	PmReq_R(LNS: 4, LCS: **1650**, LC: **3300**, MaxLC: 3300)
- Command RWL	**2752**/85

2 Model-Checking Schedulability Methodology

The main idea is to translate schedulability analysis problem into a reachability problem for timed automata and use the real-time model-checker UPPAAL to check that none of the deadlines are violated, derive worst-case blocking and response-

times and processor utilization. We refer to the previous chapter for UPPAAL concepts.

Figure 2 shows the work-flow of response-time analysis (performed by Terma A/S) and schedulability analysis using UPPAAL: the task timing informations are obtained from ASW and BSW documentation, worst-case execution times (WCET) of BSW are obtained from BSW documentation [Terma A/S(Issue 9)] and ASW timings are obtained from simulation measurements. In addition the UPPAAL model uses information about the individual task flows, i.e. the timing of resource locks, CPU execution and suspension.

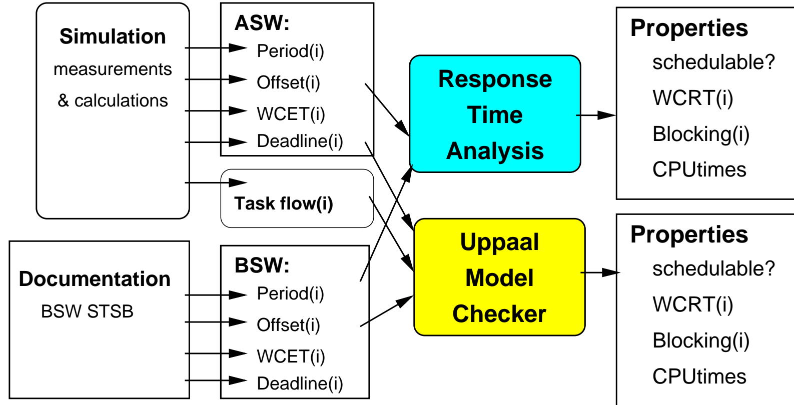


Fig. 2: Work-flow of schedulability analysis.

The UPPAAL framework consists of the following process models: a fixed priority preemptive CPU scheduler, a number of task models, and one process for ensuring global invariants. We provide different templates for task models: one for periodic tasks and several for tasks with dependencies, all of which are parameterised with explicit sequence of task actions and may be customised to a particular resource sharing protocol. We also investigate the scalability of the approach by allowing different best-case execution times (BCET) as a percentage discount from WCET. In practice it is possible to put realistic BCETs, but we choose this parametrisation for the sake of systematic exploration. Our approach uses the same task descriptions as [Palm(2006)].

The following outlines the main modelling ingredients:

- One template for the CPU scheduler.
- One template for the "idle" task to keep track of CPU usage times.
- One template for all BSW tasks, where resources are locked based on priority inheritance protocol.
- One template for the MainCycle ASW task, which is released periodically, starts other ASW tasks and locks resources based on the priority ceiling protocol.

- One template for all other ASW tasks, which are released by synchronisations, and locks resources based on priority ceiling protocol.
- Task specialisation is performed during process instantiation by providing individual list of operations encoded into a *flow* array.
- Each task (either ASW or BSW) uses the following clocks and data variables:
  - Task and its clocks are parameterised by an identifier *id*.
  - A local clock x controls periodic releases of the task. The task then moves to an error state if x is greater than its deadline.
  - A local clock *sub* controls progress and execution of individual operations.
  - A local integer *ic* is an operation counter.
  - The worst-case response-time for task *id* is modelled by a stopwatch $WCRT[id]$ which is reset when the task is started and is allowed to progress only when the task is ready (global invariant $WCRT[id]' == ready[id]$ ensures that). The worst-case response-time is estimated as maximum value of $WCRT[id]$.
  - An `error` location is reachable and *error* variable is set to *true* if there is a possibility to finish after deadline.

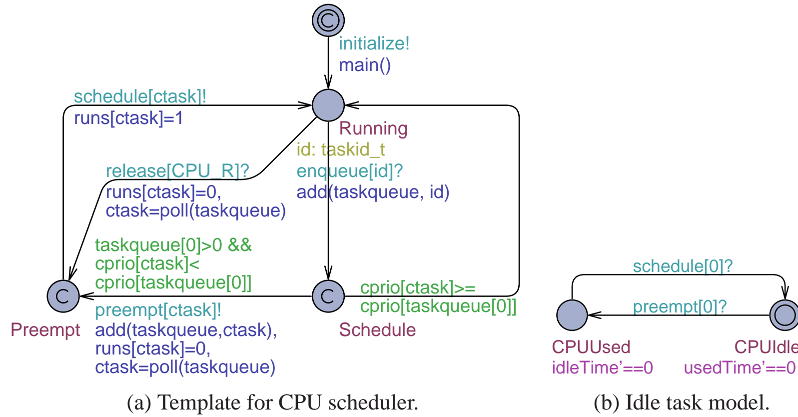
Further we explain the most important model templates, while the complete model is available for download at http://www.cs.aau.dk/~marius/Terma/ .

2.1 Scheduler Model

Figure 3a shows the model of the scheduler. In the beginning, the Scheduler initialises the system (computes the current task priorities by computing default priority based on `id` and starts the tasks with zero offset) and in location `Running` waits for tasks to become ready or current task to release the CPU resource. When some task becomes ready, it adds itself to the `taskqueue` and signals on the `enqueue` channel, thus moving the Scheduler to location `Schedule`. From the location `Schedule`, the Scheduler compares the priority of a current task `cprio[ctask]` with the highest priority in the queue `cprio[taskqueue[0]]` and either returns to `Running` (nothing to reschedule) or preempts the current task `ctask`, puts it into `taskqueue` and schedules the highest priority task from `taskqueue`.

A task releases the CPU by a signal `release[CPU_R]`, in which case the Scheduler pulls the highest priority task from `taskqueue` and optionally notifies it with broadcast synchronisation on channel `schedule`.

The `taskqueue` always contains at least one ready task: IdleTask. Figure 3b shows how IdleTask reacts to Scheduler events. It also computes the CPU usage time using stopwatch `usedTime` and the total CPU load is then calculated as $\frac{\text{usedTime}}{\text{globalTime}}$.



(a) Template for CPU scheduler. (b) Idle task model.

Fig. 3: Models for CPU scheduler and the simplest task.

2.2 *Tasks Templates*

Task template is a generalization of a task process. We provide three task templates which share the same timed automata structure except some minor differences: BSW (started periodically, uses priority inheritance), ASW (started by other task, uses priority ceiling) and MainCycle (started periodically, starts other tasks and uses priority ceiling). The templates are instantiated with a concrete task description: period, offset, deadline and resource usage sequence we call task flow.

Figure 4 shows a template used by MainCycle which is started periodically. At first MainCycle waits for `Offset` time to elapse and moves to location `Idle` by setting the clock x to `Period`. Then the process is forced to leave the `Idle` location immediately, to signal other ASW tasks, add itself to the ready task queue and arrive to location `WaitForCPU`. When MainCycle receives notification from the scheduler it moves to location `GotCPU` and starts processing commands from the *flow* array.

Declaration of task flow array type is shown in Fig. 5a: `flow_t` is an array of operations `operation_t`, and operations are tuples of operation type `optype_t`, resource identifier `resid_t` and a timing argument `time_t` which is an integer.Figure 5b shows the beginning of the flow for the primary function task.

There are four types of operations:

1. LOCK is executed from location `tryLock` where the process attempts to acquire the resource. It blocks if the resource is not available and retries by adding itself to the processor queue again when the resource is released. It continues to location `Next` by locking the resource if the resource is available.
2. UNLOCK simply releases the resource and moves on to location `Next`. The implementation of locking and unlocking for both protocols is straightforward and fits into 28 lines of code.

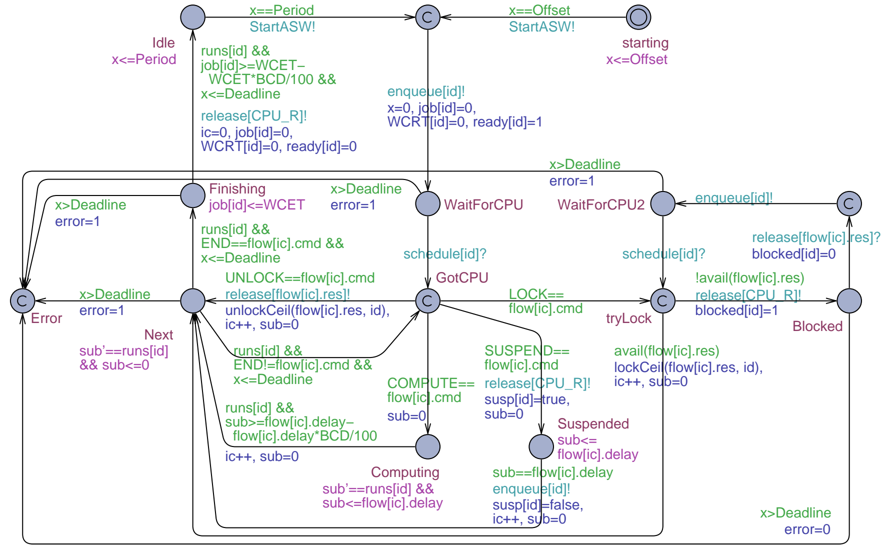


Fig. 4: MainCycle task: periodically starts ASW functions.

optype_t	::=	END | COMPUTE | LOCK | UNLOCK | SUSPEND
resid_t	::=	Icb_R | Sgm_R | PmReq_R | Other_R
time_t	::=	int[0, 10000000]
operation_t	::=	optype_t resid_t time_t
flow_t	::=	operation_t*

(a) Declaration of task flow type.

ic:	cmd:	res:	delay:
0	LOCK	Icb_R	0
1	COMPUTE	CPU_R	400
2	SUSPEND	CPU_R	1200
3	UNLOCK	Icb_R	1200
4	COMPUTE	CPU_R	20177
	...	...	...
18	END		

(b) Flow of primary functions task.

Fig. 5: Structure and an instance of task flow.

3. SUSPEND releases the processor for the specified amount of time, adds itself to the queue and moves to location `Next`. The task progress clock $job[id]$ is not increasing but the response measurement clock $WCRT[id]$ is.
4. COMPUTE makes the task stay in location `Computing` for the specified duration of CPU time, i.e. the clock *sub* is stopped whenever the task is preempted ($runs[id]$ is set to 0). Once the required amount of CPU time is consumed, the process moves on to location `Next`. For scalability study we relax the guard by `BCD` percent of time, allowing the task to finish slightly earlier than WCET.

From location `Next`, the process is forced by the $runs[id]$ invariant to continue with the next operation: if it is not the END and it is running, then it moves back to `GotCPU` to process next operation, and it moves to `Finishing` if it's the END. In the `Finishing` location the process consumed CPU for the remaining time so that the complete WCET is exhausted and then it moves back to `Idle`. From locations

`Next` and `Finishing` the outgoing edges are constrained to check whether the deadline has been reached since the last task release (when x was set to 0), and edges force the process into `Error` location if $x > Deadline$.

The *flow* for MainCycle is trivial: it computes for its WCET while keeping a lock on Sgm_R. A more sophisticated example of flow is shown in Listing 1 where the timing numbers are taken from description in Table 1: the task attempts to lock the resource `Icb_R`, when the resource is locked it actively uses the CPU for 400μs (because according to the description the resource is locked for 1600μs and CPU is not used for 1200μs due to suspension), then CPU is suspended for 1200μs, `Icb_R` is released and CPU is used for the remaining task execution.

Listing 1: The data processing part of operation flow for PrimaryF task.

```
const ASWFlow_t PF_f = { // Primary Functions, ----- Data processing:
    { LOCK,    Icb_R,  0 },                // 0)  acquire lock on Icb_R
    { COMPUTE, CPU_R, 1600−1200 }, // 1) execute with Icb_R being locked
    { SUSPEND, CPU_R, 1200 },          // 2)  suspend/release CPU, while Icb_R is locked
    { UNLOCK, Icb_R, 0 },                 // 3)  release lock on Icb_R
    { COMPUTE, CPU_R, 20577−(1600−1200) }, // 4) execute without Icb_R
    ...
};
```

The template for BSW tasks is almost the same as MainCycle, except that 1) BSW tasks do not have to start other ASW tasks and thus from `Idle` they go directly to `WaitForCPU` with enqueueing edge, 2) instead of the ceiling protocol (*lockCeil* and *unlockCeil*) it uses priority inheritance (*lockInh* and *unlockInh*) and 3) it boosts the owners priority by calling $boostPrio(flow[ic].res, id)$ on the edge from `tryLock` to `Blocked`. BSW tasks have their own local clock x, while MainCycle shares its x with other ASW tasks.

We use only LCS (CPU suspension time while resource is locked) and LC (total locking time) from Table 1, where we assume that LC−LCS is the CPU busy time while the resource is locked.

Listing 1 shows an example of detailed control flow structure for PrimaryF task, where the numbers mean the time duration and comments relate each step to an item in Table 1.

2.3 System Model Instantiation

Listing 2 shows how tasks are instantiated with task identifier, offset, period, flow, deadline and shared ASW clock. In total there are 32 tasks, where id=13 is reserved for priority ceiling.

Listing 2: Task instantiation.

```
//                        taskid , Offset , Period,  flow ,     WCET, Deadline
RTEMS_RTC      =       BSW(1,     0,  10000,  WCET_f,   13,       1000);
AswSync_SyncPulseIsr=BSW(2,   0,250000,  WCET_f,   70,       1000);
Hk_SamplerIsr   =       BSW(3,62500,125000, WCET_f, 70,       1000);
 ...
mainCycle =    MainCycle(16,20000,250000,  400,   230220,  ASWclock);
```

```
...
primaryF  =          ASW(21,StartASW,Done, PF_f, 34050, 59600, ASWclock);
...
Bkgnd_P        =     BSW(33,    0,250000,  WCET_f, 200,  250000);
```

Listing 3 shows system declaration with a . The variable `cycle` counts cycle number as an heuristic progress measure which allows UPPAAL to use the sweep-line method to reduce the verification memory consumption. The cycle is incremented after a period of 250ms and is being reset after some specified `CYCLELIMIT` in the Global process. The process Global also takes care of global invariants on $job[i]$ and $WCRT[i]$ stopwatches of each task i.

Listing 3: System declaration using UPPAAL priorities.

```
system Scheduler, RTEMS_RTC, AswSync_SyncPulseIsr, Hk_SamplerIsr, SwCyc_CycStartIsr,
     SwCyc_CycEndIsr, Rt1553_Isr, Bc1553_Isr, Spw_Isr, Obdh_Isr, RtSdb_P_1, RtSdb_P_2, RtSdb_P_3,
     FdirEvents, NominalEvents_1, mainCycle, HkSampler_P_2, HkSampler_P_1, Acb_P, IoCyc_P,
     primaryF, rCSControlF, Obt_P, Hk_P, StsMon_P, TmGen_P, Sgm_P, TcRouter_P, Cmd_P,
     NominalEvents_2, secondF_1, secondF_2, Bkgnd_P, IdleTask, Global;
progress { cycle; }
```

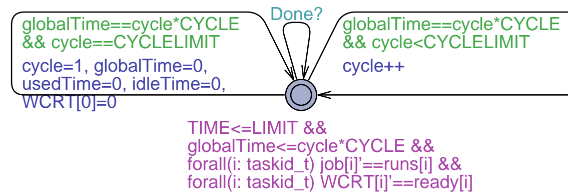


Fig. 6: Global process enforce invariants on stopwatches and cyclic progress.

2.4 *Verification Queries*

The following is a list of queries used to check schedulability properties:

- Check if the system is schedulable (the error state is not reachable): `E<> error`
- Check if any task can be blocked at all: `E<> exists(i:taskid_t) blocked[i]`
- Find the total worst CPU usage: `sup: usedTime, idleTime`
- Find the worst-case response-times: `sup: WCRT[0], WCRT[1], ... WCRT[33]`
- Find worst-case blocking times, where $B[i]$ is a stopwatch growing when task i is blocked: `sup: B[0], B[1], B[2], ... B[33]`

A `sup`-query explores the entire reachable state space and computes the maximum (supremum) value of an argument expression. This is particularly useful for computing several bounds at once.

3 Results

Our results provide three important pieces of information: visualisation of a schedule in a Gantt chart, worst-case response-times estimates and CPU utilisation and verification benchmarks.

A Gantt chart can be used to visualise a trace of the system, thus providing a rich picture for inspection. For example, the generated Gantt chart in Figure 7 shows that Cmd_P is blocked more than 5 times during the first cycle, while blocking times for PrimaryF (21) and StsMon_P (25) are significantly long only starting from the second cycle.

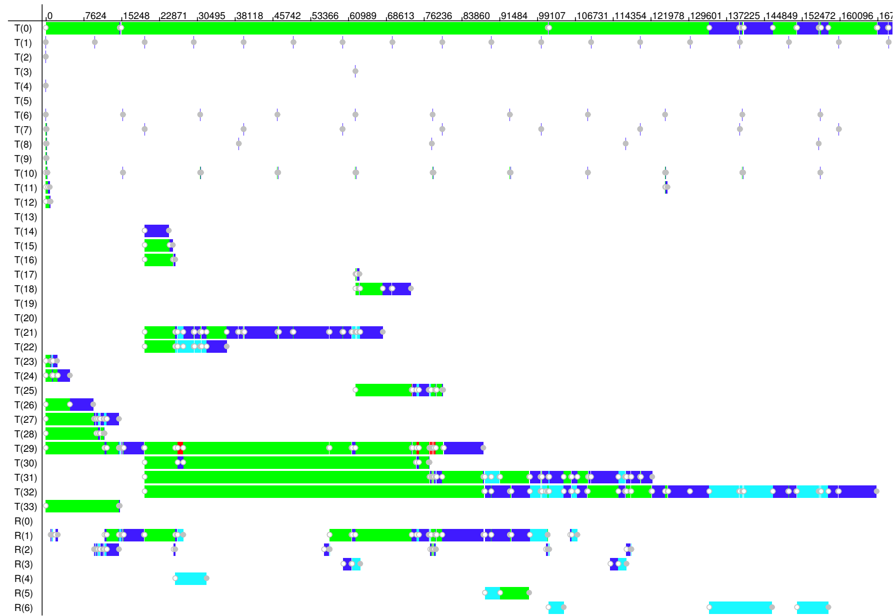


Fig. 7: Gantt chart of the first cycle, generated by UPPAAL TIGA: task T(i) is green when ready, blue – executing, red – blocked, cyan – suspended, resource R(j) is blue when locked and owner uses CPU, green – locked but the owner is preempted, cyan – locked but owner is suspended.

In [Palm(2006)] the CPU utilisation for a 20-250ms window is estimated as 62.4%. Our estimate for the entire worst-case cycle is 63.65% which is slightly larger, possibly due to the fact that it also includes the consumption during the 0-20ms window. See [Mikučionis et al(2010)Mikučionis, Larsen, Rasmussen, Nielsen, Skou, Palm, Pedersen, and Hou for additional insight on how the cycle limit affects verification resources and results.

Table 2 shows the worst-case response-times obtained from UPPAAL analysis with 0%, 5% and 10% BCET deviation from WCET in comparison with response-

times acquired by Terma. We note that in all cases the WCRT estimates provided by UPPAAL are smaller (hence less pessimistic) than those originally obtained [Palm(2006)]. In particular, we note that the task PrimaryF (task 21) is found to be schedulable using model-checking with up to 10% deviation for best-case execution times, but most probably not schedulable from 14% (a trace leading to deadline violation is found), in contrast to the original negative result obtained by Terma.

Table 2: Specification, blocking and worst-case response-times of individual tasks.

		Specification			WCRT			
ID	Task	Period	WCET	Deadline	Terma	0%	5%	10%
1	RTEMS_RTC	10.000	0.013	1.000	0.050	0.013	0.013	0.013
2	AswSync_SyncPulseIsr	250.000	0.070	1.000	0.120	0.083	0.083	0.083
3	Hk_SamplerIsr	125.000	0.070	1.000	0.120	0.070	0.070	0.070
4	SwCyc_CycStartIsr	250.000	0.200	1.000	0.320	0.103	0.103	0.103
5	SwCyc_CycEndIsr	250.000	0.100	1.000	0.220	0.113	0.113	0.113
6	Rt1553_Isr	15.625	0.070	1.000	0.290	0.173	0.173	0.173
7	Bc1553_Isr	20.000	0.070	1.000	0.360	0.243	0.243	0.243
8	Spw_Isr	39.000	0.070	2.000	0.430	0.313	0.313	0.313
9	Obdh_Isr	250.000	0.070	2.000	0.500	0.383	0.383	0.383
10	RtSdb_P_1	15.625	0.150	15.625	4.330	0.533	0.533	0.533
11	RtSdb_P_2	125.000	0.400	15.625	4.870	0.933	0.933	0.933
12	RtSdb_P_3	250.000	0.170	15.625	5.110	1.103	1.103	1.103
14	**FdirEvents**	250.000	5.000	230.220	7.180	5.553	5.553	5.553
15	**NominalEvents_1**	250.000	0.720	230.220	7.900	6.273	6.273	6.273
16	**MainCycle**	250.000	0.400	230.220	8.370	6.273	6.273	6.273
17	HkSampler_P_2	125.000	0.500	62.500	11.960	5.380	7.350	8.153
18	HkSampler_P_1	250.000	6.000	62.500	18.460	11.615	13.653	14.153
19	Acb_P	250.000	6.000	50.000	24.680	6.473	6.473	6.473
20	IoCyc_P	250.000	3.000	50.000	27.820	9.473	9.473	9.473
21	**PrimaryF**	250.000	34.050	59.600	**65.47**	54.115	56.382	58.586
22	**RCSControlF**	250.000	4.070	239.600	76.040	53.994	56.943	58.095
23	Obt_P	1000.000	1.100	100.000	74.720	2.503	2.513	2.523
24	Hk_P	250.000	2.750	250.000	6.800	4.953	4.963	4.973
25	StsMon_P	250.000	3.300	125.000	85.050	17.863	27.935	28.086
26	TmGen_P	250.000	4.860	250.000	77.650	9.813	9.823	9.833
27	Sgm_P	250.000	4.020	250.000	18.680	14.796	14.880	14.973
28	TcRouter_P	250.000	0.500	250.000	19.310	11.896	11.906	14.442
29	Cmd_P	250.000	14.000	250.000	114.920	94.346	99.607	101.563
30	**NominalEvents_2**	250.000	1.780	230.220	102.760	65.177	69.612	72.235
31	**SecondaryF_1**	250.000	20.960	189.600	141.550	110.666	114.921	122.140
32	**SecondaryF_2**	250.000	39.690	230.220	204.050	154.556	162.177	165.103
33	Bkgnd_P	250.000	0.200	250.000	154.090	15.046	139.712	147.160

On a Linux server with Intel Xeon E5420 2.5GHz processor UPPAAL takes 2min 40s to verify that the system is schedulable, 6min 30s to find WCRTs with 0% BCET deviation. In case of 10% BCET deviation it took slightly over 6 days to establish schedulability and slightly over 7 days of 6 parallel runs to find all WCRTs. Table 3 shows the amount of verification resources UPPAAL requires to verify schedulability with different task execution time windows and model time limits. In this study we used compact data structure (CDS) to store the clock valuations in contrast to difference bound matrices (DBM) in previous study, which explains why the verification is slower, but the memory usage is limited and varies very little across model time limits.

In addition UPPAAL reported that the system is not schedulable when the task execution time window is larger than 14%. We found that the cycle limit granularity (used to define the progress measure) affects performance as well as the outcome: the larger cycles lead to error state being reachable because larger cycles result in the coarser stop-watch over-approximation. For example, binary search method revealed that with a 20% task execution window the error is reachable within the first 250ms period when the cycle is larger than 8017*ms* and otherwise it is not. However the error is reachable in the second 250ms period even if the cycle is as small as 2ms (verification took 24 hours).

Table 3: 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), memory in MB, time in seconds.

limit	0%			5%			10%			14%		
	states	mem	time	states	mem	time	states	mem	time, s	states	mem	time
1	1300	51.2	1.47	485077	83.0	903.1	1481162	124.1	4962.8	3348246	186.9	23986.5
2	2522	53.7	2.45	806914	96.8	1619.9	2414679	139.7	7755.2	5253778	198.7	33299.2
4	4981	54.5	4.62	1499700	97.2	2881.8	4421630	138.3	13720.0	9231399	274.6	51176.6
8	9928	54.7	8.48	2828776	97.8	5325.1	9093562	156.5	31122.3	18240030	364.6	102932.4
16	19805	55.3	16.11	5366015	112.0	9952.0	17798572	176.0	60124.5	35432003	520.4	158816.7
∞	196336	58.8	159.64	52728344	553.9	97507.4	181869652	1682.2	530604.9	error may be reachable		

4 Discussion

We have shown how UPPAAL can be applied for schedulability analysis of a system with a single CPU, fixed priorities preemptive scheduler, mixture of periodic tasks and tasks with dependencies, and mixed resource sharing protocols. Worst-case response-times (WCRT), blocking times and CPU utilisation are estimated by using model-checker according to detailed task models. Our modelling patterns use stopwatches in a simple and intuitive way. A break-through in verification scalability for large systems (more than 30 tasks) is achieved by employing the sweep-line method.

The task templates are demonstrated to be generic through many instantiations with arbitrary computation sequences and specialised for particular resource sharing. The framework is modular and extensible to accommodate a different scheduler and control flow can be expanded with additional instructions if some task algorithm is even more complicated. In addition, UPPAAL allows easy visualisation of the schedule in Gantt chart and the system behaviour can be examined in both symbolic and concrete simulators.

The case study results include a self-contained non-ambiguous model which formalises many informal assumptions described in [Palm(2006)] in human language. The verification results demonstrate that the timing estimates correlate with figures

from the response-time analysis [Palm(2006)]. The worst-case response-time of PrimaryF is indeed very close to its deadline, but overall, all estimates by UPPAAL are lower (more optimistic) and they all ($WCRT_{21}$ in particular) are below deadlines, whereas the classical response-time analysis found that PrimaryF may not finish before deadline and does not provide any more insight on how the deadline is violated or whether such behaviour is realizable.

By relaxing the lower bound of task execution time we showed that the system is probably not schedulable if BCET deviates from WCET by 15% or more. We found that it is better to start exploration with small task execution windows with large progress cycles first and limit the model time (effectively limiting the verification resources), then progress gradually with larger windows and then use smaller cycles to refine over-approximation. The large task execution windows (e.g. 20% with small progress cycles, or simple case of 50% with large cycles) can take days just to find the error and potentially much longer if there is no error.

We plan to conduct a similar study to allow sporadic tasks and apply statistical model-checking methods to investigate the probability of deadline violation as a cheaper means to detect errors.

So far we have not addressed margin analysis (as part of response-time analysis), but we see no principle obstacle to use the binary search method to find upper bounds for task execution times.

4.1 Related Work

Process algebraic approach has resulted in many methods for specification and schedulability analysis of real-time systems. For example [Ben-Abdallah et al(1998)Ben-Abdallah, Choi, Clarke, Ki provide an overview of this long tradition.

In [Waszniowski and Hanzálek(2008)] it is shown how a multitasking application running under a real-time operating system compliant with an OSEK/VDX standard can be modelled by timed automata. Use of this methodology is demonstrated on an automated gearbox case study and the worst-case response-times obtained from model-checking is compared with those provided by classical schedulability analysis showing that the model-checking approach provides less pessimistic results due to a more detailed model and exhaustive state-space exploration.

The Times tool [Amnell et al(2002)Amnell, Fersman, Mokrushin, Pettersson, and Yi] can be used to analyse single processor systems, however it supports only highest locker protocol (simplified priority ceiling protocol) [Fersman(2003)]. Approaches like [Bøgholm et al(2008)Bøgholm, Kragh-Hansen, Olsen, Thomsen, and Larsen] and [Brekling et al(2009)Brekling, Hansen, and Madsen] provides external transformation into UPPAAL [Behrmann et al(2004)Behrmann, David, and Larsen] timed-automata for schedulability analysis.

References

[Amnell et al(2002)Amnell, Fersman, Mokrushin, Pettersson, and Yi] Amnell T, Fersman E, Mokrushin L, Pettersson P, Yi W (2002) TIMES – a tool for modelling and implementation of embedded systems. In: TACAS '02: Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, Springer-Verlag, London, UK, pp 460–464

[Behrmann et al(2004)Behrmann, David, and Larsen] Behrmann G, David A, Larsen K (2004) A tutorial on Uppaal. Lecture Notes in Computer Science pp 200–236

[Ben-Abdallah et al(1998)Ben-Abdallah, Choi, Clarke, Kim, Lee, and Xie] Ben-Abdallah H, Choi JY, Clarke D, Kim YS, Lee I, Xie HL (1998) A process algebraic approach to the schedulability analysis of real-time systems. Real-Time Systems 15:189–219, URL http://dx.doi.org/10.1023/A:1008047130023, 10.1023/A:1008047130023

[Bøgholm et al(2008)Bøgholm, Kragh-Hansen, Olsen, Thomsen, and Larsen] Bøgholm T, Kragh-Hansen H, Olsen P, Thomsen B, Larsen KG (2008) Model-based schedulability analysis of safety critical hard real-time java programs. In: Bollella G, Locke CD (eds) JTRES, ACM, ACM International Conference Proceeding Series, vol 343, pp 106–114

[Brekling et al(2009)Brekling, Hansen, and Madsen] Brekling A, Hansen M, Madsen J (2009) Moves – a framework for modelling and verifying embedded systems. In: Microelectronics (ICM), 2009 International Conference on, pp 149–152, DOI 10.1109/ICM.2009.5418667

[Burns(1994)] Burns A (1994) Preemptive priority based scheduling: An appropriate engineering approach. In: Principles of Real-Time Systems, Prentice Hall, pp 225–248

[Fersman(2003)] Fersman E (2003) A generic approach to schedulability analysis of real-time systems. Acta Universitatis Upsaliensis

[Mikučionis et al(2010)Mikučionis, Larsen, Rasmussen, Nielsen, Skou, Palm, Pedersen, and Hougaard] Mikučionis M, Larsen KG, Rasmussen JI, Nielsen B, Skou A, Palm SU, Pedersen JS, Hougaard P (2010) Schedulability analysis using uppaal: Herschel-planck case study. In: Margaria T (ed) ISoLA 2010 – 4th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation, Springer, vol Lecture Notes in Computer Science

[Palm(2006)] Palm S (2006) Herschel-Planck ACC ASW: sizing, timing and schedulability analysis. Tech. rep., Terma A/S

[Terma A/S(Issue 9)] Terma A/S (Issue 9) Software timing and sizing budgets. Tech. rep., Terma A/S

[Waszniowski and Hanzálek(2008)] Waszniowski L, Hanzálek Z (2008) Formal verification of multitasking applications based on timed automata model. Real-Time Systems 38(1):39–65