Schedulability Analysis Using Uppaal: Herschel-Planck Case Study

Marius Mikučionis[1], Kim Guldstrand Larsen[1], Jacob Illum Rasmussen[1], Brian Nielsen[1], Arne Skou[1], Steen Ulrik Palm[2], Jan Storbank Pedersen[2], and Poul Hougaard[2]

[1] Aalborg University, 9220 Aalborg Øst, Denmark,
{ marius | kgl | illum | bnielsen | ask } @cs.aau.dk
[2] Terma A/S, 2730 Herlev, Denmark,
{ sup | jnp | poh } @terma.com

Abstract. We propose a modeling framework for performing schedulability analysis by using Uppaal real-time model-checker [2]. The framework is inspired by a case study where schedulability analysis of a satellite system is performed. The framework assumes a single CPU hardware where a fixed priority preemptive scheduler is used in a combination with two resource sharing protocols and in addition voluntary task suspension is considered. The contributions include the modeling framework, its application on an industrial case study and a comparison of results with classical response time analysis.

Keywords: 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 [5] provide generic frameworks which assume worst case scenario where worst case execution time and blocking times are estimated and then worst case response times are calculated and compared w.r.t. deadlines. Often, such conservative scenarios are never realized and thus negative results from such analysis are often too pessimistic. The idea of our method is to base the schedulability analysis on a system model with 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.

Related Work. During the last years, timed automata modelling and analysis of multitasking applications running under real-time operating systems has received substantial research effort. Here the goals are multiple: to obtain less pessimistic worst-case response time analysis compared with classical methods for single-processor systems or to relax the constraints be of period task arrival times of classical scheduling theory to task arrival patterns that can be described using timed automata.

In [13] 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 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.

Times tool [1] can be used to analyse single processor systems, however it supports only highest locker protocol (simplified priority ceiling protocol) [8]. An approach of [4] provides Java code transformation into Uppaal [2] timed-automata for schedulability analysis. Similarly, we use the model-checking framework provided by Uppaal, where the modelling language is extended with stop-watches and C-like code structures allows to express preemption, suspension and mixed resource sharing using two different protocols, in a more intuitive way without a need for more complex model transformations or workarounds.

A framework from [7] provides a generic set of tasks and resources that can be instantiated with concrete parameters (specific offsets, release times, deadlines, dependencies, periods) and processor resources together with their scheduling policies (i.e., preemption vs. non-preemption, earliest deadline first, fixed priority, first-in-first-out). The instantiated system can then be analysed for schedulability in a precise manner as all the concrete information is taken into account. This means that the framework will be able to verify schedulability of some systems that would otherwise be declared "non-schedulable" by other methods. Although the framework is general to cover multi-processor systems it does not tackle passive resource sharing protocols like priority ceiling or inheritance.

The remainder of the paper is organised as follows: in the next Section 2 we provide a brief overview of the Herschel-Planck satellite mission, its software setup and the response time analysis already carried out by Terma. Section 3 describes our model-based methodology for solving the schedulability problem, Section 4 presents the results of our method and compares it to traditional response time analysis. Finally, Section 5 discusses conclusions and future work.

2 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 architec-

ture. The architecture consists of a single processor, real-time operating system (RTEMS), basic software layer (BSW) and application software (ASW).

Terma A/S has performed an extended worst case response time analysis described in [5] by analysing [11] and [12] resulting in [10] (we provide the necessary details of those documents in this paper). 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 framework uses 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 elements of multi-processor systems since parts of activities are executed elsewhere and the classical worst case response analysis (applicable to single-processor systems) is pushed into extreme. One of the results of [10] 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.

The system is required to be schedulable in four instances: Herschel in nominal and event modes and Planck in nominal and event modes. The processor consumption should be less than 45% for Herschel and less than 50% for Planck.

In response time analysis, in order to prove schedulability it is enough to calculate worst case response times (WCRT) for every task and compare it with its deadline: if for every task the WCRT does not exceed the correspond deadline the system is schedulable.

Figure 1 shows the work-flow performed by Terma A/S: the deadline requirements are obtained from ASW and BSW documentation, worst case execution times (WCET) of BSW are obtained from BSW documentation [12] and ASW timings are obtained from time measurements. The tasks are carefully picked and timings aggregated (documented in [10]) and processed by the proprietary Terma tool SCHEDULE which performs worst case response time analysis as described in [5] and the outcome is processor utilisation and worst case response times (WCRT) for each task. The system is schedulable when for every task i $WCRT_i$ is less than its $Deadline_i$, i.e. the task always finishes before its deadline. We use the index i as a global task identifier.

We provide only the formulas used in response time analysis and refer to [5] for details on how this analysis works. The important property of the analysis is that it always takes conservative estimates of blocking times even though the actual blocking may not appear, thus resulting in too pessimistic response times. $WCRT_i$ is calculated by recursive formula for task computation windows $w(q)$ which may overlap with another task release (due to deadline extending past period), where q is the number of window starting with 0, $hp(i)$ is a set of tasks with higher priority than i. Then the longest window is taken as WCRT:

$$w_i^{n+1}(q) = Blocking_i + (q+1)\,WCET_i + \sum_{j \in hp(i)} \left\lceil \frac{w_i^n(q)}{Period_j} \right\rceil WCET_j$$

$$Response_i(q) = w_i^n(q) - qPeriod_i$$

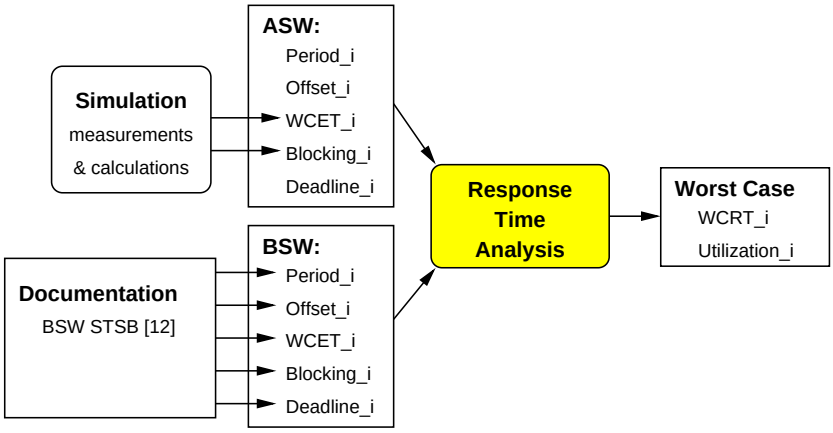


Fig. 1: Work-flow of schedulability analysis [10].

$$WCRT_i = \max_q Response_i(q)$$

$Blocking_i$ denotes the blocking time when task i is waiting for a shared resource being occupied by a lower priority task. Blocking times calculation is specific to the resource sharing protocol used. BSW tasks use priority inheritance protocol and thus their blocking times are calculated using the following equation:

$$Blocking_i = \sum_{r=1}^{R} usage(r,i)\, WCET_{CriticalSection}(r)$$

ASW tasks use priority ceiling protocol and therefore their blocking times are:

$$Blocking_i = \max_{r=1}^{R} usage(r,i)\, WCET_{CriticalSection}(r)$$

The matrix $usage(r,i)$ captures how resource r is used by the tasks: $usage(r,i) = 1$ if r is used by at least one task with a priority less than task i and at least one task with a priority greater than or equal to task i, otherwise $usage(r,i) = 0$.

Some BSW tasks are periodic and some sporadic, but we simplify the model by considering all BSW tasks as periodic. ASW tasks are started by periodic task `MainCycle`. In order to obtain more precise results, [10] splits the analysis into 0-20ms and 20-250ms windows, distinguishes two operating modes and analyse six cases in total separately. However, one case shows up as not schedulable anyway and application of our framework improves this result by showing that all tasks finish within their deadlines.

Resource Usage by Tasks. Some tasks require 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 a sum of worst case execution times by duration of analysed time window.

3 Model-Based Schedulability Methodology

This section explains the principles and concepts used throughout the modelling framework and then describes the modelling templates in detail.

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 find worst case blocking and response times, processor utilisation and to check whether all the deadlines are met. In our modelling framework clocks and stopwatches control task release patterns, track task execution progress, check response time against deadline bound and thus all the computations are performed by model-checker according to the model, in contrast to carefully customised specific formula.

The framework consists of the following process models: fixed priority preemptive CPU scheduler, a number of task models and one process for ensuring global invariants. We provide several templates for task models: for periodic tasks and for tasks with dependencies, all of which are parameterised with concrete program control flow and may be customised to a particular resource sharing protocol. Our approach takes the same task descriptions as [10] and produces results which are more optimistic and provides the proof that all the tasks will actually finish before the deadline.

We use stopwatches to track task progress and stop the task progress during preemption. In Uppaal, the stopwatch support is implemented through a concept of derivative over clock, where the derivative can be either 1 (valuation progresses with a rate of 1 as regular clocks) or 0 (valuation is not allowed to progress – the clock is stopped). Syntactically stop-watch expressions appear in invariant expressions in a form of $x' == c$, where x is declared of type `clock` and c is an integer expression which evaluates to either 0 or 1. The reachability analysis of stopwatch automata is implemented as an over-approximation in Uppaal, but the approximation still suffices for safety properties like checking if a deadline can ever be violated.
The following outlines the main modelling ingredients:

- One template for the CPU scheduler.
- One template for "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 MainCycle ASW task, which is released periodically, starts other ASW tasks and locks resources based on priority ceiling protocol.
- One template for all other ASW tasks, which is 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 *flow* array of structures.

- Each task (either ASW or BSW) uses the following clocks and data variables:
  - Task and its clocks are parameterised by identifier *id*.
  - Execution time is modelled by a stopwatch $job[id]$ which is reset when the task is started and stopped by a global invariant when the task is not being run on the processor. A worst case execution time (WCET) guard ensures that task cannot finish before WCET elapses. To ensure progress, the clock $job[id]$ is constrained by an invariant of $WCET$ so that the task releases the processor as soon as it has finished computing.
  - A local clock x controls when the task is released and is reset upon task is released. 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.
  - 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). In addition $WCRT[id]$ is reset when the task is finished in order to allow model checker to apply active clock reduction to speed up analysis as the value of this clock is no longer used. 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.

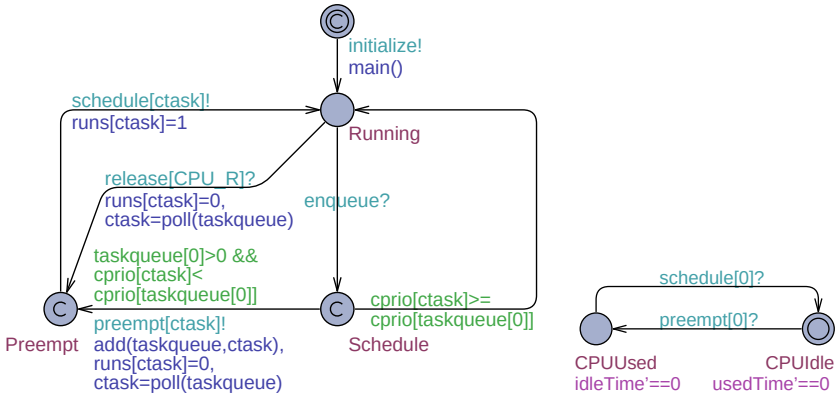
Further we explain the most important model templates, while the complete model is available for download at `http://www.cs.aau.dk/~marius/Terma/` .

3.1 Processor Scheduler

Figure 2a shows the model of CPU scheduler. In the beginning 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 `enqueue` channel, thus moving Scheduler to location `Schedule`. From location `Schedule`, the Scheduler compares the priority of a current task `cprio[ctask]` with 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`.

The processor is released 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 sending is performed always in non-blocking way as receivers may ignore broadcast synchronisations).

The `taskqueue` always contains at least one ready task: IdleTask. Figure 2b shows how IdleTask reacts to Scheduler events and computes the CPU usage time with stopwatch `usedTime`, the total CPU load is then calculated as $\frac{\text{usedTime}}{\text{globalTime}}$.



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

Fig. 2: Models for CPU scheduler and the simplest task.

3.2 Tasks Templates

Figure 3 shows the parameters which describe each periodic task: period duration showing how often the task is started, offset showing 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.

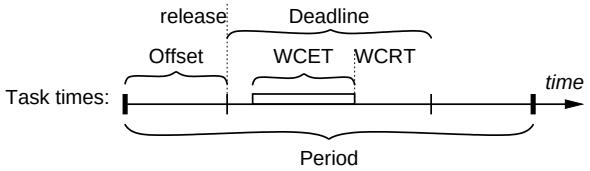


Fig. 3: Periodic task execution parameters.

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 `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 scheduler it moves to location `GotCPU` and starts processing commands from the *flow* array. There are four types of commands:

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 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 is shown in Listing 1.2.
3. SUSPEND releases the processor for 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.

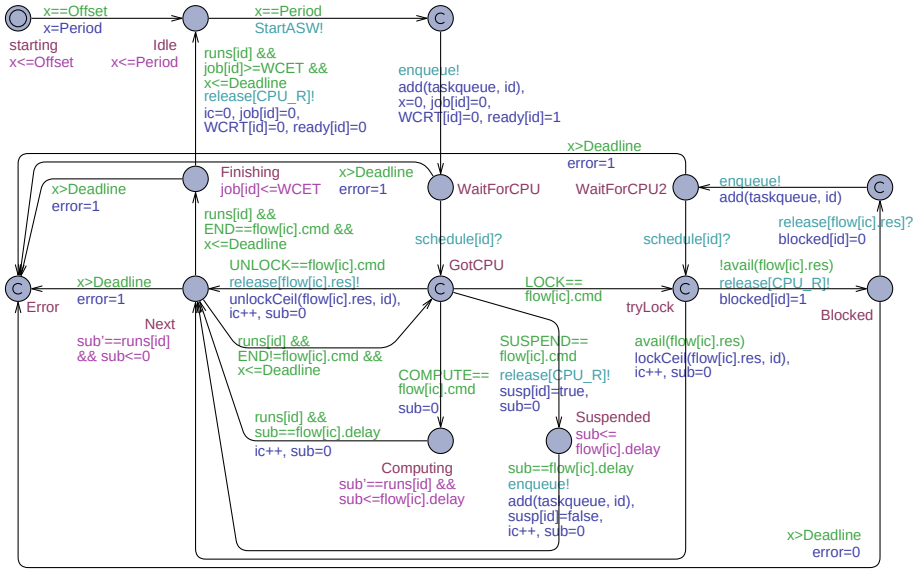


Fig. 4: MainCycle task: periodically starts ASW functions.

4. COMPUTE makes the task stay in location `Computing` for at least the specified duration of pure running time, i.e. the clock *sub* is stopped whenever the task is preempted and $runs[id]$ is set to 0. Once the required amount of CPU time is consumed, the process moves on to location `Next`.

From location `Next`, the process is forced by $runs[id]$ 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 `Finishing` location the process consumed CPU for the remaining time so that 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 very simple: it computes for its WCET while keeping a lock on Sgm_R. A more sophisticated example of flow is in Listing 1.1. We do not know the exact times the resources are locked and the points in time are chosen arbitrarily, thus it may not necessarily lead to worst-case blocking timed for higher priority tasks.

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 ceiling protocol (*lockCeil* and *unlockCeil*) it uses 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.

Other ASW tasks are started by MainCycle, thus instead of broadcast shout synchronisation on `StartASW` channel they have receive synchronisation on `StartASW`. Also, they share the same clock x with MainCycle, because response time is measured from the same $20ms$ offset (as in [10], so that the results are comparable).

Figure 1 shows the description of PrimaryF from [10] as an example that we used to create *flow* structure. This particular description consists of six activities.

Table 1: The description of PrimaryF task from [10] inputs.

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

Each activity is described by two numbers (CPU time / BSW service time, BSW service time is included in CPU time, thus is not used in our model), 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.

From this description we use only LCS and LC, where we assume that LC-LCS is the CPU busy time while the resource is locked. Listing 1.1 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. Listing 1.2 shows functions for priority inheritance and priority ceiling protocols, which use *owner* and *cprio* to track current resource owner and task priority.

Listing 1.1: Operation flow for PrimaryF task.

```
1  const ASWFlow_t PF_f = { // Primary Functions:
2      { LOCK,   Icb_R, 0 },          // 0) ----- Data processing
3      { COMPUTE, CPU_R, 1600-1200 }, // 1) computing with Icb_R
4      { SUSPEND, CPU_R, 1200 },      // 2)  suspended with Icb_R
5      { UNLOCK, Icb_R, 0 },          // 3)
6      { COMPUTE, CPU_R, 20577-(1600-1200) }, // 4) computing without Icb_R
7      { COMPUTE, CPU_R, 3440 },      // 5) ----- Guidance
8      { LOCK,   Sgm_R, 0 },          // 6) ----- Attitude determination
9      { COMPUTE, CPU_R, 1218-121 },  // 7) computing with Sgm_R
10     { SUSPEND, CPU_R, 121 },       // 8)  suspended with Sgm_R
11     { UNLOCK, Sgm_R, 0 },          // 9)
12     { COMPUTE, CPU_R, 3751-(1218-121) }, //10) computing without Sgm_R
```

```
    { COMPUTE, CPU_R, 42 },     //11) ----- Perform extra checks
    { LOCK,   PmReq_R,0 },      //12) ----- SCM controller
    { COMPUTE, CPU_R, 3300-1650 }, //13) computing with PmReq_R
    { SUSPEND, CPU_R, 1650 },   //14) suspended with PmReq_R
    { UNLOCK, PmReq_R, 0 },     //15)
    { COMPUTE, CPU_R, 3479-(3300-1650) }, //16) computing without PmReq_R
    { COMPUTE, CPU_R, 2752 },   //17) ----- Command RWL
    FIN, FIN, FIN, FIN, FIN, FIN, FIN, FIN, FIN, FIN, FIN // fill the array
};
```

Listing 1.2: Resource locking based on two different protocols.

```
/** Check if the resource is available: */
bool avail(resid_t res) { return (owner[res]==0); }
/** Lock the resource based on priority ceiling protocol: */
void lockCeil(resid_t res, taskid_t task) {
    owner[res] = task; // mark resource occupied by task
    cprio[task] = ceiling[res]; // assume the priority of resource
}
/** Unlock the resource based on priority ceiling protocol: */
void unlockCeil(resid_t res, taskid_t task){
    owner[res] = 0; // mark resource as released
    cprio[task] = def_prio(task); // return to default priority
}
/** Boost the priority of resource owner based on priority inheritance protocol: */
void boostPrio(resid_t res, taskid_t task) {
    if (cprio[owner[res]] <= def_prio(task)) {
        cprio[owner[res]] = def_prio(task)+1;
        sort(taskqueue);
    }
}
/** Lock the resource based on priority inheritance protocol: */
void lockInh(resid_t res, taskid_t task) {
    owner[res] = task; // mark resource occupied by task
}
/** Unlock the resource based on priority inheritance protocol: */
void unlockInh(resid_t res, taskid_t task) {
    owner[res] = 0; // mark resource as released
    cprio[task] = def_prio(task); // return to default priority
}
```

3.3 System Model Instantiation

Listing 1.3 shows how tasks are instantiated. Listing 1.4 shows system declaration using priorities which help enforce the specified priorities in verification. The resulting model is deterministic, thus the expected state space shape is narrow (single sequence of steps) but potentially very deep.

Initial experiments showed that in fact the state space is so deep that UPPAAL exhausts the memory in a few minutes by storing most of the state space just to check for loops and ensure the verification termination. To address this issue a sweep-line method [6] is used. The basic idea behind sweep-line method is to store only those passed states which have the greatest progress measure and purge the rest, thus effectively releasing and reusing most of the memory.

Listing 1.3: 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);
SwCyc_CycStartIsr= BSW(4,   0,250000, WCET_f, 20,     1000);
```

```
6  SwCyc_CycEndIsr=  BSW(5,200000,250000, WCET_f, 100,  1000);
7  Rt1553_Isr      =     BSW(6,   0,  15625, WCET_f,  70,     1000);
8  Bc1553_Isr      =     BSW(7,   0,  20000, WCET_f,  70,     1000);
9  Spw_Isr         =     BSW(8,   0,  39000, WCET_f,  70,     2000);
10 Obdh_Isr        =     BSW(9,   0,250000, WCET_f,  70,     2000);
11 RtSdb_P_1       =    BSW(10,   0,  15625, WCET_f, 150,   15625);
12 RtSdb_P_2       =    BSW(11,   0,125000, WCET_f, 400,   15625);
13 RtSdb_P_3       =    BSW(12,   0,250000, WCET_f, 170,   15625);
14 // #13 is reserved for ASW resource priority ceiling
15 FdirEvents      =ASWspor(14,20000,250000, WCET_f, 5000, 230220);
16 NominalEvents_1=ASWspor(15,20000,250000, WCET_f, 720, 230220);
17 mainCycle =  MainCycle(16,20000,250000, 400,  230220, ASWclock);
18 HkSampler_P_2 =   BSW(17,62500,125000, WCET_f, 500, 62500);
19 HkSampler_P_1 =   BSW(18,62500,250000, WCET_f, 6000, 62500);
20 Acb_P          =   BSW(19,200000,250000,WCET_f, 6000, 50540);
21 IoCyc_P        =   BSW(20,200000,250000,WCET_f, 3000, 50540);
22                  //ASW: id, start, finish ,   flow,   WCET, Deadline
23 primaryF =          ASW(21,StartASW,Done, PF_f, 34050, 59600, ASWclock);
24 rCSControlF=        ASW(22,StartASW,Done, RCS_f, 4070, 239600, ASWclock);
25 Obt_P          =   BSW(23,   0,1000000,Obt_f,   1100, 100000);
26 Hk_P           =   BSW(24,   0,250000, WCET_f, 2750, 250000);
27 StsMon_P       =   BSW(25,62500,250000,StsMon_f,3300, 125000);
28 TmGen_P        =   BSW(26,   0,250000, WCET_f, 4860, 250000);
29 Sgm_P          =   BSW(27,   0,250000, Sgm_f,   4020, 250000);
30 TcRouter_P     =   BSW(28,   0,250000, WCET_f, 500, 250000);
31 Cmd_P          =   BSW(29,   0,250000, Cmd_f, 14000, 250000);
32 NominalEvents_2=  BSW(30,20000,250000, WCET_f, 1780, 230220);
33 secondF_1      =      ASW(31,StartASW, Done, SF1_f, 20960, 189600, ASWclock);
34 secondF_2      =      ASW(32,StartASW, Done, SF2_f, 39690, 230220, ASWclock);
35 Bkgnd_P        =   BSW(33,   0,250000, WCET_f, 200, 250000);
```

Periodic models—like the ones for schedulability—do not have inherent progress measure, so we propose to create an artificial one based on how many cycles the system has executed so far, and then reset it to zero once it reaches a pre-specified limit. In fact, such drops in progress measure are tolerated by generalised sweep-line method [9], which is also implemented in UPPAAL. The progress measure is based on the variable *cycle* defined on line 8 in Listing 1.4, incremented by guarded loops in Global process (Fig. 5) after each 250ms and is reset together with all the global clocks to zero once the CYCLELIMIT is reached. The process Global also takes care of global invariants on $job[i]$ and $WCRT[i]$ stopwatches of each task i.

Listing 1.4: System declaration using UPPAAL priorities.

```
1 system Scheduler, Bkgnd_P < secondF_2 < secondF_1 < NominalEvents_2 < Cmd_P <
2     TcRouter_P < Sgm_P < TmGen_P < StsMon_P < Hk_P < Obt_P < rCSControlF <
3     primaryF < IoCyc_P < Acb_P < HkSampler_P_1 < HkSampler_P_2 <
4     mainCycle < NominalEvents_1 < FdirEvents < RtSdb_P_3 < RtSdb_P_2 < RtSdb_P_1
          < Obdh_Isr <
5     Spw_Isr < Bc1553_Isr < Rt1553_Isr < SwCyc_CycEndIsr < SwCyc_CycStartIsr <
6     Hk_SamplerIsr < AswSync_SyncPulselsr <
7     RTEMS_RTC, IdleTask, Global;
8 progress { cycle; }
```

Further, we explore some values of CYCLELIMIT in order to minimise the verification resources. We postulate that a good heuristics is to explore at least to a hyper-period (least common multiple of periods) of all the periodic processes before reseting the cycle counter. There are the following different periods in the system: 125000, 15625, 20000, 39000, 250000 and 1000000μs, therefore a potential hyper-period is 39000000μs, or 156 cycles of 250ms each, but it can be much larger due to non-trivial resource sharing and task interaction.

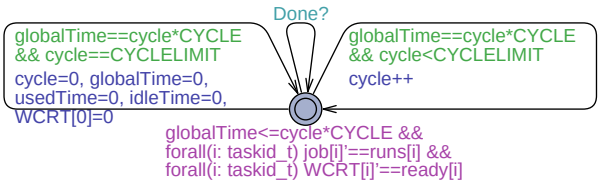


Fig. 5: Global process enforce invariants on stopwatches and cyclic progress.

3.4 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, which is useful for computing several bounds at once. However, in such queries, Uppaal treats the specified clocks as active, therefore the exploration can be significantly slower when the clock list is large. Therefore, we create a separate model to estimate blocking times instead WCRT by purging expressions with $WCRT[id]$ and adding $B[id]$ reset statements on edges from `tryLock` to `Blocked` in order to save half of expensive stopwatches.

4 Results

The results of our model-based framework consist of three parts: visualisation of a schedule in Gantt chart, worst case response times estimates and CPU utilisation estimation with verification benchmarking based on cycle limit.

Visualisation. A Gantt chart can be used to visualise a trace of the system, thus providing a rich picture for inspection. For example, in the generated chart, it can be seen that Cmd_P is blocked more than 5 times during the first cycle, while blocking times for PrimaryF and StsMon_P are significantly long only starting from the second cycle. Listing 1.5 shows a chart declaration accepted by Uppaal TIGA [3] which assigns colours for each line (T for task lines, R for resource lines) based on the state (ready, running, blocked or suspended; locked and used, locked and preempted or locked and suspended respectively).

Listing 1.5: Specification for Gantt chart.

```
1 gantt {
2   T(i:taskid_t):
3     (ready[i] && !runs[i]) -> 1,// green: ready
4     (ready[i] && runs[i]) -> 2, // blue: running
5     (blocked[i]) -> 0,          // red:   blocked
```

```
6       susp[i] -> 9;                  // cyan: suspended
7     R(i: resid_t):
8       (owner[i]>0 && runs[owner[i]]) -> 2, // blue: locked and actively used
9       (owner[i]>0 && !runs[owner[i]] && !susp[owner[i]]) -> 1, // green: locked, preempted
10      (owner[i]>0 && susp[owner[i]]) -> 9; // cyan: locked and suspended
11  }
```

Verification and CPU Load Estimates. UPPAAL takes about 2min (112s) to verify that the system is schedulable and about 3 times as much to find WCRT on a Linux laptop PC with Intel Core 2 Duo 2.2GHz processor.

In [10] CPU utilisation for 20-250ms window is estimated as 62.4%, and our estimate for entire worst case cycle is 63.65% which is slightly larger, possibly due to the fact that it also includes the consumption during 0-20ms window.

Table 2 shows UPPAAL verification resources used for estimating WCRT and CPU utilisation for various cycle limits. The instances where cycle limit is a divisor or a multiple of a hyper-period (156) are in bold. Notice that for such cycle limits the verification resources are orders of magnitude lower, and there is nearly perfect linear correlation between cycle limit and resource usage in both sub-sequences when evaluated separately (both coefficients are ≥ 0.993).

Table 2: Verification resources and CPU utilisation estimates.

cycle	Uppaal resources			Herschel CPU utilization				
limit	CPU, s	Mem, KB	States, #	Idle, μs	Used, μs	Global, μs	Sum, μs	Used, %
1	465.2	60288	173456	91225	160015	250000	251240	0.640060
2	470.1	59536	174234	182380	318790	500000	501170	0.637580
3	461.0	58656	175228	273535	477705	750000	751240	0.636940
4	474.5	58792	176266	363590	636480	1000000	1000070	0.636480
6	474.6	58796	178432	545900	955270	1500000	1501170	0.636847
8	912.3	58856	352365	727110	1272960	2000000	2000070	0.636480
13	507.7	58796	186091	1181855	2069385	3250000	3251240	0.636734
16	1759.0	58728	704551	1454220	2545850	4000000	4000070	0.636463
26	541.9	58112	200364	2363640	4137530	6500000	6501170	0.636543
32	3484.0	75520	1408943	2908370	5091700	8000000	8000070	0.636463
39	583.5	74568	214657	3545425	6205745	9750000	9751170	0.636487
64	7030.0	91776	2817704	5816740	10183330	16000000	16000070	0.636458
78	652.2	74768	257582	7089680	12411420	19500000	19501100	0.636483
128	14149.4	141448	5635227	11633480	20366590	32000000	32000070	0.636456
156	789.4	91204	343402	14178260	24821740	39000000	39000000	0.636455
256	23219.4	224440	11270279	23266890	40733180	64000000	64000070	0.636456
312	1824.6	124892	686788	28356520	49643480	78000000	78000000	0.636455
512	49202.2	390428	22540388	46533780	81466290	128000000	128000070	0.636455
624	3734.7	207728	1373560	56713040	99286960	156000000	156000000	0.636455

Herschel CPU utilisation estimate does not improve much, therefore we conclude that individual cycles are very similar. The sum of idle and used times is slightly larger than global supremum meaning that some cycles are only slightly more stressed than others.

Worst Case Response Times. Table 3 shows the response timed from UPPAAL analysis in comparison to response time analysis by Terma. For most of BSW tasks (1-12,17-18) resource patterns are not available and thus UPPAAL could not determine their blocking times. Blocking times by Terma also include the suspension times related to locking of resources. We note that in all cases the WCRT estimates provided by UPPAAL are smaller (hence less pessimistic) than those originally obtained [10]. In particular, we note that the task PrimaryF (task 21) is found to be schedulable using model-checking in contrast to the original negative result obtained by Terma.

Table 3: Specification, blocking and worst case response times of individual tasks.

		Specification			Blocking times			WCRT		
ID	Task	Period	WCET	Deadline	Terma	UPPAAL	Diff	Terma	UPPAAL	Diff
1	RTEMS_RTC	10.000	0.013	1.000	0.035	0	0.035	0.050	0.013	0.037
2	AswSync_SyncPulseIsr	250.000	0.070	1.000	0.035	0	0.035	0.120	0.083	0.037
3	Hk_SamplerIsr	125.000	0.070	1.000	0.035	0	0.035	0.120	0.070	0.050
4	SwCyc_CycStartIsr	250.000	0.200	1.000	0.035	0	0.035	0.320	0.103	0.217
5	SwCyc_CycEndIsr	250.000	0.100	1.000	0.035	0	0.035	0.220	0.113	0.107
6	Rt1553_Isr	15.625	0.070	1.000	0.035	0	0.035	0.290	0.173	0.117
7	Bc1553_Isr	20.000	0.070	1.000	0.035	0	0.035	0.360	0.243	0.117
8	Spw_Isr	39.000	0.070	2.000	0.035	0	0.035	0.430	0.313	0.117
9	Obdh_Isr	250.000	0.070	2.000	0.035	0	0.035	0.500	0.383	0.117
10	RtSdb_P_1	15.625	0.150	15.625	3.650	0	3.650	4.330	0.533	3.797
11	RtSdb_P_2	125.000	0.400	15.625	3.650	0	3.650	4.870	0.933	3.937
12	RtSdb_P_3	250.000	0.170	15.625	3.650	0	3.650	5.110	1.103	4.007
14	**FdirEvents**	250.000	5.000	230.220	0.720	0	0.720	7.180	5.153	2.027
15	**NominalEvents_1**	250.000	0.720	230.220	0.720	0	0.720	7.900	5.873	2.027
16	**MainCycle**	250.000	0.400	230.220	0.720	0	0.720	8.370	6.273	2.097
17	HkSampler_P_2	125.000	0.500	62.500	3.650	0	3.650	11.960	5.380	6.580
18	HkSampler_P_1	250.000	6.000	62.500	3.650	0	3.650	18.460	11.615	6.845
19	Acb_P	250.000	6.000	50.000	3.650	0	3.650	24.680	6.473	18.207
20	IoCyc_P	250.000	3.000	50.000	3.650	0	3.650	27.820	9.473	18.347
21	**PrimaryF**	250.000	34.050	**59.600**	5.770	0.966	4.804	**65.470**	**54.115**	11.355
22	**RCSControlF**	250.000	4.070	239.600	12.120	0	12.120	76.040	53.994	22.046
23	Obt_P	1000.000	1.100	100.000	9.630	0	9.630	74.720	2.503	72.217
24	Hk_P	250.000	2.750	250.000	1.035	0	1.035	6.800	4.953	1.847
25	StsMon_P	250.000	3.300	125.000	16.070	0.822	15.248	85.050	17.863	67.187
26	TmGen_P	250.000	4.860	250.000	4.260	0	4.260	77.650	9.813	67.837
27	Sgm_P	250.000	4.020	250.000	1.040	0	1.040	18.680	14.796	3.884
28	TcRouter_P	250.000	0.500	250.000	1.035	0	1.035	19.310	11.896	7.414
29	Cmd_P	250.000	14.000	250.000	26.110	1.262	24.848	114.920	94.346	20.574
30	**NominalEvents_2**	250.000	1.780	230.220	12.480	0	12.480	102.760	65.177	37.583
31	**SecondaryF_1**	250.000	20.960	189.600	27.650	0	27.650	141.550	110.666	30.884
32	**SecondaryF_2**	250.000	39.690	230.220	48.450	0	48.450	204.050	154.556	49.494
33	Bkgnd_P	250.000	0.200	250.000	0.000	0	0.000	154.090	15.046	139.044

5 Discussion

We have shown how the UPPAAL model-checker can be applied for schedulability analysis of a system with 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 model-checker according to the system model structure. Modelling patterns use stopwatches in a simple and intuitive way. A breakthrough in verification scalability for large systems (more than 30 tasks) is achieved by employing sweep-line method. Even better control over verification resources can be achieved by carefully designing progress measure.

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 toolkit 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 assumptions described in [10] in human language. The verification results demonstrate that the timing estimates correlate with figures from the response time analysis [10]. The worst case response time of PrimaryF is indeed very close to deadline, but overall all estimates by UPPAAL are lower (more optimistic) and they all ($WCRT_{21}$ in particular) are below deadlines, whereas the 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.

References

1. Amnell, T., Fersman, E., Mokrushin, L., Pettersson, P., Yi, W.: 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. pp. 460–464. Springer-Verlag, London, UK (2002)
2. Behrmann, G., David, A., Larsen, K.: A tutorial on Uppaal. Lecture Notes in Computer Science pp. 200–236 (2004)
3. Behrmann, G., Cougnard, A., David, A., Fleury, E., Larsen, K.G., Lime, D.: UPPAAL-TIGA: Time for playing games! In: Proceedings of the 19th International Conference on Computer Aided Verification. pp. 121–125. No. 4590 in LNCS, Springer (2007)
4. Bøgholm, T., Kragh-Hansen, H., Olsen, P., Thomsen, B., Larsen, K.G.: Model-based schedulability analysis of safety critical hard real-time java programs. In: Bollella, G., Locke, C.D. (eds.) JTRES. ACM International Conference Proceeding Series, vol. 343, pp. 106–114. ACM (2008)
5. Burns, A.: Preemptive priority based scheduling: An appropriate engineering approach. In: Principles of Real-Time Systems. pp. 225–248. Prentice Hall (1994)
6. Christensen, S., Kristensen, L., Mailund, T.: A Sweep-Line method for state space exploration. In: Tools and Algorithms for the Construction and Analysis of Systems, pp. 450–464 (2001), http://dx.doi.org/10.1007/3-540-45319-9_31
7. David, A., Illum, J., Larsen, K.G., Skou, A.: Model-Based Design for Embedded Systems, chap. Model-Based Framework for Schedulability Analysis Using UPPAAL 4.1, pp. 93–119. CRC Press (2010)
8. Fersman, E.: A generic approach to schedulability analysis of real-time systems. Acta Universitatis Upsaliensis (2003)
9. Kristensen, L., Mailund, T.: A generalised Sweep-Line method for safety properties. In: FME 2002:Formal Methods—Getting IT Right, pp. 215–229 (2002), http://dx.doi.org/10.1007/3-540-45614-7_31
10. Palm, S.: Herschel-Planck ACC ASW: sizing, timing and schedulability analysis. Tech. rep., Terma A/S (2006)
11. Terma A/S: Herschel-Planck ACMS ACC ASW requirements specification. Tech. rep., Terma A/S (Issue 4/0)
12. Terma A/S: Software timing and sizing budgets. Tech. rep., Terma A/S (Issue 9)
13. Waszniowski, L., Hanzálek, Z.: Formal verification of multitasking applications based on timed automata model. Real-Time Systems 38(1), 39–65 (2008)