An Exact Schedulability Test for Non-Preemptive Self-Suspending Real-Time Tasks — Extended Version —

Beyazit Yalcinkaya[1,2], Mitra Nasri[1,3], Björn B. Brandenburg[1],
[1] *Max Planck Institute for Software Systems (MPI-SWS), Germany* [2] *Middle East Technical University, Turkey*
[3] *Delft University of Technology, Netherlands*

***Abstract*—Exact schedulability analysis of limited-preemptive (or non-preemptive) real-time workloads with variable execution costs and release jitter is a notoriously difficult challenge due to the scheduling anomalies inherent in non-preemptive execution. Furthermore, the presence of self-suspending tasks is well-understood to add tremendous complications to an already difficult problem. By mapping the schedulability problem to the reachability problem in timed automata (TA), this paper provides the first exact schedulability test for this challenging model. Specifically, using TA extensions available in UPPAAL, this paper presents an exact schedulability test for sets of periodic and sporadic self-suspending tasks with fixed preemption points that are scheduled upon a multiprocessor under a global fixed-priority scheduling policy. To the best of our knowledge, this is the first exact schedulability test for non- and limited-preemptive self-suspending tasks (for both uniprocessor and multiprocessor systems), and thus also the first exact schedulability test for the special case of global non-preemptive fixed-priority scheduling (for either periodic or sporadic tasks). Additionally, the paper highlights some subtle pitfalls and limitations in existing TA-based schedulability tests for non-preemptive workloads.**

I. Introduction

The current state of the art in worst-case execution time analysis (WCET) favors *non-preemptive execution* due to its inherent predictability advantages [1]. To counteract latency spikes that may arise from long-running jobs, it is common practice to insert fixed *preemption points* to break jobs into a sequence of *segments*, where each such segment is executed non-preemptively, but the job overall is *limited-preemptive* (i.e., the job can be preempted, but only at well-known points in its control-flow graph, which aids WCET analysis). Furthermore, when jobs engage in synchronous I/O (e.g., network communication, storage devices), synchronize via semaphores, or offload computation to accelerators such as GPUs, DSPs, or other co-processors, segments may be separated by *self-suspensions* (i.e., after one segment completes, the next segment may not immediately be available for execution) [2].

A sound schedulability analysis of this practically relevant class of workloads, on either uni- or multiprocessors, is however presently beyond the reach of the state of the art, due to a number of challenges. For one, the analysis of limited-preemptive (or non-preemptive) real-time workloads is already on uniprocessors a notoriously difficult problem due to the scheduling anomalies that arise from non-preemptive execution. Additionally, the complex effects of self-suspensions cause further timing anomalies that, if not carefully handled, may result in *unsound* schedulability tests (as explained in detail in Chen et al.'s recent survey [2]). In fact, as we show in this paper, non-preemptive self-suspending tasks (either periodic or sporadic) suffer from anomalies that render it *unsafe* to treat suspensions as execution segments (i.e., to simply over-approximate processor demand). This is in contrast to the preemptive case, in which such a *suspension-oblivious* approach is commonly used and has been shown to be safe [2,3].

This paper. We provide the first exact schedulability test for this difficult setting, that is, a test that decides whether a set of segmented, self-suspending tasks scheduled non-preemptively upon either uni- or multiprocessors will meet all deadlines. By extension, our test also applies to limited-preemptive tasks (i.e., non-suspending tasks with preemption points). This is also the first exact test for the special case of global non-preemptive scheduling (without suspensions or preemption points).

Our test models the system as a synchronized network of *timed automata* (TA) [4] and maps the schedulability problem to a reachability query. In extensive experiments that evaluate the scalability of the solution with regard to the number of cores, tasks, segments, magnitude of release jitter, and system utilization, it is shown to scale up to 60 periodic tasks scheduled on 2 cores, 30 tasks on 4 cores, and 15 tasks on 8 cores.

Besides scaling to small- and even moderately-sized workloads, the proposed test also provides the first exact baseline against which future, more efficient, but inexact schedulability tests can be compared and thus lays an important foundation for future research into self-suspensions and limited-preemptive models. Additionally, the paper highlights some subtle pitfalls and limitations in existing TA-based schedulability tests for non-preemptive workloads.

Prior work. To the best of our knowledge, no exact schedulability test for non-preemptive (or limited-preemptive) self-suspending tasks has been proposed to date (for either uniprocessor or multiprocessor platforms).

Several exact schedulability tests have been introduced for *preemptively* scheduled uni- and multiprocessor platforms (i.e., for global fixed-priority scheduling) [5]–[8] and for non-preemptive scheduling upon uniprocessor platforms for both sporadic [9,10] and periodic tasks [11]. None of these tests supports self-suspending tasks. Further, even for the special case of non-preemptive global scheduling without self-suspensions or preemption points, no exact test has been proposed to date.

The conference version of this extended tech report appears in the *Proceedings of the 2019 Conference on Design, Automation and Test in Europe (DATE 2019)*, pages 1222–1227, March 2019, IEEE.

The TA formalism has been previously leveraged in a number of schedulability tests for preemptive tasks scheduled by a global scheduling policy upon a multiprocessor platform [12]–[18]. These tests, however, are not designed for self-suspending tasks, cannot handle blocking times caused by non-preemptive execution, and some use *stopwatches* (e.g., [14,17,18]) that render the TA reachability problem *undecidable* [19]. As we discuss in Sec. III, the use of stopwatches significantly limits the practical applicability of the tests. There are further a few TA-based schedulability tests for non-preemptive tasks [14,20,21]. These tests, however, are limited to uniprocessors and do not support self-suspensions or preemption points.

Finally, prior work has yielded several sufficient schedulability tests for global non-preemptive scheduling that target either sporadic [22]–[25] or periodic tasks [26], but these tests do not support self-suspending or limited-preemptive tasks.

II. System Model and Background

We consider a multiprocessor system with m identical processors and a set of n independent, self-suspending tasks $\tau = \{\tau_1, \tau_2, \dots, \tau_n\}$. Each task τ_i is represented by a tuple $(T_i, D_i, O_i, P_i, k_i, \mathcal{S}_i)$, where T_i is the period (or equivalently, the minimum inter-arrival time), $D_i \leq T_i$ is the relative deadline, O_i is the initial offset, P_i is the priority, k_i is the number of segments, and $\mathcal{S}_i$ is the vector of segments of the task. Smaller values of P_i indicate higher priorities. We assume global fixed-priority scheduling and that $P_1 < \ldots < P_n$.

The j^{th} segment of a task τ_i is denoted by a tuple $S_{i,j} = ([s_{i,j}^{min}, s_{i,j}^{max}], [C_{i,j}^{min}, C_{i,j}^{max}])$, where $s_{i,j}^{min}$ and $s_{i,j}^{max}$ are the *best-case suspension* time (BCST) and *worst-case suspension time* (WCST), and $C_{i,j}^{min}$ and $C_{i,j}^{max}$ are the *best-case execution time* (BCET) and the *worst-case execution time* (WCET) of the segment, respectively. Our model implicitly supports *release jitter*, i.e., the first suspension prior to the first execution segment represents the release jitter of the task.[1] Both the BCST and WCST can be zero for any or all of the segments, which turns such a segment boundary into a fixed preemption point (i.e., for a limited-preemptive task, $s_{i,j}^{min} = s_{i,j}^{max} = 0$ for $1 < j \leq k_i$). Once a segment starts execution it is not preempted until it finishes. If $k_i = 1$ for every $\tau_i \in \tau$, then the problem reduces to non-preemptive global scheduling.

We assume $C_i + X_i \leq D_i$, where $C_i = \sum_{j=1}^{k_i} C_{i,j}^{max}$ is the total execution time and $X_i = \sum_{j=1}^{k_i} s_{i,j}^{max}$ is the total suspension time of task τ_i. The system utilization is given by $U = \sum_{i=1}^{n} u_i$, where $u_i = C_i/T_i$ is the utilization of task τ_i. The *execution time ratio* of τ_i is given by $\beta_i = C_i/(C_i + X_i)$.

Timed automata. A *timed automaton* is a finite-state machine extended with a finite set of real-valued clocks that progress monotonically at the same rate and measure the time spent after their latest resets [4].[2] UPPAAL[3] extends the formal definition of timed automata with integer variables, structured data types, C-like programming constructs, specialized locations, and synchronization channels for modeling, simulating, and verifying real-time systems by defining them as *networks of timed automata*. In UPPAAL, the *state* of a system, i.e., a network of timed automata, is defined by the present locations of all timed automata, the values of all clocks, and the values of any discrete variables. Hence, the number of locations, clocks, and discrete variables has a significant effect on the size of the state space as well as the verification time.

Our schedulability test uses three special features of UPPAAL: *committed locations*, *urgent channels*, and *broadcast channels*. Whenever a state includes a committed location, the next transition *must* be one of the outgoing transitions of (one of) the committed location(s). Whenever a transition with an *urgent channel* is enabled, it *must* be taken without any time delay. Finally, a *broadcast channel* is used for synchronizing more than two timed automata: whenever a transition with a broadcast channel is taken, all other enabled transitions that use the same channel must be taken, too. When one location has multiple enabled transitions that use the same broadcast channel, UPPAAL selects one of them non-deterministically.

III. Motivation and Pitfalls

In this section, we discuss a few potential pitfalls and challenges related to the design of TA-based schedulability tests and the analysis of non-preemptive self-suspending tasks.

Stopwatch limitations. Since UPPAAL introduced *stopwatches* in version 4.1, several TA-based schedulability tests have used stopwatches for the analysis of preemptive and non-preemptive tasks [14,17,18]. However, in our experiments (Sec. V), we found that stopwatch-based tests are unable to provide any concrete schedulability answer even for very simple task sets. That is, due to the underlying undecidability result [19], instead of concluding that a task set is "schedulable" or "not schedulable," the evaluated stopwatch-based tests yield "may not be schedulable" as the analysis result for almost all task sets, unless the task set has only two tasks and the total sum of WCETs is less than the shortest period. To the best of our knowledge, this is a new observation; prior work on stopwatch-based tests [14,17,18] did not report on empirical evaluations of the tests in the context of non-preemptive workloads.

Impossible event ordering. The ordering of the start times of jobs plays a crucial role in the analysis of non-preemptive tasks since it determines the amount of blocking incurred by high-priority tasks. To be *exact*, a schedulability test must hence discount any impossible orderings. However, some previous studies (e.g., [14,21]) have modeled tasks as *independent* TAs without synchronizing transitions related to their respective job releases. Consequently, tasks with the same release time could be scheduled in any order, including in orders contrary to their assigned priorities, which is actually impossible when a deterministic scheduling algorithm such as fixed-priority scheduling is used. As a concrete example, consider the following periodic task set, which is schedulable on a uniprocessor under non-preemptive fixed-priority scheduling,

[1] For example, if a task τ_i has a minimum and maximum release jitter equal to $[r_i^{min}, r_i^{max}]$, then $s_{i,1}^{min} = r_i^{min}$ and $s_{i,1}^{max} = r_i^{max}$.

[2] This paper assumes the reader to be familiar with the timed automata formalism; an overview and tutorial may be found in [19,27].

[3] http://www.uppaal.com

which is however deemed *not schedulable* by David et al.'s analysis [21] due to the inclusion of impossible event orderings.

Counterexample 1. *When applying David et al.'s test [21] to the periodic task set* $\tau_1 = (3, 3, 0, 1, 1, \{([0, 0], [1, 1])\})$ *and* $\tau_2 = (6, 6, 0, 2, 1, \{([0, 0], [3, 3])\})$*, UPPAAL reports the following scenario to result in a deadline miss: at time 0,* τ_2 *enters its* `ready` *location before* τ_1.[4] *Next, UPPAAL activates the scheduler, which dispatches the only job in the ready queue, i.e.,* τ_2 *(at time 0). This is allowed as the scheduler and task automata are independent of each other [21], i.e., UPPAAL is allowed to non-deterministically pick* any *enabled transition, including the scheduler transition. Finally,* τ_1 *also moves to its* `ready` *location (still at time 0). However, the processor has already been allocated, so* τ_2 *blocks* τ_1 *for three time units. Consequently,* τ_1 *is reported to miss its deadline at time 3. This, however, is actually impossible, since a fixed-priority scheduler will always select the higher-priority task* τ_1 *if both* τ_1 *and* τ_2 *release a job at exactly the same time (note the absence of release jitter). The problem can be avoided if, at time 0, both tasks are forced to synchronously move to their* `ready` *locations before the scheduler can be called.*

UPPAAL's scheduling framework [14], which uses stopwatches, reports "may not be schedulable" for this example.

Suspension-oblivious analysis is unsound. The following example shows that, given a self-suspending task set (either periodic or sporadic), if suspension segments are analyzed as if they were execution segments (i.e., using a suspension-oblivious approach [2]), then the resulting task set may be deemed schedulable while the original task set is in fact *not schedulable*. Hence, naïvely accounting for suspensions as execution time is not safe for limited-preemptive self-suspending tasks.

Counterexample 2. *Consider three tasks* $\tau_1 = (20, 6, 1, 1, 2, \{([0, 0], [1, 1]), ([1, 1], [1, 1])\})$, $\tau_2 = (20, 20, 2, 2, 1, \{([0, 0], [3, 3])\})$*, and* $\tau_3 = (20, 20, 0, 3, 1, \{([0, 0], [3, 3])\})$*. Fig. 1(a) shows that the task set is schedulable if suspension time is treated as execution time since* τ_1 *can only suffer from one low-priority blocking, while in reality it can suffer from two such blockings and hence is not schedulable as shown in Fig. 1(b). This counterexample holds also for sporadic tasks.*

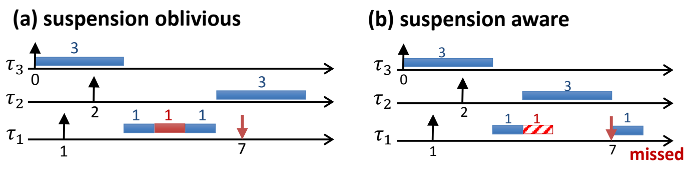


Fig. 1. **(a)** suspension-oblivious and **(b)** suspension-aware schedules.

The main contribution of this paper is a new TA-based exact schedulability test that avoids all of the aforementioned pitfalls.

IV. Proposed Schedulability Test

Our solution includes three TA templates for tasks (TASK), the scheduler (SCHED), and an event synchronizer (SYNC), as shown in Figs. 2(a)–(c).[5] Fig. 2(d) provides details of the functions used in the timed automata. A system with n tasks requires n TASK instances, and one SCHED and one SYNC instance each. The SYNC automaton uses a broadcast channel `synch`, defined in Fig. 2(d), whose receivers are the TASK automata. The SCHED automaton uses n high-priority urgent channels called `run` to send signals to the TASK automata. The system is schedulable if, in any reachable state, no TASK automaton resides in its **Missed** location.

TASK. This automaton models a periodic, segmented, self-suspending task with initial offset and release jitter. Each TASK automaton uses two clocks: t keeps track of the arrival time (and deadline) of the task, while x keeps track of the execution and suspension time of each segment. The initial offset of the task is enacted in the **Start** location, in which the automaton is forced to stay for `offset` time units. Next, the automaton enters the **Suspended** location, which realizes both release jitter and suspensions. A task stays **Suspended** non-deterministically for x units of time, where `s_min()`$\leq x \leq$ `s_max()`. The functions `s_min()` and `s_max()` return the minimum and maximum suspension duration of the current segment of the task (indicated by `seg_idx`), respectively.

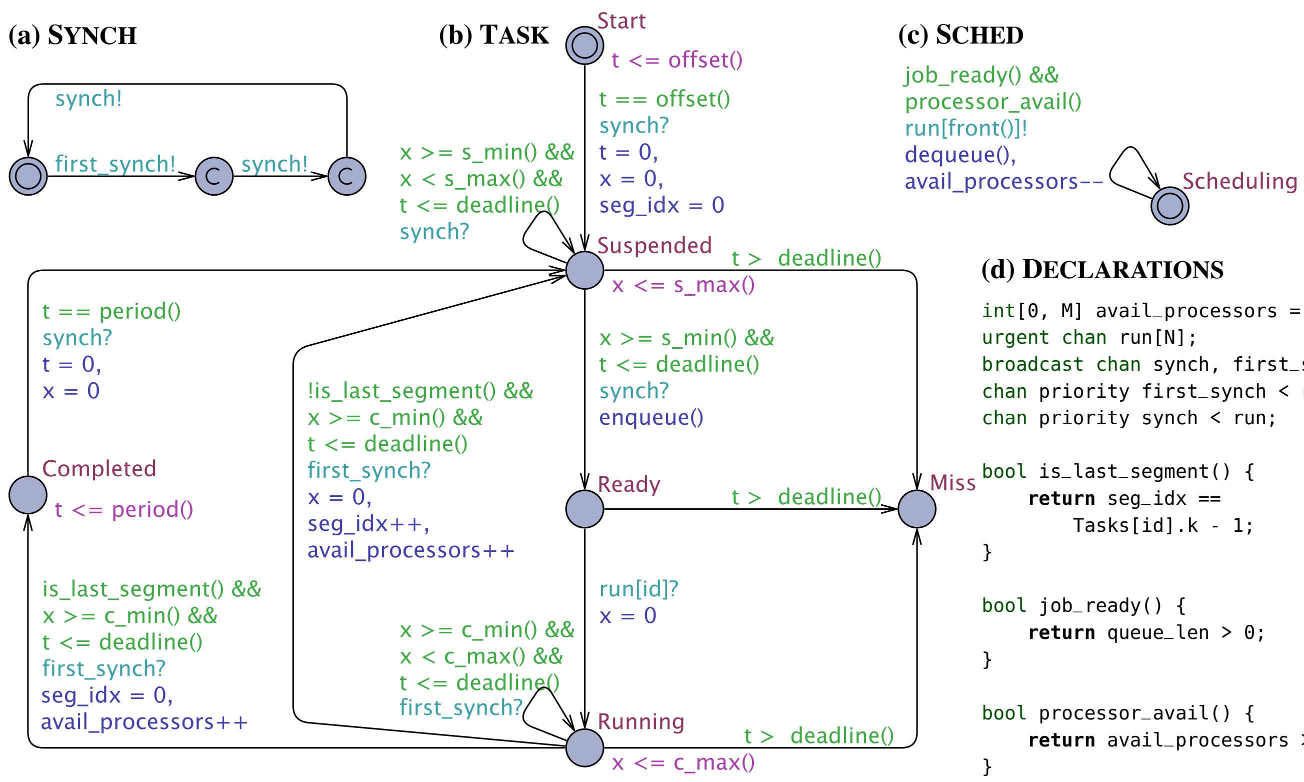
When the suspension time (or initial jitter delay) has passed, the task enters the **Ready** location, where it waits until it receives a `run` signal from the scheduler to start its execution and enter the **Running** location. During this transition, it decreases the number of available processors by one as it starts executing on one of the processors. The task remains in the **Running** location until some time within the minimum and maximum execution time of the current segment, denoted by `c_min()`and `c_max()`, respectively. When the task completes the execution of its current segment, it either enters the **Completed** location (if `seg_idx` indicates the last segment of the task), or goes back to the **Suspended** location to model the next suspension. Whenever the task leaves the **Running** location, it increments the current segment index as well as the number of available processors. If a task is not completed before its deadline, i.e., it is still in the **Suspended**, **Ready**, or **Running** location when t exceeds `deadline`, then it enters the **Missed** location. A task enters the **Completed** location when it completes the execution of its last segment, and stays there until the next arrival time (i.e., the end of its period, as is indicated by `t == period()`).

The TASK automaton can be easily modified to realize a sporadic task by **(i)** adding a self-loop to the **Completed** location with guard "`synch?`" and removing the location invariant "`t <= period()`", and **(ii)** replacing "`t == period`" with "`t >= period`" on the transition from **Completed** to **Suspended**. The resulting automaton is shown in Fig. 4 and discussed in the Appendix.

SCHED. This automaton is similar to the scheduler model used

[4]Please refer to the original paper for an illustration.

[5]Fig. 2 differs in a minor way from the conference version. Specifically, the SYNC automaton first sends a `first_synch` signal followed by two `synch` signals (in the conference version, it simply sends three `synch` signals in a row). This modification is required to address a corner case in the analysis of limited-preemptive (non-suspending) tasks.



(d) DECLARATIONS

```
int[0, M] avail_processors = M;
urgent chan run[N];
broadcast chan synch, first_synch;
chan priority first_synch < run;
chan priority synch < run;

bool is_last_segment() {
    return seg_idx ==
        Tasks[id].k - 1;
}

bool job_ready() {
    return queue_len > 0;
}

bool processor_avail() {
    return avail_processors > 0;
}
```

Fig. 2. The **(a)** SYNC, **(b)** TASK, and **(c)** SCHED automata with **(d)** key declarations. Locations marked with a 'C' are committed locations. Initial locations are indicated by an inner circle. The declarations of `front()`, `enqueue()`, and `dequeue()` have been omitted as they implement a standard priority queue.

by David et al. [14,21] and realizes a global work-conserving fixed-priority scheduler using a priority queue that stores a sorted list of tasks whose current segment is ready for execution. The first task in the list has the highest priority among all ready tasks. This automaton schedules a task as soon as there is a ready task in the queue and at least one processor available, in which case it sends a `run` signal to the highest-priority task.

SYNC. The purpose of SYNC is to synchronize the TASK automata such that all tasks that release a segment at a given time t' enter their **Ready** location at the same time *before* the scheduler is triggered. This construct serves to avoid the impossible event ordering problem mentioned in Sec. III.

Specifically, the SYNC automaton uses a broadcast channel to synchronize the TASK automata. In UPPAAL, any receiver of a broadcast channel is forced to activate the transition that has become enabled upon receiving the broadcast signal. As a result, any two tasks that are in their respective **Suspended** locations and release their next segments at time t', upon receiving the `synch` signal, must move to the **Ready** location at the same time. This ensures that, whenever SCHED is activated, all ready tasks have indeed entered their **Ready** location.

Adding a broadcast signal to transition guards forces them to react upon receiving a signal on the broadcast channel. Hence, in order to allow TASK automata to stay in their **Suspended** or **Running** locations for some time, we have added self-loop transitions to these locations. Upon receiving a `synch` signal (or, respectively, a `first_synch` signal), two of the outgoing transitions of these locations will be enabled, and since UPPAAL non-deterministically chooses one of them, the tasks can stay in the **Suspended** or **Running** locations for the duration of their current segment.

Another important detail of the SYNC automaton is that it always sends a sequence of three synchronization signals (one `first_synch`followed by two `synch` signals) due to the two *committed* locations that follow the **Init** location in the SYNC automaton. This design covers corner cases in which a task finishes its execution *exactly* at the end of its current period and hence must be able to reach the **Ready** location from the **Running** location in the same instant (i.e., at the period boundary). This requires a forced multi-step transition from **Running** to **Ready** via **Completed** and **Suspended** without any passage of time, which in turn requires first a `first_synch` signal, as indicated in the transitions out of the **Running** location, and two subsequent `synch` signals to move from **Completed** to **Suspended** and then to **Ready**.

The reason for distinguishing between the `first_synch` and `synch` signals, and for using the `first_synch` signal to leave the **Running** location, is to align the SYNC automaton's sequence of three committed transitions with the TASK automaton's multi-step transition. Specifically, this construction ensures that whenever a task leaves the **Running** location, it will receive a sufficient number of follow-up `synch` signals (i.e., two) to let the task automaton reach the **Ready** location if needed, which is required to ensure an exact analysis of limited-preemptive tasks that do not suspend in between segments.

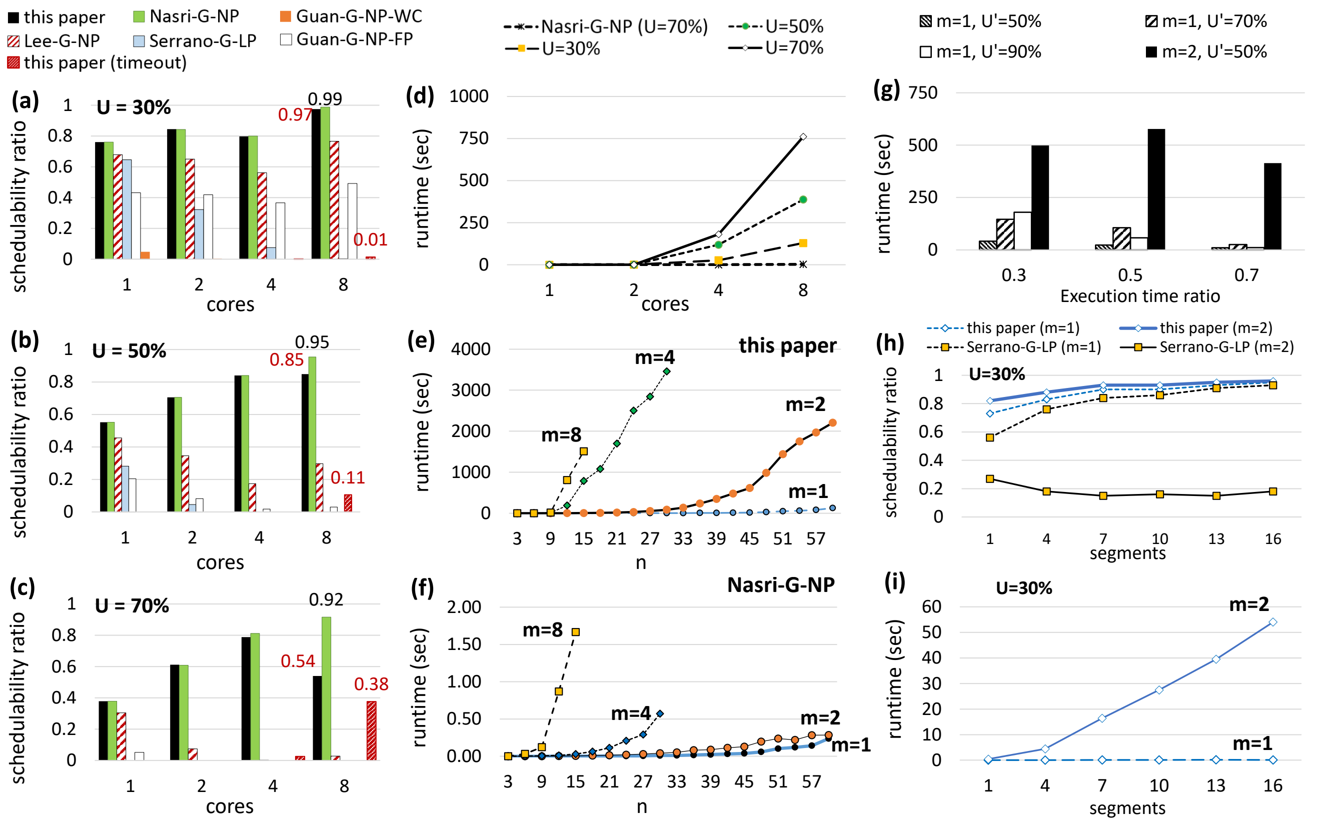


Fig. 3. Experiments with various parameters. **(a, b, c, d)** The schedulability ratio and runtime in Exp1; **(e, f)** the runtime in Exp2 for both the TA-based test and Nasri-G-NP; **(g)** the runtime in Exp3; and **(h, i)** the schedulability ratio and runtime in Exp4.

V. Experimental Results

This section answers the following questions: **(i)** How is the proposed test's runtime affected by various system parameters, such as the number of cores, tasks, etc., and **(ii)** to what extent does it improve schedulability gain w.r.t. the state of the art?

We compared the proposed test against existing schedulability tests for *global non-preemptive* (G-NP) and *global limited-preemptive* (G-LP) scheduling in terms of *average runtime* and *schedulability ratio* (i.e., the fraction of task sets deemed schedulable for a particular set of parameters). We considered the following baselines: Guan et al.'s test [24] for any work-conserving G-NP policy (Guan-G-NP-WC), three tests for fixed-priority G-NP scheduling by Guan et al. [24], Lee et al. [25], and Nasri et al. [26] (denoted Guan-G-NP-FP, Lee-G-NP, and Nasri-G-NP, respectively), and Serrano et al.'s test for limited-preemptive scheduling [28] (Serrano-G-LP).

It should be noted that, with the exception of Nasri-G-NP [26], all baseline tests were designed specifically for sporadic tasks. We hence expect them to exhibit some degree of inherent pessimism when applied to periodic workloads (i.e., some feasible periodic workloads become infeasible if tasks exhibit sporadic arrivals). We focus on periodic tasks in the following and report on experiments involving sporadic tasks in the Appendix.

We generated periodic task sets following the guidelines of the Autosar benchmark introduced by Kramer et al. [29]. Specifically, for a given number of tasks n, we sampled the (non-uniform) distribution of common periods ($\{1, 2, 5, 10, 20, 50, 100, 200, 1000\}$ms) reported by Kramer et al. [29] to randomly draw a realistic period for each task. Due to UPPAAL's restricted support for integer variables and parameters, we multiplied each period by ten to obtain integer execution times while ensuring that periods can still be stored in 16-bit integer variables. Next, given a target total utilization U, we used the *RandFixSum* algorithm [30] to generate a random utilization value u_i for each task, from which we obtained C_i as $u_i \cdot T_i$. We took k_i and β_i as input parameters for each experiment and derived $X_i = (C_i - \beta_i C_i)/\beta_i$. Based on C_i and X_i, we used the RandFixSum algorithm [30] to distribute C_i among the $C_{i,j}^{max}$ values and X_i among the $s_{i,j}^{max}$ values, respectively. Finally, for each segment, we assigned $C_{i,j}^{min} = 0.1 \cdot C_{i,j}^{max}$ and $s_{i,j}^{min} = 0.1 \cdot s_{i,j}^{max}$, respectively. Tasks were assumed to have implicit deadlines and to not have release jitter, and were assigned rate-monotonic priorities. Moreover, we discarded task sets that did not pass even a trivial simulation-based necessary test that considers only one execution scenario in which each task executes for C_i time units and suspends for X_i time units (i.e., where all job parameters are maximal).

We performed four experiments by varying **(Exp1)** the number of cores and total utilization U (for $U \in \{0.3m, 0.5m, 0.7m\}$, $n = 10$, one segment per task, and no suspensions); **(Exp2)** the number of tasks n (for $m \in \{1, 2, 4, 8\}$, U=$0.3 \cdot m$, one segment per task, and no suspensions); **(Exp3)** the maximum execution time ratio β (for $m \in \{1, 2\}$,

$U' \in \{0.5m, 0.7m, 0.9m\}$, where $U'{=}(C_i + X_i)/T_i$, $n = 5$, and up to 5 segments separated by suspensions); and **(Exp4)** the maximum number of execution segments (for $m = 2$, $n = 10$, $U{=}0.3m$, and no suspensions). For each parameter combination, we generated and analyzed at least 100 task sets. For instance, Exp2 evaluated 27,000 task sets for all parameter values, with 1,300 jobs per hyperperiod on average (with a minimum of 3 and a maximum of 10,728 jobs).

For each task set, we ran an instance of UPPAAL's `verifyta` tool (a mainly single-threaded computation) on an Intel Xeon E7-8857 v2 processor clocked at 3 GHz. A task set was deemed unschedulable either if it was rejected by the proposed test or if the proposed test could not reach a conclusion within one hour. We applied the same timeout to all other tests as well. Moreover, to avoid wasting compute resources, we stopped an experiment (e.g., in Exp2, stopped increasing n) when the analysis of more than 50% of the task sets timed out. Fig. 3 reports the results of our experiments.[6] When reporting average runtimes, we consider only schedulable task sets (i.e., we exclude unschedulable task sets) to avoid biasing the results since tests can typically reject unschedulable task sets very quickly without exploring the whole state space.

Schedulability. Figs. 3(a,b,c) and (h) confirm that our exact test identifies many more schedulable task sets than prior G-NP and G-LP tests (when applied to periodic tasks). Our results show that, for the tested workloads, Nasri-G-NP is actually as precise as the exact test, even though it is technically only a sufficient test if $m > 1$ [26]. In fact, thanks to its efficient, interval-based state-space exploration approach, which in contrast to UPPAAL is tailored to the schedulability problem, Nasri-G-NP was able to identify more schedulable task sets than the exact test within the one-hour time limit, as for instance can be seen in Figs. 3(b,c). In future work, it would be promising to extend Nasri-G-NP to the full task model considered herein (i.e., limited-preemptive, self-suspending workloads).

Moreover, our test reveals considerable pessimism in the state-of-the-art schedulability test for limited-preemptive tasks (Serrano-G-LP [28]), which can be observed in Fig. 3(h). While our test confirms that schedulability improves with an increase in the number of cores, the test of Serrano et al. [28] actually indicates the opposite: the number of task sets deemed schedulable by Serrano-G-LP drops considerably when multiple cores are available. This reveals further room for improvement in state-of-the-art global limited-preemptive schedulability analyses and demonstrates the benefit of having an exact baseline to compare sufficient tests against.

Runtime. As expected and as shown in Figs. 3(d,e,g,i), the exact test's runtime increases rapidly when U, m, the number of execution segments, or the length of suspension segments increase since each of these changes increases the number of execution scenarios that must be explored. The decrease in the runtime of the analysis in Fig. 3(g) is due to the fact that with the increase in the *execution time ratio*, the suspension segments become shorter and hence the number of interleavings that must be considered by the analysis is reduced. This makes the analysis faster. Finally, comparing Figs. 3(d) and (e), we observe that the proposed test's runtime is more affected by joint increases in m and n than by joint increases in m and U.

Discussion. While the proposed test suffers from the typical scalability limitations expected from TA-based analyses, we nonetheless found it to be able to scale to periodic workloads of nontrivial size: up to 60 tasks on 2 cores, 30 tasks on 4 cores, and 15 tasks on 8 cores. The test performs worse when the number of possible execution scenarios increases, e.g., when there are more segments or longer suspension times (see Fig. 3(g)). Similarly, its runtime grows very quickly in the presence of sporadic tasks (as reported in the Appendix).

Nonetheless, our exact test provides a useful baseline for evaluating the accuracy of other, only sufficient tests. For instance, in the special case of G-NP scheduling (i.e., no preemption points, no suspensions), Nasri-G-NP is as accurate as the exact test while scaling much better: Figs. 3(e) and (f) show the Nasri-G-NP test to be three orders of magnitude faster.

VI. Conclusion

We have proposed the first exact schedulability test for limited-preemptive (and non-preemptive) self-suspending real-time tasks scheduled upon a uniprocessor or multiprocessor platform (under a global fixed-priority scheduling policy). We mapped the schedulability problem to the TA reachability problem, discussed some subtle pitfalls and limitations of prior TA-based schedulability tests, and proposed task, scheduler, and event synchronizer automata to realize an exact test. In addition to scaling to nontrivial workload sizes before succumbing to the state-space explosion problem, our work also provides the first exact baseline against which sufficient schedulability tests can be compared and thus enables future research into self-suspensions and limited-preemptive models.

References

[1] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström, "The worst-case execution-time problem - overview of methods and survey of tools," *ACM TECS*, vol. 7, no. 3, pp. 36:1–36:53, 2008.

[2] J.-J. Chen, G. Nelissen, W.-H. Huang, M. Yang, B. Brandenburg, K. Bletsas, C. Liu, P. Richard, F. Ridouard, N. Audsley, R. Rajkumar, D. de Niz, and G. Brüggen, "Many suspensions, many problems: a review of self-suspending tasks in real-time systems," *Real-Time Systems*, 2018.

[3] F. Cerqueira, G. Nelissen, and B. Brandenburg, "On strong and weak sustainability, with an application to self-suspending real-time tasks," in *Proc. of ECRTS*, vol. 106, 2018, pp. 26:1–26:21.

[4] R. Alur and D. L. Dill, "A theory of timed automata," *Theoretical computer science*, vol. 126, no. 2, pp. 183–235, 1994.

[5] T. P. Baker and M. Cirinei, "Brute-force determination of multiprocessor schedulability for sets of sporadic hard-deadline tasks," in *Proc. of OPODIS*, 2007, pp. 62–75.

[6] V. Bonifaci and A. Marchetti-Spaccamela, "Feasibility analysis of sporadic real-time multiprocessor task systems," *Algorithmica*, vol. 63, no. 4, pp. 763–780, 2012.

[7] A. Burmyakov, E. Bini, and E. Tovar, "An exact schedulability test for global FP using state space pruning," in *Proc. of RTNS*, 2015.

[6]Fig. 3 has been updated to reflect the aforementioned tweak of the Sync automaton and hence differs in minor ways from the conference version. However, none of the general conclusions have been altered.

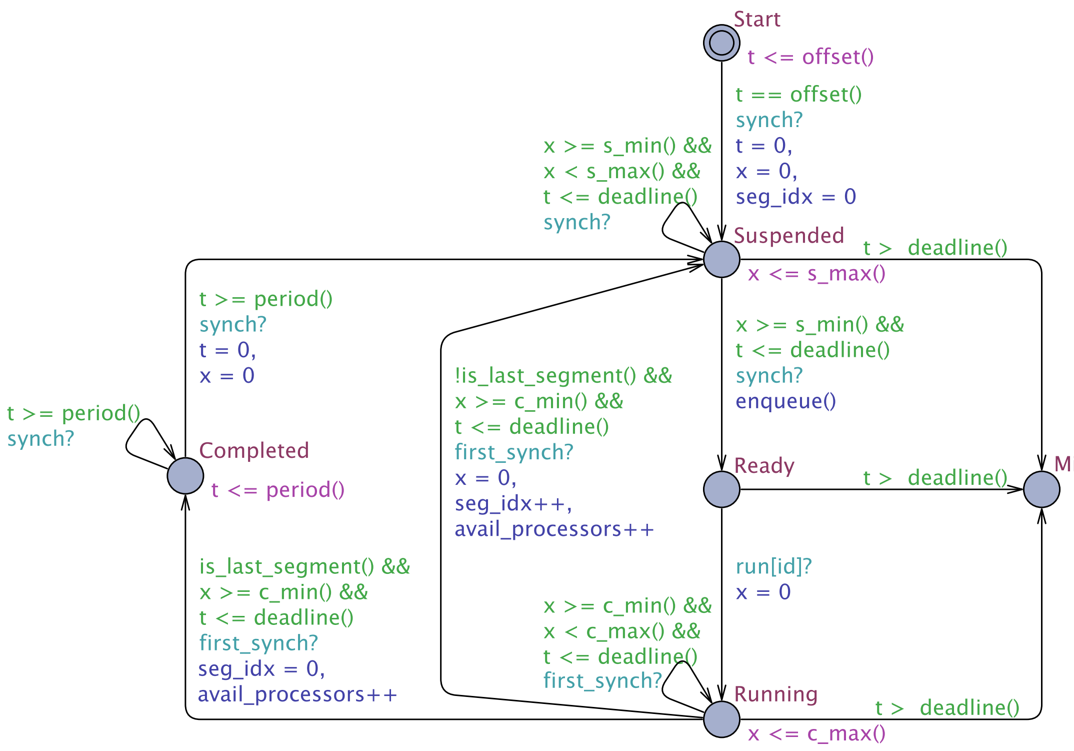


Fig. 4. The TASK automaton for sporadic tasks.

[8] Y. Sun and G. Lipari, "A pre-order relation for exact schedulability test of sporadic tasks on multiprocessor Global Fixed-Priority scheduling," *Real-Time Syst.*, vol. 52, no. 3, pp. 323–355, 2016.

[9] K. Jeffay, D. F. Stanat, and C. U. Martel, "On non-preemptive scheduling of period and sporadic tasks," in *Proc. of RTSS*, 1991.

[10] R. I. Davis, A. Burns, R. J. Bril, and J. J. Lukkien, "Controller Area Network (CAN) schedulability analysis: Refuted, revisited and revised," *Real-Time Syst.*, vol. 35, no. 3, pp. 239–272, 2007.

[11] M. Nasri and B. Brandenburg, "An exact and sustainable analysis of non-preemptive scheduling," in *Proc. of RTSS*, 2017, pp. 1–12.

[12] N. Guan, Z. Gu, Q. Deng, S. Gao, and G. Yu, "Exact schedulability analysis for static-priority global multiprocessor scheduling using model-checking," in *Proc. of SEUS*, 2007, pp. 263–272.

[13] N. Guan, Z. Gu, M. Lv, Q. Deng, and G. Yu, "Schedulability analysis of global fixed-priority or edf multiprocessor scheduling with symbolic model-checking," in *Proc. of ISORC*, 2008, pp. 556–560.

[14] A. David, J. Illum, K. Larsen, and A. Skou, "Model-based framework for schedulability analysis using UPPAAL 4.1," in *Model-based design for embedded systems*, 2009, pp. 117–144.

[15] W. Sheng, Y. Gao, L. Xi, and X. Zhou, "Schedulability analysis for multicore global scheduling with model checking," in *Proc. of MTV*, 2010, pp. 21–26.

[16] M. Cordovilla, F. Boniol, E. Noulard, and C. Pagetti, "Multiprocessor schedulability analyser," in *Proc. of SAC*, 2011, pp. 735–741.

[17] F. Cicirelli, A. Furfaro, L. Nigro, and F. Pupo, "Development of a schedulability analysis framework based on PTPN and UPPAAL with stopwatches," in *Proc. of ISORC*, 2012, pp. 57–64.

[18] Z. Gu, Z. Wang, H. Chen, and H. Cai, "A model-checking approach to schedulability analysis of global multiprocessor scheduling with fixed offsets," *Int. Jour. of Embedded Syst.*, vol. 6, no. 2-3, pp. 176–187, 2014.

[19] R. Alur, C. Courcoubetis, N. Halbwachs, T. A. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine, "The algorithmic analysis of hybrid systems," *Theoretical computer science*, vol. 138, no. 1, pp. 3–34, 1995.

[20] L. Waszniowski and Z. Hanzálek, "Formal verification of multitasking applications based on timed automata model," *Real-Time Systems*, vol. 38, no. 1, pp. 39–65, 2008.

[21] A. David, K. Larsen, and A. Skou, "An introduction to schedulability analysis using timed automata," *Industrial Handbook, Deliverable no.: D5. 12, Quantitative System Properties in Model-Driven Design Project*, pp. 107–118, 2011. [Online]. Available: http://people.cs.aau.dk/~kgl/China2011/Slides/QuasimodoHandbook.pdf

[22] S. Baruah, "The Non-preemptive Scheduling of Periodic Tasks upon Multiprocessors," *Real-Time Systems*, vol. 32, no. 1, pp. 9–20, 2006.

[23] N. Guan, W. Yi, Z. Gu, Q. Deng, and G. Yu, "New schedulability test conditions for non-preemptive scheduling on multiprocessor platforms," in *Proc. of RTSS*, 2008, pp. 137–146.

[24] N. Guan, W. Yi, Q. Deng, Z. Gu, and G. Yu, "Schedulability analysis for non-preemptive fixed-priority multiprocessor scheduling," *Jour. of Syst. Arch.*, vol. 57, no. 5, pp. 536–546, 2011.

[25] J. Lee, "Improved schedulability analysis using carry-in limitation for non-preemptive fixed-priority multiprocessor scheduling," *IEEE TC*, vol. 66, no. 10, pp. 1816–1823, 2017.

[26] M. Nasri, G. Nelissen, and B. Brandenburg, "A response-time analysis for non-preemptive job sets under global scheduling," in *Proc. of ECRTS*, 2018.

[27] G. Behrmann, A. David, and K. Larsen, "A tutorial on UPPAAL," *Formal Methods for the Design of Real-Time Systems*, pp. 200–236, 2004.

[28] M. A. Serrano, A. Melani, S. Kehr, M. Bertogna, and E. Quiñones, "An analysis of lazy and eager limited preemption approaches under DAG-based global fixed priority scheduling," in *Proc. of ISORC*, 2017.

[29] S. Kramer, D. Ziegenbein, and A. Hamann, "Real world automotive benchmark for free," in *Proc. of WATERS*, 2015.

[30] R. Stafford, "Random vectors with fixed sum," Technical Report, 2006.

APPENDIX

Timed Automaton for Sporadic Tasks

Fig. 4 shows the TASK automaton for sporadic tasks. There are only a few differences between the periodic and sporadic task automata. First, in the sporadic TASK automaton, the **Completed** location has a self-loop with guard "`synch?`" to implement sporadic release behavior (i.e., a task does not necessarily release its next job after exactly T_i time units). Second, the invariant on the **Completed** location (i.e., "`t <= period()`") is removed so that the task can stay in this location even after its minimum inter-arrival time has

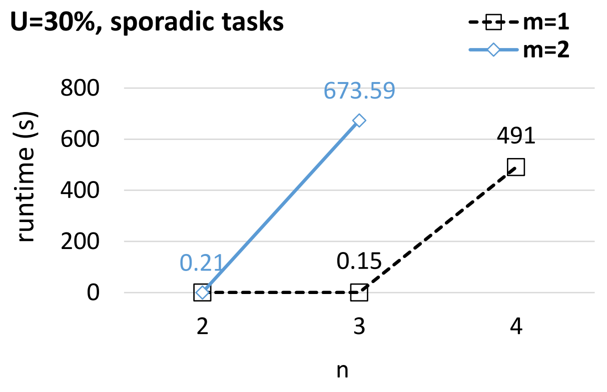


Fig. 5. The runtime of the test as a function of the number of sporadic tasks for $m = 1$ and $m = 2$ cores and $U = 30\%$.

been exceeded. Finally, we have replaced "`t == period`" with "`t >= period`" on the transition from **Completed** to **Suspended** so that this transition can be activated at any point after the minimum inter-arrival time of the task has passed.

Extended Experiments on Sporadic Task Sets

We conducted additional experiments on sporadic task sets to assess the scalability of the model shown in Fig. 4. Unfortunately, sporadic task behavior induces a much larger state space, which translates into substantially worse runtimes.

We used the same task set parameters as in Exp2 in Sec. V (varying the number of tasks for $n = \{2, 3, 4\}$) and assumed that every task is sporadic. Fig. 5 shows the observed runtime of the analysis for $m = 1$ and $m = 2$ cores and a varying number of tasks.

The results depicted in Fig. 5 clearly indicate that the runtime of the analysis increases exponentially with the increase in the number of sporadic tasks even for very small task counts. This is due to the large number of possible release scenarios in sporadic task sets. Unfortunately, for $m = 1$ and $n > 4$ as well as for $m = 2$ and $n > 3$, the analysis did not finish within the configured four-hour time budget. In conclusion, an exact analysis of sporadic workloads that scales to industrially relevant workload sizes remains a challenge for future work.