Bridging the Gap between Formal Verification and Schedulability Analysis: The Case of Robotics

Mohammed Foughali, Pierre-Emmanuel Hladik

Bridging the Gap between Formal Verification and Schedulability Analysis: The Case of Robotics

Mohammed Foughali[a], Pierre-Emmanuel Hladik[b]

[a]*Université Grenoble Alpes, CNRS, VERIMAG, Grenoble, France*
[b]*LAAS-CNRS, Université de Toulouse, CNRS, Toulouse, France*

Abstract

The challenges of deploying robots and autonomous vehicles call for further efforts to bridge the gap between the robotics, the real-time systems and the formal methods communities. Indeed, with robots being more and more involved in costly missions and contact with humans, a rigorous formal verification of their behavior in the presence of various real-time constraints is crucial. In order to increase our trust in its results, such verification should be carried out on models that are as close as possible to reality, and thus take into account hardware and OS specificities such as the number of cores provided by the robotic platform and the scheduling policy. In this paper, we propose a novel *binary-search-inspired technique* that allows to extend timed automata models of robotic specifications with dynamic-priority schedulers. Given a number of cores, the extended models can then be checked against various real-time and behavioral properties, including schedulability, within the same model checking framework. Our technique is implemented in an automatic translation from a robotic framework to UPPAAL, and evaluated on a real robotic case study, where it shows a significant gain in scalability as opposed to the *counting* technique used in the literature.

1. Introduction

In robotics, schedulability analysis is important as it provides answers on the deployability of an application given a hardware platform, a scheduling policy and some real-time requirements (*e.g.* hard or weakly hard real-time [7]). However, such analysis needs to be consolidated with the verification of other important behavioral and real-time properties like *liveness*, *bounded response* and *safety*. This need is flagrant in, *e.g.* mixed-criticality software, where some tasks are allowed to exceed their deadlines in which case some lower criticality jobs can be dropped (see for example [29]). Dually, while formal verification covers a wide range of properties, it classically abstracts away important hardware-software settings (*e.g.* number of cores, scheduling policy).

Email address: `mohammed.foughali@univ-grenoble-alpes.fr` (Mohammed Foughali)

This simplification renders verification results valid only if all tasks run in parallel at all times, which is seldom a realistic assumption.

Thus, bridging the gap between formal verification and schedulability analysis, and in a broader sense, between the formal methods and the real-time systems communities, would be of a great benefit to practitioners and researchers. One could imagine a unified framework where schedulability, but also other behavioral and real-time properties can be verified, on a model that is faithful to both the underlying robotic specification and the characteristics of the OS and the robotic platform. This is however very difficult in practice. For instance, theoretical results on schedulers are difficult to exploit in the robotic context given the complex model of tasks characterized by, for instance, a low-level fine-grain concurrency at the *functional layer*, where *components* directly interact with sensors and actuators (details in Sect. 3.1). Similarly, enriching formal models with scheduling policies usually penalizes the scalability of their verification, *e.g.* by means of model checking, even in non-preemptive settings. As an example, non-preemptive Earliest Deadline First scheduler (EDF) [27, 46] requires knowing the waiting time of tasks in order to compute their priorities. Model checking frameworks are hostile to this kind of behavior: UPPAAL [6], for instance, does not allow reading the value of a clock (to capture waiting time), which requires using *counting* methods that create further transitions in the model [26], leading to unscalable verification in the context of complex robotic systems.

In this paper, we propose a novel approach that allows schedulability analysis and formal verification of other properties within the same framework. We transform capturing waiting times from a counting problem to a *search* problem, which we solve using a binary-search-inspired technique. Integrated within a *template*, this technique allows us to automatically obtain, from functional robotic specifications, scalable formal models enriched with dynamic-priority cooperative schedulers. Our contribution is thus threefold: we (i) propose a novel approach for the general problem of capturing, at the model level, the value of time elapsed between some events, (ii) enable scalable model checking of robotic specifications against various behavioral and real-time properties, including schedulability, while taking into account hardware- and OS-related specificities and (iii) automatize the process so the formal models are obtained promptly from any robotic specification with no further modeling efforts. In addition, we provide a means of optimizing verification results based on counterexample analysis. We pay a particular attention to the readability of this paper by a broad audience in the different communities of robotics, formal methods and real-time systems. In that regard, we adopt a level of vulgarization with simple mathematical notions, together with sufficient references for further readings.

The rest of this paper is organized as follows. First, we propose a novel search technique that ensures alleviating the effect of modeling schedulers on scalability (Sect. 2). Then, in Sect. 3, we apply our search technique explained in Sect. 2 to a robotic case study. We present the *UPPAAL template* [21], which automatically generates formal (timed automata) models from robotic specifications, and show how we extend it with dynamic-priority schedulers, for a given number of cores. The extended template is then used to automatically generate UPPAAL models out of a real robotic specification, the number of cores on a real Robotnik platform and a dynamic-priority scheduling policy. Then, crucial behavioral and real-time properties are verified on the generated UPPAAL models (Sect. 4). In the same section, we show how the results can be improved using an optimization of the task model through suspension mechanisms. Finally, we evaluate the scalability of our approach while discussing the results obtained (Sect. 5), explore the related work in Sect. 6 and conclude with a summary and possible directions of future work (Sect. 7).

This work is an extension of the DETECT workshop paper [17] where we initially presented our approach and its application to a real robotic case study, with verification restricted to schedulability and schedulability-related bounded response properties. The extension we present here includes:

- The advantages of our search technique as opposed to another possible search-based solution that we call *interval test* (Sect. 2.2.3). This was only hinted at at the end of Sect. 2 of the workshop paper.

- Verification of properties that are not related to schedulability, that is properties that are neither the schedulability of tasks nor the maximum amount of time by which a task may overrun its period (Sect. 4.2). This shows further the efficiency of our approach in verifying other important properties on which one cannot obtain results using schedulability analysis techniques. The properties we verify are *liveness* and *leadsto* properties, crucial from a roboticist point of view.

- An optimization, based on suspension mechanisms, to improve results on schedulability (Sect. 4.3). We propose a new task model that allows to schedule, under the same scheduling and resources constraints, all tasks including the task scan which was not feasible in [17]. We explain how we come up with such a model after analyzing the counterexamples provided by UPPAAL, and detail our experiments that allow us, additionally, to reduce the response times for each task.

- An empirical evaluation of the scalability of our search technique as opposed to the counting one (Sect. 5.1). Properties are verified on UPPAAL models (of our robotic case study) that implement, respectively, the search method and the counting method. Then, the verification cost (time and memory consumption) is measured in both cases and compared, which shows a major gain in scalability using our search method.

- Freely available artefacts (Sect. 4.4). All experimental results are backed up with online freely accessible artefacts to allow reproducing the results,. This includes the automatic generation of UPPAAL models from robotic specifications and the verification of the generated UPPAAL models (Sect. 4), as well as the scalability evaluation (Sect. 5.1).

2. Capturing Time

In this paper, we focus on dynamic-priority *cooperative* (*i.e.* *non-preemptive*) schedulers, namely cooperative *Earliest Deadline First* (cEDF) and *Highest Response Rate Next (HRRN)*. The computations of either of these schedulers rely on a key information: the *waiting time*. Let us consider n tasks T_1 .. T_n. Whenever a core is free,

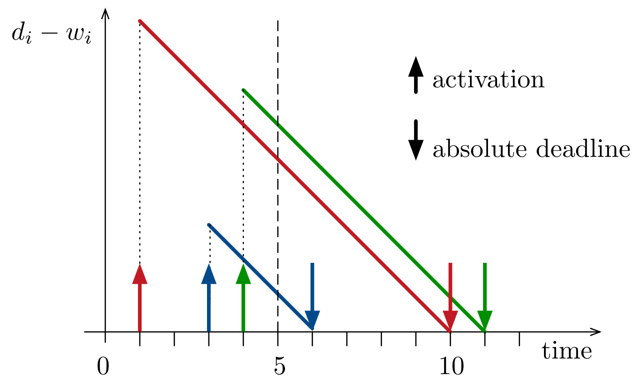


Figure 1: An illustrative Diagram of the cEDF policy. At each moment, the order of priority of tasks is directly obtained by comparing in increasing order the values of $d_i - w_i$. This value represents exactly how early the absolute deadline is (the global $time$ axis of the diagram), hence the policy name. For example, at global time 5, the blue task has the highest priority, followed by the red task and finally the green task.

w_i, the time each task T_i has been waiting in the queue so far, is used to compute its priority (see Fig. 1). In cEDF (resp. HRRN), the smaller (resp. higher) the value of $d_i - w_i$ (resp. $1 + \frac{w_i}{e_i}$), the higher the priority of T_i, where d_i is the (relative to task activation) *deadline* (resp. e_i is the *estimated execution time*) of T_i (more in Sect. 3.3). The task with the highest priority is then *released*: it is removed from the queue and a core is assigned to it. Since cores may be assigned to different tasks from an execution to another, the approach used in this paper is a global approach, *i.e.* task migration is possible [14].

Now, we need to integrate these schedulers into "model-checkable" formal models of robotic and autonomous systems. We explore thus two main formalisms: time Petri nets (TPN) and timed automata with urgencies (UTA), both extended with data variables. This is because most of paramount model checkers are based either on the former (*e.g.* Fiacre/TINA [8] and Romeo [32]) or the latter (*e.g.* UPPAAL [6] and IMITATOR [5]). Also, we already have templates that translate robotic specifications to both Fiacre/TINA [19] and UPPAAL [21]. Exploring both TPN and UTA will help us conclude which of these templates we need to extend with schedulers.

2.1. Premilinaries

We very briefly present TPN and UTA as to show the difference between these formalisms in the context of this paper. In the original "model checkable" version of each formalism, timing constraints (bounds of time intervals in TPN and clock constraints in UTA) are allowed in $\mathbb{Q}_{\geq 0} \cup \infty$, with time evolving over $\mathbb{R}_{\geq 0}$. Since we can always multiply all timing constraints by a natural that brings them to $\mathbb{N} \cup \infty$ (that is the *lowest common multiple LCM* of their denominators), we use natural constraints in our presentation. At the tooling level, this is done internally by tools like TINA, and is a requirement in frameworks like UPPAAL, where non-natural constraints are not allowed.

TPN:

Time Petri nets TPN [34] are Petri nets extended with time intervals (we only focus on closed intervals in this succinct presentation). Each transition t is associated with an interval $I(t) = [a_t, b_t]$ over $\mathbb{R}_{\geq 0}$ where $a_t \in \mathbb{N}$ (resp. $b_t \in \mathbb{N} \cup \infty$) is the *earliest* (resp. *latest*) *firing deadline* of t. The semantics of $I(t)$ is as follows: (i) firing a transition t is timeless and (ii) if t was last enabled since date d, t may not fire before $d + a_t$ and *must* fire before or at $d + b_t$ unless it is disabled before then by firing another transition. Time intervals in TPN are thus *relative* to the *enabledness* of transitions: if t is disabled, then $I(t)$ has no semantic effect. We use a version of TPN where guards and operations over data variables are possible on transitions. Note that, because of the timeless nature of firing a transition in time Petri nets (which is similar to that of taking transitions in timed automata), they are not to be confused with time**d** Petri nets [41], where firing a transition may take an arbitrary amount of time.

Timed automata with urgencies (UTA):

Timed automata TA [2] extend *finite-state Büchi automata* with real-valued clocks. The behavior of TA is thus restricted by defining (natural) constraints on the clock variables and a set of accepting states. A simpler version allowing local invariant conditions is introduced in [24], on which this paper and tools like UPPAAL rely. The syntax and semantics of TA in this paper follow those in [1] except that we refer to *switches* as *edges*. UTA [9] extend TA with a notion of urgency on edges, mainly (i) the *strong* urgency *eager*, denoted $\wr$, meaning the edge is to be taken as soon as enabled and (ii) the *weak* (by default) urgency *lazy*, meaning the edge *may* be taken when enabled. *Transitions* resulting from synchronizing some clock-constraint-free edges inherit the strongest urgency: if there is at least one $\wr$ edge in the synchronization, the resulting transition is also $\wr$. We use an extension of UTA where guards and operations over data variables are possible on edges.

TPN vs UTA:

What we need to retain for the sake of understanding this paper relates uniquely to the way time is handled in both formalisms. The main difference is that TPN feature no clocks (time intervals depend on transitions enablendess) whereas clocks in UTA evolve monotonically and independently from edges/transitions enabledness.

2.2. A High Level Presentation: Problem and Solution

We analyze the problem of capturing an arbitrary time, in both TPN and UTA models, at a framework-independent high level. We consider in each case a "process" that needs to store the value of time τ separating two events e and e', captured through the Booleans b and b', respectively. The value of τ is needed to perform further computations in the model. Since we are reasoning at a high level, we use standard algorithmic notations: $\leftarrow$ for assignment, $=$ for equality and $\neg$ for negation. In UTA, $reset(x)$ denotes resetting the valuation of clock x to zero. In graphical representations, guards are in green, operations in blue, and discontinued arcs/edges refer to missing parts of the model.

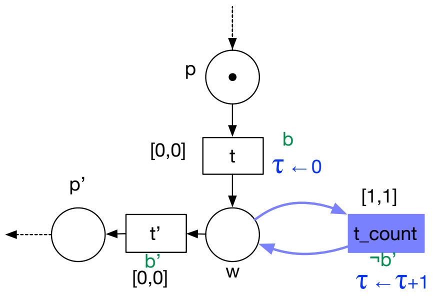


Figure 2: Capturing waiting time in TPN

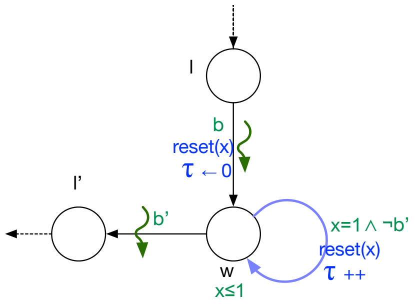


Figure 3: Capturing waiting time in UTA

Before we go any further, it is very important to distinguish between the modeling and the verification levels. Here, it is essential to capture and store τ in order to construct the model (the model depends on the value of τ, as explained for cEDF and HRRN above, and further detailed in Sect. 3.3). We cannot just use verification techniques to, for example, look for the bounds τ lies within, because the model itself relies on the *exact* value of τ for each $e- > e'$ sequence, the tracking of which is far from obvious. Indeed, TPN feature no clocks to capture τ directly in the model. Surprisingly, this is also the case for UTA: UTA-based model-checkers allow comparing a clock value to some constraints, but none of them permits *reading* such a value as to, for example, store it in a variable, since that would prevent *symbolic* representations like regions [3]. It follows that we can only approximate τ to its truncated natural value (or the natural that upper-bounds it).

2.2.1. The Classical "Counting" Method

Fig. 2 shows the classical way to capture τ in TPN. The original net is in black stroke: as soon as (denoted by the interval $[0, 0]$) b (resp. b') is true, transition t (resp. t') is fired, which unmarks place p (resp. the "waiting" place w) and marks place w (resp. p'). When p' is marked, we need the value of τ to perform further computations. The part in light blue is thus added to the net. Transition t_count, whose input and output place is w, is fired at each time unit as long as event e' is not received, which increments the value of τ. Consequently, as soon as p' is marked, τ holds the truncated natural value of the real duration d separating e and e' ($d - 1$ if d is natural).

An equivalent solution is implemented in UTA (Fig 3). Location l is to wait for event e. Eager (⟅) edges are taken as soon as their guard is true. The invariant on clock x at location w enforces taking the added edge (in light blue) at each time unit, which increments the value of τ. This method, referred to as *integer clocks*, is proposed to solve a similar problem in [26]. Note that the term "integer clocks" should not be confused with discrete-time models. Indeed, counting the waiting time here does not change the continuous nature of time in UTA.

Now, in either formalism, this solution is costly: adding transitions triggered at each time unit creates further interleavings and complexity that leads to combinatory explosion in our robotic case study (Sect. 5.1).

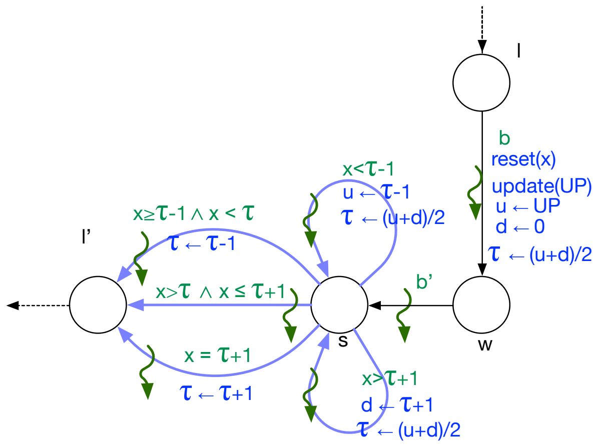


Figure 4: Capturing waiting time in UTA (search method)

2.2.2. *An optimized "Search" Method*

A key idea of this paper relies on transforming the *counting* problem into a *search* problem: instead of counting the time elapsed between e and e', we *search* for the value of τ once e' is received. This technique requires however an upper bound of τ (that is a value UP we know τ will never exceed), but this bound is flexible, that is it may vary from a sequence $e- > e'$ to another (see below).

The solution in UTA is shown in Fig 4. At location s (for *search*), at which time cannot elapse (all outgoing edges are ⇃), we undertake a binary search (aka *half-interval search*) that swings the value of τ within the bounds u (upper bound, initially UP) and d (lower bound, initially 0) till x lies within $[\tau - 1, \tau + 1]$, after which we simply assign τ the natural that lower-bounds the real value of x (by taking one of the edges from s to l'). This method is not implementable in TPN due to the absence of clocks.

As mentioned above, the value of UP may freely vary for each new $e- > e'$ sequence, that is each time location l is (re-)reached. This is done through the operation $update(UP)$, on the edge $l \rightarrow w$, which assigns a new value to UP according to some new estimation. The flexibility over UP is an important feature of our technique when solving this problem in a generic context (more in Sect. 2.2.3)

Now, we already know that, generally, binary search algorithms (logarithmic complexity) are faster than linear ones. We extrapolate that the number of times the self-loop edges at location s (in our search solution, Fig. 4) are taken is generally (and noticeably) smaller than the one of taking the self-loop at location w (in the counting solution, Fig. 3). Thus, there is a considerable gain in terms of state space size (and therefore scalability) when using the search technique, as we will confirm in Sect 5.1.

2.2.3. *An Interval Test Method*

Note that we can think of another solution, like simply testing the value of x between each pair of integers i and $i + 1$ within the range $0..UP$ on separate edges from s to l' (Fig. 5). Compared to our optimized search method (Sect. 2.2.2), this solution

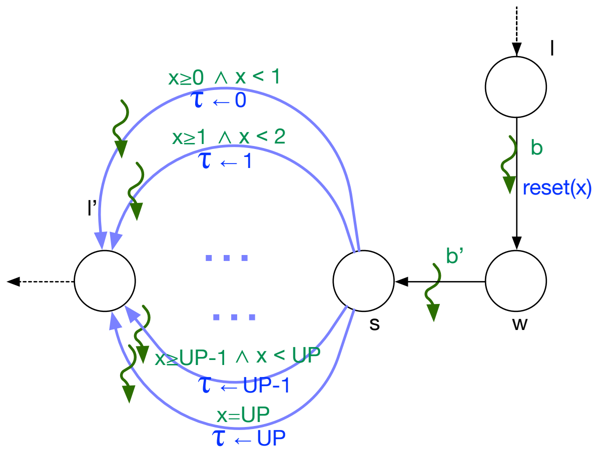


Figure 5: Capturing waiting time in UTA (interval test method)

poses a number of problems.

For instance, the interval test method requires that the upper bound UP must be itself upper bounded, that is, we need to upper-bound the values of UP for all $e->e'$ sequences. Indeed, if the value of UP is equal to, say, 10, for some $e->e'$ sequence and 15 for another $e->e'$ sequence, the number of edges connecting location s to location l' (Fig. 5) must be 15 to allow finding the value of τ in both cases. This limitation entails:

- Cumbersome models with a large number of edges (for instance, if the maximum value of UP is equal to 2000 time units, we will need 2001 edges from location s to location l'). This is disabling for a non-expert user to visualize, understand and debug their models, while in the search solution we propose, the model is uniform no matter how the value of UP varies.

- The impossibility to solve the problem when the upper bound of UP is unknown. Indeed, in such a case, the interval test method is not suitable (the upper bound of the interval within which testing edges are created is unknown), as opposed to our solution where one needs simply to update the value of UP before assigning it to u on the edge from l to s (Fig. 4). This renders the interval test solution less generic: for instance, in the context of schedulability, it would not work for variable deadline constraints the upper bound of which is not necessarily known (see examples in [45]).

In sum, the interval test method is not suitable for the general problem of capturing variable time separating several occurrences of events, and its cumbersome models are not convenient for a non-expert user. For this reason, and for a better readability of this paper, this method will be discarded henceforth as we will focus on opposing the novel (search) method to the (counting) one used in the literature.

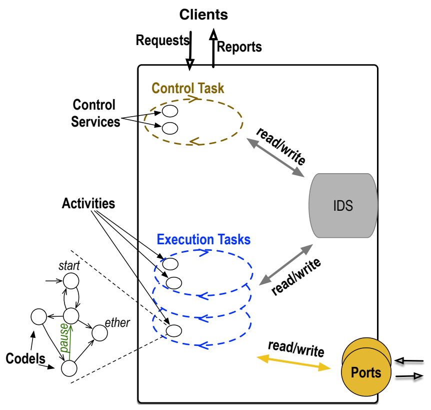


Figure 6: A generic GenoM3 component

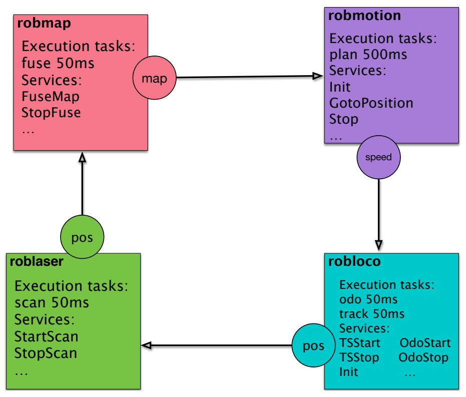


Figure 7: The RobNav application

3. Application To Robotic Systems

In this section, we aim to implement our method in order to obtain formal models of robotic specifications, enriched with dynamic-priority-based cooperative schedulers and the hardware constraints (number of cores) of a real platform. Such models will be then evaluated against a number of important behavioral and real-time properties, including schedulability (Sect. 4).

In previous work, we developed mathematically proven translations from the robotic framework GenoM3 (Sect. 3.1) to (TPN-based) Fiacre/TINA [18, 19] and (UTA-based) UPPAAL [21], which are implemented in automatic generators known as *templates*. Now, we only extend the UPPAAL template (since our search method, Sect. 2, is only implementable in UTA) with EDF and HRRN schedulers, given a number of cores.

Before we go into details of the extension of the template, we first briefly present GenoM3 in a high-level way (Sect. 3.1). Then, since the UPPAAL template output is proven faithful to the semantics of GenoM3 [21, 20], we explain some of GenoM3's important behavioral and real-time aspects using an automatically generated UPPAAL model of a GenoM3 *component* (Sect. 3.2).

3.1. GenoM3*:*

GenoM3 [20] is a component-based framework for specifying and implementing functional layer specifications. Fig. 6 shows the organization of a GenoM3 component. *Activities*, executed following *requests* from external *clients*, implement the core algorithms of the functionality the component is in charge of, for example, reading laser sensor, navigation. Two types of tasks are therefore provided: (i) a *control Task* to *process* requests, *validate* the requested activity if the processing returns no errors, and send *reports* to the clients and (ii) *execution task(s)* to execute activities. Tasks (resp. components) share data through the *Internal Data Structure IDS* (resp. *ports*).

An execution task, periodic, is in charge of a number of activities. With each period, it will run sequentially, among such activities, those that have been already validated by the control task. Activities are *finite-state machines* FSM, each state called a *codel*, at which a chunk of C or C++ code is executed. Each codel specifies a worst-case execution time (WCET) on a given platform, and the possible *transitions* following its execution. Taking a *pause* transition or a transition to the special codel `ether` ends the execution of the activity. In the former (resp. latter) case, the activity is resumed at the next period (resp. *terminated*).

IDS, ports & concurrency: At the OS level, tasks are parallel threads, with fine-grain concurrent access to the IDS and the ports. For execution tasks, the concurrency is at the codels level: a codel (in its activity, run in an execution task) locks only the memory fields required for its execution (simultaneous readings are allowed). Control tasks, on the other hand, use IDS fields when processing client requests and are thus in concurrency with the execution tasks (through the codels that use the same IDS fields). This aspect renders generalizing results on optimal schedulers very difficult in the context of robotics, as referred to in Sect 1. In the remainder of this paper, a codel that is *in conflict* (cannot execute at the same time) with another codel (or with the control task) because of this locking mechanism is called *thread unsafe* (*thread safe* otherwise). Because of the concurrency over ports, codels in conflict may belong to different components.

Case study:

In this paper, we consider a variation of the RobNav application developed by fellow researchers at LAAS (Fig. 7, technical details in [18]). The G$^{en}_{o}$M3 specification includes four components interacting to achieve autonomous terrestrial navigation. There are five execution tasks. Additionally, each component has a control task. The total number of tasks is therefore nine. The presentation in this paper focuses mainly on execution tasks and is greatly simplified. For more details on control tasks (*e.g.* how they are activated) and more complex aspects (*e.g.* interruption of activities), the interested reader may refer to [20].

The components collaborate to achieve a navigation as follows. The task odo of component ROBLOCO is in charge of writing the port **pos**, with the current position of the robot, based on the data it reads from the wheels sensors. Such a position is fed to the component ROBLASER. The latter's task scan is in charge of reading the laser sensor and augmenting the position in **pos** with the laser perception of its environment, and writing the result to the port **laser**. Task fuse of component ROBMAP uses the information on the **laser** port to update the map of the robot and its environment in the port **map**. The map in **map** (produced by component ROBMAP) and the position in **pos** (produced by component ROBLOCO) are used to compute the appropriate speed, by task plan of component ROBMOTION to reach a goal position, which is written on port **speed**. Finally, the loop closes with the task track of component ROBLOCO using **speed** to apply it to the robot controller. Note that task track of ROBLOCO is particularly critical due to a hardware contraint: the robot controller communicates at a fixed rate of 20 Hz, equivalent to the task's period (50 ms).

3.2. UPPAAL Template

We show in Fig. 8 a very simplified version of the automatically generated UPPAAL model of the periodic execution tasks odo and track (component ROBLOCO) from our case study (one time unit in the model is equal to 1 ms). This model follows the implementation model shown in [21], proven faithful to the semantics of G$^{en}_{o}$M3 [21, 20]. The *urgency* process is to enforce $\wr$ transitions through the urgent channel *exe*: UPPAAL supports $\wr$ transitions only resulting from the synchronization of two or more clock-constraint-free edges (it does not $\wr$ edges as such). Note that not all activities are shown.

Each task t is composed of a *manager* (to execute, at its location $manage$, activities sequentially), a *timer* (to send, through the Boolean $tick\_t$, period signals to the manager), and a number of activities the task executes. The $next()$ function browses the array tab_t, whose cells are records with two fields: n (activity name) and s (activity status), and returns the index of the first activity that is previously validated by the control task and still not executed in this cycle (an information retrieved through the s fields). The manager and the activities use this function, together with the variables $lock\_t$ and $turn\_t$, to communicate: the manager computes the identity of the next activity to execute and gives it the control (through $turn\_t$ and $lock\_t$). The activity will then execute until it pauses (*e.g.* reaching $track\_pause$ in *TrackSpeedStart*) or terminates (*e.g.* reaching $ether$ in *InitPosPort*), in which case it computes the identity of the next activity to execute (in i) and gives the control back to the manager. When there are no more activities to execute (i is equal to the size of tab_t and the manager has the control through $lock\_t$), the manager transits back to its initial location $start$.

Now, at the activity level, a signal is transmitted, when the activity pauses or terminates (through the Boolean $finished\_t$), to the control task (not shown here), so the latter informs the client that requested such activity and updates the status of the activity in tab_t. A thread-unsafe codel c is represented using two locations, c and c_exec (*e.g.* $compute$ and $compute\_exec$ in *TrackOdoStart*). The guards and operations over the array of Booleans mut ensure no codels in conflict (*e.g.* codel $track$ in *TrackSpeedStart* and codel $compute$ in *TrackOdoStart*) execute simultaneously, and the urgency on $c \to c\_exec$ edges ensures the codel executes (or loses some resources) *as soon as* it has the required resources. The invariants on locations c_exec reflect the fact that a codel is executed in its WCET at most. For thread-safe codels, c_exec locations are not needed, and the invariant is thus associated with c locations. Therefore, time at location c_exec (resp. c) is not allowed to progress further than the WCET of the thread-unsafe (resp. thread-safe) codel c. The guards $x > 0$ on the edges of the form $c \to (c_{exec} \to$ if codel c is thread unsafe) reflect the fact that a codel needs a non-zero time to execute. In practice, since codels often write ports and the IDS, this guard is later replaced by $x \geq bcet_c$ where $bcet_c$ is the *best-case execution time* (BCET) of codel c, that is the least amount of time allowing it to execute and write the data fields/ports it accesses. Due to the heavy computations performed in the robotic context, the BCET and WCET of a codel c are oftentimes close to one another.

As we can see, this model is highly concurrent: tasks may run on different cores and locking shared resources is fine grain (at the codels level) with simultaneous readings allowed. These features allow to maximally parallelize the tasks, but render manual verification and analytical techniques for schedulability analysis impractical.

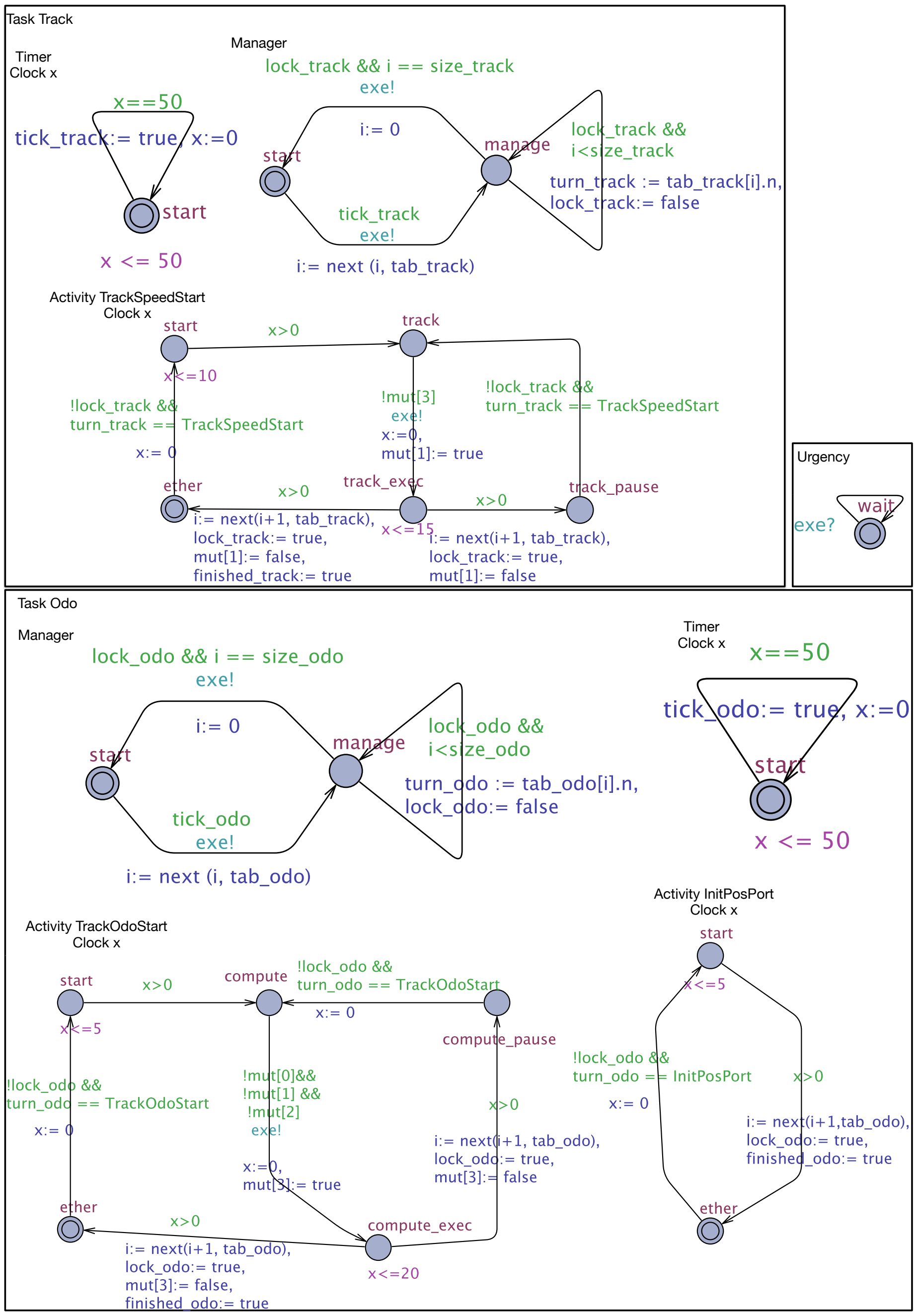


Figure 8: Partial UPPAAL model of tasks odo and track (automatically generated)

3.3. Extending With Schedulers

We show how to extend the UPPAAL template with cEDF and HRRN schedulers. First, we use the case study to exemplify on how to adapt the solution shown in Sect. 2.2.2 to efficiently and correctly integrate such schedulers. Then, we explain how control tasks, having no deadlines, are handled. Finally, we automatize the extension with schedulers within the template.

Example:

Let us get back to the ROBLOCO example. The manager processes are the only ones that will be affected. Also, we will need a *scheduler* process. Let us first introduce the constants, shared variables and channels that the scheduler and managers need to communicate and synchronize.

Constants: The number of tasks in the application is denoted by the constant $size\_sched$. An array of constant naturals $periods$ is introduced in which, with each task denoted by index i, a period $periods[i]$ is associated.

Shared variables: We need a queue (array) T of size $size\_sched$ in which we insert tasks dynamic priorities. Then, since priorities change their position when T is dequeued, we need an array p such that $p[i]$ tracks the index of T that points to the cell holding the dynamic priority of task i (that is $T[p[i]]$). Also, we need a natural len to store the number of waiting tasks, an array w to store the waiting time for each task i, and a natural s_count to store the number of tasks for which the *search* for the waiting time has already finished. Finally, the natural nc stores the number of available cores.

Channels: A handshake channel $insert$ is introduced to increment len. A broadcast channel up synchronizes with as many tasks as len to start the search operation. Besides, a broadcast channel en synchronizes the scheduler with all waiting tasks in order to diffuse the decision for each task on whether it is released (given a core to execute) or not (needs to wait further). Finally, a broadcast channel $srch$ eliminates interleaving between managers during the search operation (more explanation below). We show now the scheduler, then how the manager of odo is modified accordingly:

Scheduler: The scheduler (Fig. 9) has three locations: $start$ (initial), $update$ and $give$. The last two are *committed.* A committed location is a location that entails both an urgency and a priority. That is, whenever the system's global state contains at least a committed location, time is not allowed to progress (urgency) and the next transition in the system must involve an edge whose source location is committed (priority). In our system, this will (i) prevent unnecessary interleaving with other interactions in the system and (ii) enforce urgency on all the outgoing edges of committed locations.

The self-loop edge at location $start$, synchronized on $insert$, increments the number of waiting tasks each time a task wants to execute (we do not need a guard on this edge to ensure T is not full, because the size of T is already equal to the number of tasks in the application). From location $start$, it is possible to reach location $update$ providing there is at least one task to release.

At location $update$, an edge synchronized over the channel $srch$ allows looping as long as the search has not finished for all waiting tasks (with one search operation for all tasks at once thanks to the broadcast channel $srch$). Another edge permits reaching the location $give$ as soon as the search has finished for all waiting tasks (captured through

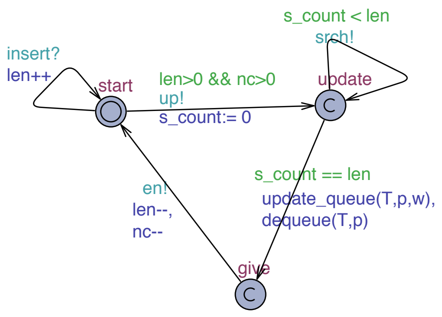


Figure 9: UPPAAL model of the scheduler

the value of s_count). On this very edge, the core of the scheduling algorithm is implemented: function $update\_queue()$ updates the dynamic priorities in each $T[p[i]]$ before the function $dequeue()$ finds the task with the highest priority and removes its priority by updating both p and T. The core of $update\_queue()$ is given later in this section.

Now, from location $give$, the initial location is immediately reached through an edge synchronized on the channel en. The number of cores as well as the number of waiting tasks is decremented as the task having the highest priority is released.

Manager: In the new manager model (Fig. 10), we have a clock x and four intermediate locations: ask, $search$, $decide$, and $error$. To meet the upper-bound condition (Sect. 2.2.2), we reason as follows. In such a real-time system, we do not tolerate that a task is still waiting (for a core) since a duration equal to its period. Thus, we enforce an urgency (through an invariant) from location ask (at which the clock x tracks the waiting time) to location $error$ as soon as the waiting time is equal to the task period. Then, at the analysis step, we make sure error is unreachable in all managers in the model. If this is not the case, the analysis becomes worthless (as soon as some manager reaches location $error$), so we must drop it and, if possible, increase the number of cores and retry.

The remaining aspects are rather straightforward considering the scheduler model and the search technique in Sect. 2.2.2 (we reuse the variable names for search bounds, u and d, from Fig. 4): $p[i]$ is updated from $start$ to ask, the edge from ask to $search$ is synchronized on up to drag all waiting tasks managers to the committed location $search$ at which they loop, synchronized on $srch$, until the search ends. When all managers reach their respective $decide$ locations, s_count is equal to len (the number of waiting tasks) and, in each manager, either the edge to $manage$ or ask is taken, depending on whether the task i is released (recognized through $p[i]$ equalling -1), or not (otherwise). In the latter case, d (resp. u), the lower (resp. upper) bound for the next search is updated to the current value of $w[i]$ (resp. $period[i]$). Finally, the task frees the core at the end of execution (operation $nc++$ on the edge from $manage$ to $start$).

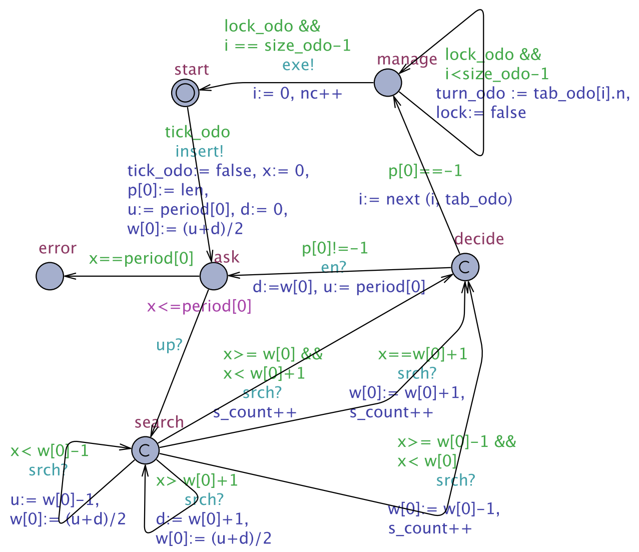


Figure 10: UPPAAL model of the odo manager (enriched)

Control tasks:

So far, we explained how the scheduler handles periodic execution tasks. On the other hand, control tasks do not have a particular deadline. Indeed, a control task is activated only when a client sends it a request, an event following which it processes the request and validates the activity corresponding to it. This absence of deadline breaks the spirit of schedulers such as EDF.

From a roboticist point of view, control tasks should have the highest priority: they must react to and process clients requests as soon as possible. Therefore, in the case of EDF, for instance, we assign to control tasks the same strictly negative deadline (*e.g.* -1), which allows any control task to be executed immediately (if a core is already available) or as soon as a core is free (otherwise) whenever it receives a request from a client. Indeed, we already know that the waiting time of an execution task is always positive (otherwise location *error* of its manager is reached, see the Unreachable Error Property UEP in Sect. 4.1), so a negative deadline gives a control task an immediate higher priority than all execution tasks in the queue. This way, the spirit of EDF is met with a classical FIFO tie-breaking for control tasks activated at the same time.

In practice, in a robotic application like ours, control tasks execute only once at the very beginning where a client sends all requests of activities involved in the application sequentially. As soon as all requests are processed, control tasks compete no more for cores which become exclusively available for execution tasks. It follows that it is important to verify that, for example, each control task eventually receives and processes the requests of which it is the destination, as we will see in Sect. 4.2.

```
 1  <'if {$argv >= 10 && $argv < 30} {'>
 2  /* scheduling */
 3  /* update dynamic priorities */
 4  void update_queue (int &T[size_sched], int &p[size_sched], int &w[size_sched]) {
 5  int i;
 6  for (i:= 0; i<size_sched; i++) {
 7          if (p[i] >= 0) {T[p[i]]:=
 8  <'if {$argv < 20} {'>lcm_p + w[i] * (lcm_p/period[i])
 9  <'} else {'>period[i] - w[i]<'}'>;}}
10  }
11  <'}'>
```

Listing 1: Automatic generation example

Automatic synthesis:

At this stage, we are ready to automatize the process. The user may pass the flag -sched to the UPPAAL template, followed by two numbers: the scheduling policy (1 for HRRN or 2 for cEDF) and the number of cores (a natural in $0..9$). For instance, the following command line generates the UPPAAL model of the GenoM3 specification *spec.gen*, that integrates a cEDF scheduler over four cores:

```
genom3 uppaal/model -sched=24 spec.gen
```

Now, the core of the UPPAAL template is enriched to automatically integrate such specificities in the generated model. As an example, Listing 1 shows the piece of the template that generates the $update\_queue()$ function. The interpreter evaluates what is enclosed in <' '> in Tool Command Language (Tcl) and outputs the rest as is. Line 1 conditions generating the function to the validity of the option passed by the programmer: the variable $argv$ captures the option passed as an integer, and since, as we have said above, this integer is $1x$ for HRRN or $2x$ for cEDF where x$\in$ $0..9$, $argv$ should be superior to 10 and strictly inferior to 30 for the option to be valid. Then, lines 8-9 generate the right dynamic-priority formula according to the specified scheduler in the option. In the case of cEDF (option comprised in $20..29$, that is $argv$ superior to 20[1], line 9), we simply subtract the waiting time $w[i]$ from the (relative) deadline, fixed to the period $period[i]$. For HRRN (option comprised in $10..19$, that is $argv$ strictly inferior to 20[2], line 8), we proceed as follows. The estimated execution time is usually an average computed dynamically. Here, we fix it statically to the period of the task (the same reasoning was followed in [19] for the Shortest Job First (SJF) scheduling policy). Then, since we can only perform integer divisions in UPPAAL, we look for the LCM lcm_p of all periods and multiply it by the priority formula. Since lcm_p is strictly positive, the comparison of priorities is not affected.

[1]the part strictly inferior to 30 is already guaranteed by the incorporating if at line 1

[2]the part superior to 10 is already guaranteed by the incorporating if at line 1

4. Results

At this point, we have a template that automatically generates the counterpart UPPAAL model from (i) a given robotic specification written in G$^{en}_{o}$M3 (ii) a cooperative dynamic-priority scheduler (cEDF or HRRN) and (iii) the number of cores in a robotic platform. We will use such template to analyze the deployability of our case study (Sect. 3.1) on the *Robotnik Summit-XL platform*[1], featuring an embedded four-core PC running Linux. That is, given the specification (behavioral and real-time aspects) of our case study and the real capacities of the robotic platform and OS (number of cores and scheduling policy), we check whether a set of requirements, crucial to the correct functioning and safety of the robot, can be satisfied.

Therefore, we automatically generate, from the case study and the number of cores on the Robotnik platform (which we vary from 1 to 4), UPPAAL models extended with cEDF and HRRN schedulers. Then, we check whether the requirements, broken into two main sets (those that are related to schedulability, and those that are not), are satisfied by formulating them as UPPAAL properties and verifying them using the UPPAAL verifier (Sect. 4.1, Sect. 4.2). Finally, we propose an optimization to improve both schedulability and maximum response time for all tasks (Sect. 4.3) and give details about how to reproduce the experiments using our online artefacts (Sect. 4.4).

Also, we derive, from our automatically generated UPPAAL models, equivalent *counting-based* models, where the managers implement the classical counting technique (detailed in Sect. 2.2.1) to count the waiting time of tasks instead of searching it. We rely on such models to evaluate the scalability of our search technique in this case study as opposed to the counting one (Sect. 5.1). For readability, we do not provide details on the equivalent counting-based UPPAAL models of our case study as (i) such models are available as public artefacts (Sect. 4.4) and (ii) implementing the counting technique, explained in Sect. 2.2.1, in the manager models is rather straightforward.

Note that all the results we obtain are identical for both cEDF and HRRN. Thus, the results presented in this paper are valid for both policies, with the condition that, in the case of HRRN only, the period of each task is considered as its static estimated execution time, as we explained under the **"Automatic Synthesis"** paragraph above.

4.1. Schedulability-Related Requirements

There are two schedulability-related requirements. The track task is *hard real-time*: as explained in Sect. 3.1, the robot controller communicates at a fixed rate of 20 Hz. Thus, new computed speeds must be sent to the controller at 50 ms, so track must always finish executing its activities within its period (R1). The remaining tasks are *soft real-time*, with the condition that the time by which a task exceeds its period must be always smaller than the period itself (R2).

[1]https://www.robotnik.eu/web/wp-content/uploads//2019/03/Robotnik_DATASHEET_SUMMIT-XL-HL_EN-web-1.pdf

Requirement R1:

In order to formulate R1 as an UPPAAL property, we reason as follows. A task is busy (waiting or executing activities) as long as its manager is not at location $start$ (we verify beforehand that location $manage$ is reachable). Thus, we check whether no new signal from the timer is sent while the manager is not at location $start$. This is expressed for task track as follows:

```
A[] (not manager_track.start imply not tick_track)
```

Which is a *safety* property that is strictly representative of the schedulability of task track (R1). We call it the *schedulability property* (SP).

However, we must recall that, in order to ensure that the verification of any property makes sense, location $error$ must be unreachable in all managers of all tasks (see for instance Fig. 10 for the manager of task odo):

```
A[] not (manager_odo.error or manager_track.error or manager_fuse.error or
     manager_scan.error or manager_plan.error)
```

This is again a safety property, which we call the *unreachable error property* (UEP). In order to check R1 (and any other requirement henceforth), UEP must hold, because reaching location $error$ in any task makes the analysis results worthless as we explained in Sect. 3.3. Therefore, we proceed as follows. We start by one core ($nc = 1$) and verify SP. If it does not hold, we increase nc and reverify. Otherwise, we make sure that UEP holds for all tasks.

The verification results using UPPAAL, which can be obtained using the links given in the artefacts (Sect. 4.4), show that SP is violated for $nc \in \{1, 2, 3\}$. As soon as we increase nc to four, SP holds. We proceed thus by verifying UEP, which we make sure it also holds. Consequently, R1 is satisfied for $nc = 4$. Thus, nc is fixed to four in the remainder of this paper's experiments. Also, as long as we bring no changes to the UPPAAL model (which will not happen before Sect. 4.3), we will no longer recall that UEP must hold in order to satisfy some requirement, since we already verified that it holds for $nc = 4$.

Requirement R2:

In order to check R2 through verifying UPPAAL properties, we ask, for each task t, for the maximum value of clock x at location $manage$. To do this, we ask UPPAAL to evaluate the following formula $sup\{manager\_t.manage\} : manager\_t.x$. Now, since $manage$ is the location at which activities are executed, the result of such evaluation corresponds to the maximum amount of time t needs to execute, starting from its period signal. This maximum value includes the time t waits in the queue, and is referred to as the *maximum response time* of t in the remainder of this paper. Then, we simply compare the maximum response time to the period of t. The results are given in table 1. All tasks are feasible, besides scan (component roblaser) that may exceed its period by up to 15 ms (which is inferior to its period). R1 and R2 are thus both met on the four-core platform, and we can provide the precise maximum amount of time by which the only non-schedulable task may overrun its periods.

t	odo	track	plan	fuse	scan
$period$	50	50	200	50	50
$sup\{manager\_t.manage\} : manager\_t.x$	40	40	60	45	65

Table 1: Maximum response time per task (four cores).

4.2. Other Requirements

So far, the requirements we checked are all related to schedulability: we verify whether a task is feasible, and if it is not, we look for the maximum amount of time by which it may exceed its deadline. However, as said in Sect. 1, verifying schedulability is often not sufficient in the context of robotic and autonomous systems. In the following, we present further requirements of our case study that are not related to schedulability, and thus cannot be checked through schedulability analysis techniques.

There are two non-schedulability-related requirements. For control tasks, it is very important to ensure that any request received by a client at any time is eventually processed (R3). Additionally, requested activities must eventually start executing (R4). Due to the low-level mutual exclusion over the IDS in a G^{en}oM3 component (involving both execution and control tasks), such requirements are not necessarily satisfied. Additionally, schedulability does not provide any guarantee on either R3 or R4. For instance, task odo may be feasible while one of its requested activities (*e.g. TrackOdoStart*, Fig. 8) has never started, that is R4 is not met.

Requirement R3:

To check whether R3 is satisfied, we formulate it as an UPPAAL property using the $a\ leadsto\ b$ pattern, denoted $a\ \ --> \ \ b$, which evaluates to *true* if and only if *whenever atomic proposition* a *holds, atomic proposition* b *will eventually hold.* The below listing shows how to write, using such pattern, the UPPAAL *leadsto* property that encodes R3 for the control task of, for instance, component ROBLOCO:

```
CT_robloco.receive and cl.start --> CT_robloco.finish
```

Where $CT\_robloco$ is the UPPAAL process mapping the control task of component ROBLOCO (we recall that control tasks models are not shown in this paper), $receive$ is the location from which $CT\_robloco$ receives client requests (which we verify beforehand that it is reachable), and $finish$ is the location that $CT\_robloco$ reaches after processing any received request. Additionally, cl is the UPPAAL process (not shown here) mapping the client that sends activities requests to all components and $start$ is its initial location from which it starts sending requests when all control tasks are ready (at their respective $receive$ locations). The results show that R3 is satisfied for all control tasks in the case study.

Requirement R4:

We know that, in the UPPAAL model, an activity is initially at location $ether$, and that reaching location $start$ (from $ether$) denotes the beginning of execution of such activity (see for example Fig. 8). Thus R4 means that each UPPAAL process of each requested activity must reach at some point of the system evolution its location $start$. This can be formulated using the *liveness* formula $A <>$ as follows:

```
A <> act.start
```

Where act is the UPPAAL process of some requested activity. This property means that location $start$ of act is eventually and *inevitably* reached in all possible evolution paths of the global UPPAAL system. Our analysis (which can be reproduced using the links given in the artefacts, Sect. 4.4) shows that R4 is met for all requested activities in all components.

4.3. Improving Schedulability

In Sect. 4.1, we have seen that cEDF (and HRRN alike) allows us to satisfy both schedulability-related requirement R1 and R2 on the four-core Robotnik platform. However, the task scan is still unfeasible: it may exceed its deadline by up to 15 ms. While this is acceptable according to R2, it is still undesirable. Indeed, scan reads the laser sensor so it would be better if it could process sensor data within the deadline, especially if the robot is evolving in a highly critical context. For instance, if the sensor fails, it would be possible to detect the failure and react to it the earliest possible.

Therefore, we will try here to find a way to make all tasks schedulable, and thus make our case study suitable for hard real-time applications as well, considering the same scheduling policy (cEDF or HRRN) and the same Robotnik platform. To do so, we reiterate the experiments of checking R2 (Sect. 4.1), and analyse the counterexamples, given by UPPAAL that correspond to task scan violating its deadline.

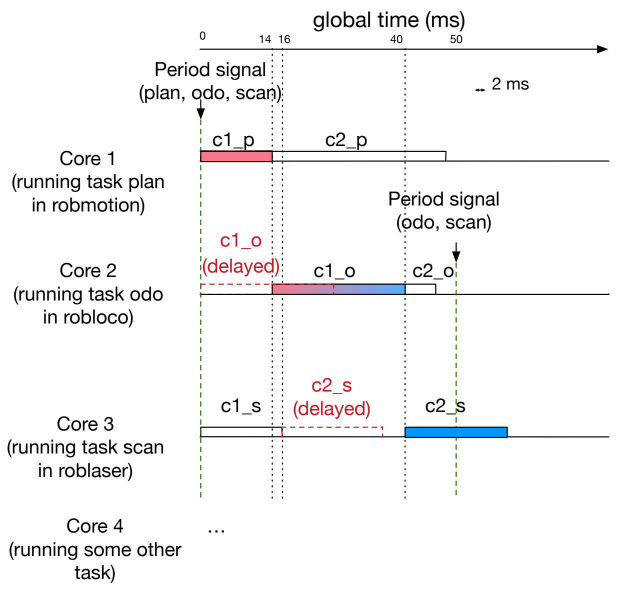
Counterexample Analysis:

The analysis shows that the violation happens each time task plan (component ROBMOTION) and odo (component ROBLOCO) are running in parallel with task scan. More precisely, some codels in activities run by plan (ROBMOTION) write the port **speed**, which is read by some codels in activities run by odo in ROBLOCO (Fig. 7). Similarly, some codels in activities run by odo (ROBLOCO) write the port **pos**, which is read by some codels in activities run by scan in ROBLASER (Fig. 7). Thus, task odo (ROBLOCO) delayed by task plan (ROBMOTION) because of the mutual exclusion over port **speed**, delays in turn task scan (ROBLASER) because of the mutual exclusion over port **pos**, which results in task scan missing its deadline.

Fig 11 gives an abstract view of this phenomenon. It is purely illustrative: it is based neither on the exact timing constraints nor on the real activities behavior of tasks in our case study. Solid (resp. dashed) stroke rectangles symbolize concrete (resp. delayed) codels execution, each codel of the form ci_x where i is some unique identifier and x the first letter of the task the codel is executed within. Colorful solid fills reflect concurrency : $c1\_p$ and $c2\_s$ are not in conflict, but each of them is in conflict with $c1\_o$ (due to the sharing of ports as explained above). Non-colored (white-filled) rectangles refer to thread-safe codels. The first period signal (to the left) is given at global time zero or some global time that is a common multiple of the periods of tasks odo, track and plan (we consider it zero for simplification). In this example, each task t is supposed to execute codels $c1\_t$ and $c2\_t$ in this order. For simplicity, execution times are fixed. In this example they have the values 14 ms, 34 ms, 26 ms, 6 ms, 16 ms and 20 ms for *c1_p*, *c2_p*, *c1_o*, *c2_o*, *c1_s* and *c2_s*, respectively.

Now, at the very start, either odo or plan has to wait, because *c1_o* is in conflict with *c1_p*, but scan may execute since *c1_s* is thread safe. In the scenario shown in Fig 11, *c1_p* seizes the shared memory (port **speed**) first, which delays the execution of *c1_o* by 14 ms. In parallel, *c1_s* starts executing. At global time 14 ms, *c1_p* finishes executing which means that *c1_o* may start. Two milliseconds later, *c1_s* finishes its execution, which means that task scan needs to execute *c2_s* as soon as global time is equal to 16 ms. However, this is not possible because *c1_o*, in conflict with *c2_s* already started executing, which delays the start of *c2_s* to global time 40 ms, and thus makes task scan miss its period.

This phenomenon originates from task plan, although it is actually not in conflict with task scan: if task odo (in particular codel *c1_o*) had not been delayed by task plan (in particular codel *c1_p*), scan would have not missed its period. Task odo, on the other hand, is not affected at all by its conflicts with tasks plan and scan: it would still respect its deadline in the other possible scenario, that is when *c1_o* seizes the shared port first.



global time (ms)
0
14
16
40
50
Period signal (plan, odo, scan)
2 ms
c1_p
c2_p
Core 1 (running task plan in robmotion)
Period signal (odo, scan)
c1_o (delayed)
c1_o
c2_o
Core 2 (running task odo in robloco)
c2_s (delayed)
c1_s
c2_s
Core 3 (running task scan in roblaser)
Core 4 (running some other task)
...

Figure 11: Abstract example: task scan missing its deadline

Thus, exceeding the period of task scan has nothing to do with the number of cores, but with the concurrency aspect between codels over shared memory (more precisely, ports in this case, Sect. 3.1). This is what we confirm when increasing nc further than four: the results remain the same, all tasks are schedulable except scan, which may exceed its period by up to 15 ms. This behavior is similar to the classical priority inversion in scheduling theory (see chapter 2 in [30]) that caused the failure of the Mars Pathfinder spacecraft in 1997.

In the example given in Fig 11, we can solve the problem by hard-cording shared memory access rules at the OS level. However, such solution is tedious as it would entail utilizing low-level OS libraries of shared memory management in order to hard-code precedence between codels in accessing ports. Also, the changes made to memory access may easily affect task track (in ROBLOCO), in conflict with both odo and scan (in ROBLOCO and ROBLASER, respectively), the most critical task in our case study (requirement R1, Sect. 4.1). Finally, Fig 11 is just an illustrative example, as we recall, which does not reflect the real complexity we have in the case study, with several activities and dozens of interleaving scenarios.

Therefore, the solution of this problem needs to be exploitable by the robotic programmer within their area of expertise. In other words, we need to find a lightweight approach that works in all possible scenarios and is easily implementable at the underlying robotic application level, that is the G$^{en}_{o}$M3 specification or implementation. This is not an easy task especially that we cannot, due to hardware (sensors and actuators) constraints, modify period values or offset period signals in the style of [37].

A suspension-based solution:

The trick is borrowed from the notion of "self-suspension", classically used in contexts where tasks access external devices [33]. The reasoning is as follows. Since task plan schedules comfortably under four cores (table 1), suspending its execution at its very beginning for some (statically defined) amount of time would relieve task odo from memory constraints, and eventually lead to task scan meeting its deadline requirements. We need however to know the value of suspension time, or suspension delay, that we call sd, such that we manage to make task scan schedulable without side effects on the schedulability of the remaining tasks.

We proceed through an empirical evaluation of sd. Firstly, we extend the model of task plan, more precisely its manager, to take the suspension (at the very beginning of the task) into account (Fig. 12). At location $delay$, the manager waits, using an additional clock y, for sd time units before it reaches location $manage$ to start executing the activities of task plan. This is the only difference with the basic model without suspension (*e.g.* given previously in Fig. 10 for task odo). Then, we tune the value of sd while verifying the schedulability of all tasks by (i) verifying UEP (Sect. 4.1), since the UPPAAL model has changed and will change for each new value of sd, and (ii) directly asking UPPAAL for the maximum value of clock x in location $manage$ of each manager (exactly like we did in Sect. 4.1 to obtain table 1).

Fig. 13 shows a plot of the empirical results, with sd varying between 0 ms and 50 ms (we make sure UEP holds in each case). The Y axis corresponds to the UPPAAL response when we ask for the maximum response time of each task (including the waiting time), using the same property in table 1. Any value of sd in the gray (light or dark alike) area, that is in the interval $I = [\mathit{1}, \mathit{35}]$ allows to schedule all tasks. In this interval, the maximum response time of task scan, the only task that was not feasible before adding the suspension delay, drops from 65 ms to 50 ms, which is exactly equal to its period. Moreover, we may isolate the interval $I' = [\mathit{11}, \mathit{30}]$, inside the interval I, which corresponds to the least maximum response time of all tasks combined: 35 ms, 25 ms, 40 ms, 50 ms, and 60 ms for odo, track, fuse, scan and plan respectively.

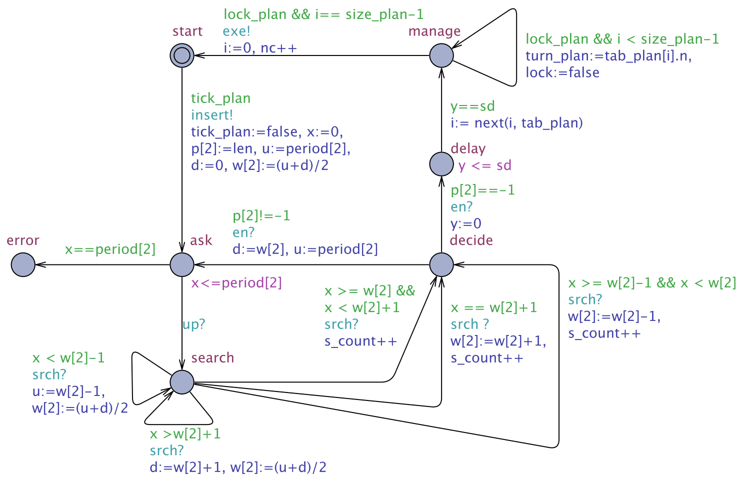


Figure 12: UPPAAL model of the manager of task plan (with suspension)

This means that, while suspending task plan with any value of sd within I ensures schedulability for all tasks, choosing sd within I' guarantees, in addition, the smallest maximum response time for all tasks.

Note that results on schedulability and/or maximum response time may only worsen for values of sd above 50. Indeed, the experiments show that the pattern of maximum response times repeats itself periodically (in $[51, 100]$, then in $[101, 150]$, etc.) for all tasks except plan whose maximum response time keeps growing until plan is no longer shcedulable. This is why we restrict the plot we show in Fig. 13 to the interval $[0, 50]$.

Effect on other properties:

Our solution improves schedulability as well as the overall maximum response times, which means that schedulability-related requirements R1 and R2 (Sect. 4.1) are largely met with a considerable optimization. However, we need to make sure that the remaining requirements R3 and R4 are still satisfied under the new solution, that is with a suspension delay comprised within interval I, *i.e.* between 1 ms and 35 ms (or, ideally, within interval $I' = [11, 30]$ as explained above).

Therefore, we follow the same steps as in Sect. 4.2 to verify liveness and leadsto properties corresponding to R3 and R4. The results (see Sect. 4.4 for practical details) show that both requirements are met for any value of sd in interval I (and thus for any

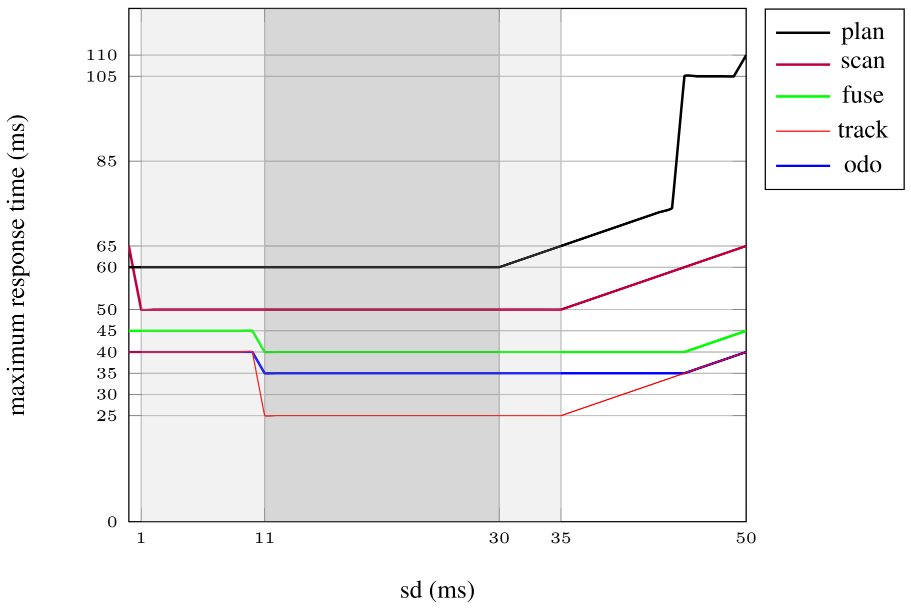


Figure 13: Effect of sd values on tasks schedulability

value of sd in interval I', since $I' \subseteq I$).

Returning to G$^{en}_{o}$M3:

Our results are exploitable by robotic programmers. Indeed, they may consider suspending task plan at the beginning of each period by, ideally, an amount of time in I', using dedicated functions such as `usleep()`. This may be done through, for instance, inserting such function call in the implementation of the task, generated from G$^{en}_{o}$M3 specifications for the PocoLibs or ROS-Com middleware, just prior to codels invocation.

4.4. Artefacts

All of our experiments material is freely available at `https://github.com/Mo-F/sched-artefact`, with a detailed README file. The repository includes the G$^{en}_{o}$M3 files specifying the robotic components involved in the case study. Additionally, it provides the automatically generated UPPAAL models (as extended with a cEDF scheduler) as well as the *query* files with all the properties that we verified, to allow reproducing the results in both cases: before and after the suspension-based optimization. In the after-optimization case, we provide also all the results that produced the plot given in Fig. 13. Moreover, there are instructions on how to generate the UPPAAL model from the G$^{en}_{o}$M3 components if the user wants to reproduce the experiments wholly following the complete transformation and verification chain. Finally, the equivalent counting-based UPPAAL models are freely accessible, which the user may use to try

		Property			
		SP (nc=1)	SP (nc=2)	SP (nc=3)	SP (nc=4)
Verification Cost (Search-Based Model)	time (s)	0.35	0.63	83	5
	Memory (Mb)	34	46	1620	799
Verification Cost (Counting-Based Model)	time (s)	10	21	> 465	10
	Memory (Mb)	1659	1691	OOM	1629

Table 2: Verification cost of schedulability of task track (search vs. counting).

to verify the same properties and compare the verification cost with that of the automatically generated (search-based) models.

5. Discussion

5.1. Scalability

As explained in Sect. 2, we expect our novel binary-search-based technique to outperform the classical counting one. We need to confirm such expectations in our experimental setting. For that, as said at the beginning of Sect. 4, counting-based UPPAAL models, equivalent to the automatically generated ones from our case study (which, we recall, rely on our search technique, see for example the manager in Fig. 10), are derived and used to verify the same properties as in Sect. 4.1 and Sect. 4.2.

The results are, without surprise, identical with both models, with the difference that the counting-based model fails to scale for some properties and provides thus no answer as UPPAAL runs out of memory (OOM). The experiments show that, in all cases, our search technique scales much better than the counting one. For instance, table 2 shows the verification cost (time and memory consumption) for the UPPAAL property that we named SP in Sect. 4.1, that is the schedulability property that encodes requirement R1, on a MacBook Pro laptop with 4 Gb of RAM dedicated to the UPPAAL verifier. In this example, our search technique performs from two to 30 times (resp. two to 50 times) better than the counting one in terms of verification time (resp. memory consumption). In addition, for $nc = 3$ (three cores), UPPAAL consumes the whole 4 Gb of memory available without succeeding to give an answer with the counting-based model. That is, on any computer with 4 Gb of RAM, we would not have been able to verify SP for three cores, and thus would have failed to know whether the requirements are satisfied for this particular setting.

However, even with our technique, model checking still suffers from the general problem of state space explosion. For instance, we encounter scalability issues as soon as we try to implement preemptive schedulers, or apply our approach to (significantly) larger case studies such as the *Osmosis* experiment detailed in Sect. 2.4.1 of the PhD thesis [16], involving ten components and over twenty tasks.

5.2. Design Optimisation

As shown in Sect. 4.3, using model checking, through UPPAAL, allows us to progress from yes-or-no verification to answering the questions “why no?” and “can we make it a yes?”. Through the analysis of counterexamples provided by the tool, we manage to draw redesign principles that allow us to improve the verification results, and even satisfy some properties that did not hold beforehand. Such redesign principles are accessible to the robotic programmer and applicable within the robotic implementation.

Nevertheless, this is not obvious in the general case. Besides the user-friendly interface of UPPAAL allowing to replay counterexamples, the analysis of the latter is out of reach of practitioners with no formal background and no advanced knowledge of the tool. In other words, contrary to the forward chain from robotic specification to verification in UPPAAL, the details of which we conceal using automatic generation, the backward chain, that is redesigning based on counterexamples analysis, still lacks an important step to make it convenient for robotic programmers.

5.3. Summary

The results are encouraging. First, schedulability is verified for all tasks and, if schedulability is violated, the precise upper bound of the time the period is exceeded is retrieved. Second, we further verify other properties than schedulability, equally important to the correct functioning and safety of the robot. All this is done automatically at both the modeling (template) and verification (model checker) levels, while taking into account the real hardware and OS specificities. In addition, we manage, based on counterexample analysis, to optimize the scheduling using suspension delays, which results in feasibility of all tasks as well as a considerable reduction of their maximum response time.

However, we do not know whether we can obtain better results (*e.g.* schedulability with a smaller number of cores, reducing further the load on cores) with preemptive schedulers. Indeed, we may not rely on generic theoretical results to know whether preemptive schedulers may perform better than cooperative ones in the case of robotics, and, unfortunately, preemption generally does not scale with model checking (Sect. 6). Furthermore, counterexample analysis is done manually, which is not suitable for non-expert practitioners. Possible directions to deal with these issues are given in Sect. 7.

6. Related Work

Real-time analysis and model checking in robotics:

Bridging the gap between analytical techniques (*e.g.* in schedulability analysis) and model checking is generally not explored at the functional layer of robotic and autonomous systems. On one hand, works focusing on model checking [28, 47, 36, 35] ignore hardware and OS constraints (number of cores and scheduling policy) which restricts the validity of results to only when the number of cores in the platform is at least equal to that of the robotic tasks, which is usually an unrealistic assumption. On the other hand, real-time analysis of functional robotic components [44, 40, 22], mainly focusing on schedulability, is non automatic and gives no guarantees on other important behavioral and timed properties such as liveness, safety and bounded response.

Moreover, although schedulability analysis of preemptive and non-preemptive policies has been well studied on mono- and multi-processor architectures [14] (especially for fixed priority and EDF), its theoretical findings are still hard to generalize to the case of robotics. This is due, mainly, to the fact that such analysis is generally pessimistic and, more importantly, limited to simple task execution models that do not allow the complexity of a robotic application to be expressed. For instance, the experiments detailed in [39] show how, contrary to general theoretical results on schedulability analysis, some non preemptive schedulers perform better than preemptive ones in the case of a mobile robot application. This limitation on task models makes it hard to adapt existing approaches, for example [48] where both schedulability and control problems are considered, to the case of robotics.

Model-checking for schedulability:

Several works propose using generic model-checking tools to perform schedulability analysis. These studies differ in the underlying task models, the considered scheduling policies and the employed model-checking methods. The majority of works [49, 13, 43, 12, 10, 50] use Timed Automata with UPPAAL as model-checker and some with Stopwatches. In contrast, the paper [23] uses of the symbolic model-checker nuSMV [11], while [31] uses an approach based on the transformation of time Petri nets into linear hybrid automaton and the verification of the transformed models with the HyTECH model-checker [25]. The majority of the scheduling policies studied are preemptive fixed priorities and EDF for mono- or multi-processor architectures. Only [12] proposes different schedulers such as LLF and LLREF but this work is based on a deterministic task model and is quite similar to a simulator. More recently, [50] focuses on a non-preemptive policy with self-suspending tasks. Without surprise, papers that present experimental evaluations note scalability issues both in terms of the number of tasks and in relation to non-deterministic variables, *i.e.*, variation in execution times, uncertainty about task starts, etc. Moreover, all of these works rely on basic task models that are not suitable for robotics.

The trend of using model-checking-based techniques to verify schedulability has even led to the development of tools and prototypes that are specific for schedulability analysis, such as TIMES [4] and POLA [38]. Unfortunately, such tools are too high-level to implement complex robotic applications, which prevents their use as a uniform environment to verify various real-time and behavioral properties, including schedulability, in the robotic context. Furthermore, they target mainly preemptive schedulers, and consequently suffer from scalability issues in large applications.

Capturing time in formal models:

To the best of our knowledge, enriching formal models of robotic applications with dynamic-priority cooperative schedulers is a non-explored research direction. Still, the problem that arises, *i.e.* storing arbitrary time values in variables to construct the model, has been already encountered in other domains. It is the case of [26], where the authors use the counting technique that we explained in Sect. 2.2.1, to which they refer as *integer clocks*, to perform arithmetics on clock values stored in natural variables. Such integer clocks, relying on a classical counting algorithm, lead to unscalable models in the case of large robotic applications like the one we verify in this paper.

Comparison to our previous work:

In [19], we extended the Fiacre template with First Come First Served (FCFS) and Shortest Job First (SJF) cooperative schedulers. We concluded that we need to integrate more "intelligent" schedulers with dynamic priorities (*e.g.* cEDF and HRRN), which we efficiently achieve in this paper using a novel binary-search-based technique. Practitioners can thus automatically generate, from any robotic specification, a formal model enriched with cEDF or HRRN, on which various properties can be verified within the same framework, UPPAAL, with majors gains in terms of scalability. Finally, compared to our paper [17], we verify also properties that are not related to schedulability, propose a self-suspension-based solution to schedule all tasks and provide public artefacts to reproduce all experiments (as detailed at the end of Sect. 1).

7. Conclusion

In this paper, we elaborate an effort to bridge the gap between the robotics, the real-time systems and the formal methods communities. We aim at providing, automatically, formal models of robotic specifications that take into account the actual hardware and OS specificities. In order to take into account optimized (dynamic-priority) schedulers, we propose a scalable *search* method that we implement and automatize within the UPPAAL template developed in [21]. The results are encouraging, and allow to deploy our case study on a four-core robotic platform while fulfilling real-time requirements, which we could not achieve using the classical FCFS and SJF schedulers available since [19].

Additionally, we propose to improve the deployability of our case study by making all tasks in the application schedulable, and improving their maximum response time. To that end, we analyze the counterexamples provided by UPPAAL for tasks that were not initially feasible on the platform. Afterwards, we propose inserting suspension delays and identify the values of such delays that allow to schedule all tasks and reduce the maximum response times. Therefore, we do not only verify the properties, but further propose redesigning guidelines to improve the results and guarantee a better deployability of the system.

We still face scalability issues when we try to implement preemptive schedulers. Yet, such schedulers may further improve the deployability. It follows that a possible direction of future work is to consider the extension of the UPPAAL-SMC (*Statistical Model Checking*) template [21] with preemptive schedulers in order to verify the properties statistically (up to some high probability). Although exploring this direction would restrict us to non-critical contexts, works like [15] may help us deal with the severe lack of probabilistic requirements in the robotics domain (*i.e.* to answer the question of what could be considered as a "sufficiently high probability" for a robotic application).

Also, analyzing counterexamples is not convenient for non-expert practitioners. A possible direction of future work is to bridge G$^{en}_{o}$M3 and UPPAAL in the "backward" direction: return counterexamples as G$^{en}_{o}$M3 traces, exploitable by robotic programmers. In order to achieve such promising yet tedious task (especially because of the timed nature of UPPAAL), we need first to formally define what a trace is in the context of G$^{en}_{o}$M3. Our work on formalizing the latter [20] would be a good starting point.

Finally, our models do not go all the way to more low-level specificities, such as cache interferences and the effects of task migration on processor load and timing constraints. A possible direction of future work is to extend our models to take into account such aspects. The recent work [42] uses UPPAAL to model cache interferences, and may inspire us to progress in this direction.

References

[1] Rajeev Alur. Timed automata. In *proc. of the International Conference on Computer Aided Verification*, CAV, pages 8–22. Springer, 1999. DOI: 10.1007/3-540-48683-6_3.

[2] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994. DOI: 10.1016/0304-3975(94)90010-8.

[3] Rajeev Alur, Costas Courcoubetis, and David L. Dill. Model-checking in dense real-time. *Information and computation*, 104(1):2–34, 1993. DOI: 10.1006/inco.1993.1024.

[4] Tobias Amnell, Elena Fersman, Leonid Mokrushin, Paul Pettersson, and Wang Yi. TIMES: a tool for schedulability analysis and code generation of real-time systems. In *Proc. of the International Conference on Formal Modeling and Analysis of Timed Systems*, FORMATS, pages 60–72. Springer, 2003. DOI: 10.1007/978-3-540-40903-8_6.

[5] Étienne André, Laurent Fribourg, Ulrich Kühne, and Romain Soulat. Imitator 2.5: A tool for analyzing robustness in scheduling problems. In *Proc. of the International Symposium on Formal Methods*, FM, pages 33–36. Springer, 2012. DOI: 10.1007/978-3-642-32759-9_6.

[6] Gerd Behrmann, Alexandre David, and Kim G. Larsen. A tutorial on UPPAAL. In *Proc. of the Formal Methods for the Design of Real-Time Systems*, SFM-RT, pages 200–236. Springer, 2004. DOI: 10.1007/978-3-540-30080-9_7.

[7] Guillem Bernat, Alan Burns, and Albert Liamosi. Weakly hard real-time systems. *IEEE transactions on Computers*, 50(4):308–321, 2001. DOI: 10.1109/12.919277.

[8] Bernard Berthomieu, Pierre-Olivier Ribet, and François Vernadat. The tool TINA – construction of abstract state spaces for Petri nets and time Petri nets. *International Journal of Production Research*, 42(14):2741–2756, 2004. DOI: 10.1080/00207540412331312688.

[9] Sébastien Bornot, Joseph Sifakis, and Stavros Tripakis. Modeling urgency in timed systems. In *Proc. of the International Symposium on Compositionality*, COMPOS, pages 103–129. Springer, 1998. DOI: 10.1007/3-540-49213-5_5.

[10] Franco Cicirelli, Angelo Furfaro, Libero Nigro, and Francesco Pupo. Development of a schedulability analysis framework based on ptpn and uppaal with stopwatches. In *Proc. of the IEEE/ACM 16th International Symposium on Distributed Simulation and Real Time Applications*, DS-RT, pages 57–64. IEEE, 2012. DOI: 10.1109/DS-RT.2012.16.

[11] Alessandro Cimatti, Edmund Clarke, Fausto Giunchiglia, and Marco Roveri. NuSMV: a new symbolic model checker. *International Journal on Software Tools for Technology Transfer*, 2(4):410–425, 2000. DOI: 10.1007/s100090050046.

[12] Mikel Cordovilla, Frédéric Boniol, Eric Noulard, and Claire Pagetti. Multiprocessor schedulability analyser. In *Proc. of the ACM Symposium on Applied Computing*, SAC, pages 735–741. ACM, 2011. DOI: 10.1145/1982185.1982345.

[13] Alexandre David, Jacob Illum Rasmussen, Kim Guldstrand Larsen, and Arne Skou. *Model-based Framework for Schedulability Analysis Using Uppaal 4.1*, pages 93–119. Computational Analysis, Synthesis, and Design of Dynamic Systems. CRC Press, 1 edition, 2009. ISBN 978-1-4200-6784-2.

[14] Robert I. Davis and Alan Burns. A survey of hard real-time scheduling for multiprocessor systems. *ACM Computer Survey*, 43(4):35:1–35:44, 2011. DOI: 10.1145/1978802.1978814.

[15] José Louis Díaz, Daniel García, Kanghee Kim, Chang-Gun Lee, Lucia Lo Bello, José Maria López, Sang Lyul Min, and Orazio Mirabella. Stochastic analysis of periodic real-time systems. In *Proc of the IEEE Real-Time Systems Symposium*, RTSS, pages 289–300. IEEE, 2002. DOI: 10.1007/s11241-008-9053-6.

[16] Mohammed Foughali. *Formal Verification of the Functional Layer of Robotic and Autonomous Systems*. PhD thesis, Institut national des sciences appliquées de Toulouse, 2018.

[17] Mohammed Foughali. On reconciling schedulability analysis and model checking in robotics. In *Proc. of International Conference on Model and Data Engineering*, MEDI, pages 32–48. Springer, 2019. DOI: 10.1007/978-3-030-32213-7_3.

[18] Mohammed Foughali, Bernard Berthomieu, Silvano Dal Zilio, Félix Ingrand, and Anthony Mallet. Model checking real-time properties on the functional layer of autonomous robots. In *Proc. of the International Conference on Formal Engineering Methods*, ICFEM, pages 383–399. Springer, 2016. DOI: 10.1007/978-3-319-47846-3_24.

[19] Mohammed Foughali, Bernard Berthomieu, Silvano Dal Zilio, Pierre-Emmanuel Hladik, Félix Ingrand, and Anthony Mallet. Formal verification of complex robotic systems on resource-constrained platforms. In *Proc. of the Conference on Formal Methods in Software Engineering*, FormaliSE, pages 2–9. ACM, 2018. DOI: 10.1145/3193992.3193996.

[20] Mohammed Foughali, Silvano Dal Zilio, and Félix Ingrand. On the Semantics of the GenoM3 Framework. Technical Report 19036, LAAS/CNRS, 2019.

[21] Mohammed Foughali, Félix Ingrand, and Cristina Seceleanu. Statistical model checking of complex robotic systems. In *Proc. of the International Symposium on Model Checking Software*, SPIN, pages 114–134. Springer, 2019. DOI: 10.1007/978-3-030-30923-7_7.

[22] Nicolas Gobillot, Charles Lesire, and David Doose. A design and analysis methodology for component-based real-time architectures of autonomous systems. *Journal of Intelligent & Robotic Systems*, 96(1):123–138, 2019. DOI: 10.1007/s10846-018-0967-5.

[23] Nan Guan, Zonghua Gu, Mingsong Lv, Qingxu Deng, and Ge Yu. Schedulability analysis of global fixed-priority or EDF multiprocessor scheduling with symbolic model-checking. In *Proc. of the 11th IEEE Symposium on Object Oriented Real-Time Distributed Computing*, ISORC, pages 556–560. IEEE, 2008. DOI: 10.1109/ISORC.2008.74.

[24] Thomas Henzinger, Xavier Nicollin, Joseph Sifakis, and Sergio Yovine. Symbolic model checking for real-time systems. *Information and computation*, 111(2): 193–244, 1994. DOI: 10.1006/inco.1994.1045.

[25] Thomas Henzinger, Pei-Hsin Ho, and Howard Wong-Toi. HyTech: A model checker for hybrid systems. In *Proc. of the International Conference on Computer Aided Verification*, pages 460–463. Springer, 1997. DOI: 10.1007/3-540-63166-6_48.

[26] Xiaowan Huang, Anu Singh, and Scott A. Smolka. Using integer clocks to verify clock-synchronization protocols. *Innovations in Systems and Software Engineering*, 7(2):119–130, 2011. DOI: 10.1007/s11334-011-0152-5.

[27] Mehdi Kargahi and Ali Movaghar. Non-preemptive earliest-deadline-first scheduling policy: A performance study. In *Proc. of the IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, MASCOTS, pages 201–208. IEEE, 2005. DOI: 10.1109/MASCOTS.2005.44.

[28] Moonzoo Kim and Kyo Chul Kang. Formal Construction and Verification of Home Service Robots: A Case Study. In *Proc. of the International Symposium on Automated Technology for Verification and Analysis*, ATVA, pages 429–443. Springer, 2005. DOI: 10.1007/11562948_32.

[29] Karthik Lakshmanan, Dionisio De Niz, Ragunathan Rajkumar, and Gabriel Moreno. Resource allocation in distributed mixed-criticality cyber-physical systems. In *Proc. of the International Conference on Distributed Computing Systems*, ICDCS, pages 169–178. IEEE, 2010. DOI: 10.1109/ICDCS.2010.91.

[30] Insup Lee, Joseph Y-T. Leung, and Sang H. Son. *Handbook of Real-Time and Embedded Systems*. Chapman & Hall/CRC, 1st edition, 2007. ISBN 1584886781, 9781584886785.

[31] Didier Lime and Olivier H. Roux. Formal verification of real-time systems with preemptive scheduling. *Real-Time Systems*, 41(2):118–151, 2009. DOI: 10.1007/s11241-008-9059-0.

[32] Didier Lime, Olivier H. Roux, Charlotte Seidner, and Louis-Marie Traonouez. Romeo: A parametric model-checker for Petri nets with stopwatches. In *Proc. of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS, pages 54–57. Springer, 2009. DOI: 10.1007/978-3-642-00768-2_6.

[33] Cong Liu and James H Anderson. Suspension-aware analysis for hard real-time multiprocessor scheduling. In *Proc. of the Euromicro Conference on Real-Time Systems*, ECRTS, pages 271–281. IEEE, 2013. DOI: 10.1109/ECRTS.2013.36.

[34] Philip Merlin and David Farber. Recoverability of Communication Protocols: Implications of a Theoretical Study. *IEEE Transactions on Communications*, 24 (9):1036–1043, 1976. DOI: 10.1109/TCOM.1976.1093424.

[35] Alvaro Miyazawa, Pedro Ribeiro, Wei Li, Ana Cavalcanti, and Jon Timmis. Automatic property checking of robotic applications. In *Proc. of the International Conference on Intelligent Robots and Systems*, IROS, pages 3869–3876. IEEE, 2017. DOI: 10.1109/IROS.2017.8206238.

[36] Levente Molnar and Sandor Veres. System verification of autonomous underwater vehicles by model checking. In *Proc. of the OCEANS-EUROPE Conference*, pages 1–10. IEEE, 2009. DOI: 10.1109/OCEANSE.2009.5278284.

[37] Mitra Nasri, Robert I Davis, and Björn B Brandenburg. FIFO with offsets: High schedulability with low overheads. In *Proc. of the Real-Time and Embedded Technology and Applications Symposium*, RTAS, pages 271–282. IEEE, 2018. DOI: 10.1109/RTAS.2018.00035.

[38] Florent Peres, Pierre-Emmanuel Hladik, and François Vernadat. Specification and verification of real-time systems using POLA. *International Journal of Critical Computer-Based Systems*, 2(3-4):332–351, 2011. DOI: 10.1504/IJCCBS.2011.042332.

[39] Maurizio Piaggio, Antonio Sgorbissa, and Renato Zaccaria. Pre-emptive versus non-pre-emptive real time scheduling in intelligent mobile robotics. *Journal of Experimental & Theoretical Artificial Intelligence*, 12(2):235–245, 2000. DOI: 10.1080/095281300409856.

[40] Steve Qadi, Ala Goddard, Jiangyang Huang, and Shane Farritor. A performance and schedulability analysis of an autonomous mobile robot. In *Proc. of the IEEE Euromicro Conference on Real-Time Systems*, ECRTS, pages 239–248. IEEE, 2005. DOI: 10.1109/ECRTS.2005.2.

[41] Chander Ramchandani. Analysis of asynchronous concurrent systems by Petri nets. Technical report, DTIC Document, 1974.

[42] Nathanaël Sensfelder, Julien Brunel, and Claire Pagetti. Modeling cache coherence to expose interference. In *Proc. of the IEEE Euromicro Conference on Real-Time Systems*, ECRTS. IEEE, 2019. DOI: 10.4230/LIPIcs.ECRTS.2019.18.

[43] Wei Sheng, Yanyan Gao, Li Xi, and Xuehai Zhou. Schedulability analysis for multicore global scheduling with model checking. In *Proc. of the 11th International Workshop on Microprocessor Test and Verification*, pages 21–26. IEEE, 2010. DOI: 10.1109/MTV.2010.13.

[44] Jiazheng Shi, Steve Goddard, A. Lal, and Shane Farritor. A real-time model for the robotic highway safety marker system. In *Proc. of the IEEE Real-Time and Embedded Technology and Applications Symposium*, RTAS, pages 331–340. IEEE, 2004. DOI: 10.1109/RTTAS.2004.1317279.

[45] Chi-Sheng Shih, Lui Sha, and Jane Liu. Scheduling tasks with variable deadlines. In *Proc. of the IEEE Real-Time and Embedded Technology and Applications Symposium*, RTAS, pages 120–122. IEEE, 2001. DOI: 10.1109/RTTAS.2001.929874.

[46] Michael Short. On the implementation of dependable real-time systems with non-preemptive edf. *Electrical Engineering and Applied Computing*, 20, 2011. DOI: 10.1007/978-94-007-1192-1_16.

[47] Daniel Simon, Roger Pissard-Gibollet, and Soraya Arias. Orccad, a framework for safe robot control design and implementation. In *Proc. of the Workshop on Control Architectures of Robots: software approaches and issues*, CAR, 2006.

[48] Sakthivel Manikandan Sundharam, Nicolas Navet, Sebastian Altmeyer, and Lionel Havet. A model-driven co-design framework for fusing control and scheduling viewpoints. *Sensors*, 18(2):628, 2018. DOI: 10.3390/s18020628.

[49] Libor Waszniowski and Zdeněk Hanzálek. Formal verification of multitasking applications based on timed automata model. *Real-Time Systems*, 38(1):39–65, 2008. DOI: 10.1007/s11241-007-9036-z.

[50] Beyazit Yalcinkaya, Mitra Nasri, and Björn B. Brandenburg. An exact schedulability test for non-preemptive self-suspending real-time tasks. In *Proc. of the Design, Automation Test in Europe Conference Exhibition*, DATE, pages 1228–1233. IEEE, 2019. DOI: 10.23919/DATE.2019.8715111.