On Reconciling Schedulability Analysis and Model Checking in Robotics

Mohammed Foughali

On Reconciling Schedulability Analysis and Model Checking in Robotics

Mohammed Foughali

LAAS-CNRS, Université de Toulouse, CNRS, INSA, Toulouse, France
mfoughal@laas.fr

Abstract. The challenges of deploying robots and autonomous vehicles call for further efforts to bring the real-time systems and the formal methods communities together. In this paper, we discuss the practicality of paramount model checking formalisms in implementing dynamic-priority-based cooperative schedulers, where capturing the waiting time of tasks has a major impact on scalability. Subsequently, we propose a novel technique that alleviates such an impact, and thus enables schedulability analysis and verification of real-time/behavioral properties within the same model checking framework, while taking into account hardware and OS specificities. The technique is implemented in an automatic translation from a robotic framework to UPPAAL, and evaluated on a real robotic example.

1 Introduction

In robotics, schedulability analysis needs to be consolidated with the verification of other important properties such as *bounded response* and *safety*. This need is flagrant in *e.g.* mixed-criticality software, where some tasks are allowed to exceed their deadlines. Dually, important hardware-software settings (*e.g.* number of cores, scheduling policy) are classically abstracted away in formal verification. This renders verification results valid only if all tasks run in parallel at all times, which is seldom a realistic assumption.

Bridging the gap between these communities would be of a great benefit to practitioners and researchers: one could imagine a unified framework where schedulability, but also other 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 given *e.g.* the low-level fine-grain concurrency at the *functional layer* of robotic systems, where *components* directly interact with sensors and actuators (details in Sect. 3.1). Similarly, enriching formal models with *e.g.* dynamic-priority-based scheduling policies usually penalizes the scalability of their verification, even in non-preemptive settings. As an example, cooperative EDF [20] requires knowing the waiting time of tasks in order to compute their priorities. Model checking frameworks are hostile to this kind of behavior: UPPAAL [7], for instance, does not allow reading the value of a clock (to capture waiting time), which requires using discrete-time-like methods that create further transitions in the model [19], 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 model checking robotic specifications 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. 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 technique that ensures alleviating the effect of modeling schedulers on scalability (Sect. 2). Then, in Sect. 3, we present the *UPPAAL template* [15], which automatically generates formal models from robotic specifications, and show how we extend it with dynamic-priority schedulers using the solution shown in Sect. 2. In Sect. 4, we use the automatically generated models to verify properties over a real-world case study, before we explore the related work in Sect. 5 and conclude with a discussion and possible future work (Sect. 6).

2 Capturing Time

In this paper, we focus on dynamic-priority *cooperative* (*i.e. non preemptive*) schedulers, namely cooperative *Earliest Deadline First (EDF)* 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, w_i, the time each task T_i has been waiting in the queue so far, is used to compute its priority. In EDF (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.

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 extended 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 [22]) or the latter (*e.g.* UPPAAL [7] and IMITATOR [6]). Also, we already have templates that translate robotic specifications to both Fiacre/TINA [12] and UPPAAL [15]. Exploring both TPN and UTA will help us conclude on which of these templates we need to extend with schedulers.

2.1 Preliminaries

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 $\mathbb{Q}_{\geq 0} \cup \infty$. 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.

Time Petri nets TPN: Time Petri nets TPN [24] 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: 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 consider a version of TPN where guards and operations over data variables are possible on transitions.

Timed automata with urgencies UTA: Timed automata TA [4] 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 [18], on which this paper (and tools like UPPAAL) relies. The syntax and semantics of TA in this paper follow those in [2] 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 consider a version 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.

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 EDF and HRRN above, and further detailed in Sect. 3.3). We cannot just use verification techniques to *e.g.* 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 *e.g.* 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).

The classical method: Fig. 1 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).

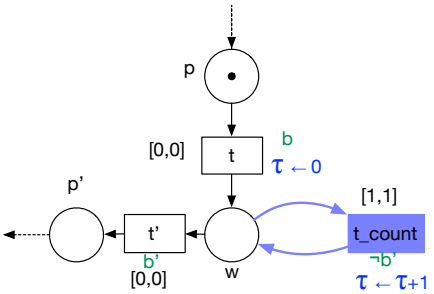

Fig. 1: Capturing waiting time in TPN

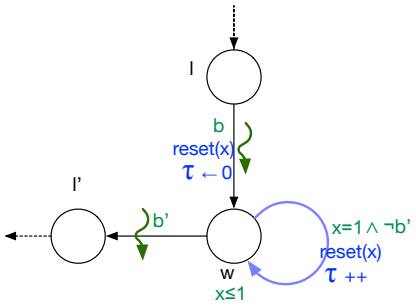

Fig. 2: Capturing waiting time in UTA

An equivalent solution is implemented in UTA (Fig 2). Location l is to wait for event e. Eager ($\wiggle$) 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 [19].

Now, in either formalism, this solution is very costly: adding transitions triggered at each time unit creates further interleavings and complexity that leads to combinatory explosion in real-world robotic case studies (Sect. 4.1).

An optimized 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). In our solution, this value may change for each sequence $e- > e'$ (UP may take a different value each time location l (or place p) is (re-)reached).

The solution in UTA is shown in Fig 3. At location s (for *search*), at which time cannot elapse (all outgoing edges are $\wiggle$), 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.

Now, we already know that, generally, binary search algorithms (logarithmic complexity) are faster than linear ones. We extrapolate that the number of times edges from location s (in the optimized solution, Fig. 3) are taken is generally (and noticeably) smaller than the one of taking the self-loop at location w (in the classical solution, Fig. 2). Thus, there is a considerable gain in terms of state space size (and therefore scalability) when using the optimized technique, as we will confirm in Sect 4.1.

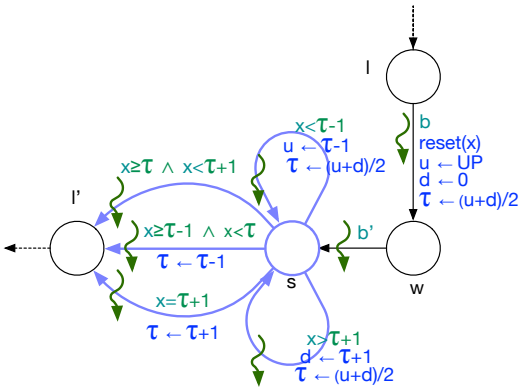

Fig. 3: Capturing waiting time in UTA (optimized solution)

Note that we can think of more optimized solutions, 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'. This would not, however, work if the value of UP varies from an $e->e'$ sequence to another, which renders the solution less generic (*e.g.* in the context of schedulability, it would not work for tasks with variable deadlines [29]).

3 Application to Robotic Systems

In previous work, we bridged the robotic framework GenoM3 (Sect. 3.1) with Fiacre/TINA [12, 13] and UPPAAL [15] through *templates*. Now, we only extend the UPPAAL template (since the optimized method, Sect. 2, is only implementable in UTA) with EDF and HRRN schedulers. The UPPAAL template output is proven faithful to the semantics of GenoM3 [14, 15]. Therefore, we present briefly GenoM3 in this section, then explain some of the former's important behavioral and real-time aspects using an example of an automatically generated UPPAAL model of a GenoM3 *component*.

3.1 GenoM3:

GenoM3 [14, 23] is a component-based framework for specifying and implementing functional layer specifications. Fig. 4 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 (*e.g.* 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 *report* 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 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 piece of C or C++ code is executed. Each codel specifies a WCET (worst case execution time) 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*).

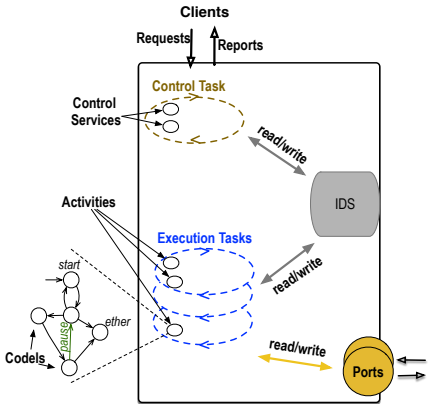

Fig. 4: A generic GenoM3 component

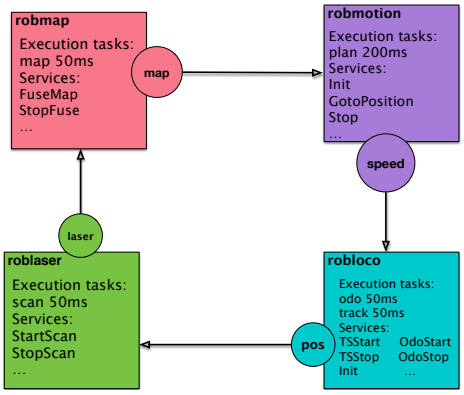

Fig. 5: The RobNav application

IDS, ports & concurrency: At ths OS level, tasks are parallel threads, with fine-grain concurrent access to the IDS and the ports: a codel (in its activity, run by a task) locks only the IDS field(s) and/or port(s) required for its execution (simultaneous readings are allowed). A codel *in conflict* (cannot execute at the same time) with another codel 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. This aspect renders generalizing results on optimal schedulers very difficult in the context of robotics, as referred to in Sect 1.

Case study: In this paper, we consider a variation of the RobNav application developed by fellow researchers at LAAS-CNRS (Fig. 5, technical details in [13]). The GenoM3 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), we refer the interested reader to [14].

3.2 UPPAAL Template

We show in Fig. 6 a very simplified version of the automatically generated UPPAAL model of the periodic execution tasks odo and track (ROBLOCO component, one time unit in the model is equal to 1 ms). This model follows the implementation model shown in [15], proven faithful to the semantics of GenoM3 [14, 15]. The *urgency* process is to enforce $\wr$ transitions through the urgent channel *exe* (UPPAAL supports $\wr$ transitions only, not $\wr$ edges). 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 $c \rightarrow c\_exec$ edges ensures the codel executes (or loses some resources) *as soon as* it has the required resources. The invariants on locations c_exec and the guards on the clock on the edges of the form $c\_exec \rightarrow$ reflect the fact that a codel is executed in a non-zero time that cannot exceed its WCET. For thread-safe codels, c_exec locations are not needed, and the invariant is thus associated with c locations.

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.

3.3 Extending with Schedulers

We show how to extend the UPPAAL template with cooperative EDF and HRRN schedulers. First, we use the case study to exemplify on how to adapt the solution shown in Sect. 2 to efficiently and correctly integrate such schedulers. Then, we automatize such integration 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)

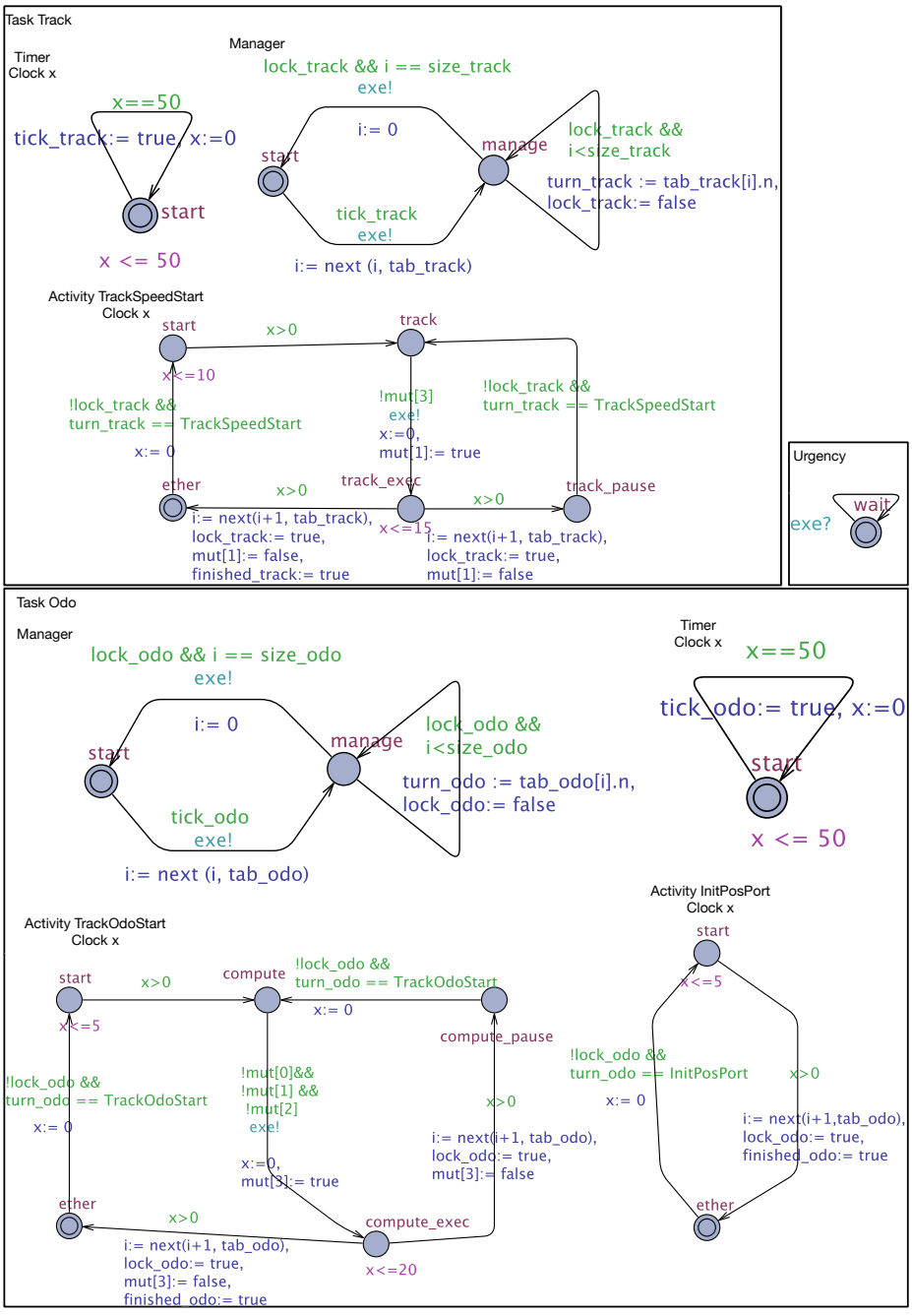

Fig. 6: UPPAAL model of tasks odo and track (automatically generated)

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. 7) has three locations: $start$ (initial), $update$ and $give$. The last two are *committed*, which (i) prevents interleaving with other interactions in the system and (ii) enforces urgency on all their outgoing edges (time cannot elapse).

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 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.

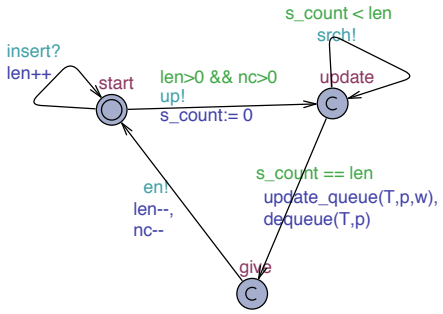

Fig. 7: UPPAAL model of the scheduler

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 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. 8), we have a clock x and four intermediate locations: ask, $search$, $decide$, and $error$. To meet the upper-bound condition (Sect. 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 start by checking whether *error* is reachable in any manager in the model, in which case we drop the analysis and increase the number of cores.
The remaining aspects are rather trivial considering the scheduler model and the search technique in Sect. 2 (we reuse the variable names for search bounds, u and d, from Fig. 3): $p[i]$ 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

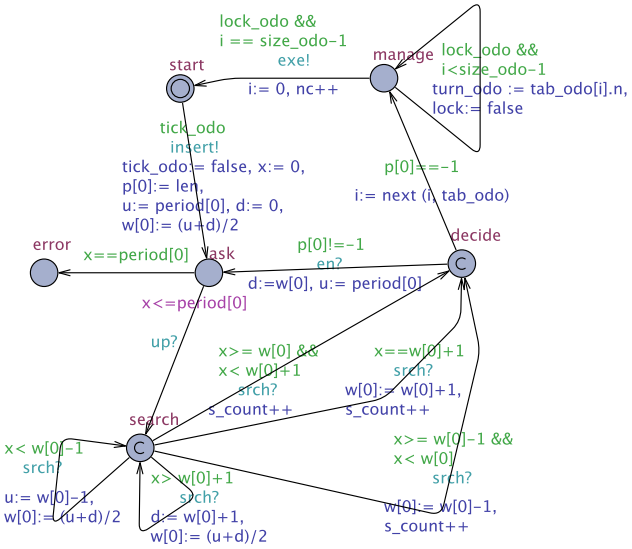

Fig. 8: UPPAAL model of the *odo* manager (enriched)

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 $start$).

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 (HRRN (1) or EDF (2)) 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 cooperative EDF 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, the listing below shows the piece of the template that generates the $update\_queue()$ function. The interpreter evaluates what is enclosed in <' '> in Tcl and outputs the rest as is. Line 1 conditions generating the function with the validity of the option passed by the programmer and lines 8-9 generates the right dynamic-priority formula according to the specified scheduler in the option. In the case of EDF, we simply subtract the waiting time $w[i]$ from the (relative) deadline, fixed to the period $period[i]$. For HRRN, 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 [12] for the 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 the priority formula by it. Since lcm_p is strictly positive, the inequality sign is not affected.

```
1  <'if {$argv >= 10} {'>
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 <'}'>
```

4 Results

We aim to analyze the deployability of the case study (Sect. 3.1) on the *Robotnik Summit-XL platform* [1], featuring an embedded four-core PC running Linux. There are two requirements, which we are unable to guarantee using classical FCFS and SJF schedulers from [12]. The track task is *hard real-time* (R1): it must always finish executing its activities within its period (new computed speeds must be sent to the controller at a fixed rate of 20 Hz). 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). R1 is a typical *schedulability* property, whereas R2 is a *bounded response* property. UPPAAL models extended with EDF and HRRN are automatically generated from the case study. The results presented below are identical for both schedulers.

Schedulability: To check the schedulability of a task t, we first make sure such a task never waits for a duration equal to its period before starting to execute its activities, that is location $error$ of its *manager* is unreachable:

```
A[] not manager_t.error
```

This *safety* property does not guarantee schedulability, but its falsification allows to quickly invalidate R1 (and is generally unacceptable for any task). Thus, we start by one core ($nc = 1$) and increase as soon as the safety property is violated for any task. We stop when nc is equal to four, the number of cores on the platform. The results show that as soon as nc reaches three, the property is satisfied for all tasks.

At this point, we fix nc to three and verify R1 (task track). The reasoning is as follows. A task is busy (waiting or executing activities) as long as its *manager* is not at location $start$ (we verify beforehand that locations ask and $manage$ are reachable in all managers). Thus, we check whether no new signal from the $timer$ is sent while the *manager* is not at location $start$:

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

This safety property is violated for $nc = 3$, which means R1 is dissatisfied, which is no longer the case as soon as we increase nc to four. We fix thus nc to four and pursue the verification for the remaining tasks in order to assess R2.

Bounded response: Now, for each task t that is not schedulable, we ask for the maximum value of clock x at location $manage$, at which activities are executed:
$sup\{manager\_t.manage\} : manager\_t.x$
Then, we simply subtract the period of t from the result to get exc_t, the maximum amount of time by which t exceeds its period.

The results for both schedulability and bounded response are given in table 1. All tasks are feasible, besides scan (component roblaser) that may exceed its period by up to 20 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 period.

t	odo	track	plan	fuse	scan
$schedulabe$	Yes	Yes	Yes	Yes	No
exc_t	/	/	/	/	20

Table 1: Verification results (four cores).

4.1 Discussion

The results are encouraging: (i) schedulability is verified for all tasks and (ii) if schedulability is violated, the precise upper bound of the time the period is exceeded is retrieved. 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. As expected, the search technique used to capture waiting times scales much better than the classical counting one: with the former, verification results are obtained within around 80 s for each property with less than 1 Gb of RAM usage, while with the latter no answer is given after several minutes and 4 Gb. However, we do not know whether we can obtain better results (*e.g.* schedulability of all tasks or shorter exceeding time) with preemptive schedulers. Indeed, we may not rely on generic theoretical results to know whether preemptive schedulers may perform better than cooperative ones in this case, and, unfortunately, preemption do generally not scale with model checking (Sect. 5). Possible directions to deal with this issue are given in Sect. 6.

5 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 [21, 25, 30] 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 [16, 17, 28], mainly focusing on schedulability, is non automatic, gives no guarantees on other important properties and is hard to extend to verify specific temporal constraints (such as bounded response). Moreover, theoretical results on optimized schedulers are hard to generalize to the case of robotics due to the complexity of multitasking models. For instance, the experiments in [26] show how, contrary to generic theoretical results, some non preemptive schedulers perform better than preemptive ones in the case of a mobile robot application.

Model-checking for schedulability: Using model-checking-based techniques to verify schedulability has been studied in the past, producing tools such as TIMES [5]. 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. 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 [19], where the authors use *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.

Comparison to our previous work: In [12], we extended the Fiacre template with FCFS and SJF cooperative schedulers. We concluded that we would need to integrate more "intelligent" schedulers with dynamic priorities, 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 EDF or HRRN, on which various properties can be verified within the same framework, UPPAAL. The results enable deploying the case study on a four-core platform.

6 Conclusion

In this paper, we elaborate an effort to bring the robotics, the real-time systems and the formal methods communities together. We aim at providing, automatically, formal models of robotic specifications that take into account the actual hardware and OS specificities. In order to consider optimized (dynamic-priority) schedulers, we propose a scalable *search* method that we automatize within the UPPAAL template developed in [15]. The obtained results are encouraging, and allow to deploy the case study on a four-core robotic platform while fulfilling real-time requirements. This work gives also insights on the use of formalisms in practice. For instance, we favor TA-based to TPN models for this particular problem, where it was the other way around in [11].

A possible direction of future work is considering preemptive schedulers. Indeed, those may further improve the deployability, but do unfortunately not scale with model checking. We are exploring the extension of the UPPAAL-SMC (*Statistical Model Checking*) template [15] with preemptive schedulers in order to verify the properties up to some high probability. In that regard, works like [10] may help us deal with the lack of probabilistic requirements in the robotics domain (what could be considered as a "sufficiently high probability" for a robotic application?). Another direction is to integrate more low-level specificities, such as cache interferences (modeled using UPPAAL in [27]), in our models as to gain a higher confidence in the verification results.

References

1. Robotnik summit-xl data sheet. https://www.robotnik.eu/web/wp-content/uploads//2019/03/Robotnik_DATASHEET_SUMMIT-XL-HL_EN-web-1.pdf.
2. R. Alur. Timed automata. In *CAV*, pages 8–22. Springer, 1999.
3. R. Alur, C. Courcoubetis, and D. Dill. Model-checking in dense real-time. *Information and computation*, 104(1):2–34, 1993.
4. R. Alur and D. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.
5. T. Amnell, E. Fersman, L. Mokrushin, P. Pettersson, and W. Yi. TIMES: a tool for schedulability analysis and code generation of real-time systems. In *FORMATS*, pages 60–72. Springer, 2003.
6. É. André, L. Fribourg, U. Kühne, and R. Soulat. Imitator 2.5: A tool for analyzing robustness in scheduling problems. In *FM*, pages 33–36. Springer, 2012.

7. G. Behrmann, A. David, and K.G. Larsen. A tutorial on uppaal. In *Formal Methods for the Design of Real-Time Systems*, pages 200–236. 2004.
8. B. Berthomieu, P-O. Ribet, and F. Vernadat. The tool TINA–construction of abstract state spaces for Petri nets and time Petri nets. *Jour. of prod. research*, 42(14):2741–2756, 2004.
9. S. Bornot, J. Sifakis, and S. Tripakis. Modeling urgency in timed systems. In *ISC: the significant difference*, pages 103–129. 1998.
10. J. Díaz, D. García, K. Kim, C-G. Lee, L. Bello, J. López, S. Min, and O. Mirabella. Stochastic analysis of periodic real-time systems. In *RTSS*, pages 289–300. IEEE, 2002.
11. M. Foughali. Toward a correct-and-scalable verification of concurrent robotic systems: Insights on formalisms and tools. In *ACSD*, pages 29–38, 2017.
12. M. Foughali, B. Berthomieu, S. Dal Zilio, P.-E. Hladik, F. Ingrand, and A. Mallet. Formal verification of complex robotic systems on resource-constrained platforms. In *FormaliSE*, pages 2–9, 2018.
13. M. Foughali, B. Berthomieu, S. Dal Zilio, F. Ingrand, and A. Mallet. Model checking real-time properties on the functional layer of autonomous robots. In *ICFEM*, pages 383–399. Springer, 2016.
14. M. Foughali, S. Dal Zilio, and F. Ingrand. On the Semantics of the GenoM3 Framework. Technical report, 2019.
15. M. Foughali, F. Ingrand, and C. Seceleanu. Statistical model checking of complex robotic systems. In *SPIN*, 2019.
16. N. Gobillot, F. Guet, D. Doose, C. Grand, C. Lesire, and L. Santinelli. Measurement-based real-time analysis of robotic software architectures. In *IROS*, pages 3306–3311. IEEE, 2016.
17. S. Goddard, J. Huang, S. Farritor, et al. A performance and schedulability analysis of an autonomous mobile robot. In *ECRTS*, pages 239–248. IEEE, 2005.
18. T. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic model checking for real-time systems. *Information and computation*, 111(2):193–244, 1994.
19. X. Huang, A. Singh, and S. Smolka. Using integer clocks to verify clock-synchronization protocols. *Innovations in Systems and Software Engineering*, 7(2):119–130, 2011.
20. M. Kargahi and A. Movaghar. Non-preemptive earliest-deadline-first scheduling policy: A performance study. In *MASCOTS*, pages 201–208. IEEE, 2005.
21. M. Kim and K. Kang. Formal Construction and Verification of Home Service Robots: A Case Study. In *ATVA*, pages 429–443. Springer, 2005.
22. D. Lime, O. Roux, C. Seidner, and L-M. Traonouez. Romeo: A parametric model-checker for Petri nets with stopwatches. In *TACAS*, pages 54–57. Springer, 2009.
23. A. Mallet, C. Pasteur, M. Herrb, S. Lemaignan, and F. Ingrand. Genom3: Building middleware-independent robotic components. In *ICRA*, pages 4627–4632. IEEE, 2010.
24. P. Merlin and D. Farber. Recoverability of Communication Protocols: Implications of a Theoretical Study. *IEEE Transactions on Communications*, 24(9):1036–1043, 1976.
25. A. Miyazawa, P. Ribeiro, W. Li, A. Cavalcanti, and J. Timmis. Automatic property checking of robotic applications. In *IROS*, pages 3869–3876. IEEE, 2017.
26. M. Piaggio, A. Sgorbissa, and R. 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.
27. N. Sensfelder, J. Brunel, and C. Pagetti. Modeling cache coherence to expose interference. In *ECRTS*, 2019.
28. J. Shi, S. Goddard, A. Lal, and S. Farritor. A real-time model for the robotic highway safety marker system. In *RTAS*, pages 331–340. IEEE, 2004.
29. C. Shih, L. Sha, and J. Liu. Scheduling tasks with variable deadlines. In *RTAS*, pages 120–122. IEEE, 2001.
30. A. Sowmya, D. Tsz-Wang So, and W. Hung Tang. Design of a Mobile Robot Controller using Esterel Tools. *Electronic Notes in Theoretical Computer Science*, 65(5):3–10, 2002.