A Two-Step Hybrid Approach for Verifying Real-Time Robotic Systems

Mohammed Foughali

A Two-Step Hybrid Approach for Verifying Real-Time Robotic Systems

Mohammed Foughali[1]

***Abstract*—Due to the severe consequences of their possible failure, robotic systems must be rigorously verified against (i) behavioral properties, such as safety and (ii) real-time properties, such as schedulability, while taking into account the real hardware (*e.g.* number of cores) and operating system (*e.g.* scheduling policy) specificities. Formal verification and schedulability analysis are popular approaches that may help with such verification, but suffer from limitations such as scalability issues (for the former) and difficulty to generalize to complex robotic tasks (for the latter), when used independently. In this paper, we propose a two-step, efficient solution that combines both approaches. The first step provides a sufficient condition for the schedulability of hard-real-time tasks in a robotic application. Then, the second step automatically generates a formal model of the application, on which other important properties may be verified formally using statistical model checking. The solution is applied to an autonomous drone case study.**

I. Introduction

A. Context & Addressed Problem

Robotic software, inherently complex, is majoritarily *component-based* (*e.g.* ROS [1], Orocos [2] and GenoM3 [3]): a number of *functional components* collaborate while interacting with the hardware. Each functional component implements complex algorithms, often organized in tasks, to perform some computations. Computations results are communicated between components to close the perception-action loop and fulfill the robot's missions.

With many robotic applications being time-critical and running on embedded platforms, robotic tasks must satisfy real-time properties, *e.g.* schedulability, while sharing limited resources. In addition, it is important that the robot behaves safely and correctly, *i.e.* that behavioral properties, such as safety and liveness, are verified. Considering the complexity of robotic components and their communication, verifying real-time and behavioral properties is a particularly hard research problem as explained hereafter.

Scenario-based testing, often used by roboticists, is unfortunately unreliable (Pecheur [4] gives an example of a full-year test failing to detect a bug in a NASA experiment). We must thus rely on more rigorous approaches such as *formal methods* and *schedulability analysis*. Formal methods are mathematically sound and can deal with both behavioral and real-time properties, but their use in robotics is impeded by scalability issues. Indeed, if the formal technique is exhaustive (*e.g.* model checking), the *state-space explosion* problem is observed in real-world robotic systems, *i.e.* their state spaces are intractable because of their sheer complexity. On the other hand, if the formal technique is non exhaustive, such as Statistical Model Checking (SMC) [5], the properties can no longer be evaluated with certainty, but with some probability, which is not sufficient in critical missions (*e.g.* if a task in a component is *hard real-time (HRT)*, its schedulability must be verified with certainty). Finally, the literature on formal verification of robotics ignores hardware and OS constraints which restricts the results validity (Sect. V). Likewise, the applicability of schedulability analysis to robotic systems is limited. First, its theoretical results are hardly generalizable to robotic tasks because the latter models are much more complex than the task models used in the real-time systems literature (Sect. II). Second, schedulability analysis leaves other important properties such as liveness and safety unattended.

[1]VERIMAG, Université Grenoble Alpes, CNRS, Grenoble, France
mohammed.foughali@univ-grenoble-alpes.fr

B. Proposition & Overall Contribution

We propose a novel two-step verification approach that combines formal methods and schedulability analysis, where neither of the two is sufficient alone (Sect. I-A). Our approach enables verifying both real-time and behavioral properties while taking into account the actual specificities of the robotic platform (mainly the number of cores and a scheduling policy). Furthermore, we provide a high level of automation, which makes our approach suitable for robotic programmers with no particular knowledge in formal methods or schedulability analysis.

Step one focuses on guaranteeing schedulability with certainty. We develop a schedulability test for HRT robotic tasks, which belong to a (mixed-) critical application, under a fixed-priority (FP) preemptive policy. If the original application, or a modified version achievable by *e.g.* modifying tasks deadlines, together with the number of cores on the robotic platform satisfy this test, then schedulability of HRT tasks is guaranteed. This will be the basis of step two, where we verify, up to a high probability, other important properties less crucial than schedulability. Such verification is done with SMC on formal models that we automatically generate from the robotic application, the number of cores and the FP scheduler (altogether proven to satisfy schedulability for HRT tasks in step one). Our approach is applied to a real autonomous drone system, developed using the robotic framework GenoM3, and the verification in step two is carried out using the formal framework UPPAAL-SMC [6].

C. Outline

The rest of this paper is organized as follows. In Sect. II, we present the verification problem: we (i) introduce robotic task models, and exemplify through presenting

978-1-7281-4403-0/20/$31.00 ©2020 IEEE

GenoM3 and the autonomous drone case study, then (ii) give examples of crucial properties in robotics, and analyze the problems preventing their verification with formal methods or schedulability analysis independently. In Sect. III, we detail our approach. Sect. IV shows and discusses the results of applying our approach to the drone case study. Finally, we compare our work to the state-of-the-art in Sect. V and conclude with possible axes of future work (Sect. VI).

II. The Verification Problem

A. Robotic Software Specificities

A robotic software, which we call a *system*, is made of communicating components (Sect. I-A). To account for timining constraints, a component encapsulates *periodic tasks*, in charge of its complex algorithms. The latter are organized within *services*. Because services are heavy and share memory resources, they are broken into small *pieces of code*, each attached to a state in a *Finite-State Machine (FSM)*, hence the popularity of FSMs in robotics. Thus, there are four "levels" in a system (from the lowest to the highest): pieces of code, services (FSMs), tasks and components.

Though not unanimous in robotics, the above organization is used by most component-based robotic frameworks designed for real-time applications, with subtle differences (*e.g.* while MAUVE [7] and Orocos[2] confound components with tasks, *i.e.* a component is a task, GenoM3 preserves both levels). Note that, since there is no standard terminology for most levels, the one we use is that of GenoM3.

1) GenoM3*:* GenoM3 [3] is the robotic framework used in this paper. We provide a generic informal description of GenoM3 with a focus on concurrency and real-time aspects. A more formal example using timed-automata is given in Sect. III-A. Note that this description is simplified for readability and to remain in the scope of this paper (*e.g. control tasks* and *aperiodic tasks* are excluded).

a) Organization: The organization of a component is shown in Fig. 1, where we can see the three component "levels" described above. Pieces of code are called *codels*. Each codel, attached to a state of a service FSM, has a Worst Case Execution Time WCET. By abuse of terminology, FSM states are simply called codels. Each task t, featuring a period, is in charge of a set of services S_t. We say that each service $s \in S_t$ is **a** service of t, and t is **the** task of s (s cannot belong to any $S_{t'}$ with $t' \neq t$). To perform their computations, codels share the *Internal Data Structure IDS* of the component. Finally, *ports* are used to communicate with other components, and are thus accessible by the codels in all components that use them.

b) Behavior: We briefly explain how a component evolves in a top-down fashion (from tasks to codels), following the *scheduler-agnostic* semantics developed in [8].

The component is piloted by an external *client* that *requests* services. Each task t, at each period, executes only the services previously requested by the client (among services in S_t) sequentially. When a service finishes executing, the task informs the client by sending a *report*. While communication between clients and tasks is taken into account in our verification (Sect. III), it is abstracted in the rest of this paper for readability and simplicity.

Each service FSM has at least two codels: `start` (at which the first execution begins) and `ether` (the termination point). A service execution ends when either (i) codel `ether` is reached (service is terminated) or another codel `c` is reached after taking a *pause transition*, *i.e.* a transition labeled *pause* (see the abstract FSM in Fig. 1), we say then the service is *paused* and refer to `c` as a *pause codel*. In the latter case, the service is resumed, at the next period of its task, starting from `c`.

c) Concurrency: Tasks (in a system), each of which executes its requested services sequentially (see previous paragraph), are run as parallel threads (assuming enough cores are available). To maximize parallelism, access to shared memory is handled at the codels level: memory resources (ports or fields of the IDS) that a codel needs for its execution are statically defined, so two codels *in conflict* (using at least a same port or a same IDS fragment) may not execute in parallel (simultaneous readings are allowed). Thus, while executing its requested services, a task needs to wait when one of such services reaches a codel in conflict with another codel, in another service being executed by another task concurrently. Following this low-level concurrency model, a codel may be either *thread safe (TS)* (not in conflict with any codel) or *thread unsafe (TU)* (otherwise). Because of ports sharing, codels in conflict may belong to different components (example in Sect. III-A).

d) Specification & Templates: While we content with graphical illustrations of GenoM3 systems, the latter are actually specified textually. Each component is written in a *dotgen* (*.gen*) file, in which tasks, services and codels are specified. A system may be then built by #-including the dotgens of the different components in another dotgen.

Templates transform GenoM3 (dotgen) specifications into Tool Command Language (Tcl) structures for automatic generation purposes. The robotic programmer can access all information in the dotgen (*e.g.* tasks periods, FSM and codels WCET), manipulate them and generate a text file in any format accordingly. We have used this mechanism in previous work to automatically generate formal models [9]. In Sect. III, we give examples of templates developed to automatize the two-step approach presented in this paper.

e) Case Study: To validate our approach, we use the Quadcopter case study from LAAS-CNRS. Fig. 1 shows its GenoM3 organization in which some names are modified for simplicity. The system contains five components collaborating to achieve autonomous aerial navigation. We give a high-level description (in terms of components and ports) on how these components collaborate (the interested reader may refer to [9] for more details on each component).

Component MIKROKOPTER processes data from the *Inertial Measurement Unit (IMU)* and the propellers sensors and

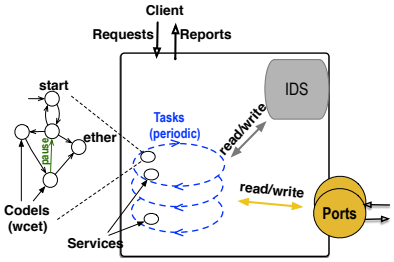

Fig. 1: A generic GenoM3 component.

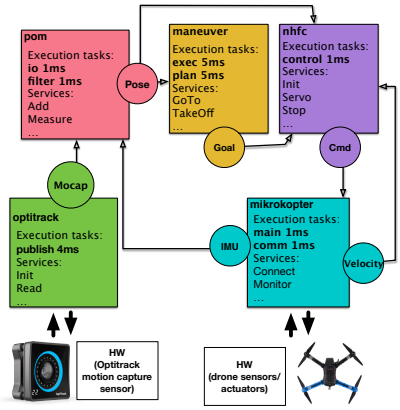

Fig. 2: The quadcopter case study (only some services are listed).

uses them to write the current IMU and velocity to ports **IMU** and **Velocity**, respectively. Component OPTITRACK processes data from the Optitrack motion capture system and writes it to port **Mocap**. Component POM reads the IMU and captured position from, respectively, ports **IMU** (MIKROKOPTER) and **Mocap** (OPTITRACK), to which it applies an *Unscented Kalman Filter UKF* to compute the estimated position of the drone that it writes to port **Pose**. Such position is fed to (i) MANEUVER, which uses it to compute an intermediary goal position that it writes to port **Goal** and (ii) NHFC, which uses it, together with the current **Velocity** (from MIKROKOPTER), to compute and update, in port **Cmd**, the velocity to reach the intermediary goal position (from **Goal** in MANEUVER). Finally, the perception-action loop closes as MIKROKOPTER reads the updated velocity in **Cmd** (NHFC) and applies it to the drone propellers.

B. The Verification Challenge

If the drone software above fails, the drone may crash, inducing economic costs and/or human injuries. We give examples of crucial properties that must be verified to avoid such failure, and explain why their verification is particularly challenging.

1) Properties of Interest: The drone system has three critical components: MIKROKOPTER, NHFC and POM. That is, tasks in these components are HRT: each must always finish executing within its period, otherwise the drone may crash. It follows that the schedulability property must be proven always true for these tasks, for all possible scenarios.

In the remaining components (less critical), tasks are allowed to miss their deadlines. However, it is still important to verify that they are *e.g.* exempt of starvation, that is being, at some point, delayed forever by critical tasks monopolizing resources. For *e.g.* tasks in MANEUVER, such starvation would make the drone hover forever without fulfilling its mission (as it may not navigate to a final goal position).

2) Difficulties: Now, in order to verify these properties, using model checking (or SMC) or schedulability analysis independently proved insufficient in robotics in general and on this drone system in particular.

a) With model checking / SMC: Model checking does generally not scale with complex robotic applications. For instance, we show in [8] that, although it performs well on the *stationary flight application* (*i.e.* component MANEUVER is excluded), model checking with state-of-the-art tools fails to scale on the *navigation application* involving all the five components (Fig. 2), with eight tasks and over 20 services broken into more than 80 codels. In the same work [8], we use SMC to verify properties up to a high probability. Though SMC provides better guarantees than scenario-based testing, it is not suitable for the schedulability property of HRT tasks which must be proven with certainty.

Another problem of model checking (and generally formal verification) in robotics is that extending formal models with scheduling algorithms usually penalizes their scalability because of (i) preemption and/or (ii) the necessity to create large models to handle schedulers [10], [11]. For the drone navigation application, the integration of schedulers in formal models (which already do not scale as explained above) produces new formal models that still do not scale, even when preemption is not allowed.

b) With schedulability analysis: From a real-time analysis point of view, we focus on three levels in GenoM3 (and generally robotic) systems: the tasks level, the services level and the codels level (component are abstracted as tasks map to cores). Robotic tasks models are thus more complex than those usually considered in real-time analysis: a robotic task executes a sequence of jobs (codels) at each period rather than one job whose WCET is known. A particular problem is the computation of the WCET of tasks, which is practically intractable. Indeed, besides the fact that a TU codel (Sect. II-A1c) may remain infinitely blocked waiting for resources (robotic frameworks do not guarantee the absence of starvation), the sequence of codels to execute in services by a task may differ from a period to another depending on *e.g.* which services are requested (Sect. II-A1b). Another problem is, even if such sequences WCET are somehow obtained, theoretical results of schedulability analysis in the literature are still unusable because the preemption model in robotics is also different (more in Sect. III-A). Finally, schedulability analysis provides no guarantees on other properties excluding schedulability.

III. A Two-Step Hybrid Approach

Our approach combines both formal verification, by means of SMC, and schedulability analysis to achieve scalable rigorous verification of crucial properties in robotics. We divide properties into two types: *Type I* covers properties that must be verified with certainty (schedulability of all HRT tasks), while *Type II* comprises properties that may be verified with a high probability (*e.g.* absence of starvation in less critical tasks). On that basis, a key idea is the following. Since model checking does not scale, then we may use SMC for *Type II* properties, but only once properties of *Type I* are verified with certainty. Thus, we first check whether we can guarantee properties of *Type I* using schedulability analysis. This is the first step of our approach, which takes into account the actual number of cores on the robotic platform and a scheduling policy (Sect. III-A). If step one is conclusive, an UPPAAL-SMC model of the considered application, number of cores and scheduler (already proven to satisfy properties of *Type I* in step one) is generated. On such formal model, we verify by means of SMC properties of *Type II*, which concludes the second step of our approach (Sect. III-A). If step one is inconclusive, we give a possible direction on how to pass it (Sect. IV-C).

A. Step One: Schedulability Analysis

Our approach is based on Response Time Analysis (RTA). First, we compute tasks WCETs, taking into account delays caused by mutual exclusion over memory (Sect. III-A1). Then, we compute tasks Worst Case Response time (WCRT) considering the concurrency over cores (Sect. III-A2).

1) Computing tasks WCET: In the following, we explain more where the difficulty of computing task WCET (Sect. II-B2) comes from, using the UPPAAL formal model of the GenoM3 task main (component MIKROKOPTER) of the drone (Fig. 2) shown in Fig.3. This model, automatically generated, is proven correct w.r.t GenoM3 semantics [12], [8]. The model is simplified for readability purposes.

Each timed automata (TA) in UPPAAL, made of *locations* and *edges* connecting them, and possibly having a *clock x*, is called a *process*. Time *invariants* (in purple) may be associated with locations, and edges may have *guards* (in green) and *operations* (in blue). Processes are arranged to fit with the "layers" vision given in Sect. II-A1: the task layer, composed of processes *timer* and *manager*, the services layer, where each underlying GenoM3 service FSM is mapped to an UPPAAL process, and the codels layer, where codels are locations in services processes. Fig. 3 shows that task main has two services: *Init* and *Apply*.

Shared variables and functions are used by processes to communicate. Array *tab_t* holds the names and "statuses" of all services of task *t*. Each of its cells contains two fields: *n*, a service name and *st*, the service status that may be either *R* (requested by a client) or *V* (for "Void", otherwise). The timer of *t* gives at exactly each period a signal, through variable *tick_t*, to the manager to start execution, by taking the edge *start* → *manage*. The operation of such edge

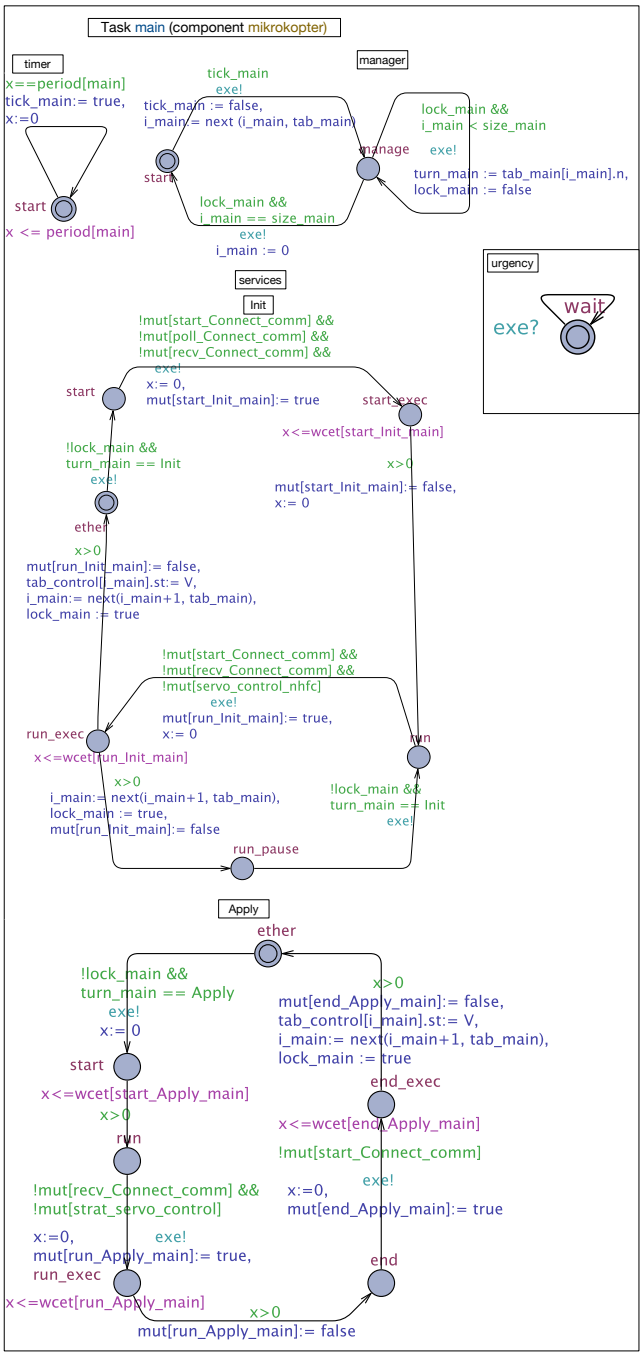

Fig. 3: Simplified UPPAAL model of task main in component MIKROKOPTER. Process *Urgency* does not belong to any component and is added to enforce urgencies, *i.e.* prevent unnecessary lazy waits (the receiver (?) edge is always ready).

searches, through function *next*(), for the index of the next requested service in *tab_t* (having status *R*) starting at index *i_t* (initially 0) and stores the result in *i_t* (the size of *tab_t* if such service does not exist). At location *manage*, the manager executes the requested services sequentially: variables *lock_t* and *turn_t* are used to pass the control to the next service to execute (computed previously through function

[1] In this case, there is an additional operation: *V* is assigned to the service status in *tab_t* to prevent *t* from executing it again at its next period.

next()). When such service finishes execution, either by terminating[1] (*e.g.* edge *end_exec* → *ether* in service *Apply*) or pausing (*e.g.* edge *run_exec* → *run_pause* in service *Init*), it computes the index of the next service to execute and gives the control back to the manager. And so, the control passes back and forth between the manager and the requested services until each of the latter has executed once (detected when *next*() hits the bottom of *tab_t*), so the manager transits back to *start* and awaits the next period.

Now, at the codels level, a codel c in a service *s* is represented either by one location *c* (if it is TS) or two locations *c* and *c_exec* (otherwise), plus a location *c_pause* if such codel is targeted by a pause transition in the underlying GenoM3 specification. The WCET of c is represented with an invariant $x \leq wcet[c\_s\_t]$ on location *c* (*c_exec* if c is TU) where *wcet* is an array of all codels WCET indexed with unique identifiers. The array of Booleans *mut* is used to handle concurrency: it tracks the execution of TU codels in the system. Therefore, guards on edges *c* → *c_exec* ensure *c* does not start executing unless no codel in conflict with *c* is currently executing, witnessed by the falseness of the corresponding fields in *mut*. For instance, codel run of service *Apply* is in conflict with codel recv (in service *Connect*, executed by the other task comm in MIKROKOPTER), and codel start (in service *servo* of task control in component NHFC), which explains the guard on the edge *run* → *run_exec* in process *Apply*. If such guard is true, codel run starts executing by taking *run* → *run_exec* through which it turns its own field in *mut* to true to prevent, in turn, codels in conflict with it to execute.

This example shows the complexity of GenoM3 (and generally robotic) tasks. From a real-time analysis perspective, we identify two problems. First, the WCET of a sequence of codels (which a task executes) is possibly infinite, because we do not know beforehand how long a TU codel needs to wait to secure the memory resources it needs. Second, even if we bound such waiting times, it is practically infeasible to compute by hand the WCETs of all possible sequences: for instance, summing the WCETs of all codels in all services in a task (assuming we bound and include waiting times in TU codels WCETs) would be a naive solution (such sum would be a coarse overapproximation that will likely prevent finding a feasible schedule). We propose a solution for both problems by, respectively (i) an implementation to bound waiting times for TU codels and (ii) an algorithm to compute the WCET of a task by traversing all possible codels sequences. We explain how solution can be automated.

a) Bounding TU codels WCET: We propose an implementation to enable computing a bound B_c on the waiting time (to acquire memory resources, *i.e.* IDS or ports) of any TU codel c. Then, we get the actual WCET of c by summing its WCET (from the GenoM3 specification) with B_c.

The implementation is inspired from multiprocessor memory-sharing protocols. Brandenburg [13] reviewed a number of such protocols, mainly categorized into *spin-based* (*busy-wait*) and *suspend-based*, and pointed out that the former are easier to implement and perform better than the latter when durations of critical sections are short. As we explained in Sect. II, FSM in robotics are designed to reduce the times of locking shared resources, which makes spin-based protocols suitable to our case. Actually, the previous reasoning fits with the reality of spinlocks being widely used in robotics (*e.g.* ROS and GenoM3 systems). In this paper, we use the MSRP protocol [14]. We benefit from the judicious remark made by Brandenburg [13] as he clears the confusion that the protocol name may create: in MSRP, the "SRP" part refers to the priority-ceiling-based protocol SRP [15], which is actually used to handle local resources only on each core/processor, and thus this part of the protocol is not relevant to our case (as all shared resources *e.g.* ports are global). In contrast, the "M" part deals with global resources (and this is exactly what we will use to bound waiting times): a TU codel c appends itself to a FIFO and its thread is spinlocked until c gets access to shared memory, and spinlocked threads are non preemptible. TS codels are not concerned as they are in conflict with no other codel in the system (Sect. II-A1c). The direct disadvantage of this approach is that all TU codels compete for the shared memory as a whole, which reduces the overall parallelism of the system (it is possible for a TU codel c to be blocked by another TU codel c' in the FIFO even though c and c' are not in mutual conflict). While we deem this loss in parallelism a fair price to pay for bounding waiting times, we are investigating new FIFO-spinlock algorithms to restore it (see future work in Sect. VI).

Let us compute B_c of a TU codel *c* in a service *s* in a task *t*. We assume there are *n* tasks and *m* cores ($m < n$). In worst case scenarios, the thread trying to execute *c* spinlocks after already $m-1$ threads are in the spinlock FIFO (for accessing shared memory). Since each thread corresponds to a GenoM3 task that (i) is sequential and (ii) spinlocks only when trying to execute a TU codel, the first $m-1$ entries of the FIFO are occupied by TU codels each in a distinct GenoM3 task, different than *t*. In the worst case, each t' of the $m-1$ tasks already spinlocked is trying to execute TU codel c' with the largest WCET among the TU codels of all services in t'. Thus, B_c is upper-bounded by the sum of the WCET of codels c'. To get that sum, we proceed as follows. (1) For each task $t' \neq t$, we find, within all its services, the largest WCET of all TU codels. (2) We sort, in a decreasing order, the values found in (1). (3) B_c is equal to the sum of the first $m-1$ values sorted in (2).

Once B_c computed, we sum it with $WCET_c$ to get the *actual* WCET of *c* (including the maximum time it may wait for memory access). To make codels actual WCETs computations accessible to robotic programmers, we make use of the template mechanism (Sect. II-A1d). We give in listing 1 an example that performs steps (1) and (2) of the algorithm above, then writes (to a file) the list output by (2) for any TU codel in any service in task *t*. The template generator evaluates everything enclosed in <' '> (resp. <" ">) in Tcl without output (resp. and outputs the result),

```
1   <' set wcets_max [list] '>
2   <' foreach comp [dotgen components] { '>
3   <'   foreach t_prime [$comp tasks] { '>
4   <'   if {t_prime == t} {continue}
5   <'   set max_wcet 0 '>
6   <'      foreach s [$t_prime services] { '>
7   <'        foreach c [$s codels] { '>
8   <'        if {[llength [$c mutex]] && ([$c wcet] >
        $max_wcet)} {'>
9   <'        set max_wcet [$c wcet]} '>
10  <'        } '>
11  <'      } '>
12  <'   lappend wcets_max $max_wcet '>
13  <'   } '>
14  <' } '>The list is <"[lreverse [lsort wcets_max]]">
```

Listing 1: Generating largest WCET of TU codels per task.

and outputs the rest as is. Line 4 excludes task t, and line 8 conditions considering codel c only when it is TU through the non-emptiness of the field [$c *mutex*], a ready-to-use list containing all codels in conflict with c. The last line writes to a file the list after sorting it in a decreasing order.

Thus, at the end of these computations, we have the actual WCET of all codels, which we call simply WCET in the rest of this paper (that is, the WCET provided by GenoM3 if c is TS, summed with B_c if c is TU). Our approach to compute B_c is generic, and may thus be pessimistic in some cases. For instance, if the scheduler is partitioned, some of the $m-1$ largest elements of *wcets_max* (listing 1) may belong to tasks allocated to the same core as t, and thus B_c is overestimated. However, this generecity brings a valuable advantage. Indeed, since the computation is affinity-independent, the roboticist performs this step only once and, if some HRT tasks do not pass the schedulability test (Sect. III-A2), may try to find a better affinity by reallocating tasks based on the timing constraints already computed (the affinity does not affect such constraints). This is explained further in Sect. IV.

b) Deducing tasks WCET: We call each possible (full) codel sequence executed by task t a *hyperjob*. The largest WCET of all hyperjobs in t is then simply the WCET of t.

Therefore, to compute the WCET of t, we proceed as follows. (1) For each service s in task t, we sum the WCETs of codels involved in each possible path (starting either at codel `start` or some pause codel, and ending either at `ether` or some pause codel). (2) We find, for each s, the value of the largest among the sums computed in (1). (3) We sum the values found in (2). (4) we repeat (1), (2), (3) for all tasks in the GenoM3 system. Thus, this algorithm will give the maximum time to execute the longest possible path in all services in t, which corresponds to the largest WCET of all possible hyperjobs in t (*i.e.* the WCET of t).

The above algorithm being classical in model checking, the idea is to benefit from the already existing UPPAAL template [8] to achieve it. Yet, we know that the overall UPPAAL model of this application does not scale. The good news is, however, we do not need to consider the system as a whole: since WCETs are now known for all codels, we may adapt services processes of task t to allow computing the maximum time of their possible paths (step (1) above) without considering the rest of the system.

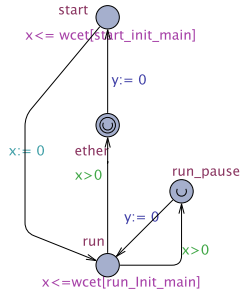

Fig. 4: Modified process of *Apply* for WCET task computation.

First, locations *c_exec* are no longer needed: location c is enough, the invariant bound of which is the WCET of codel c (Sect. III-A1a). That being done, interactions of each service with services outside t cease to exist (since bounds B_c are now included in TU codels WCET, all guards and operations involving the *mut* array are removed). Then, we (i) make all *ether* and *c_pause* locations *urgent* (time cannot progress at them) and add, to each service process of t, a clock y reset to 0 at all edges leaving *ether* or *c_pause* locations. This way, y tracks the time of each possible path from location *start* (or any *c_pause* location) to location *ether* (or any *c_pause* location). We have thus what we need for step (1) of the algorithm above, and may remove all the remaining non-clock guards and operations in the services of t. It follows that there are no more interactions between any service process in t and the rest of the system, which means we can verify each service separately.

Fig. 4 shows the result of these changes to the UPPAAL process of service *Apply* (Fig. 3). Now, all we need to do is ask UPPAAL for the maximum value of clock y at location *ether* and each location *c_pause* using the UPPAAL query pattern $sup\{p.l\} : p.y$ (with p being the process name and l the location name), store the results and repeat the operation for each service in task t, which corresponds to step (1) of the algorithm above. Then, we perform (2) and (3), then repeat the whole process for all other tasks (step (4)) to get the WCET of all tasks in the GenoM3 system.

2) Analysis: Once tasks WCET computed, we compute their WCRT for RTA analysis. We recall that schedulability tests from the literature are not applicable to robotic tasks even when they take memory-sharing into account. For instance, standard task and scheduling models assume a task executes only one job at each release. This means that, if we use available tests, we should treat each hyperjob as one job and, since such hyperjob is likely to include a TU codel, make it non preemptible (Sect. III-A1a). Consequently, we will most likely end up with a set of non-preemptible tasks, which renders preemptive scheduling useless.

Thus, we need to perform schedulability analysis based on the model in Fig. 5: each hyperjob may be preempted at the end of each codel. The reason for this is rather straightforward: in robotics, elementary pieces of code (codels in

GenoM3) are designed by roboticists as the smallest pieces (of the algorithm they belong to) that must be performed with no intermediary perturbations. TU codels present another feature that consolidates the rationale of codels non-interruptibility: their interruption may compromise their memory-dependent computations.

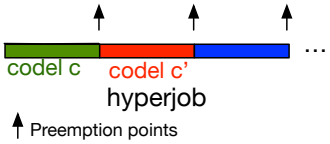

Fig. 5: hyperjob model.

a) Scheduling assumptions: In this paper, we use a partitioned fixed-priority FP scheduler. There are two main reasons. First, partitioned FP is very popular in domains related to robotics, such as automotive systems (*e.g.* in AUTOSAR [16]), since it removes the cost of task migration. Second, global schedulers are known to perform poorly compared to partitioned ones, even though this might result from over-pessimism of tests in global approaches [17].

The task set of the robotic system is the union of HRT and less critical tasks $\tau = \tau_h \cup \tau_l$. There are two priorities: pr_h (rep. pr_l), the high (resp. low) priority, assigned to all tasks in τ_h (resp. τ_l). The platform features m cores $C_1 ... C_m$ ($m < |\tau|$). Let $Aff_i \subseteq \tau$ be the affinity of core C_i, that is the set of tasks allocated to it. Then, $Aff_i = Aff_{il} \cup Aff_{ih}$ where $Aff_{il} = \tau_l \cap Aff_i$ (resp. $Aff_{ih} = \tau_h \cap Aff_i$) is the set of low (resp. high) priority tasks allocated to C_i. Since the algorithm is partitioned, each task is statically allocated to only one core, that is $\forall i,j \in 1..m, i \neq j : Aff_i \cap Aff_j = \emptyset$. The size of the queue of C_i is equal to the cardinality of its affinity $|Aff_i|$.

b) Computing tasks WCRT: Following the model in Fig. 5, a task is a set of hyperjobs $t = \{hj_1 ... hj_{|t|}\}$. A hyperjob is an ordered set of codels $hj = \{c_1 ... c_{|hj|}\}$. If a codel c belongs to a hyperjob in t, we may say simply that c belongs to t. When needed, we use the superscript $(^t)$ to denote that a hyperjob or a codel belongs to task t, and the double subscript jk to denote that a codel c_j belongs to hyperjob hj_k. At each period P_t, one (depending on the evolution of the system) of hyperjobs in t is to be executed. The WCRT of t defines an upper bound on the time separating the moment a_i, at which a hyperjob hj_i is activated (arrives in a core queue), and the moment f_i, at which it finishes its execution and frees the core, that is $WCRT_t = max_{i \in 1..|t|}(f_i - a_i)$ (eq. 1). Let $r_i \in [a_i, f_i)$ be the moment hj_i is *released*, that is a core is given to it and it starts to execute (f_i is excluded because hj_i cannot execute in a zero time). By inserting r_i in eq. 1 we get $WCRT_t = max_{i \in 1..|t|}(f_i - r_i + r_i - a_i)$ which we may upper bound $WCRT_t \leq (max_{i \in 1..|t|}(f_i - r_i) + max_{i \in 1..n}(r_i - a_i))$ (ineq. 2). Now, we know that the left-hand operand of the right-hand side of ineq. 2 is the WCET of t which we already computed in Sect. III-A1. We call the remaining operand the *Worst case Waiting Time* $WWT_t = max_{i \in 1..|t|}(r_i - a_i)$ (eq. 3). Therefore $WCRT_t \leq WWT_t + WCET_t$ (ineq. 4).

A hyperjob hj of a high-priority task t allocated to core C_i ($t \in Aff_{ih}$) worst position in the prioritized queue of C_i is equal to $|Aff_{ih}|$. The worst waiting time of hj corresponds to this very position (hyperjobs of tasks in Aff_{ih}, having the same priority pr_h as t, are already in the queue, so hj has to wait for them to finish). Now, in this worst situation, the worst case is when the hyperjob at the head of the queue cannot start immediately as a low-priority task hyperjob hj' is still not preempted (we recall that preemption points are set at the end of each codel, Fig. 5). It follows that the worst waiting time for hj is equal to the sum of the WCET of all $|Aff_{ih}| - 1$ hyperjobs (each belonging to a task $t' \in Aff_{ih} \backslash \{t\}$) in the queue plus the WCET of the codel of hj' being currently executed. We maximize such worst waiting time for all hyperjobs in t to get WWT_t (see eq. 3). To account for the waiting needed for high-priority hyperjobs, we maximise the WCET of all hyperjobs in each task $t' \in Aff_{ih} \backslash \{t\}$ and sum them (1). Then, we add to the value obtained in (1) the waiting for preemption by maximizing the WCET of codels in low-priority tasks $t'' \in Aff_{il}$ (2). (1) is simply the sum of the WCET of tasks $t' \in Aff_{ih} \backslash \{t\}$ and in (2) we add the WCET of the longest codel in tasks $t'' \in Aff_{il}$, which gives us the following bound for any task t allocated to core C_i:

$$WWT_t \leq \Sigma_{t' \in Aff_{ih} \backslash \{t\}} WCET_{t'} + max^{t'' \in Aff_{il}}_{\substack{l \in 1..|t''| \\ k \in 1..|hj_l^{t''}|}} (WCET_{c_{kl}^{t''}}) \quad (ineq.\ 5)$$

We sum WWT_t with $WCET_t$ to upper-bound $WCRT_t$ (see ineq. 4). Finally, we state the schedulability test for HRT tasks $\forall t \in \tau_h : WCRT_t \leq P_t$ (ineq. 6).

While pessimistic, this test is sufficient: if the maximum time a task t needs from its activation to its end is inferior than its deadline (period), then t is schedulable. We trade off optimism for *sustainability*: Burns and Baruah [18] show that RTA-based FP schedulability tests are sustainable in the sense that they remain valid even if some tasks manage to execute in less than their WCET.

B. Step Two: Formal Verification

If all HRT tasks in the GenoM3 system pass the schedulability test in step one, we may verify other - less critical - properties using SMC, otherwise we should redesign the system (Sect. IV). We automatize the generation of UPPAAL-SMC models by extending the template presented in [8].

First, we make sure that the WCET computations, made with the help of UPPAAL (Sect. III-A), still hold in UPPAAL-SMC models. This is a simple proof. As shown in [8], the only difference between UPPAAL-SMC and UPPAAL models is at the level of services, where non-deterministic edges may have custom probabilities. To give an example, let us get back to Fig 3. In process *Init*, there are two edges out of location *run_exec*. In UPPAAL, these edges are equiprobable (chances to take one or the other are equal). In UPPAAL-SMC, one may use custom probabilities (that sum to one) on such edges, a mechanism which we exploited in [8] to insert experiment-based probabilities. Now, w.r.t the computations made in Sect. III-A, this difference has no

```
1   <' for {set k 1} {$k <= [length $Aff]} {incr k} { '>
2   <' if {[$t name] in [lindex $Aff $k]} { '>
3   <' set index $k '>
4   <' break} '>
5   process manager ...
6   ...
7   start -> ask {guard tick_<"[$t name]">; sync
        insert_<"$index">; assign tick_<"[$t name]">:= false;},
8   ...
```

Listing 2: Generating an edge for manager of task t.

impact since, for HRT tasks, we need to explore all paths anyway, no matter how big or small is the probability to take each of them. Second, we need to integrate the FP scheduler in the UPPAAL-SMC model and automatize it in the template. We show how this is done hereafter.

a) Integrating the FP scheduler: We add a process *C_i* to handle the FP scheduling for each core C_i on the robotic platform. Fig. 6 shows a generic example of such process, and how the managers processes are modified accordingly (we show only one manager). *C_i* has three locations: *idle* (the core is idle), *decide* (the core queue is dequeued) and *busy* (the core is being used). On the managers side (of each task in Aff_i), an intermediary location *ask* (the task is activated but still waiting to get the core) is added.

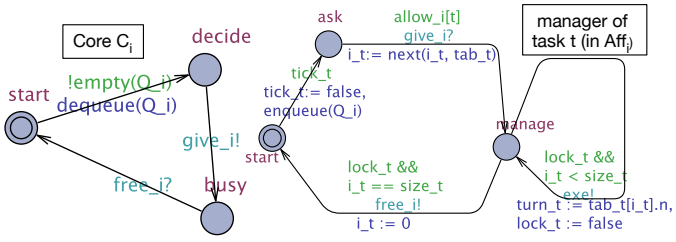

Fig. 6: Integrating scheduler model

C_i and the managers communicate as follows. The manager inserts (function *enqueue*()) the name of *t* in *Q_i*, the prioritized queue of *C_i*. On the edge *idle* → *decide*, *C_i* dequeues *Q_i*, and, depending on the dequeued value, updates the shared array of initially false Booleans *allow_i*, indexed with the names of tasks in Aff_i (if the dequeued value is *t*, it turns *allow_i*[*t*] to true). At location *decide*, C_i synchronizes with the manager of the only task *t* such that *allow_i*[*t*] = *true* (computed in the previous step), through the urgent channel *give_i*, which allows such manager to move to location *manage* (at which it starts executing a hyperjob) and moves *C_i* to location *busy*. *C_i* will then wait till the manager finishes executing the hyperjob and releases the core, which is done through the urgent channel *free_i*. Finally, the model handles also preemption, but this is not shown in this paper for space and readability reasons.

b) Automatization: The programmer provides the affinity of cores $Aff = \{Aff_1..Aff_m\}$ as a Tcl list of lists. Listing. 2 shows how we generate the edge *start* → *ask*, in the manager process of task *t* in the system, in the UPPAAL *.xta* textual format. The *for* loop (lines 1-4) finds the *i* subscript of core C_i to which *t* is allocated. Then, in line 7, the edge *start* → *ask*, with the right subscript for channel *insert_i* (in the synchronization block, keyword "sync"), is generated.

IV. Results

We apply our approach to verify important properties on the drone navigation under a partitioned FP policy and the number of cores on the drone platform. The latter embeds an ODROID-C0 card featuring a four-core processor.

A. Step One

We comply with the notation given in Sect. III-A: $m = 4$ (number of cores), $\tau_h = \{main, comm, io, filter, control\}$ (the set of HRT, *i.e.* high-priority tasks, those of the critical components MIKROKOPTER, POM and NHFC), $\tau_l = \{publish, plan, exec\}$ (the set of low-priority tasks, those of components OPTITRACK and MANEUVER). Following the steps given in Sect. III-A1, we compute the actual WCET of all TU codels in the system and update such WCET accordingly, then compute the WCET of the five HRT tasks in the system (table. I). For each of the remaining three tasks, we identify the codel having the largest WCET (table. II). We recall tasks periods in table III.

HRT task	WCET (ms)
main	0.51
comm	0.47
io	0.68
filter	0.55
control	0.52

TABLE I: WCET of HRT tasks.

Task	WCET of longest codel (ms)
publish	0.3
plan	0.4
exec	0.4

TABLE II: Longest-codel WCET in low-priority tasks.

Task	Period (ms)	Task	Period (ms)
main	1	control	1
comm	1	publish	4
io	1	plan	5
io	1	exec	5

TABLE III: Tasks periods.

Core	Affinity
C_1	{*main*, *comm*}
C_2	{*io*, *plan*}
C_3	{*filter*, *publish*}
C_4	{*control*, *exec*}

TABLE IV: Initial affinity.

HRT task	WCRT (ms)
main	0.98
comm	0.98
io	1.08
filter	0.85
control	0.92

TABLE V: WCRT of HRT tasks considering the initial affinity (table IV).

An issue that arises is how to allocate tasks to cores. It stems from the bin-packing problem, known to be NP-hard. In this paper, the way we allocate tasks is inspired by the *first-fit decreasing* heuristic. We start by allocating *m* high-priority tasks (in τ_h) to the *m* cores, then repeat until all tasks in τ_h are allocated. Then, we do the same for low-priority tasks (thus allocation is by *decreasing* priority). The *first-fit* part is left to after running the schedulability test on HRT tasks (if such test fails). This allocation is not exactly what the original heuristic does, but in our case, it intuitively tends to reduce the WCRT of HRT tasks in the application. Indeed, such WCRT increases with the number of HRT tasks allocated to the core (ineq. 5), and so allocating first HRT tasks minimizes the maximum number of HRT

tasks allocated to a core C_i, upper-bounded by $\lceil \tau_h / m \rceil$. The (decreasing) affinity we start with is given in table IV.

We are now ready for schedulability analysis: we apply ineq. 5 (using the values from table. I and table. II) for each task t to upper bound $WCRT_t$, then compare the latter with the period P_t from table III (ineq. 6). The results (table V), show that all HRT tasks pass the schedulability test except for task io, whose WCRT is 80 μs larger than its period.

At this point, we may try to change the affinity without modifying the decreasing pattern (no more than two HRT tasks per core). Here, the generecity of the approach in Sect. III-A1a allows us to reason only using the timing constraints from tables I and II, which remain valid regardless of the chosen affinity. We notice that, by permuting the allocation of low-priority tasks publish and plan, all HRT tasks pass the schedulability test (table. VI). This new affinity guarantees schedulability for all HRT tasks in the system (table. VII) and will be thus the basis of step two.

Core	Affinity
C_1	{*main*, *comm*}
C_2	{*io*, *publish*}
C_3	{*filter*, *plan*}
C_4	{*control*, *exec*}

TABLE VI: New affinity (by permuting tasks in blue in the initial affinity in table IV).

HRT task	WCRT (ms)
main	0.98
comm	0.98
io	0.98
filter	0.95
control	0.92

TABLE VII: WCRT of HRT tasks considering the new affinity (table VI).

B. Step Two

We generate, from the affinity in table VI, the number of cores and the GenoM3 system, an UPPAAL-SMC model. In the latter, schedulability for HRT tasks (Sect. IV-B), is guaranteed by construction (step one).

Now, using UPPAAL-SMC, we guarantee, up to a high probability, that low-priority tasks never starve, a less critical, yet important property (Sect. II-B1). To do so, we reason as follows. We know that in any task manager (Fig. 4), location *manage* denotes that a hyperjob is being executed. Thus, the absence of starvation means that (i) location *manage* is reachable and (ii) whenever it is reached, location *manage* is eventually left (back to location *start*). (i) is a reachability property while (ii) is a *leadsto* property which UPPAAL-SMC does not support. This is a limitation of the tool and not intrinsic to SMC.

Fortunately, there is a simple workaround if we augment the manager model (Fig. 4) with a clock x that is reset to 0 whenever any location is left. Thus, if the value of x is upper-bounded, then *manage* (i) is reachable (otherwise x would be unbounded at *start* or *ask*) and (ii) eventually left (otherwise x would be unbounded at *manage*), which corresponds to the same (i) and (ii) above. We may thus transform the two-step reachability-leadsto property into a safety property as we query the UPPAAL-SMC verifier to estimate the probability of x being bounded by value *max_x*, *e.g.* for task plan: $Pr[\leq b]([]\, manager\_plan.x \leq max\_x)$ (with b being a time bound for SMC simulations). We call $prob_t$ the probability of satisfying this property by a task t.

	$prob_t \in$			
t	$x_{max} = 4$	$x_{max} = 5$	$x_{max} = 6$	$x_{max} = 7$
optitrack	[0, 0.004]	[0, 0.004]	[0.996, 1]	[0.996, 1]
plan	[0, 0.004]	[0, 0.004]	[0.996, 1]	[0.996, 1]
exec	[0, 0.004]	[0, 0.004]	[0, 0.004]	[0.996, 1]

TABLE VIII: Verification results (step two).

We set the statistical parameters to a high confidence ($\alpha = 0.02$) and precision ($\epsilon = 0.002$), which means that the highest probability we can obtain for $prob_t$ is $99.8\% \pm \epsilon$ *i.e.* $prob_t \in [0.996, 1]$ with a confidence $100\% - \alpha = 98\%$. For each task t, we set max_x to P_t and raise it until such highest probability is reached. Table VIII gives the results for all low-priority tasks. In sum, all low-priority tasks are starvation-free with a 99.8% probability as soon as $x\_max = 7$ ms. UPPAAL-SMC takes up to 25 minutes to verify each property, a value that grows exponentially if we try to tighten the precision further: with ϵ tending toward zero SMC tends toward classical model checking and thus scalability is threatened as we noticed in [8].

C. Discussion

We prove, with certainty, the schedulability for all HRT tasks in the application, while proposing a scheduling policy on the drone platform. Also, we prove with a high probability that low-priority tasks never starve for cores. Thus, considering the real robotic platform and the affinity and scheduling algorithm we propose, the GenoM3 system of the drone guarantees the latter does not crash because HRT constraints are not met, and is highly likely to fulfill its navigation missions (Sect. II-B1).

However, it is possible, for other applications, that step one (Sect. III-A) is not conclusive, that is we fail to find an affinity that allows all HRT tasks to pass the schedulability tests. In this case, we may consider redesigning the application by *e.g.* changing the periods, which is however not always feasible because periods may be dictated by hardware constraints (*e.g.* sensor frequency).

V. Related Work

A. Rigorous Verification of Real-Time Robotic Applications

Popular robotic environments like ROS [1] are not suitable for real-time applications [19]. This is one consequence of the hard research problem of verifying time-constrained robotic systems. Schedulability analysis and formal methods have been explored to tackle this problem.

One of the main issues hindering the use of schedulability analysis is the generalization of tests to robotic task models [7]. Some robotic software initiatives try to tackle this issue [2], [20], [7]. In particular, MAUVE [7] supports specification, implementation and analysis of real-time constraints. However, such initiatives focus on schedulability analysis and thus leave important properties such as reachability and bounded response unattended.

On the other hand, a major challenge of using formal verification is bridging robotic software, not formally

founded, with formal methods. Proposed solutions range from ad-hoc non-reusable formalization [21], [22] to formal frameworks for robotics [23]. Another difficulty is the lack of scalability of exhaustive verification techniques due to the complexity and size of robotic systems. Non-exhaustive techniques such as SMC, used in [24], are not suitable for critical applications where schedulability of HRT tasks must be verified with certainty. Besides, to the best of our knowledge and except for our efforts (Sect. V-B), the literature on formal verification in robotics (including works cited here) ignores hardware and OS scheduling constraints, which restricts the results validity to the unrealistic assumption of all tasks running in parallel at all times.

B. Our previous work

In [25], [11], [10] we proposed automated support to verify various properties of robotic applications under different scheduling policies by means of model checking. Such support is not suitable for the drone navigation application because of scalability issues. In [8], we proposed an automated approach based on SMC to verify, up to a high probability, a number of properties. This approach is not suitable either for the drone system because SMC guarantees are not enough for critical properties such as the schedulability of HRT tasks.

C. Comparison to related work

In this paper, we combine both schedulability analysis and formal verification to achieve a rigorous verification of complex robotic systems. Unlike related work, we cover both schedulability and other important properties, while taking into account hardware and OS constraints, on robotic systems that do not scale with model checking. Furthermore, we provide an automated support to enable using our approach with no prior knowledge of either schedulability analysis or formal verification.

VI. Conclusion

We describe an automated two-step approach to rigorously verify complex (mixed-) critical robotic applications. It combines schedulability analysis and formal verification and is suitable for robotic applications that do not scale with model checking. Our approach is automated for non-expert users and validated on a real drone case study.

We give two examples of future work directions. First, as said in Sect. IV, UPPAAL-SMC does not support *leadsto* properties and nesting operators in general. Though we often find alternatives to formulate the properties we want to verify, this limitation may be disabling for non experts. We are investigating strengthening our approach with automated modeling in BIP-SMC [26], supporting full Bounded LTL and Weighted MTL logics with nesting. Second, the "global memory" FIFO spinlock that we borrow from MSRP leads to a loss in parallelism (Sect. III-A1a). We are working on a new shared-memory-locking algorithm inspired from *group locks* [27], where locking overheads need to be carefully considered.

References

[1] M. Quigley, B. Gerkey, K. Conley, J. Faust, T. Foote, J. Leibs, E. Berger, R. Wheeler, and A. Ng, "ROS: an open-source Robot Operating System," in *ICRA workshop on open source software*, 2009, p. 5.

[2] P. Soetens and H. Bruyninckx, "Realtime hybrid task-based control for robots and machine tools," in *ICRA*. IEEE, 2005, pp. 259–264.

[3] A. Mallet, C. Pasteur, M. Herrb, S. Lemaignan, and F. Ingrand, "Genom3: Building middleware-independent robotic components," in *ICRA*. IEEE, 2010, pp. 4627–4632.

[4] C. Pecheur, "Verification and validation of autonomy software at NASA," NASA Ames Research Center, Tech. Rep., 2000.

[5] A. Legay, B. Delahaye, and S. Bensalem, "Statistical model checking: An overview," in *RV*. Springer, 2010, pp. 122–135.

[6] A. David, K. G. Larsen, A. Legay, M. Mikučionis, and D. B. Poulsen, "Uppaal SMC tutorial," *STTT*, vol. 17, no. 4, pp. 397–415, 2015.

[7] N. Gobillot, C. Lesire, and D. Doose, "A design and analysis methodology for component-based real-time architectures of autonomous systems," *Journal of Intelligent & Robotic Systems*, vol. 96, pp. 123–138, 2019.

[8] M. Foughali, F. Ingrand, and C. Seceleanu, "Statistical model checking of complex robotic systems," in *SPIN*. Springer, 2019, pp. 114–134.

[9] M. Foughali, "Formal verification of the functional layer of robotic and autonomous systems," *PhD Thesis, INSA Toulouse*, 2018.

[10] M. Foughali and P.-E. Hladik, "Bridging the gap between formal verification and schedulability analysis: The case of robotics," *Journal of Systems Architecture*, vol. 101, pp. 817–830, 2020.

[11] M. Foughali, "On reconciling schedulability analysis and model checking in robotics," in *MEDI*. Springer, 2019, pp. 32–48.

[12] M. Foughali, S. Dal Zilio, and F. Ingrand, "On the Semantics of the GenoM3 Framework," LAAS/CNRS, Tech. Rep. 19036, 2019.

[13] B. B. Brandenburg, "Scheduling and locking in multiprocessor real-time operating systems," *PhD Thesis, University of North Carolina at Chapel Hill*, 2011.

[14] P. Gai, M. Di Natale, G. Lipari, A. Ferrari, C. Gabellini, and P. Marceca, "A comparison of MPCP and MSRP when sharing resources in the Janus multiple-processor on a chip platform," in *RTCSA*. IEEE, 2003, pp. 189–198.

[15] T. P. Baker, "Stack-based scheduling of realtime processes," *Real-Time Systems*, vol. 3, no. 1, pp. 67–99, 1991.

[16] A. Wieder and B. B. Brandenburg, "On spin locks in autosar: Blocking analysis of FIFO, unordered, and priority-ordered spin locks," in *RTSS*. IEEE, 2013, pp. 45–56.

[17] G. Gracioli, A. A. Fröhlich, R. Pellizzoni, and S. Fischmeister, "Implementation and evaluation of global and partitioned scheduling in a real-time OS," *Real-Time Systems*, vol. 49, no. 6, pp. 669–714, 2013.

[18] A. Burns and S. Baruah, "Sustainability in real-time scheduling," *Jour. of Computing Science and Engineering*, vol. 2, no. 1, pp. 74–97, 2008.

[19] Y. Maruyama, S. Kato, and T. Azumi, "Exploring the performance of ROS2," in *EMSOFT*, 2016, pp. 1–10.

[20] C. Schlegel, A. Steck, D. Brugali, and A. Knoll, "Design abstraction and processes in robotics: From code-driven to model-driven engineering," in *SIMPAR*. Springer, 2010, pp. 324–335.

[21] M. Kim and K. C. Kang, "Formal construction and verification of home service robots: A case study," in *ATVA*. Springer, 2005, pp. 429–443.

[22] L. Molnar and S. Veres, "System verification of autonomous underwater vehicles by model checking," in *OCEANS-EUROPE*. IEEE, 2009, pp. 1–10.

[23] A. Miyazawa, P. Ribeiro, W. Li, A. Cavalcanti, and J. Timmis, "Automatic property checking of robotic applications," in *IROS*. IEEE, 2017, pp. 3869–3876.

[24] M. Hazim, H. Qu, and S. Veres, "Testing, verification and improvements of timeliness in ROS processes," in *TAROS*, 2016, pp. 146–157.

[25] M. Foughali, "Toward a correct-and-scalable verification of concurrent robotic systems: insights on formalisms and tools," in *ACSD*. IEEE, 2017, pp. 29–38.

[26] B. Mediouni, A. Nouri, M. Bozga, M. Dellabani, A. Legay, and S. Bensalem, "SBIP 2.0: Statistical model checking of stochastic real-time systems," *ATVA*, p. 536.

[27] A. Block, H. Leontyev, B. B. Brandenburg, and J. H. Anderson, "A flexible real-time locking protocol for multiprocessors," in *RTCSA 2007*. IEEE, 2007, pp. 47–56.