Formal Verification of Complex Robotic Systems on Resource-Constrained Platforms

Mohammed Foughali, Bernard Berthomieu, Silvano Dal Zilio, Pierre-Emmanuel Hladik,
Félix Ingrand and Anthony Mallet
LAAS-CNRS, Université de Toulouse, CNRS, Toulouse, France
firstname.lastname@laas.fr

ABSTRACT

Software constitutes a major part of the development of robotic and autonomous systems and is critical to their successful deployment in our everyday life. Robotic software must thus run and perform as specified. Since most of these systems are used in a hard real-time context, the schedulability of their tasks is a crucial property. In this work, we propose to use formal methods to check whether the tasks of a robotic application are schedulable with respect to a given hardware platform. For this, we automatically translate functional components specified in GenoM into FIACRE, a formal language for timed systems. The generated models integrate realistic real-time schedulers based on the FCFS and the SJF cooperative policies. We use then the model-checker TINA to assert schedulability properties. We carry out experiments on a real robotic system, namely a quadcopter flight controller. We demonstrate that, on its actual hardware, schedulability properties can be formally expressed and verified. We give examples on how we can check other important behavioral and timed properties on the same synthesized models.

1 INTRODUCTION

Robotic software is complex. The reliability of such software is thus challenging to assess. While roboticists tend to rely on testing campaigns, simulators and following good coding practices, such techniques are inherently non-exhaustive. They are therefore insufficient given the increasing involvement of autonomous systems in human environments (e.g. robotic surgery, service robots) and costly missions (e.g. space exploration). Indeed, one should rather rely on formal proofs to guarantee that, for instance, no faulty behavior with catastrophic consequences would ever occur while the autonomous robot is fulfilling its missions. In a hard real-time application, such a failure may be caused, for example, by a periodic task missing its deadline (not satisfying the schedulability property).

Robots/Autonomous systems software is often organized through two *layers* (or *levels*), *decisional* and *functional* [3]. Formal methods are more commonly used on decisional components [11, 31, 34]. This is mainly due to the fact that decisional specifications are mostly model based with well-defined semantics.

In this paper, we focus only on the functional level. We propose a common ground between automatic verification and schedulability analysis. Our main contribution is enabling verification of both schedulability and other behavioral and timed properties while taking into account the actual constraints of the robotic platform, namely the number of processors/cores and the scheduling policy. Indeed, enriching the model so it includes these constraints gives a greater value to the verification results as the model is more realistic and does not ignore the hardware real specifications. For this, we automatically generate formal models that integrate the scheduling policies and the resources of the robotic hardware. This is done by bridging GenoM3 (Sec. 2.1), a tool to specify robotic functional components, with Fiacre (Sec. 2.2), a formal language for timed systems, using templates fed with information on the resources of the robotic hardawre: Fiacre models are automatically synthesized from GenoM3 specifications given the actual resources and two cooperative schedulers: (i) First-Come-First-Served (FCFS) and (ii) Shortest-Job-First (SJF) (Sec. 3). The model checker TINA is used on the synthesized Fiacre models to formally verify schedulability properties, and possibly other important real-time properties. A quadcopter real-world example is used as a case study on which we illustrate, discuss and compare verification results (Sec. 4).

*This work was supported in part by the EU CPSE Labs project funded by the H2020 program under grant agreement No 644400.

2 PRELIMINARIES

2.1 GenoM3

GenoM3 [22] is a component-based framework for designing and specifying the functional level of robotic applications. Functional components are in charge of functionalities with varying complexity (may range from simple low-level control *e.g.* velocity control to more elaborate computations such as simultaneous localization and mapping “SLAM" and navigation). These components are controlled by a supervisor, *aka* external client, that sends requests to the services (see Sect. 2.1.1 below) they want the components to deliver.

2.1.1 Components. Each component is organized as shown in Fig. 1. Hereafter, we briefly introduce the different constituents of a GenoM3 component:

Services : The *services* encapsulate the algorithms of the component. Services are associated with *requests* (with the same name). The service algorithm may require a *short* or a *long* computation time. *Short* services are known as *control services* and are executed by the *control task* (see below). *Long* services, known as *activities*, are executed by *execution tasks* (see below). Activities are modeled as *finite-state machines*, each state associated to a *codel*. A codel is a C or C++ function that specifies the possible *transitions* (codels to execute next) subsequent to its execution. It may also specify its *Worst Case Execution Time (WCET)*. We abuse notation so that the term *codel* may refer to either itself or the state it is associated to.

Control Task : A component always has a *control task* that processes *services requests* and sends *reports* (from/to external clients); it also executes control services and activates and interrupts activities.

Execution Task(s) : Aperiodic or periodic, they execute activities.

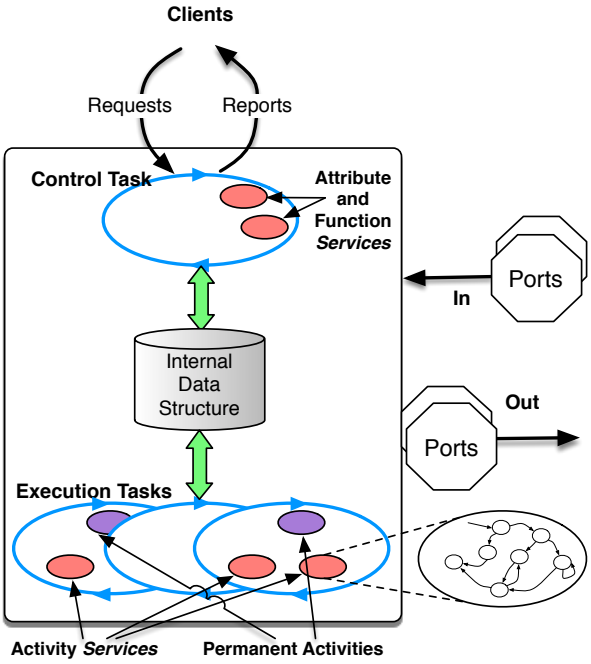

Figure 1: A generic GenoM3 component.

Internal Data Structure (IDS) : A local *internal data structure* containing the internal data of the component is shared by all services.

Ports : Store shared data with other components.

For the sake of readability, the description above is minimal. Indeed, the behavior of a GenoM3 component is dictated by complex mechanisms defining *e.g.* how, when, and how often activities are executed by their respective tasks. For more details, we refer the interested reader to [13] and [14].

2.1.2 Templates Approach. The GenoM3 template mechanism was initially introduced to cope with the *middleware independency* problem [22]: templates are used to generate the components, specified generically, for various middleware.

A GenoM3 template has access to all the input component(s) information (e.g. tasks and their periods, activities and their codels) and there is no restriction on what it can synthesize. For instance, the template code in listing 1 generates the output shown in listing 2 when called with the component MIKROKOPTER (Fig. 2) that has two execution tasks main and comm. The interpreter evaluates thus only what is enclosed in *markers* (<"> or <"">) in Tcl and reproduces the rest as is. This gives developers the freedom to write templates that output files in any desired language. A more practical example is given in Sect. 3.3.

There are already templates to synthesize: the component itself for mainly two middleware (PocoLibs[1] and ROS-Com [27]); client libraries to control the component (*e.g.* JSON, C, OpenPRS), etc.

2.1.3 Pocolibs vs ROS-Com. Between Pocolibs and ROS-Com, we choose the former for the implementation of our case study as it is more suitable for real-time applications. This is due to the fact that ROS-Com treats ports (Sect. 2.1.1) as *topics* while pocolibs implements them as a shared memory. A ROS topic has one or more *publishers* and *subscribers*. A publisher (respect. subscriber) writes (respect. reads) the topic. With the ROS-Com implementation, a publisher message is buffered then pushed into each subscriber queue by an internal ROS thread, additional to those created for each GenoM3 task in the application. This boils down to writing for each subscriber, which creates a load proportional to the number of subscribers. On the other hand, each port in Pocolibs is a data field properly protected to prevent simultaneous writings and allow simultaneous readings. A codel that reads/writes a port may thus do so whenever it is executed by its task in its own thread. The Pocolibs implementation is thus the better choice to guarantee real-time requirements as (i) no additional threads are required and (ii) the number of a port readers/writers has no side effect on jobs loads.

[1]https://git.openrobots.org/projects/pocolibs

```
The component <"[$component name]"> has <"[llength [$component
      tasks]]"> execution task(s):

<' foreach t [$component tasks] { '>

    Task <"[$t name]">

<' } '>
```

Listing 1: A simple template code for illustration.

```
The component mikrokopter has 2 execution task(s):

    Task main

    Task comm
```

Listing 2: Output of Listing 1 when called with MIKROKOPTER (Fig. 2).

2.1.4 Concurrency. During the execution of an activity in its task, a codel may need to access fields in the IDS. GenoM3 uses a mutual exclusion function that ensures a fine-grain locking of resources while tasks run concurrently. Only the fields of the IDS needed by a codel are locked while that codel is running, forcing those in conflict with it (needing the same fields) to wait (simultaneous readings are allowed). A codel presenting a conflict is called *non thread safe*. Non-thread-safe codels are subject to both *urgency* and *non determinism*. That is, if several codels are waiting for the same resources, one of them, non-deterministically, will start executing *as soon as* the resources are free. The control task may also need to access fields of the IDS to process requests or execute control services, which might make it also in conflict with some codels. In the Pocolibs implementation, access to ports is also mutually exclusive, which enables conflicts between codels belonging to different components.

This fine-grain locking guarantees a high level of concurrency (more details in [13]), but makes reasoning about vital properties, such as schedulability of periodic tasks and reactivity of control

tasks (more in Sect. 4), very difficult. For schedulability, for instance, using classical analytical methods is practically unfeasible, as it would be particularly tedious and error-prone to try to extract manually the worst case response time of a task in a complex robotic application where conflicts between numerous codels are very common.

2.2 Fiacre and TINA

2.2.1 The Fiacre Language. Fiacre [5] is a high-level, formal specification language designed to represent both the behavioral and timing aspects of reactive systems. Fiacre has been used in several projects pertaining to various domains such as robotics [14] and avionics [8].

A Fiacre *process* is the building unit of Fiacre descriptions. Processes describe state machines with timing aspects based on the *Strong Time Semantics* of Time Petri nets [23]. Processes can be composed into *components* that may themselves be hierarchically composed. Both processes and components may interact and/or communicate through *synchronization ports* and/or *shared variables*. Fiacre is a strongly typed language, meaning that type annotations are exploited in order to avoid unchecked run-time errors. In our examples, we will use FIFO queues and arrays, which are native datatypes in Fiacre. We also use *interval types*, denoted `n..m` (where n and m are integer constants with $n \leq m$), that define integer values in the range $[n, m]$. Fiacre provides more complex data types, such as tagged union and records and supports user-defined *functions*.

A Fiacre specification can also declare properties on the form of *patterns* [2]. Patterns provide a user-friendly way to express temporal and timed properties. For instance, Fiacre provides a pattern called `leadsto`; the property *a* `leadsto` *b* abbreviates the LTL expression $\Box\ (a \Rightarrow \Diamond b)$, which states that an event `b` (such as service termination) will always follow an event `a` (such as service start) in the future. Patterns can be extended with a *scope modifier*, like `within`, to express a timing constraint between the occurrence date of the two events. For example, the *timed* property `a leadsto b within [0,3[` translates to: `b` will always follow `a` in less than 3 units of time. Fiacre implements an observer process to encode such a timed property, reducing it to a reachability property. While this method may be costly as it implies a larger state space, it allows on-the-fly verification (the interested reader may find more about the *leadsto within* observer in [14]).

2.2.2 The TINA Toolbox. TINA [7], for TIme Petri Nets Analyzer, is a framework for specifying, analyzing and verifying Time Petri nets (TPN for short) and their extensions with *e.g.* data, inhibitor arcs and priorities. TPN state spaces are infinite due to the dense nature of time. Finite abstractions known as *State Classes* are available since [6]. The State classes construction, known as the *State Class Graph* (*SCG*), is suitable for LTL model-checking as it preserves markings and traces. The tools provided by TINA feature a variety of SCG constructions and support model checking for LTL and modal μ-calculus. On-the-fly verification is possible for some types of properties, *e.g.* safety properties. TINA model checkers produce a counterexample when a property is violated which can be analyzed for diagnosis purposes.

2.2.3 Verification of Fiacre descriptions. A compiler called `frac` is used to translate Fiacre descriptions into enriched TPN. It also translates the properties declared within a Fiacre description into the format supported by the TINA model checker specified as the compilation target. The verification is then performed within TINA using the compiled description and properties. The compilation process is fully automatic and transparent for the user.

3 SCHEDULERS

In this section, we show how we model the schedulers in Fiacre. We refer the reader to [14] and [13] for examples on how GenoM3 constituents (*e.g.* execution tasks, non-thread-safe codels) and mechanisms (*e.g.* concurrency, execution of activities) were mapped into Fiacre.

The schedulers are formally modeled in Fiacre. An advantage of this approach is that it enables model-checking of the schedulability property at the same time as other behavioral and timed properties. Another advantage is that, by taking into account the hardware resources (processors/cores) and integrating the scheduling policies, the verification results have a greater value as the real hardware specificities are considered.

We chose two multicore cooperative (non-preemptive) schedulers: FCFS and SJF where a task cannot be preempted when executing active activities. These schedulers are adapted to our Quadcopter hardware and easily implementable on a real-time OS (see Sect. 4). They are also suitable for model-checking as they prevent preemption. To ease the comprehension of the modeling details, we give in the sequel brief definitions of the *queues* in Fiacre and the primitives that operate on them.

3.1 Fiacre Queues

Fiacre provides the special type *queue* that allows the implementation of bounded First-In-First-Out (FIFO) queues. Hereafter the primitives that may operate on a bounded queue q:

- empty q: returns **true** if q is empty, **false** otherwise.
- full q: returns **true** if q is full, **false** otherwise.
- enqueue (q,e): returns a queue equal to q with the element e inserted at the back.
- append (q,e): returns a queue equal to q with the element e inserted at the front.
- dequeue q: returns a queue equal to q deprived of its front element.
- first q: returns the front element of q.

3.2 First-Come-First-Served (FCFS) Scheduler

The preemptive FCFS scheduling policy is based on serving jobs in the order of their arrival while allowing higher priority tasks to preempt lower priority ones. Here, we choose a cooperative version of FCFS (preemption is not allowed). For more details about the policy, both versions (preemptive and cooperative) are studied in [29]. To model the scheduler, three shared variables are introduced into the Fiacre model. The first one is a queue, named `fifo`, which represents the list of tasks identifiers ordered by tasks activation dates. Tasks identifiers, *i.e.* the elements of `fifo`, are positive integers ranging from 1 to N, where N is the number of tasks in the robotic specification. The second shared variable is an array

```
1  process scheduler (&fifo: queue N of 1..N, &launch: array 1..N of
       bool, &cores: 0..P) is
2  states start
3  from start
4    on (not empty fifo) and (cores > 0);
5    wait [0,0];
6    cores:= cores-1;
7    launch [first fifo]:= true;
8    fifo := dequeue fifo;
9    to start
```

Listing 3: Fiacre model of the FCFS scheduler

```
1  process Taskmanager_n (..., &tick_n: bool, &fifo: queue N of
       1..N, &launch: array 1..N of bool, &cores: 0..P) is
2  states ask, start, manage
3  from ask
4         wait [0,0];
5         on tick_n; tick_n:= false;
6         fifo:= enqueue(fifo, n); to start
7  from start
8         wait [0,0]; on launch[n];
9         lock_n:= 1; to manage
10 from manage
11        wait [0,0]; on lock_n=0;
12        cores:= cores+1; launch[n]:= false; to ask
```

Listing 4: Fiacre model of a task manager

```
1  process Timer_n (&tick_n: bool) is
2    states start
3    from start
4      wait [PERIOD,PERIOD];
5      tick_task_n := true;
6      to start
```

Listing 5: Fiacre model of a timer

of boolean values, named `launch`, that represents the signals to release tasks. Each task is statically associated to a specific index of this array, *i.e.* the task whose identifier is `i` can execute only when `launch[i]` is set to `true`. The last shared variable, `cores`, is an integer that ranges from 0 to P. P is the number of cores provided by the platform and is thus the initial value of `cores`.

Listing 3 gives an overview of the scheduler Fiacre model. It is a process parameterized with `fifo`, `launch` and `cores` (line 1). It has only one state called *start* (line 2). A self-loop transition is taken only when the `fifo` is not empty **and** there is at least one available core (line 4). Such a transition is then urgently (line 5) fired (line 9). The firing allows the first task in the queue to execute by assigning an available core (line 6), sending the right release signal through `launch` (line 7) and dequeuing the `fifo` (line 8). Note that, since the operation of line 7 depends on the value of *first fifo*, there are as many possible self-loop transitions as tasks in the underlying robotic specification, but the choice is deterministic. Note also that a more readable encoding of the scheduler is possible by using more explicit task identifiers. For instance, one may use a *union* type for task identifiers as task names instead of integers.

Now, we show the relation between the scheduler and periodic tasks. A periodic task is modeled by a `Taskmanager` and a `Timer`. Listing 4 shows the `Taskmanager` process for a generic task (periodic or not), with identifier `n` associated to the index `n` of the array `launch`. The process `Taskmanager` has three states (line 2): `ask`, `start` and `manage`. The transition from `ask` to `start` (lines 3-6) represents the activation of the task at the beginning of its period. The period signal is given by the `Timer` as it assigns *true* to the variable `tick_n` each *PERIOD* unit(s) of time (Listing 5). Upon activation, `tick_n` becomes *false* and the task identifier is inserted at the back of `fifo`. To prevent queue overflow, we set the size of `fifo` to the number of tasks in the robotic specification. State `start` (lines 7-9 in Listing 4) is used to wait on the release signal, that happens *as soon as* `launch[n]` becomes true (this urgency is expressed by the timing constraint `wait [0,0]`). In this transition, the variable `lock_n` is used to allow the codels of active activities (not presented here) to start executing. Finally, state `manage` remains the current state of the manager until the end of the execution, that is until variable `lock_n` is set to 0. As soon as that occurs, the task manager releases the core and transits back to `ask` (line 12) to wait for the next period signal.

3.3 Shortest-Job-First (SJF) Scheduler

The SJF scheduler, *aka* SPN (Shortest Process Next), is a cooperative scheduler based on priorities related to the jobs estimated execution time (*aka* burst time) [21]. That is, the insertion of a job in the waiting queue is based on its Estimated Execution Time (EET) rather than its activation date. For instance, for the jobs a and b with the respective EETs $t_a = 1$ and $t_b = 2$, there is no possible configuration where b is before a in the waiting queue (while such configuration is possible in FCFS). Jobs with equal EET will be sorted in a FIFO fashion, as in FCFS.

To encode such a scheduler, we need first to define the EET for a GenoM3 task. Normally, EETs are computed dynamically in SJF which renders its implementation on OS difficult. There exist methods that predict the EET of a job based on *e.g.* an average of its previous execution times. Here, we find it rather convenient to assign EETs statically. Indeed, the robotic programmer assigns periods to the tasks in their components according to the tasks estimated load. Accordingly, we will consider the period as the EET for a periodic execution task. For aperiodic tasks, including the control tasks, the programmers expect these to react and execute as promptly and quickly as possible. Their EET is thus considered shorter than any EET of a periodic task. Assigning static EETs will ease the implementation of SJF on the OS (Sect. 4).

The Fiacre encoding is therefore similar to that of FCFS 3.2 except that the insertion in the queue *fifo* is done through the Fiacre recursive function *insert_sjf* (Listing 6) rather than the classical *enqueue* primitive. This function ensures that the jobs are inserted according to their respective EET if different, and to their activation date otherwise.

The function *eet*, called within *insert_sjf*, returns for each task its EET. Listing 7 shows how we can generate such a function from a GenoM3 component c using a template function (Sect. 2.1.2). In line 1, the expression between markers will be replaced by the number of tasks in c (the +1 is for counting the control task as well, always present in a GenoM3 component). The statement **case ... of** (line 3) is similar to the **switch case** statement in the *C* language. The first

```
1  function insert_sjf (q: queue N of 1..N, t: 1..N) : queue N of
       1..N is
2  var temp: 1..N
3  begin
4          if (empty(q) or eet(t) < eet(first(q))) then
5                  return append(q,t)
6          end if;
7          temp:= first(q);
8          return append(insert_sjf (dequeue(q), t),temp)
9  end
```

Listing 6: Queue insertion function for the SJF scheduler

```
1  function eet (t: 1..<"[expr [llength [$c tasks]] + 1]">) : nat is
2  begin
3          case t of
4          1 →return 0
5  <'set k 2
6  foreach task [$c tasks] {
7          if {![catch {$task period}]} {'>
8          | <"$k"> →return <"[$task period]">
9  <'} else {'>
10         | <"$k"> →return 0
11 <'}
12 incr k}'>
13         end
14 end
```

Listing 7: Generating the function *eet* for a component *c*

clause of the **case ... of** statement (line 4) returns 0 for the control task, encoded by the integer 1. This ensures that the control task EET is always smaller than any EET of a periodic execution task. The same goes for aperiodic execution tasks (line 10). For periodic execution tasks, the function simply returns their periods (line 8).

4 EXPERIMENTS AND DISCUSSION

In this section, two Fiacre models are automatically generated from a real-world Quadcopter specification (Fig. 2): one integrates a cooperative FCFS scheduler and the other an SJF scheduler (as seen in Sect. 3)[2]. These models are verified in order to assess schedulability properties on the underlying GenoM3 components. We also give a couple of examples of other behavioral and timed properties that we can verify on the same models. The considered hardware is the quad-core ODROID-C0 board running Ubuntu 14.04. We refer the interested reader to [13] for more details on the Quadcopter components shown in Fig. 2. In either context, FCFS or SJF, the affinity, *i.e.* the core on which the tasks will run, is set as any task may run on any available core. As mentioned in Sec. 2.1, GenoM3 components interact with external clients in order to be able to execute services. We add a client corresponding to a stationary flight (hovering) application that involves all the components shown in Fig. 2 except MANEUVER. We have then ten tasks, and consequently a queue of size ten.

[2]Details on how Fiacre models (without schedulers) are automatically generated from GenoM3 specifications are available in [14]

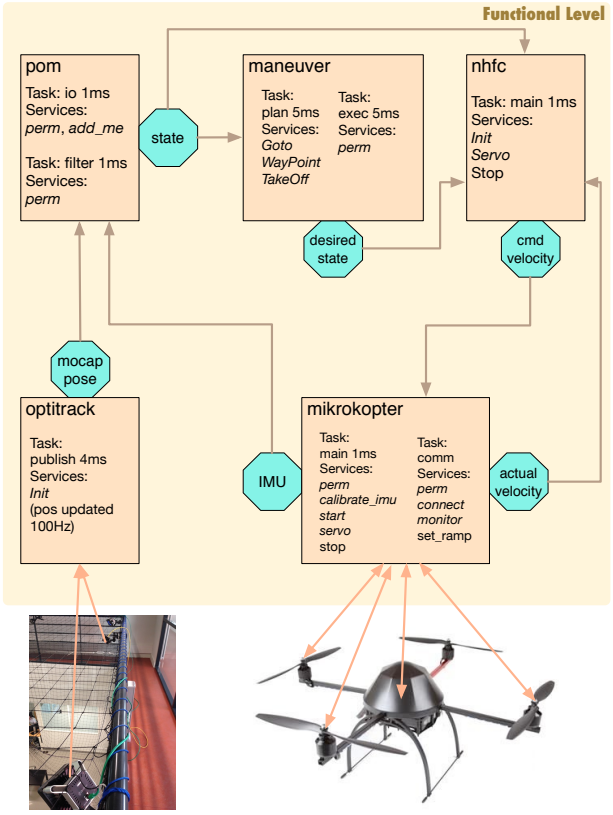

Figure 2: The quadcopter functional level. Only a partial list of services is presented. Activities are in *Italic* font.

4.1 Schedulability with FCFS

Scheduler Implementation. The cooperative FCFS scheduler can be easily implemented by using the real-time policy SCHED_FIFO on Ubuntu with all tasks given the same priority.

Schedulability Properties. As mentioned in 2.1.4, schedulability properties are hard to reason upon analytically due, especially, to the fine-grain mutual exclusion implemented in GenoM3. We express the schedulability properties for periodic tasks within the Fiacre specification as invariants. A task is schedulable iff the value of the timer tick is never reset to `true` (by its `timer`) while the task is waiting for the core allowance (its `manager` is in its state `start`) or executing (its `manager` is in its state `manage`). For instance, the schedulability property is expressed for the task io (component POM) as follows:

```
property sched_io is always (not (pom/io_manager/state ask)
                  ⇒not (pom/io_manager/value tick_odo))
```

We build the SCG and assert the schedulability properties for each periodic task. We prove that all tasks are schedulable, considering the given hardware and a cooperative FCFS scheduling policy. We also prove that the minimum number of cores to ensure such schedulability is 3 as the task filter (component POM) misses its deadline when this number is reduced to 2. Table 1 summarizes

the results according to the number of cores. It gives the details of the SCG (its size and the time needed to build it) in each case. Note that the verification time is negligible compared to the SCG construction time, thus not given here. All the verification process (SCG construction and verification of properties) is carried out on a laptop with Intel Core i5 2.7 GHz and 8 GB of RAM.

4.2 Schedulability with SJF

Scheduler Implementation. Since the EETs are assigned statically, this scheduler is easily implementable using the same real-time policy as that used for FCFS, namely SCHED_FIFO on Ubuntu, with the following differences:

- Priorities: Priorities are assigned according to the rules explained in 3.3: the smaller the period, the higher the priority, and aperiodic tasks have the same, highest priority.
- Preemption: To prevent higher priority tasks from preempting lower priority ones, a semaphore is initialized to the number of cores. A wait operation on this semaphore is added at the beginning of each task and a signal operation at the end.

Schedulability Properties. The verification results are given in Table 2. With SJF, the minimum number of cores to ensure the schedulability of all tasks is 2.

4.3 FCFS vs SJF

We prove that using SJF improves the schedulability of tasks as the application may be run on a dual-core platform while the minimum number of cores needed with FCFS is 3. By analyzing the traces of counterexamples, we realize that the scenarios where the task filter (component POM) misses its deadline correspond to execution paths where the task publish (component OPTITRACK) position in the queue precedes that of filter. These paths are, however, eliminated in the SJF context thanks to the *insert_sjf* function (listing 6) since the EET of publish (4ms) is larger than the EET of filter (1ms).

4.4 Other Properties

The same model, either with FCFS or SJF, is readily usable to verify other behavioral and timed properties like the ones verified in [14]. Results of verification on such a model are more realistic in the sense that it integrates the real hardware on which the robotic application is implemented with a realistic scheduling policy. Also, since the model is automatically generated, one may carry out similar experiments on other applications with different hardware constraints. In the sequel, we give two examples of behavioral and timed properties that we can verify on the FCFS model. In both examples, the number of cores is fixed to 4.

Liveness. In contrast to schedulability for periodic tasks, liveness is a key property for sporadic ones. It is important, for instance, to verify that a control task always finishes its execution to make sure that further requests can be received and processed in the future. GenoM3 does not provide any guarantee of such a property, especially in the presence of non-deterministic mutual exclusion (Sect. 2.1.4). We use the Fiacre pattern *leadsto* (Sect. 2.2) to formulate the liveness property, *e.g.* for MIKROKOPTER's control task:

```
property live_comm is (mikrokopter/CT/state start) leadsto
     (mikrokopter/CT/state idle)
```

The control task is initially in the state *idle*. When a request is received, it transits to *start*. Different behaviors corresponding to processing the request are possible from *start* to *finish* through intermediate states. When execution finishes, a transition is taken back to *idle* from *finish*. The property above expresses that if the control task starts executing, it will eventually go back to *idle* (after execution finishes) and therefore other requests can be received. This property evaluates to *true* for all control tasks in the specification. The SCG size and construction time are the same as in table 1 (first row, 4 cores).

Bounded response. The aperiodic task comm keeps, through its permanent activity, polling data in order to store it in MIKROKOPTER's IDS. An important timed property to verify is the minimum and maximum time that could elapse between receiving the data and actually storing it. Computing such bounds is tedious and subject to errors considering (i) the mutual exclusion between comm's permanent activity codels and other codels within and outside MIKROKOPTER and (ii) the different concurrent scenarios under the scheduling policy. We rely on the Fiacre pattern *leadsto within* (Sect. 2.2) to express the property as follows:

```
property bounded_comm is (mikrokopter/comm_permanent/state
     poll) leadsto (mikrokopter/comm_permanent/state end)
     within [min,max]
```

with *poll* and *end* being, respectively, the codel where data is received and the codel where processing such data is finished. The bounds *min* and *max* are tuned manually while verifying the property on the fly so the construction of the SCG is stopped as soon as the property is violated. We prove *min* to be equal to 0.27ms and max to be equal to 0.32ms. These results are acceptable since the upper bound is three times less than the smallest period in the specification. The new SCG is 7 923 690 classes large (roughly 600 000 classes more induced by the observer) and is built in 374.408s. The whole verification process, including tuning and on-the-fly verification, lasted about 20mn.

5 RELATED WORK

The next two paragraphs give references to some of the works on formal verification and schedulability analysis of functional robotic components. In each category, we outline the main limitations. We then show through the last paragraph how our contribution addresses such limitations and thus distinguishes itself from the related work.

Formal verification. In [12, 19, 32], functional components of simple robotic applications are manually translated into ESTEREL [9] models. The latter are model-checked for various real-time properties ranging from reachability to bounded liveness. In [1], D-Finder, an invariant-based verification tool, is used to formally verify BIP (formal framework) models synthesized from functional robotic specifications. The verification of ROS [26] robotic specifications is the subject of several works [16, 17, 35]. We find in the literature also works using other techniques such as *Discrete-Time Markov Chains* models [10]. The contributions are thus numerous yet still

Cores	SCG (size)	SCG (time)	sched_main (MIKROKOPTER)	sched_main (NHFC)	sched_publish (OPTITRACK)	sched_io (POM)	sched_filter (POM)
4	7 338 151	351.840s	True	True	True	True	True
3	10 459 826	485.764s	True	True	True	True	True
2	10 788 413	391.040s	True	True	True	True	False
1	40 545	0.880s	False	False	False	False	False

Table 1: Schedulability results with FCFS

Cores	SCG (size)	SCG (time)	sched_main (MIKROKOPTER)	sched_main (NHFC)	sched_publish (OPTITRACK)	sched_io (POM)	sched_filter (POM)
4	6 210 003	301.691s	True	True	True	True	True
3	7 986 495	333.289s	True	True	True	True	True
2	7 008 957	244.609s	True	True	True	True	True
1	32 049	0.670s	False	False	False	False	False

Table 2: Schedulability results with SJF

face fundamental issues. First, the high complexity (number of components, level of parallelism, time constraints, etc.) of functional layer specifications leads to scalability problems. As a result, unrealistic abstractions are made such as ignoring timing constraints (*e.g.* [1]). Second, most functional-level software (*e.g. ROS*) are not model based and do not use formal languages. Their formalization is thus time-consuming and non-reusable, in the sense that it has to be redone for each new application (*e.g.* [12, 16, 17, 32]). Last, hardware constraints (numbers of processors/cores and scheduling policy) are ignored while applying automatic verification to the functional components. Results are thus valid only when the number of processors/cores in the platform is at least equal to that of the robotic tasks, which is seldom the case in reality. In [24], the first two issues are tackled using RoboChart, while the last issue is not addressed. RoboChart models are automatically translated into CSP [28]. In contrast to GenoM3, RoboChart is not a robotic framework (its models are, so far, not executable on robotic platforms). That is, each robotic application, initially specified in a robotic framework, needs to be modeled first in RoboChart before it can be verified in CSP. This reduces the reusability of the method from a roboticist point of view. In [14], we particularly address the first two issues and set the last one as a line of future work.

Schedulability analysis. The goal of schedulability analysis is to check whether tasks in a system meet their temporal requirements (*e.g.* deadlines) following a certain scheduling policy on a given hardware. For robotic systems, schedulability is mainly checked using classical real-time analytical approaches. For instance, the *worst-case response time* approach is used in [25] while *utilization factor* and *dedicated analytical bound* are used in [30] and [33], respectively. In [15], the authors use MAUVE, a functional robotic framework, to specify their components, but the schedulability is checked again with an analytical approach (worst-case response time). Schedulability analysis approaches are both non automatic and difficult to extend to verify specific temporal constraints (such as end to end). Moreover, the absence of formal models of the underlying robotic application prevents the verification of other vital behavioral and timed properties in robotics e.g. undesirable state(s) are never reached or some data is always processed within a time interval.

Our contribution. The contribution presented in this paper is thus an extension to that presented in [14] and compares favorably to previous works. Schedulability analysis is automatized and properties such as liveness and bounded response are verified given the real hardware setting (number of processors/cores and two different scheduling policies). Within the same framework, engineers may verify both schedulability and other real-time properties in a fully automatic manner. The integration of the number of processors/-cores and the scheduling policies of the real robotic platform into the generated formal model makes the results more accurate. Indeed, ignoring hardware constraints, as mentioned above, restricts the validity of the verification results to (often) unrealistic assumptions about the hardware. The chosen scheduling policies are suitable for robotic applications (*e.g.* FCFS with priorities was used in [15]). This overcomes the limitation of [13], where we emphasized that the chosen non-deterministic scheduler was not adapted for real-time applications.

6 CONCLUSION

We present an approach for verifying schedulability properties as well as other real-time properties (liveness and bounded response) on robotic functional components using model checking. Our effort distinguishes itself from the related work presented in Sec. 5 in being, simultaneously, automatic (and therefore reusable), rigorous (*e.g.* all timing constraints are taken into account) and realistic with respect to the actual hardware constraints, *i.e.* the number of available cores and the scheduling policies, as it integrates them into the generated models. Overall, to our knowledge, this is the first work that attempts to find a common ground between formal verification and schedulability analysis in robotics, while taking hardware resources and scheduling policies into account. Verification results are therefore more reliable as models are closer to the real hardware-software setting.

For future work, further cooperative scheduling algorithms should be explored. Most optimized cooperative schedulers such as cooperative EDF [18] and Highest Response Ratio Next (HRNN) require, however, tracking not only the execution time but also the waiting time of jobs in the queues. The experience acquired from our work here indicates that TPN-based languages, such as Fiacre, where time intervals depend on transitions enabledness, are not suitable for such schedulers. It follows that *Timed Automata* (TA) [4] are a promising alternative to capture the time elapsed in queues using *clocks*. We plan thus to integrate these schedulers to TA models of GenoM3 specifications and compare the results with those presented here. This will be possible by modeling the schedulers in the TA-based model checker UPPAAL [20] and integrating them to the recently developed GenoM3-to-UPPAAL template.

The work in this paper is also a basis for future work on automatic source code generation from a verified formal model. A prototype has already been built from Fiacre models to generate *C* code on a *Xenomai* platform. In this context, we highlight that the schedulers models presented in our work here, associated with the behavioral model of the robotic application, are an effective way to guarantee temporal constraints of the system during its execution. On the long term, the execution model will allow us to perform runtime verification and enforce desired properties online.

Overall, the work presented in this paper is beneficial to the robotics, the formal methods, and the real-time systems communities. It confronts a model-checking framework to a real-word complex application and, by considering the hardware constraints and automatizing the process, advances the state of the art toward a more correct and less tedious formal verification of robotic and autonomous applications. This work may be, consequently, considered as a step forward in the direction of a safe deployment of autonomous systems in real-world critical applications in human environments.

REFERENCES

[1] T. Abdellatif, S. Bensalem, J. Combaz, L. de Silva, and F. Ingrand. 2012. Rigorous design of robot software: A formal component-based approach. *Robotics and Autonomous Systems* 60, 12 (2012), 1563–1578.

[2] Nouha Abid, Silvano Dal Zilio, and Didier Le Botlan. 2012. Real-Time Specification Patterns and Tools. In *Formal Methods for Industrial Critical Systems*. 1–15.

[3] R. Alami, R. Chatila, S. Fleury, M. Ghallab, and F. Ingrand. 1998. An architecture for autonomy. *The International Journal of Robotics Research* 17, 4 (1998), 315–337.

[4] Rajeev Alur and David L Dill. 1994. A theory of timed automata. *Theoretical computer science* 126, 2 (1994), 183–235.

[5] B. Berthomieu, J.-P. Bodeveix, P. Farail, M. Filali, H. Garavel, P. Gaufillet, F. Lang, and F. Vernadat. 2008. Fiacre: an Intermediate Language for Model Verification in the Topcased Environment. In *ERTSS*.

[6] Bernard Berthomieu and Miguel Menasche. 1983. An enumerative approach for analyzing time Petri nets. In *IFIP Congress*.

[7] B. Berthomieu, P.O. Ribet, and F. Vernadat. 2004. The tool TINA – Construction of Abstract State Spaces for Petri Nets and Time Petri Nets. *Journal of Production Research* 42, 14 (2004).

[8] P A Bourdil, Bernard Berthomieu, and E Jenn. 2014. Model-Checking Real-Time Properties of an Auto Flight Control System Function. In *ISSREW*.

[9] Frédéric Boussinot and Robert de Simone. 1991. The ESTEREL Language. In *Proceeding of the IEEE*, Vol. 79. 1293–1304.

[10] Giuseppe Cicala, Ali Khalili, Giorgio Metta, Lorenzo Natale, Shashank Pathak, Luca Pulina, and Armando Tacchella. 2016. Engineering approaches and methods to verify software in autonomous systems. In *IAS*.

[11] Alessandro Cimatti, Marco Roveri, and Piergiorgio Bertoli. 2004. Conformant planning via symbolic model checking and heuristic search. *Artificial Intelligence* 159, 1-2 (2004), 127–206.

[12] B Espiau, K Kapellos, and M Jourdan. 1996. Formal verification in robotics: Why and how?. In *International Symposium on Robotics Research*. Springer, 225–236.

[13] Mohammed Foughali. 2017. Toward a Correct-and-Scalable Verification of Concurrent Robotic Systems: Insights on Formalisms and Tools. In *ACSD*.

[14] M. Foughali, B. Berthomieu, S. Dal Zilio, F. Ingrand, and A. Mallet. 2016. Model Checking Real-Time Properties on the Functional Layer of Autonomous Robots. In *International Conference on Formal Engineering Methods*. Springer, 383–399.

[15] Nicolas Gobillot, Fabrice Guet, David Doose, Christophe Grand, Charles Lesire, and Luca Santinelli. 2016. Measurement-based real-time analysis of robotic software architectures. In *IROS*.

[16] Raju Halder, José Proença, Nuno Macedo, and André Santos. 2017. Formal verification of ROS-based robotic applications using timed-automata. In *Proceedings of the 5th International FME Workshop on Formal Methods in Software Engineering*. IEEE Press, 44–50.

[17] Jeff Huang, Cansu Erdogan, Yi Zhang, Brandon Moore, Qingzhou Luo, Aravind Sundaresan, and Grigore Rosu. 2014. ROSRV: Runtime verification for robots. In *International Conference on Runtime Verification*. Springer, 247–254.

[18] Mehdi Kargahi and Ali Movaghar. 2005. Non-preemptive earliest-deadline-first scheduling policy: a performance study. In *Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, 2005. 13th IEEE International Symposium on*. IEEE, 201–208.

[19] M. Kim and K. C. Kang. 2005. Formal Construction and Verification of Home Service Robots: A Case Study. In *Automated Technology for Verification and Analysis*. 429–443.

[20] K. G. Larsen, P. Pettersson, and W. Yi. 1997. UPPAAL in a nutshell. *International Journal on Software Tools for Technology Transfer (STTT)* 1, 1 (1997), 134–152.

[21] Simone Lupetti and Dmitrii Zagorodnov. 2006. Data popularity and shortest-job-first scheduling of network transfers. In *Digital Telecommunications,, 2006. ICDT'06. International Conference on*. IEEE, 26–26.

[22] A. Mallet, C. Pasteur, M. Herrb, S. Lemaignan, and F. Ingrand. 2010. GenoM3: Building middleware-independent robotic components. In *Robotics and Automation (ICRA), 2010 IEEE International Conference on*. IEEE, 4627–4632.

[23] P. M. Merlin and D. J. Farber. 1976. Recoverability of Communication Protocols: Implications of a Theoretical Study. *IEEE transactions on Communications* 24, 9 (1976), 1036–1043.

[24] A. Miyazawa, P. Ribeiro, W. Li, A. L. C. Cavalcanti, and J. Timmis. 2017. Automatic Property Checking of Robotic Applications. In *The International Conference on Intelligent Robots and Systems*.

[25] Ala' Adel Qadi, Steve Goddard, Jiangyang Huang, and Shane Farritor. 2005. A performance and schedulability analysis of an autonomous mobile robot. In *ECRTS*.

[26] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y Ng. 2009. ROS: an open-source Robot Operating System. In *ICRA workshop on open source software*, Vol. 3. Kobe, 5.

[27] M. Quigley, B. Gerkey, K. Conley, J. Faust, T. Foote, J. Leibs, E. Berger, R. Wheeler, and A. Y. Ng. 2009. ROS: an open-source Robot Operating System. In *ICRA workshop on open source software*, Vol. 3. Kobe, 5.

[28] Andrew William Roscoe. 2010. *Understanding concurrent systems*. Springer Science & Business Media.

[29] Uwe Schwiegelshohn and Ramin Yahyapour. 1998. Analysis of first-come-first-serve parallel job scheduling. In *SODA*, Vol. 98. 629–638.

[30] Jiazheng Shi, Steve Goddard, Anagh Lal, and Shane Farritor. 2004. A real-time model for the robotic highway safety marker system. In *RTAS*.

[31] Reid Simmons, Charles Pecheur, and G Srinivasan. 2000. Towards Automatic Verification of Autonomous Systems. In *IROS*.

[32] A. Sowmya, D. Tsz-Wang So, and W. Hung Tang. 2002. Design of a Mobile Robot Controller using Esterel Tools. *Electronic Notes in Theoretical Computer Science* 65, 5 (2002), 3–10.

[33] John D Sweeney, Huan Li, Roderic A Grupen, and Krithi Ramamritham. 2003. Scalability and schedulability in large, coordinated, distributed robot systems. In *ICRA*.

[34] Kai Weng Wong and Hadas Kress-Gazit. 2016. Need-based coordination for decentralized high-level robot control. In *IROS*.

[35] Safdar Zaman, Gerald Steinbauer, Johannes Maurer, Peter Lepej, and Suzana Uran. 2013. An integrated model-based diagnosis and repair architecture for ROS-based robot systems. In *Robotics and Automation (ICRA), 2013 IEEE International Conference on*. IEEE, 482–489.