Toward a Correct-and-Scalable Verification of Concurrent Robotic Systems: Insights on Formalisms and Tools

Mohammed Foughali

▶ To cite this version:

Mohammed Foughali. Toward a Correct-and-Scalable Verification of Concurrent Robotic Systems: Insights on Formalisms and Tools. International Conference on Application of Concurrency to System Design (ACSD 2017), Jun 2017, Zaragoza, Spain. 10p., 10.1109/ACSD.2017.10 . hal-01515012

HAL Id: hal-01515012
https://hal.science/hal-01515012

Submitted on 28 Apr 2017

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Toward a Correct-and-Scalable Verification of Concurrent Robotic Systems: Insights on Formalisms and Tools*

Mohammed Foughali[1]

Abstract—**Formal verification of robotic functional components is extremely important. Indeed, with the growing involvement of autonomous systems in everyday life, we may no longer rely on classical testing and simulation to establish our trust in them. However, the formalization of such systems is challenging considering the various existing formalisms and their respective advantages/drawbacks. One may express more easily in one formalism and verify more easily in another depending on the aspects/properties they are modeling/verifying. Furthermore, both the reusability of the formalization and the scalability of the obtained formal models are crucial elements in the verification process. In this paper, we present modeling concurrency aspects of robotic functional components in Time Petri Nets, Timed Automata and Timed Automata extended with urgencies. Formal models are automatically generated and verification is conducted on each of them. Both the expressiveness of the formalisms and scalability of the obtained models are evaluated and future directions are consequently outlined.**

I. Introduction

Robotic and autonomous systems involvement in costly missions (e.g. space exploration) and/or human environments (e.g. robotic surgery) keeps growing with time. Such a growth questions the level of trust we are willing to put in these systems. Indeed, we still tend to rely on extensive tests and heavy simulations for certification while such techniques are inherently non exhaustive. Formal methods constitute a promising alternative to consolidate trustworthiness of robotic and autonomous systems.

Robotic/autonomous specifications are very often organized through two layers: the decisional and the functional one [1]. In this paper, we only focus on applying formal methods to the latter. The reason is quite straightforward. Decisional descriptions are usually formal and thus formal methods are directly applicable. This explains why literature is rich in works of formal verification of decisional functions [2], [3], [4], [5]. In contrast, functional specifications are non formal and remarkably complex, which renders their verification extremely challenging.

Functional descriptions are nowadays deployed through *components* [6], [7], [8]. Formal verification of functional robotic components faces two main problems. First, due to their non formal nature, their formalization is often hard, error prone and non reusable (manual). Second, the complexity of functional components leads quickly to state space explosion, which forces the specialists to go through exaggerated abstractions such as removing time constraints. At least one of these issues is present in [9], [10], [11], [12], [13].

The problems pertaining to formalization are often treated slightly or not treated at all. This is however worrying since the efficiency of formalization, its correctness and its reusability are crucial to both the feasibility and accuracy of verification. Indeed, one formalism may be more suitable than the other in modeling a certain aspect of the application while it is the other way around for another aspect. The choice of the most suitable formalism and associated verification techniques is far from being obvious, yet very important. With regards to *Time Petri Nets* “à la Merlin” [14] and *Timed Automata* [15], few formal works comparing their expressiveness and trying to bridge the gap between them exist in the literature [16], [17], [18]. There is, however, an important disconnection between such leading formal works and the real-world, complex applications proposed by the robotics community. This gap shows no signs of shrinking as formal methods are out of a roboticist field of expertise. A rare contribution that actually brings Timed Automata and Time Petri Nets together with relatively complex examples put on verification is presented in [19] and discussed in section V.

In this paper, we go through the abovementioned issues. Formal methods are applied to components written in G$^{en}_{o}$M3, a robotic software equipped with an automatic synthesis mechanism. Non reusability is thus resolved as formal models are automatically generated out of the functional components. On the modeling level, we focus on presenting and discussing modeling the fine-grain locking of resources characteristic to G$^{en}_{o}$M3. Time Petri Nets (TPN), *Timed Safety Automata* (TA) and *Timed Safety Automata extended with urgencies* (UA) are used. We compare the power of such formalisms to express the said concurrency aspect and the feasibility of verification in each case using Fiacre/TINA for TPN and BIP/RTD-Finder for TA/UA. Formal models are automatically generated, the complexity of the specification is conserved all along the process and all timing constraints are taken into account. We stand halfway between the formal methods and the robotic communities in a clear attempt to make this paper readable to both of them. Formal definitions are therefore often avoided as we rely mostly on natural language and simple mathematical concepts to develop our ideas.

The rest of this paper is organized as follows. First, G$^{en}_{o}$M3, its automatic synthesis mechanism and the concurrency within and between its components are presented through an example in section II. Afterwards, section III presents and discusses (i) modeling the concurrency in TA and UA as well as verification feasability with RTD-Finder (section III-A) and

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

[1]LAAS-CNRS, Université de Toulouse, CNRS, Toulouse, France {firstname.lastname}@laas.fr

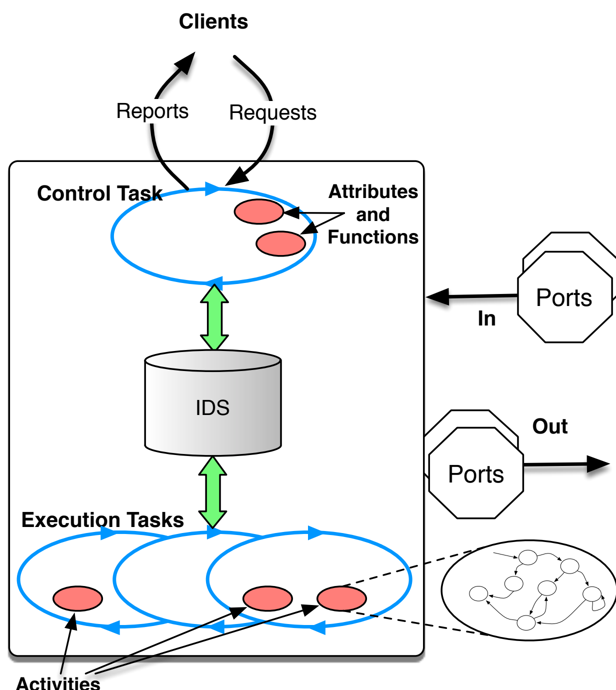

Fig. 1: A generic GenoM3 component.

(ii) modeling such concurrency in TPN and verification of the components with TINA (section III-B). We debate the results in section IV and conclude with learned lessons and future actions in section V.

II. GenoM3 COMPONENTS

GenoM3 [20] is a component-based framework used to develop and deploy robotic functional specifications. In the overall LAAS architecture [21], functional components act as "servers" in charge of functionalities which may range from simple low-level driver control (e.g. the velocity control of the propellers of a drone, camera, etc) to more integrated computations (e.g. navigation, mapping, etc).

We consider that a typical component is a *program* which needs to handle and manage the following aspects:

Inputs and Outputs : a component interacts with external clients and other components. For the former, the control flow, it must handle *requests* and send back *reports* from/to clients. For the latter, the data flow, it must provide a mechanism to share data with other components.

Algorithms : the core algorithms needed to implement the functionality the component is in charge of must be appropriately organized as to preserve the reactivity of the component and the schedulability of the various possibly concurrent algorithms. A component may have just one service to provide, but most of the time, there are a number of such services associated to the considered robotic functionality.

Internally shared data : the various algorithms, possibly concurrent, running in the component, may have to share state variables, parameters, etc. which represent the internal state of the component.

To achieve such requirements of a functional component, we propose to organize each one as shown in Fig. 1. Hereafter, we briefly introduce the different entities of a GenoM3 component:

Control Task : A component always has an implicit *control task* that manages the control flow by processing *requests* and sending *reports* (from/to external clients); activates and stops services, etc.

Execution Task(s) : Aside from the *control task*, whose reactivity must remain short, one may need one or more *execution tasks*, aperiodic or periodic, in charge of longer computations.

Services : The core algorithms of the component are encapsulated within *services*. *Services* are often associated to a *request* (with the same name). They can be either *fast* or slow. Fast services are known as *control services* and are directly executed by the control task. Control services may be attributes (setters/getters) or functions (quick computations). Slow services are known as *activities* and they are executed by execution tasks. They are broken into different *states*, each state associated to a *codel*. A codel specifies a C or C++ function. At a given state of an *activity*, the execution task runs the codel associated to that state.

IDS : A local *internal data structure* is provided for all the *services* to share parameters, computed values or state variables of the component. It is appropriately accessed by the *codels* when they need to read or write it (section II-B).

Ports : They specify the shared data, in and out, the component needs or produces from/for other components. Access to ports is mutually exclusive.

Note that this description is greatly simplified. Indeed, GenoM3 components contain several aspects the formalization of which is difficult and non obvious. For instance, there are a number of rules concerning the behavior of execution tasks: how, when, and how often they execute their associated activities. These aspects are out of the scope of this paper but the interested reader may refer to [22] for some of them and their formalization.

A. Templates Approach

The template mechanism was first proposed to deal with the *middleware dependency* problem, classical in robotics. A *robotic middleware* is the piece of software that permits the functional components to communicate with the operating system and with one another. In this context, GenoM3 developers proposed a novel approach back then known as *template*

```
<' foreach s [$component services] { '>

    Service <"[$s name]">

<' } '>
```

Listing 1: A simple template code snippet.

mechanism [20]. In brief, the components are specified in a generic way using GenoM3. Templates are used to generate the components for various middleware. Hence, the component achieves a complete independency from the middleware.

A template, when called by GenoM3 on a given component specification, has access to all the information contained in the specification such as services names and types, ports and IDS fields needed by each codel, execution tasks periods, etc. Through the template interpreter (Tcl), one specifies what they need the template to synthesize. For instance, Listing 1 shows a simple template function that outputs the names of all the services of a given component. The interpreter evaluates anything enclosed in *markers* (<''> or <"">) in Tcl and leaves the rest as is.

Since there is no restriction on what a template may synthesize, we use this powerful mechanism to automatically generate formal models. As of today, GenoM3 has three formal-model templates, a Fiacre template [22], and two BIP templates (one for UA and one for TA).

B. Concurrency

As said in the introduction, this paper will focus mainly on modeling concurrency aspects of GenoM3. GenoM3 tasks run in parallel assuming the number of available cores is sufficient. Therefore, the shared resources need to be correctly managed. The latest releases of GenoM3 improve the parallelism level in particular. Only the field(s) of the IDS/ports needed by a codel are now locked while such a codel is running and simultaneous readings are allowed. This guarantees a maximal level of parallelism.

Let us introduce our quadcopter navigation as the verification case study through which we also illustrate the concurrency aspect of GenoM3. Fig. 2 presents the 5 components involved in our quadcopter functional layer. Each box corresponds to a component, and each octagon is a port. Ports are written (out) by the components they are attached to, and read (in) through the arrow pointing to the reading component. Inside each box, we list the execution tasks, their periods (if any), and a partial list of the services provided by this component. The *perm* keyword refers to *permanent activities*. Unlike regular activities which are started upon request from a client, permanent activities are started when the task is spawned, but otherwise follow the same execution rules as any other activity (see below). Note that this figure does not present the clients in charge of sending requests and analyzing reports (out of scope of this paper).

- MIKROKOPTER is the component in charge of the quadcopter low-level hardware. The quadcopter is controlled by applying a velocity to each propeller, and produces the current velocities, as well as its current IMU (Inertia Measurement Unit) values. It has two execution tasks i) comm, aperiodic, in charge of polling and parsing data from the hardware (to get the current propellers velocity and IMU) and storing them in the IDS. ii) main, periodic at 1KHz, which reads the **cmd velocity** port, manages the servo control and writes the two ports **IMU** and the propellers **actual velocity**.
- OPTITRACK is the component handling the current position of the quadcopter as perceived by our "OptiTrack" motion capture system. It has one execution task publish that provides the current position of the quadcopter in the **mocap pose** port. Its period is 4ms.
- POM merges the **mocap pose** position produced by OPTITRACK and the **IMU** from MIKROKOPTER and produces an Unscented Kalman filtered position in port **state**. It has two execution tasks io and filter both periodic at 1KHz.
- MANEUVER is the navigation component, it has two execution tasks exec and plan both periodic at 5ms. Given a position or waypoints to navigate to, it reads the **state**, and computes a trajectory to reach it, producing in **desired state** the intermediate positions to fly to.
- NHFC (Near Hovering Flight Controller) is the core of the flight controller. Running one task main at 1KHz, it reads the **actual velocity** port of the propellers, the current position in the **state** port of POM, and the desired position (port **desired state**) of MANEUVER and produces the proper **cmd velocity** port containing the desired velocity of the propellers (which is then read by MIKROKOPTER) to reach and hover near this position.

Through an excerpt of the specification of MIKROKOPTER (listing 2), we introduce hereafter the concurrency of GenoM3 tasks. In order to do so, we see first how an activity is specified in GenoM3. An activity is defined through its states, transitions and codels. The execution task main is in charge of the activity *Servo* (line 12). *Servo* has three states: `start`, `exec` and `end` (lines 13 through 15) to which a terminal state called `ether` is added. Except the latter, each state defines a codel to execute (for example, the codel `mk_servo_start` is associated to the state `start`, line 13), the possible transitions at the end of such an execution (for example, the state `exec` transits to itself or to the state `end`, line 14) and, optionally, the WCET of the codel ($4\mu s$ for `mv_servo_end`, line 15). An activity may be incompatible with a set of activities that may include the activity itself. Line 16 specifies that the activity *servo* interrupts itself, it is thus *self-incompatible*. A self-incompatible activity may not run a new instance unless the currently running one is interrupted and terminated. Therefore, it may not have more than one *active* (running) instance at any given time, contrary to self-compatible activities that may have as many active instances as the memory may handle. In the rest of this paper, we extend the term *codel* to refer indifferently to itself or the state it is associated to. This choice makes it easier

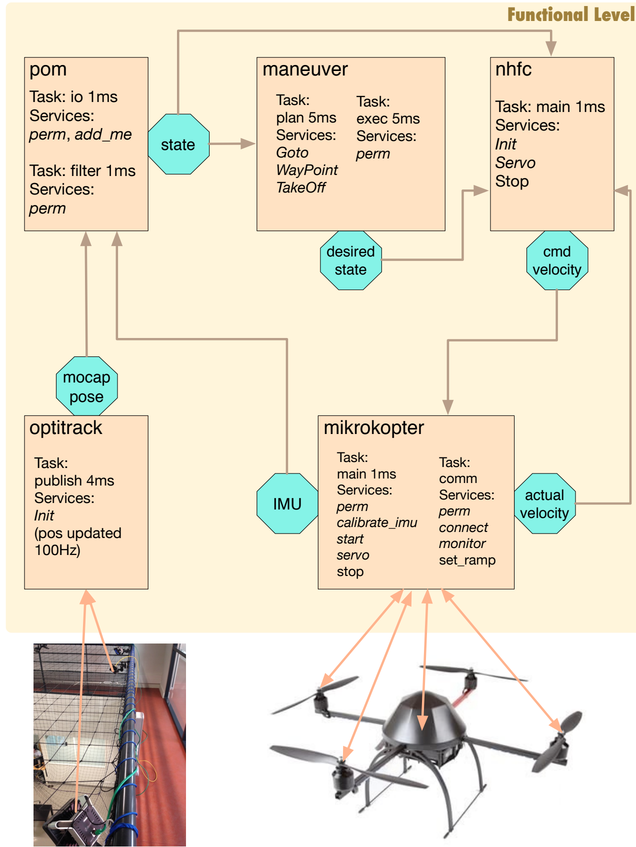

Fig. 2: The quadcopter functional level. Only a partial list of services is presented. Activities are in *Italic* font.

to formulate the different explanations given in section III.

The attribute *set_ramp* writes (out) the IDS field *servo_data* (line 5). Since an attribute is a control service, its execution is carried out by the control task. On the other hand, the codel `mk_servo_main` associated to the state `exec` of the activity *servo* (line 14) reads (in) the same IDS field, namely *servo_data* (codel's arguments, line 14). This codel is executed by the execution task main, in charge of the activity *servo*. This is a perfect example of how concurrency works in G$^{en}_{o}$M3. Since the attribute and the activity codel are run by two different tasks in parallel, whichever starts executing first locks the field *servo_data* of the IDS and forces the other to wait till it finishes its execution. **As soon as** such a field is unlocked, the waiting entity executes, there is therefore a notion of **urgency**. We say that the codel `exec` and the attribute *set_ramp* are **in conflict**. G$^{en}_{o}$M3 offers a function that ensures a correct implementation of concurrency while preserving a maximum parallelism level as explained above (simultaneous readings are allowed). This function returns, for each codel/control service, the list of codels/control services in conflict with it. A codel/control service that is in conflict with at least one codel/control service is referred to as *non-thread-safe*. There is also a notion of **non determinism**. That

```
1  component mikrokopter {
2    ...
3
4      /* control services declaration */
5    attribute set_ramp(out servo_data);
6    ...
7
8    /* activities declaration */
9
10    activity servo() {
11     doc  "Control the propellers according to the given
          velocities";
12     task main;/* execution task in charge of the activity */
13     codel<start> mk_servo_start(out scale) yield exec wcet
          26us;
14     codel<exec> mk_servo_main(in servo_data, inout scale,
          ...) yield exec, end wcet 12us;
15     codel<end> mk_servo_end(in conn) yield ether wcet 4 us;
16     interrupts servo;
17   };
```

Listing 2: Excerpt from the G$^{en}_{o}$M3 specification of the MIKROKOPTER component.

is, if several non-thread-safe entities are waiting for the same resources, there is no rule defining which of the waiting entities will lock such resources as soon as they are released. Note that mutual exclusion between G$^{en}_{o}$M3 entities is not transitive, i.e if B and C are in conflict with A, then B is not necessarily in conflict with C (A, B and C being codels/control services).

Not to go beyond of the scoop of this paper, our description of the mechanisms offered by G$^{en}_{o}$M3 to specify and deploy functional components remains partial. Still, one can see the complexity of G$^{en}_{o}$M3 components and the challenge of applying formal methods to them. For instance, in the quadcopter example above, we have 5 modules, 13 tasks, 20 activities, and more than 65 codels overall.

III. Modeling and Verification

The formalization and the verification of G$^{en}_{o}$M3 components face several challenges. For instance, G$^{en}_{o}$M3 reflects many software aspects the formalization of which is unnatural and error prone, mainly interruption routines, termination of activities, etc. Also, the modeling process should not be decoupled from the verification one. That is, it is not sufficient to model correctly, but also efficiently. This efficiency rimes with both scalability and convenience. The user should understand the generated models and express the properties they want to verify easily, and models should remain as scalable as possible. It is therefore very important to preserve compositionality throughout the modeling process.

This section will focus on modeling the concurrency aspect detailed in section II-B in TPN, TA, and UA. Both compositionality and timing constraints are preserved all along the verification process. The power of the different formalisms to express the urgency characterizing the way G$^{en}_{o}$M3 deals with concurrency is particularly assessed. The verifiability of the global systems with the proposed tools is then evaluated.

All along this section, we consider three non-thread-safe

codels $C1$, $C2$ and $C3$ belonging to, respectively, activities $A1$, $A2$ and $A3$. These activities are run by three different tasks. $C1$ and $C3$ are in conflict with $C2$ but not in mutual conflict. Consequently, $C1$ (respect. $C3$) may not run if $C2$ is executing, and vice versa. However, $C1$ and $C3$ may run in parallel. For the sake of readability and simplicity, we suppose that each of these codels has only one *input* and only one *output* codel. That is, for instance, only one codel in $A1$ has a transition to $C1$ and $C1$ has only one outgoing transition.

A. With TA/UA

a) With TA: Timed Automata is a theory for modeling and verification of timed systems. In the original version of the theory [15], Timed Automata extend *finite-state Büchi automata* with real-valued clocks. The behavior of such automata is therefore restricted by defining constraints on the clock variables and a set of accepting states. A simpler version allowing local invariant conditions and known as *Timed Safety Automata* is introduced in [23]. In this paper, we focus on Timed Safety Automata and refer to them as Timed Automata or TA for short. The syntax and semantics of TA in this paper follow those in [24] except that we refer to *switches* as *transitions*.

We target in this section modeling the G^{en}oM3 concurrency/mutual exclusion aspects in TA. Fig. 3 shows such modeling. The system is modeled as TA *components* connected through *synchronization constraints* on labels. When two or more labels are connected, the respective transitions are synchronized. The global system is the synchronous product of the TA components with respect to the different synchronization constraints. Synchronization constraints are represented as thick colored lines in Fig. 3. For the sake of readability, merely the synchronization constraints involving the execution and end of execution of $C2$ are represented in Fig. 3. Note that (i) the components $A1$, $A2$ and $A3$ are represented only partially (the discontinued transitions denote missing parts before/after them) and (ii) labels/clocks belonging to different components are different, even if they have the same name. These two simplifications apply also to the UA model (in the sequel).

Each non-thread-safe codel C is modeled within its activity component by two locations, *C_wait* and *C_exe*. If *C_wait* is the current location, then C is waiting for resources to execute. In contrast, location *C_exe* denotes that C is executing. To imitate the execution time we define a local invariant on *C_exe* with an upper bound equal to the codel's WCET. Each C has a *lock_C* TA component that determines whether C may execute (location *ready*), is executing (location *execute*) or is idle/needs to wait further (otherwise).

Let us depict how it works with an example. The component *lock_*$C2$ is initially at the location *Init*. Unless one of the codels in conflict with $C2$, i.e. $C1$ or $C3$, starts executing, *lock_*$C2$ transits to *ready* as $C2$ reaches the location *C2_wait*. Here, an urgency occurs as no time may elapse at the location *ready* (component lock_$C2$). Therefore, $C2$ transits immediately to its location *C2_exe* while its lock transits to *execute* (in case $C1$ or $C3$ is also waiting to execute, non-determinism applies). By the end of execution, *lock_*$C2$ transits back to *Init*. Now, if $C1$ starts executing, *lock_*$C2$ transits to *level_1*. If $C2$ reaches its location *C2_wait* during $C1$'s execution, it is the end of the latter that allows the transition to *ready* (component lock_$C2$) and consequently the possibility to execute $C2$.

TA do not express urgencies naturally, hence the heterogeneity of the *lock* components from a codel to another. Let C be a non-thread-safe codel and N the number of codels in conflict with it. The proposed solution induces a linear proportionality between N and the number of locations/transitions of the *lock* component associated with C:

- The number of locations in the *lock* component associated with C is equal to N+3
- The number of transitions within the *lock* component associated with C is equal to 3N+5.

For instance, when N is equal to 8 (which is rather usual for a G^{en}oM3 non-thread-safe codel), the number of locations and transitions is, respectively, 11 and 29. Note that another representation is possible using Timed Automata extended with variables where each *lock* component has a local variable. This allows a more compact and homogeneous representation of the *lock* components (Fig. 4). However, such a representation changes nothing in terms of the global product of the components.

b) With UA: Timed Automata with Urgencies (UA, [25]) extend Timed Automata (TA) with a notion of urgency on transitions. In this paper, we use two types of urgencies:

- **eager**: the transition is to be taken as soon as enabled. This is the *strong* urgency.
- **lazy**: no urgency, as in standard TA. Naturally, it's the *weak* urgency.

Fig. 5 shows the modeling of G^{en}oM3 concurrency/mutual exclusion aspects in UA. As in TA, we use synchronization constraints among different components. Applying a synchronization constraint on transitions with no conditions on clocks and with different urgencies produces a transition with the strongest urgency. For instance, the synchronization constraint involving the transition labeled *resfree* (component $A2$), the transition labeled *take* (component lock_$C2$), and the transitions labeled *check* (components lock_$C1$ and lock_$C3$) results in an *eager* transition (conjunction of eager, lazy, lazy and lazy). Therefore, if the codel $C2$ is waiting to execute, i.e component $A2$ is in the location *C2_wait*, it will transit to *C2_exe* (start executing) as soon as the components lock_$C1$, lock_$C2$ and lock_$C3$ are in their respective locations *free* (in case $C1$ or $C3$ is also waiting to execute, non-determinism applies). Such a transition is concomitant to switching lock_$C2$ to its location *taken* which will prevent any codel in conflict with $C2$ to execute until $C2$ finishes its execution. The rest of the model is quite similar to the TA model. For the sake of readability, only one synchronization constraint (that of $C1$) defining codels end of execution is represented in Fig. 5. Also, the *lazy* type of urgency is omitted (being the "default" type).

c) Automatic Synthesis and Verification: After formalizing all the remaining aspects/entities of G^{en}oM3 components,

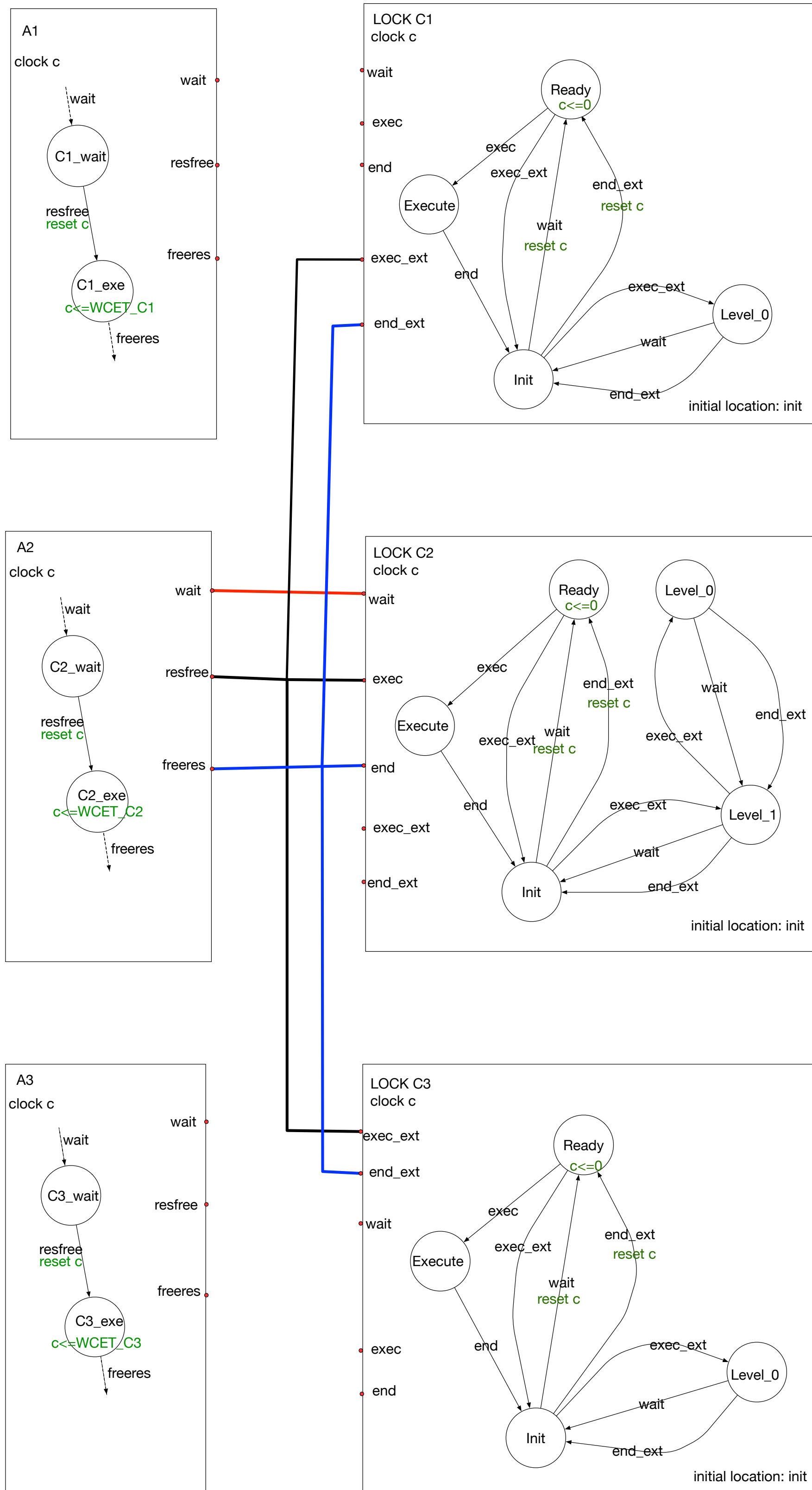

Fig. 3: Concurrency and mutual exclusion in TA.

two templates are developed to automatically synthesize, respectively, UA and TA models in the RT-BIP [26] syntax. RT-BIP associated verification tool is known as RTD-Finder [27]. It is a deductive, compositional verification tool that overap-

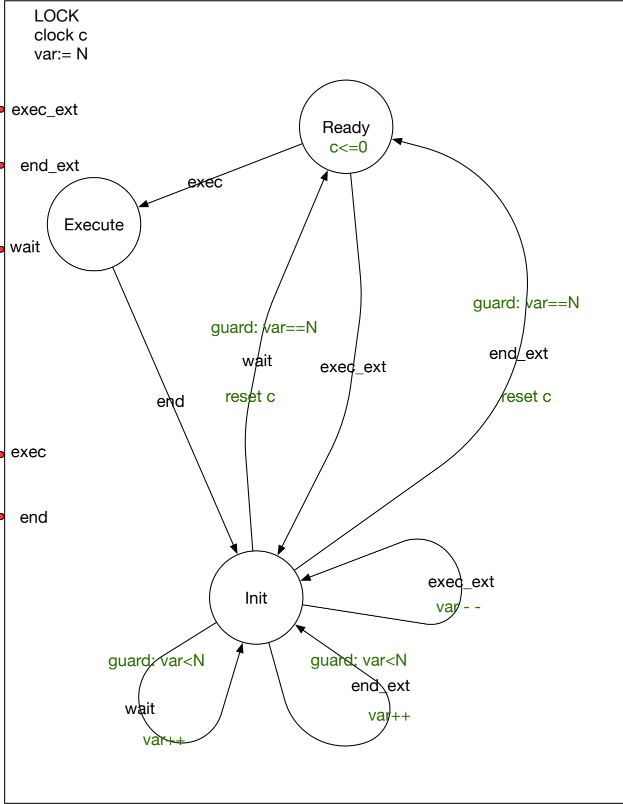

Fig. 4: The lock components with a local variable.

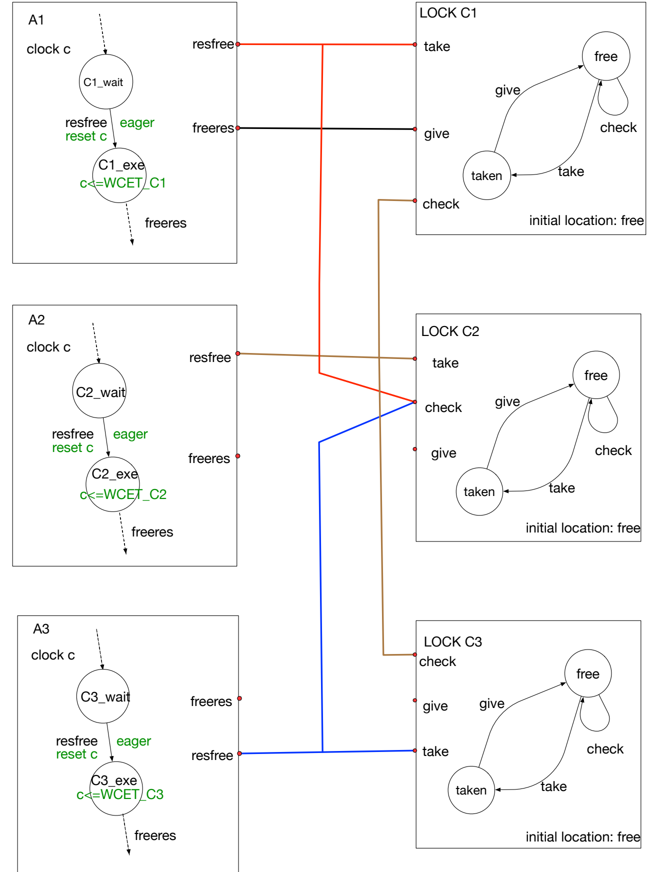

Fig. 5: Concurrency and mutual exclusion in UA.

proximates the reachable state space using invariants. RTD-Finder aims to overcome the state explosion problem often encountered in complex real-time systems modeled as networks of TA. RTD-Finder extends its untimed version D-Finder [28] with the possibility to reason over time by considering *history clocks* in the generation of the components local invariants.

We automatically generate the RT-BIP models from our quadcopter application specification using the RT-BIP templates. Two models are thus obtained, a TA- and a UA-based model. The TA model is relatively large due to the proportionality between the number of locations/transitions of a given lock and the codels in conflict with its associated codel. The verification process leads quickly to a memory blow-up even when tested on our simplest components. We could not verify the UA and extended TA models since RTD-Finder does not take urgencies/variables into account.

B. With TPN

a) Modeling: Time Petri Nets [14], TPN for short, are Petri nets extended with intervals on transitions. Each transition T is associated with the positive reals efd_T and lfd_T where:

- efd_T is bounded and denotes the *earliest firing deadline* of T
- lfd_T is possibly unbounded and is superior or equal to efd_T. It is called the *latest firing deadline* of T.

In this paper, T activation, or *firing* in TPN terminology, follows the *Strong Time Semantics* of TPN introduced in [29]. That is, if T was last enabled at time d, T may not fire before $d + efd_T$ and *must* fire before or at $d + lfd_T$ unless it is disabled before then by the firing of another transition.

A TPN is referred to as *safe* if it is *1-bounded*, i.e the maximum marking of each of its places is 1. In this section, the TPN components we use for modeling are all safe, and thus we define a composition rule that is similar to the one of automata. Synchronization constraints define which transitions are to synchronize. The resulting transition (from synchronization) is associated with the interval resulting from the conjunction of the respective intervals of the synchronized transitions. The input (respect. output) places of the resulting transition is the union of the input (respect. output) places of the synchronized transitions. The global TPN is the product of the TPN components following these rules by applying the synchronization constraints. Fig. 6 shows how we model, using TPN components and synchronization constraints, the concurrency of the execution of $C1$, $C2$ and $C3$ while finely locking resources.

Each non-thread-safe codel C has a *lock_C* TPN compo-

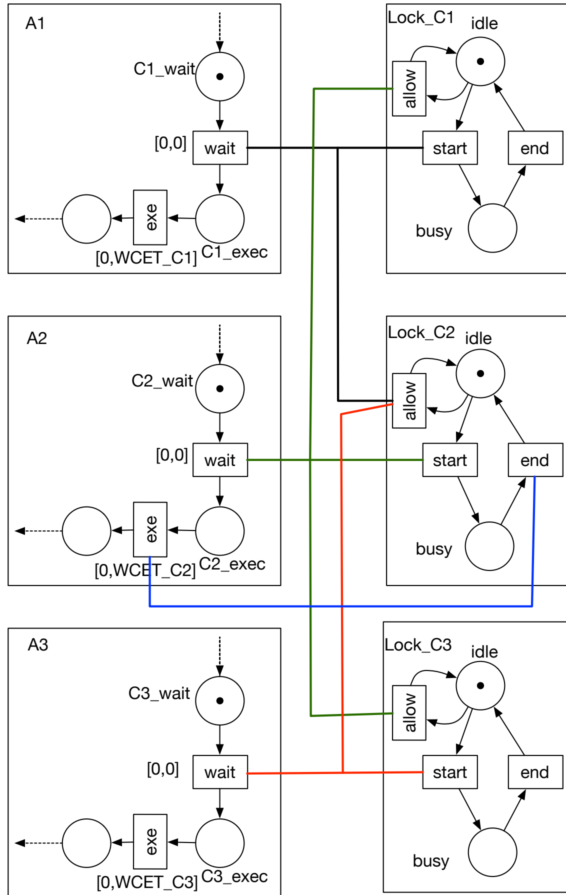

Fig. 6: Concurrency and mutual exclusion in TPN.

nent. The marking of that TPN defines whether C is executing (place *busy* marked) or not (either idle or waiting to execute, place *idle* marked). Firing the transition *start* unmarks *idle* and marks *busy*, firing *end* does the contrary. The transition *allow* is enabled when *idle* is marked and its firing preserves the marking of the net.

Despite the lock component, C is modeled within its activity by two places, *C_wait* and *C_exec*. If C_wait is marked then C is waiting for resources to execute. In contrast, if C_exec is marked then C is executing.

In Fig. 6, synchronization constraints are represented by colored thick lines connecting transitions of different components. For instance, allowing $C2$ to execute is dictated by the synchronization constraint that involves its transition *wait* (component $A2$), the transition *start* of its own *lock* (component lock_$C2$) and all the transitions *allow* of the locks of the codels in conflict with $C2$ (components lock_$C1$ and lock_$C3$). Let T be the transition resulting from applying such a constraint. T is associated with the firing interval [0,0] (the conjunction of [0,0],[0,∞[,[0,∞[and [0,∞[). Hence, **as soon as** all the input places of T, i.e *C2_wait* (component $A2$), *idle* (component lock_$C2$) and all the places *idle* of the lock components of the codels in conflict with $C2$ (components lock_$C1$ and lock_$C3$), are marked, T is immediately fired (in case $C1$ or $C3$ is also waiting to execute, it might get immediately disabled instead as non-determinism applies). Urgencies are then well expressed in a relatively easy way. This is mainly due to the fact that firing intervals in TPN depend on the enabledness of their associated transitions.

Let T' be the result of applying the synchronization constraint connecting *exe* (component $A2$) to *end* (component lock_$C2$). T' is then associated with the firing interval [0,WCET_C2] (where WCET_C2 is the WCET of $C2$). When T is fired, *C2_exec* (component $A2$) and *busy* (component lock_$C2$) are marked and T' is consequently enabled. According to TPN firing rules, T' must be fired within the next WCET_C2 units of time. This imitates the execution time of the codel, that may take any positive value inferior to its WCET. Note that for the sake of readability, only one synchronization constraint pertaining to the end of execution is represented in the figure, namely that of $C2$. Also, the default firing intervals (i.e. [0,∞[) are not shown. The components $A1$, $A2$ and $A3$ are represented only partially (the discontinued arcs denote missing parts before/after them). Transitions/places belonging to different components are different even if they have similar names.

b) Automatic Synthesis and Verification: All GenoM3 aspects, including the concurrency one, are formalized in Fiacre [30], a high-level specification language for describing compositionally both the behavioral and timing aspects of concurrent systems. Fiacre is relevant in this context since it derives its expression of timing constraints from TPN. Fiacre models are automatically generated from GenoM3 specifications [22] then automatically compiled into *Time Transition Systems*, TTS for short, namely TPN extended with data. On the resulting TTS, we verify important real-time properties to the robotic programmer using TINA model checkers [31]. The latter rely on model checking techniques where a *States Class Graph* (SCG) is constructed with a possibility to verify the properties on the fly. SCGs are introduced in [32] as finite abstractions to the typically infinite nature of TPN state spaces due to the dense nature of time. One may refer to [33] and [34] for later improvements of the technique. In this section, we give an example of the verified properties, namely the schedulability of execution tasks. In order to do so, we consider the global Fiacre model of the five modules communicating through ports. We generate an *application-specific* client corresponding to a stationary flight which involves ten tasks (all the components shown in Fig. 2 except MANEUVER).

In [22], we suppose that the robotic platform contains as many cores as the GenoM3 tasks involved in the application. Here, we consider the number of cores of the robotic platform. For that matter, our template integrates a cooperative non-deterministic scheduler into the generated model. Results are given for the actual hardware platform, namely the quad-core ODROID-C0 board.

We refer to a periodic execution task as *schedulable* if it never misses its deadline. The problem of verifying the schedulability of tasks is inherently complex in GenoM3 because of the mutual exclusion between codels. While a given

task is executing an activity, it may have to wait when it reaches one of its codels since that codel needs a resource (a field of the IDS or a port) already in use by a codel run by another task (of the same or another component). Hence, verifying the schedulability is more complicated than just summing the WCETs of the codels and comparing the result to the task period. Furthermore, the lack of cores may cause a delay between the period signal and the actual start of the task execution, raising the risk of missing the deadline, i.e. the next period signal.

We formulate schedulability properties as invariants. This allows constructing a coarser SCG that does not preserve firing sequences. Therefore, the verification is time and memory efficient. We prove, in a few to several minutes, that all tasks are schedulable, considering the ODROID-C0 constraints. We note that in case of using a less powerful hardware, e.g. with two cores, only publish (OPTITRACK) remains schedulable. Under a cooperative non-deterministic scheduling policy, we prove that in order to satisfy schedulability properties for all the periodic tasks in the application, a hardware with three cores minimum is indispensable. In all cases, the SCG contains barely ten million states.

Note that the generated model limits the number of active instances to two per activity. While this sounds fair for self-incompatible activities that may not have more than one active instance anyway, it is an additional constraint on self-compatible ones (it does not exist in the underlying system). As soon as this limit is raised to four, the system scales no more. The user should thus be aware that the obtained results exclude the possibility of having four instances of the activity *monitor* (MIKROKOPTER) (the only self-compatible activity of the application). The model does not scale either when we consider a random navigation application that involves all of the five components, i.e thirteen tasks overall.

IV. Discussion

A. On the formalisms

When the theory of TA was introduced in [15], the authors argued that a main advantage of TA compared to other formalisms like Timed Petri Nets [35] is the ability to express the time elapsed in a whole path (not only between taking two successive transitions). This advantage is contrasted with a less obvious, yet equal, compositional expressiveness when compared to TPN with regards to urgencies. Indeed, as seen in section III-A, expressing urgencies in TA is relatively painful in compositional contexts and leads to larger representations compared to those based on TPN. We extrapolate that this was one of the motivations to introduce UA and benefit from both the abovementioned advantage (expressing time easily through a whole execution path) and an easy and efficient expression of urgencies. In [36], the authors corroborate as they describe why UA are needed in many real-world contexts (such as robotics).

B. On the tools

The experience depicted through this paper constitutes a useful feedback on RTD-Finder. RTD-Finder developers are currently investigating the memory blow-up. Consequently, they are developing a new version of the tool where the *linear method* [37] replaces Binary Decision Diagrams. They are also working on extending RTD-Finder to UA to avoid scalability issues with the large TA models. TINA, on the other hand, gives promising results despite the scalability issues encountered when greatly alleviating the assumptions or considering a more complex application than a stationary flight control. It is still important to point out that preemptive scheduling leads immediately to a state-space explosion, a non surprising result with model checking.

V. Conclusion

In this paper, we formalize functional components and automatize such formalization. We develop few templates that bridge G^{en}oM3, our component-based robotic software with Fiacre and RT-BIP, formal languages based on TPN, TA and UA. Important real-time properties are verified on the resulting models. Compared to the related work on formal verification of robotic functional components (section I), our approach is simultaneously automatic (automatically synthesized formal models) and rigorous (all timing constraints considered). Our models remain mostly scalable. Compared to our work in [22], we consider hardware constraints into the model, we extend our templates to another formal framework and share the learned lessons of the whole experience.

We attempt to enrich the formal work presented in [19] with the lessons learned from our experience. For instance, while the authors propose a method to translate TPN to TA, it follows that they rather use TA extended with variables as an output. This is an issue for TA-based tools that do not support variables. Furthermore, each TPN transition is modeled as an (extended) TA with one clock which gives large (extended) TA models. The authors argue that this is not an issue since only clocks of simultaneously enabled transitions are *active*. In a real robotic context, this might be still problematic as dozens of transitions might be simultaneously enabled (e.g. at a given moment, a client might send any of dozens of possible requests, several codels in conflict are ready to execute, etc.). Also, one may question the need of TA when TPN models scale since the main advantage of the translation was the verifiability of quantitative properties with TA tools. Indeed, techniques such as the Fiacre *patterns* [38] emerged since then to express and verify quantitative properties of the style "marking M leads to marking M' within n units of time" using observers on TPN. As robotic programmers, we still did not encounter a case where a property of interest to us could not be expressed within a TPN model. Still, TPN observers induce larger models whereas TA/UA are suitable for *TCTL* which gives a natural way of verifying quantitative properties.

The work in progress to extend RTD-Finder to urgencies will permit us to benefit from its powerful compositional technique to deal with scalability issues. The integration of

urgencies in TA-based verification tools is quite recent. In [36], the authors claim that their work is the first that integrates urgencies to a model checker, namely SGM [39]. The well-known model checking tool UPPAAL [40] allows now *urgent channels*. UPPAAL and SGM rely on model checking techniques. To our knowledge, no non-deterministic-TA-based tool that relies on a deductive approach supports urgencies so far.

On the limitations of this paper, cooperative non-deterministic scheduling is seldom used in reality. We envisage integrating more realistic scheduling policies. These may include cooperative FIFO with fixed priority and pre-emptive policies. For the latter, further techniques will be sought to verify the resulting models as they do not scale with model checking. In this context, Statistical Model Checking tools such as UPPAAL-SMC [41] may be considered. Also, both TINA and RTD-Finder are offline verification tools. Online verification will allow us to monitor our components online and enforce desired properties. RT-BIP UA template has an *execution model* branch that calls the actual codels within the model and runs the components online with the *BIP-Engine*. The latter is currently under development to support real-time multithreaded behaviors.

References

[1] R. Alami, R. Chatilla, S. Fleury, M. Ghallab, and F. Ingrand, "An Architecture for Autonomy," *IJRR*, 1998.

[2] Y. Abdeddaïm, E. Asarin, M. Gallien, F. Ingrand, C. Lesire, and M. Sighireanu, "Planning Robust Temporal Plans: A Comparison Between CBTP and TGA Approaches." in *ICAPS*, 2007.

[3] A. Cimatti, M. Roveri, and P. Bertoli, "Conformant planning via symbolic model checking and heuristic search," *AI*, 2004.

[4] D. Hähnel, W. Burgard, and G. Lakemeyer, "GOLEX—bridging the gap between logic (GOLOG) and a real robot," in *KI Advances in Artificial Intelligence*, 1998.

[5] R. Simmons and C. Pecheur, "Automating Model Checking for Autonomous Systems," in *AAAI Spring Symposium on Real-Time Autonomous Systems*, 2000.

[6] D. Brugali, "Model-Driven Software Engineering in Robotics," *IEEE RAM*, 2015.

[7] A. Elkady and T. Sobh, "Robotics Middleware: A Comprehensive Literature Survey and Attribute-Based Bibliography," *Journal of Robotics*, 2012.

[8] N. Mohamed, J. Al-Jaroodi, and I. Jawhar, "Middleware for Robotics: A Survey," in *RAMECH*, 2008.

[9] B. Espiau, K. Kapellos, and M. Jourdan, "Formal verification in robotics: Why and how?" in *ISRR*, 1996.

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

[11] M. Kim and K. C. Kang, "Formal Construction and Verification of Home Service Robots: A Case Study," in *Automated Technology for Verification and Analysis*, 2005.

[12] G. Brat, E. Denney, D. Giannakopoulou, J. Frank, and A. K. Jónsson, "Verification of autonomous systems for space applications," in *Aerospace Conference, 2006 IEEE*, 2006.

[13] T. Abdellatif, S. Bensalem, J. Combaz, L. de Silva, and F. Ingrand, "Rigorous design of robot software: A formal component-based approach," *Robotics and Autonomous Systems*, 2012.

[14] P. M. Merlin and D. J. Farber, "Recoverability of Communication Protocols: Implications of a Theoretical Study." *IEEE Trans. Comm.*, 1976.

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

[16] B. Bérard, F. Cassez, S. Haddad, D. Lime, and O. H. Roux, "Comparison of the expressiveness of timed automata and time petri nets," in *International Conference on Formal Modeling and Analysis of Timed Systems*. Springer, 2005, pp. 211–225.

[17] B. Berthomieu, F. Peres, and F. Vernadat, "Bridging the gap between timed automata and bounded time petri nets," in *International Conference on Formal Modeling and Analysis of Timed Systems*. Springer, 2006, pp. 82–97.

[18] D. Lime and O. H. Roux, "Model checking of time petri nets using the state class timed automaton," *Discrete Event Dynamic Systems*, vol. 16, no. 2, pp. 179–205, 2006.

[19] F. Cassez and O. H. Roux, "Structural translation from time petri nets to timed automata," *Journal of Systems and Software*, vol. 79, no. 10, pp. 1456–1468, 2006.

[20] A. Mallet, C. Pasteur, M. Herrb, S. Lemaignan, and F. Ingrand, "GenoM3: Building middleware-independent robotic components," in *ICRA*, 2010.

[21] F. Ingrand, S. Lacroix, S. Lemai-Chenevier, and F. Py, "Decisional autonomy of planetary rovers," *JFR*, 2007.

[22] M. Foughali, B. Berthomieu, S. Dal Zilio, F. Ingrand, and A. Mallet, "Model Checking Real-Time Properties on the Functional Layer of Autonomous Robots," in *ICFEM*, 2016.

[23] T. A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine, "Symbolic model checking for real-time systems," in *Logic in Computer Science, 1992. LICS'92., Proceedings of the Seventh Annual IEEE Symposium on*. IEEE, 1992, pp. 394–406.

[24] R. Alur, "Timed automata," in *International Conference on Computer Aided Verification*. Springer, 1999, pp. 8–22.

[25] S. Bornot, J. Sifakis, and S. Tripakis, "Modeling urgency in timed systems," in *Compositionality: the significant difference*. Springer, 1998, pp. 103–129.

[26] A. Basu, M. Bozga, and J. Sifakis, "Modeling Heterogeneous Real-Time Components in BIP," in *SEFM*, 2006.

[27] L. Astefanoaei, S. B. Rayana, S. Bensalem, M. Bozga, and J. Combaz, "Compositional verification for timed systems based on automatic invariant generation," *arXiv preprint arXiv:1506.04879*, 2015.

[28] S. Bensalem, M. Bozga, T.-H. Nguyen, and J. Sifakis, "D-finder: A tool for compositional deadlock detection and verification," in *International Conference on Computer Aided Verification*. Springer, 2009, pp. 614–619.

[29] M. Pezze and M. Young, "Time petri nets: A primer introduction," in *Tutorial presented at the Multi-Workshop on Formal Methods in Performance Evaluation and Applications, Zaragoza, Spain*, 1999.

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

[31] B. Berthomieu, P. O. Ribet, and F. Vernadat, "The tool TINA - Construction of abstract state spaces for Petri nets and Time Petri," *International Journal of Production Research*, 2004.

[32] B. Berthomieu and M. Menasche, "An enumerative approach for analyzing time Petri nets." *IFIP Cong. Series*, vol. 9, pp. 41–46, 1983.

[33] B. Berthomieu and M. Diaz, "Modeling and verification of time dependent systems using time petri nets," *IEEE transactions on software engineering*, vol. 17, no. 3, pp. 259–273, 1991.

[34] J. Lilius, "Efficient state space search for time petri nets," *Electronic Notes in Theoretical Computer Science*, vol. 18, pp. 113–133, 1998.

[35] C. Ramchandani, "Analysis of asynchronous concurrent systems by petri nets," DTIC Document, Tech. Rep., 1974.

[36] P.-A. Hsiung, S.-W. Lin, Y.-R. Chen, C.-H. Huang, J.-J. Yeh, H.-Y. Sun, C.-S. Lin, and H.-W. Liao, "Model checking timed systems with urgencies," in *International Symposium on Automated Technology for Verification and Analysis*. Springer, 2006, pp. 67–81.

[37] S. Bensalem, M. Bozga, B. Boyer, and A. Legay, "Incremental generation of linear invariants for component-based systems," in *Application of Concurrency to System Design (ACSD), 2013 13th International Conference on*. IEEE, 2013, pp. 80–89.

[38] N. Abid, S. Dal Zilio, and D. Le Botlan, "Real-time specification patterns and tools," in *International Workshop on Formal Methods for Industrial Critical Systems*. Springer, 2012, pp. 1–15.

[39] F. Wang and P.-A. Hsiung, "Efficient and user-friendly verification," *IEEE Transactions on Computers*, vol. 51, no. 1, pp. 61–83, 2002.

[40] K. G. Larsen, P. Pettersson, and W. Yi, "Uppaal in a nutshell," *International journal on software tools for technology transfer*, vol. 1, no. 1-2, pp. 134–152, 1997.

[41] P. Bulychev, A. David, K. G. Larsen, M. Mikučionis, D. B. Poulsen, A. Legay, and Z. Wang, "Uppaal-smc: Statistical model checking for priced timed automata," *arXiv preprint arXiv:1207.1272*, 2012.