Runtime Verification of Timed Properties in Autonomous Robots

Mohammed Foughali, Saddek Bensalem, Jacques Combaz, Félix Ingrand

Runtime Verification of Timed Properties in Autonomous Robots

Mohammed Foughali*†, Saddek Bensalem*, Jacques Combaz*, Félix Ingrand‡
*Université Grenoble Alpes, VERIMAG, Grenoble, France
† Corresponding author (mohammed.foughali@univ-grenoble-alpes.fr)
‡ LAAS-CNRS, Université de Toulouse, Toulouse, France

***Abstract*—Throughout the last few decades, researchers and practitioners are showing more and more interest in using formal methods in order to predict and prevent software failures in robotic and autonomous systems. However, the applicability of formal methods to such systems is limited due to several factors. For instance, robotic specifications are often non-formal which makes their formalization hard and error prone, and their translation into formal models ad-hoc and non automatic. Furthermore, the complexity and size of robotic applications lead most often to scalability issues with exhaustive techniques such as model checking. In this paper, we investigate the use of runtime verification as an alternative to model checking for the rigorous verification of large robotic systems. To do so, we first develop a sound and automatic translation from the robotic framework GenoM3 to the real-time version of the BIP formal language. Then, we apply the translation to a real-world case study the formal models of which do not scale with model checking, and use the BIP Engine to execute the generated BIP model, verify properties online, and adequately react to their possible violation. The experiments are carried out on a real Robotnik robot and show the efficiency of our approach in verifying timed properties, that is when the amount of time separating events is important.**

I. Introduction

a) Formal Methods and Robotics: Robotic software is systematically tested, both on the field and via simulators (*e.g.* Gazebo [26]). Such tests often fail, however, to rise to the level of guarantees required to safely deploy robotic systems in costly missions (*e.g.* space exploration) and applications involving direct contact with humans (*e.g.* home assistants). The authors of [27], for instance, report on a software bug that, while never occurred during thousands of hours of simulations and over 450 km of field tests, disqualified the autonomous vehicle *Alice* from the 2007 *Defense Advanced Research Projects Agency (DARPA)* urban challenge.

Formal methods are seemingly a promising alternative. However, in order to propose robust solutions based on such methods, one needs first to face the disparate nature of modern robotic software. Indeed, the latter, characterized by some level of autonomy, is classically *component based*, with components belonging to either the *decisional* (high-level) layer, in charge of deliberative functions (*e.g.* planning and learning [21], [22]) or the *functional* (low-level) layer, directly in charge of sensors and actuators [37]. The level of integration of formal methods in robotic software differs according to these two layers (a detailed overview is given in [15, Sect. 1]). On the one hand, formal verification is common at the decisional layer, where most models (with the notable exception of learning models) have formal, unambiguous semantics. On the other hand, bridging functional components with formal methods is particularly hard as such components are classically developed and deployed via non-formal frameworks (*e.g.* ROS [36]). Furthermore, the simplest robotic applications today involve several functional components that use complex interaction mechanisms and are subject to various timing constraints, which leads to scalability issues with exhaustive techniques like model checking. Non-exhaustive techniques, such as *Statistical Model Checking (SMC)* and *Runtime Verification (RV)*, are promising scalable alternatives, but their use in robotics is still limited. For instance, most of the works on RV in robotics (i) mix formal *monitors* with non-formal robotic specifications, which makes it hard to trust the overall execution, and/or (ii) do not consider *timed properties*, where the time separating events is crucially important (Sect. V).

b) Contributions: In this paper, we focus on the functional layer and propose a rigorous solution to the above issues. Firstly, we develop a translation from the robotic framework GenoM3 to the real-time formal language BIP, that is (i) sound: formal models are faithful to their underlying robotic specifications and (ii) automatic: the BIP model of any GenoM3 specification may be generated automatically. Secondly, we generate the BIP model of a real case study that does not scale with offline model checking. The generated BIP model, augmented with a timed-property monitor, is executed using the *BIP Engine*. Such execution allows to (i) verify the timed property at runtime and (ii) efficiently react to its failure.

c) Outline: The rest of this paper is organized as follows. In Sect. II, we present the formal language BIP, the robotic framework GenoM3 and the case study. Afterwards, Sect. III shows how we soundly and automatically translate any GenoM3 specification into a BIP model. We then apply our translation to the case study and show how we use our approach to verify and react to the possible violation of a crucial timed property on the real Robotnik platform (Sect. IV). Finally, we explore the related work in Sect. V and conclude with possible future work (Sect. VI).

II. Preliminaries

A. BIP

In this paper, we use RT-BIP [4] (the Real-Time version of BIP [6]) and refer to it simply as BIP. BIP is a component-based language for modeling, executing and analyzing real-time systems. A system is represented by a set of components (**B**ehavior) synchronized through *connectors* which define **I**nteractions. Conflicts between interactions may be resolved using **P**riorities. Complex systems can be built using *compound* components, which encapsulate sub-systems made of components constrained with interactions and priorities. The underlying formalism of BIP is **T**imed **A**utomata [5] extended with **D**ata and **U**rgencies [8], which we refer to as **DUTA**.

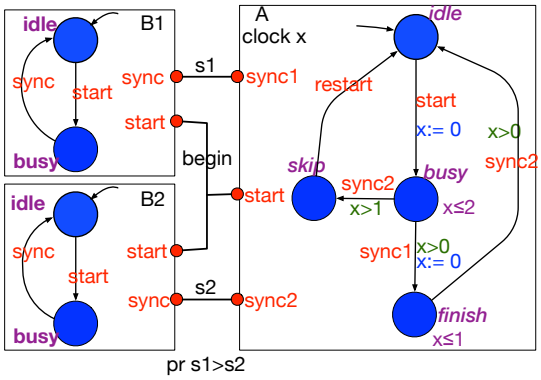

Fig. 1: A BIP example (graphical)

Atom components are the simplest type of components, *i.e.* with no hierarchies. An atom component is a DUTA: it is a timed automaton that may (i) operate on a set of local data variables and (ii) have *eager* edges, denoted $\wr$ (see below). *Ports*, which label edges, may be *exported* to interact with other components and are thus the building blocks of connectors, the only means of communication between components (shared variables are not allowed). Connectors may be *rendezvous* (strong synchronizations) or *broadcasts* (weak synchronizations) that can both be multiparty (involve any number of ports). Each connector defines a set of possible interactions. An interaction involving at least one $\wr$ edge is urgent and must thus be taken (or disabled by taking a concurrent interaction in the system) as soon as enabled. In the remainder of this paper, graphical representations of BIP models are provided for illustration purposes (BIP does not provide a graphical interface). In such representations, atom components are DUTA (initial locations recognized through sourceless incoming edges, locations names and invariants in purple, guards in green, operations (updates) in blue, and notations are generic *e.g.* $=$ for equality and $:=$ for assignment).

Example: Fig. 1 shows a simple example of three BIP atom components interacting through connectors with priority rules. Edges are labeled with ports (*e.g.* the edge from location *idle* to location *busy* in component A is labeled with the port *start*), that may be exported (*e.g.* the port *sync* of component $B1$ is exported for strong synchronization, denoted by the little red circle), or not (*e.g.* port *restart* of component A). Exported ports build connectors (*e.g. s1* is a *rendezvous* connector that involves the ports *sync1* of A and *sync* of $B1$). The priority pr denotes that the interactions through $s1$ have a higher priority than those possible through $s2$.

In the textual specification, we first define the types of ports and connectors (listing 1). Here, ports are basic without parameters (line 2). The *define* keyword is used to specify the possible interactions through the connector (broadcast or rendezvous, line 5). If *c_sync2* was a broadcast connector type, p' would be the way to define p as the sender.

Then, the atom types are defined. We show only the atom type a for component A, in listing 2. The keyword *provided* (*e.g.* line 19) is for guards and *do* (*e.g.* line 20) for operations (in a C-like notation). Line 11 specifies the initial location *idle* (together with this statement, initialization of local variables

```
1       /* port types */
2       port type Basic()
3       /* connector types */
4       connector type c_sync (Basic p, Basic q)
5       define p q
6       end
7       connector type c_sync2 (Basic p, Basic q, Basic r)
8       define p q r
9       end
```

Listing 1: Ports/Connectors types for the BIP example in Fig. 1

may be performed, example in Sect. III-B2). After specifying the edges, invariants are defined (lines 34-35).

```
1  /* atoms types */
2  atom type a()
3  clock x
4  port Basic restart()
5  export port Basic sync1()
6  export port Basic sync2()
7  export port Basic start()

9  state idle, busy, finish,
       skip

11 initial to idle

13 on start
14 from idle to busy
15 do {x= 0;}

17 on sync1
18 from busy to finish
19 provided (x>0)
20 do {x= 0;}

22 on sync2
23 from busy to skip
24 provided (x>1)
25 do {x= 0;}

27 on restart
28 from skip to idle

30 on sync2
31 from finish to idle
32 provided (x>0)

34 invariant inv1 at busy
       provided (x ≤2)
35 invariant inv2 at finish
       provided (x ≤1)
36 end
```

Listing 2: Atom type a for component A in Fig. 1

Finally, in listing 3, we build the compound component (line 2) by instantiating the atom components (lines 3-4) and the connectors (lines 6 to 8) and defining the priority pr (line 10). The ": $*$" after the names of connectors *s1* and *s2* (line 10) denotes all the possible interactions. Thus, pr states that any possible interaction through *s1* has a higher priority than any possible interaction through *s2*.

As stated in the beginning of this section, priorities and interactions restrict the possible behaviors of components. In this example, for instance, there is no reachable global state where the current location of A is *busy* and that of *B1* (or *B2*) is *idle* (because of the sole interaction forced by the connector *begin*). Similarly, location *skip* of A is unreachable (because of the application of priority *pr*).

1) The BIP Engine: The back-end compiler of BIP generates source code in C++ for execution purposes. The BIP Engine ensures a correct execution of the generated source code following the semantics of BIP. Furthermore, concrete C/C++

```
1       /* compound definition */
2       compound type example()
3       component a A()
4       component b B1(), B2()
5       /* connectors */
6       connector c_sync s1(A1.sync1, B1.sync)
7       connector c_sync s2(A1.sync2, B2.sync)
8       connector c_sync2 begin(A1.start, B1.start, B2.start)
9       /* priorities */
10      priority pr s1:*>s2:*
```

Listing 3: Building the compound of the example in Fig. 1

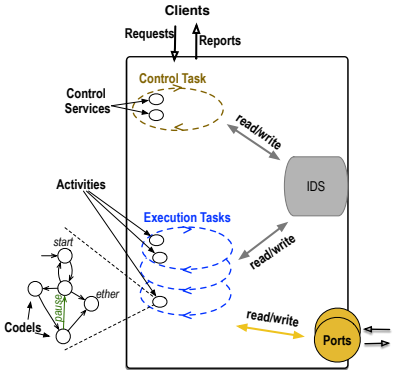

Fig. 2: A generic GenoM3 component

code can be executed together with interactions by using the same keyword *do* on edges or connectors (examples in Sect.III-B4). Therefore, the BIP Engine is a faithful, formally founded executer of BIP models, possibly enriched with C or C++ code. In addition, one may, using the Engine (i) monitor the execution online with regard to *e.g.* timing constraints and (ii) react to the violation of such constraints at runtime.

B. GenoM3

GenoM3 [31] is a framework for specifying and implementing robotic functional components. A component is typically assigned one functionality *e.g.* processing a sensor data or computing an actuator command. A GenoM3 component is organized as shown in Fig. 2. *Activities*, executed following *requests* from external *clients*, implement the core algorithms of the component. Two types of tasks are therefore provided: (i) a *control task* to *process* requests, *validate* the requested activity (if the processing returns no errors) and *report* to the clients and (ii) *execution task(s)* to execute activities. Tasks (resp. components) share data in the *Internal Data Structure IDS* (resp. *ports*). For the sake of simplicity, we omit in this paper *control services* and aperiodic execution tasks (for more details, we provide complete semantics in [20], [18]).

```
1 activity Measure()
2 {
3  doc   "Read a measure and publish in the <<IMU>> port.";
4  codel <start>  MsStart(port out IMU) yield get, ether
       wcet 1ms;
5  codel <get> GetImu(inout imu_driver, out imu, out
       timestamp) yield write_p, ether wcet 2ms;
6  codel <write_p> WritePort(in imu, port out IMU) yield
       pause::get, ether wcet 1.5ms;
7  codel <stop> MeasureStop() yield ether wcet 2ms;
8  interrupts  Measure;
9  task  update;
10 };
```

Listing 4: Activity *Measure* (comp. IMUDRIVER, Fig. 3)

1) Behavior: Below is a description of a component behavior, supported with the specification of activity *Measure* (listing 4) of component IMUDRIVER of our case study (Sect. II-B3). A less informal explanation (using DUTA) is given in Sect. III.
Activities. An activity is a *finite-state machine* FSM executed by the execution task it specifies (*e.g.* line 9 specifies that *Measure* is executed by the task update). An activity may need to *interrupt* other activities before executing (*e.g.* line 8 specifies that activity *Measure* interrupts itself, that is new instances interrupt currently running ones).
An FSM defines the activity behavior through *codels* and *transitions*. Except for some restrictions (*e.g.* mandatory `start` and `ether` codels, see below), the programmer is free to define the FSM behavior according to their needs (the FSM shown in Fig. 2 is a "generic" example that does not impose any behavioral model). A *codel* is a state at which a piece of C/C++ code is executed (by abuse of terminology, we say the codel is executed). A codel specifies (i) the IDS/ports data its execution requires (*e.g.* codel `write_p` reads the IDS field *imu* and writes the port **IMU**, line 6) and (ii) the possible *transitions* subsequent to its execution using the keyword *yield* (*e.g.* executing codel `get` returns either codel `write_p` or codel `ether`, line 5). Taking a transition labeled "pause" *pauses* the activity until the next period of its execution task (*e.g.* taking `pause::get` pauses *Measure* at codel `get` until the next period of task update, line 6). A codel may (optionally) specify a WCET, namely its worst case execution time on a given platform (*e.g.* `start` has a WCET of 1 ms, line 4). Any activity FSM has always the codels `start` (entry point) and `ether` (end point with no successors and no code attached to it, this codel is not specified by the user). When the latter is reached, the activity is *terminated*. The codel `stop`, if exists, is executed when the activity is interrupted (*e.g.* line 7).
Control task. The control task processes requests (resp. sends reports) from (resp. to) clients, and manages validation and interruption of activities. If a request for activity A is processed with no errors, A is validated, and its execution task is instructed to execute it after interrupting (executing `stop` codels) and terminating (reaching `ether` codels) all the activities A needs to interrupt. Upon completion of any activity, the control task sends a report to the client that initially requested it.
Execution tasks. With each period, an execution task runs, sequentially, all the activities it is in charge of, that were previously validated by the control task. The execution of an activity ends when the latter is paused or terminated. In the former case, the activity is resumed at the next period.
IDS, ports & concurrency. GenoM3 tasks are run as parallel threads at the OS level, with fine-grain concurrent access to IDS/ports data: a codel (in its activity, run by a task) locks only the IDS field(s) and/or port(s) required for its execution. A codel *in conflict* (cannot execute at the same time) with some other codel(s) because of this locking mechanism is called *thread unsafe TU* (*thread safe TS* otherwise). By default, `ether` codels are all thread safe (no code attached to them). Because of the concurrency over ports, codels in conflict may belong to different components (example in Sect. III-B1c).
2) Templates: GenoM3 provides *templates* for automatic generation purposes. A template may access all component information (*e.g.* tasks periods, activities FSM) and generate text files with no restrictions (example in Sect. III-B5). For example, the "implementation" templates generate glue code for a middleware (*e.g.* PocoLibs [2], ROS-Comm [36]). Such templates are extended so that codels execution time, average and WCET, is reported upon completion of components execution.

3) Case study: Our case study involves a robot equipped with a Laser Range Finder LRF for laser-based potential-field navigation [23]. The GenoM3 specification includes eight components (Fig. 3). Each box corresponds to a component, and octagons are ports written by the components they are attached to and read by other components through an arrow. The components run on the real *Robotnik* robot and also in simulation using Gazebo. Next, we give a high-level description of components roles. LASERDRIVER, GPSDRIVER, and IMUDRIVER read the LRF, the Global Positioning System GPS and the Inertial Measurement Unit IMU sensors and update respectively ports **Laser**, **GPS** and **IMU**. ROBOTDRIVER handles the communication between ports **Odometry** and **Cmd** and the Robotnik ROS topics. POM reads data from ports **Odometry**, **IMU** and **GPS** to which it applies an Unscented Kalman Filter UKF to produce an estimated position in port **Pose**. NAVIGATION computes, given the current position from **Pose**, intermediate positions to reach a goal position, which it writes to port **Target**. POTENTIALFIELD applies the potential-field navigation algorithm (using information on the goal position (port **Target**), the current position (port **Pose**), and the obstacles (port **Laser**)), and updates the robot velocity accordingly in port **PFCmd**. Finally, SAFETYPILOT uses the velocity (port **PFCmd**), together with the laser perception (port **Laser**), to compute through the activity *StopIfObstacle* a safe command that it writes to port **Cmd**. Component POM, running both its tasks at 100 Hz, has the highest frequency.

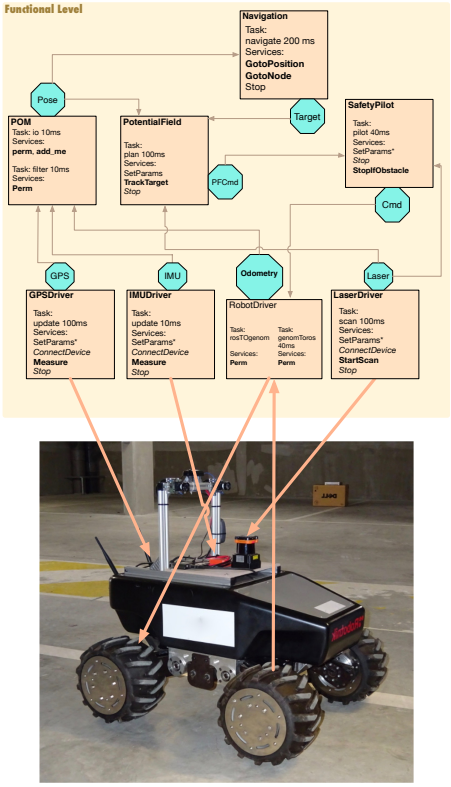

Fig. 3: Autonomous navigation case study.

III. GenoM3 TO BIP

In this section, we show how GenoM3 specifications are translated into BIP in a sound and automatic manner. Firstly, we explain the need of such translation, as well as the choice of BIP as a target language (Sect. III-A). Then, we present the translation and its automatization (Sect III-B).

A. Motivation

GenoM3 already provides automatic translations to the model checkers TINA and UPPAAL, as well as UPPAAL-SMC, the statistical extension of UPPAAL, which were used to obtain encouraging results on a couple of case studies [17], [19], [20]. However, the case study we consider in this paper (Sect. II-B3) scales neither with TINA nor with UPPAAL, and using UPPAAL-SMC leaves some *confidence* problems open (Sect. V). A key contribution of this paper is to use RV as an alternative to tackle such scalability issues (Sect. I). Yet, neither UPPAAL nor TINA provide an RV environment *per se*, able to run code on a real robotic platform. Thus, we choose the BIP language as to be able to use its powerful Engine (Sect. II-A1) for RV purposes. Additionally, instead of adding (formal) BIP monitors, of the desired properties to verify, to (non formal) GenoM3 specifications, we choose to fully translate GenoM3 specifications into BIP in a correct and automatic manner, augment them with monitors to verify the desired properties, and delegate both the execution and RV to the BIP Engine. This method (i) allows the robotic programmer to obtain formal executable models, faithful to their robotic specifications, with no effort (ii) increases our trust in the deployed models, as they are fully handled through a formal and rigorous execution Engine and (iii) reduces the resource usage as compared to methods mixing non formal specifications with formal monitors (Sect. V).

B. Translation

In [20], [18], we propose semantics for GenoM3 and a sound translation of such semantics to DUTA. In [20], we further derive DUTA implementation models, as restricted by the robotic middleware, then map them to UPPAAL to get a sound and automatic translation from GenoM3 to UPPAAL. In this paper, we use directly the DUTA implementation models (already proven faithful to their GenoM3 counterpart) and show how we map them automatically to BIP to obtain a correct and automatic translation from GenoM3 to BIP.

1) DUTA implementation: We define the DUTA implementation of a generic GenoM3 component in a top-down fashion, from the component to the activities. To simplify the presentation, the control task is not represented and a `stop` codel (resp. WCET W_c) is mandatorily defined for each activity (resp. each codel `c` excluding `ether`) in the specification. In the remainder of this paper (i) $\{\Theta\}[|||_{i\in 1..n}\ P_i]$ denotes the parallel composition of n DUTA over a set of shared variables U_s the initial valuations of which is given by the function Θ and (ii) μ is the function that returns for each TU codel `c` the names of all codels in conflict with `c`.

a) Component: A component $Comp$ containing n execution tasks T_i is the parallel composition $Comp = \{\Theta\}[|||_{i\in 1..n}\ T_i]$. The set of shared variables U_s contains one Boolean r_c for each TU codel `c` (in each activity in each T_i in $Comp$), initially false ($\Theta(r\_c) = False$ for all $r\_c \in U_s$). These Booleans

allow implementing correctly the concurrency between all TU codels in the component (see the example in Sect. III-B1c).

b) Execution tasks: An execution task T is the parallel composition $T = \{\Theta\}[Tim||M||(\|_{i\in 1..m}\ A_i)]$ where Tim is the *timer*, M the task *manager*, and $\|_{i\in 1..m}\ A_i$ the composition of all the m activities T is in charge of. U_s contains:

- sig: a Boolean (for period signal), initially False,
- run: an array of m cells, starting at index 0. Each cell is a record of two fields: an activity "name" a and its "status" s. The latter may be $void$ (activity already terminated or not requested), $busy$ (activity being executed nominally) or $inter$ (activity being interrupted). Thus, each cell of run of index i holds the name of activity A_{i+1} and its current status (initially $void$, *i.e.* $\forall i \in 0..m-1 : \Theta(run[i].s) = void$),
- Π: (the control passing variable with $\Theta(\Pi) = M$) is either equal to M or any activity name A,
- i: an integer (search index) in $0..m$, initially zero.

We show each of the DUTA involved in the composition. We first give definitions/illustrations, then exemplify.

We start with the models of Tim (the timer, Fig. 4) and M (the manager, Fig. 5), both generic (parameterized with the period value Per). There are two functions used by M: (1) $update(run)$ updates the status fields of run according to the instructions of the control task (not represented here), (2) $next(run, i)$ browses run, starting from i, and returns the index of the first cell satisfying $s \neq void$ ($|run|$, *i.e.* m, if such an element does not exist or $i = m$). For activities, whose behavior is mostly user defined (Sect. II-B1), we give a definition to derive the DUTA implementation from the robotic specification. Because of mutual exclusion, the name of the activity is added as a subscript to each of its codels names.

Definition 1. ***Activity*** *A. The DUTA model of an activity A is mapped from the latter's specification as follows:*
(1) Clocks X: the DUTA has one clock $x \in X$,
(2) Locations L: each codel c *is mapped to a location $c \in L$. A location $c_{exec} \in L$ (resp. $c_{pause} \in L$) is added for each TU codel (resp. TS codel targeted by a pause transition)* c*. The initial location is $ether_A$,*
(3) Edges $E = E^N \cup E^A$ are either nominal or additional:

- *Nominal edges E^N: each non-pause transition from* c *to* c' *in the specification is mapped to an edge $c \to c'$ (resp. $c_{exec} \to c'$) in E^N if codel* c *is TS (resp. TU). Each pause transition from* c *to* c' *in the specification is mapped to a pause edge $c \to c''$ (resp. $c_{exec} \to c''$) in E^N if codel* c *is TS (resp. TU) where $c'' = c'$ (resp. $c'' = c'_{pause}$) if codel* c' *is TU (resp. TS). There are three disjoint sets in E^N: $E^N = E^P \cup E^T \cup E^X$. E^P is the set of pause edges, E^T the set of termination edges of the form $\to ether_A$ and E^X the set of the remaining (execution) edges.*
- *Additional edges $E^A = E^S \cup E^I \cup E^M \cup E^R$ where E^S contains the starting edge $ether_A \to start_A$, E^I the interruption edges from $ether_A$ and each location targeted by a pause edge (in E^P) to $stop_A$, and E^M the mutual exclusion edges $c \to c_{exec}$ (resp. E^R the resume edges $c_{pause} \to c$) for each pair of locations $\{c, c_{exec}\}$ (resp. $\{c, c_{pause}\}$) in 2^L. All edges in E^A are ⟆*

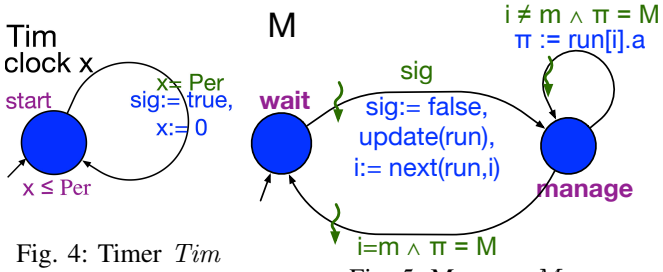

Fig. 4: Timer Tim

Fig. 5: Manager M

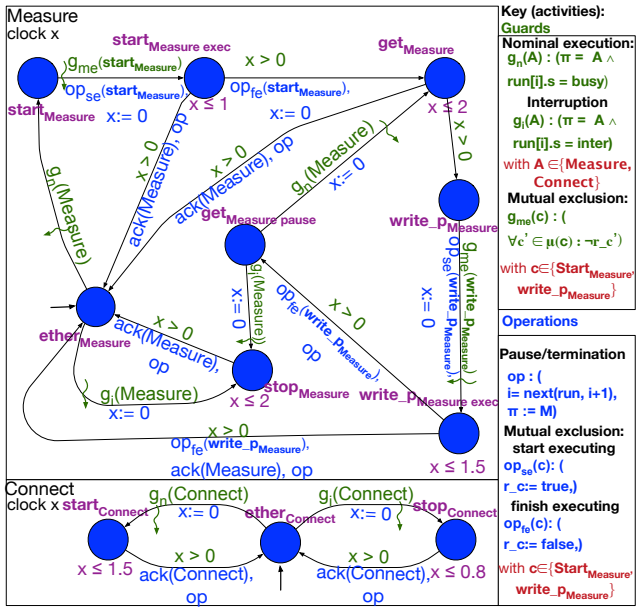

Fig. 6: DUTA implementation of activities *Measure* and *Connect* in task update (component IMUDRIVER, Fig. 3).

Guards, invariants and operations:
(4) Each location c (resp. c_{exec}) mapping a TS (resp. TU) codel c *is associated with the invariant $x \leq W_c$,*
(5) Each incoming (resp. outgoing) edge of a location in L that is associated with an invariant is augmented with the operation (resp. guard) $x := 0$ (resp. $x > 0$),
(6) Each edge in $E^T \cup E^P$ is augmented with the operations $\Pi := M, i := next(run, i + 1)$,
(7) Each edge in $E^S \cup E^I \cup E^R$, as well as each edge $c \to c_{exec}$ in E^M such that c is targeted by a pause edge (in E^P), is guarded with $\Pi = A$,
(8) Each edge in E^M is augmented with the operation $r\_c := true$ (see shared variables in Sect. III-B1a).
(9) Finally, (i) the guard of each edge in E^M is conjuncted with the expression $\forall c' \in \mu(c) : \neg r\_c'$ and (ii) $r\_c := false$ is added to the operations of each edge $c_{exec} \to$ in E^N.

Nominal edges map the underlying activity transitions explicitly specified by the programmer (*e.g.* transitions in listing 4). In contrast, additional edges enforce internal GenoM3 actions given by its semantics: starting (E^S), concurrency (E^M), resuming after a pause at a TS codel (E^R), as well as interruptions (E^I). The set E^R relates to TS codels only (it is empty if there is no TS codel targeted by a pause transition

in the specification). Indeed, a TU codel c is already mapped into two locations c (invariant free) and c_{exec}, connected with an edge $c \rightarrow c_{exec}$ in E^M, guarded with resource availability (rules (2), (3), and (9)). Thus, if codel c is targeted by a pause transition in the specification, resuming after a pause will correspond to taking $c \rightarrow c_{exec}$ in E^M. This explains why edges $c \rightarrow c_{exec}$ in E^M such that c is targeted by a pause edge in E^P need the clause $\Pi = A$ in their guards (rule (7)). If c is a TS codel targeted by a pause (*e.g.* codel get in activity *Measure*, see the latter's DUTA in Sect. III-B1c), an invariant-free location c_{pause} is necessary (rule (2)) to wait until the control through Π is back (guard on the edge $c_{pause} \rightarrow c$ in E^R, rule (7)), otherwise a timelock might occur (details in [18] and [20]). The remaining aspects of *Definition 1* are best clarified via a practical example (Sect. III-B1c).

c) Example: Let us illustrate through an example how activities evolve following the DUTA implementation, and how this coincides with the behavior in Sect. II-B1. We consider again the component IMUDRIVER that has one execution task update in charge of two activities: *Measure* (already shown in listing 4, Sect. II-B) and *Connect* (listing 5). All codels are TS except for start and write_p (activity *Measure*) which use the port **IMU** concurrently with codels in component POM (Fig. 3). We apply *Definition 1* to get the DUTA implementation of both activities (Fig. 6, a key is provided for readability). M and Tim of task update are retrieved from Fig. 5 and Fig. 4, respectively (by replacing Per with 10 ms, the period of task update). In the following, we explain the DUTA behavior with all references made to figures 4, 5 and 6.

```
1 activity Connect()
2 {
3   doc  "Connect the serial line to the hardware device";
4   codel <start>  start_device(inout imu_driver, in
        device, in baudrate, out device_open) yield ether
        wcet 1.5 ms;
5   codel <stop>  stop_device() yield ether wcet 0.8 ms;
6   interrupts  Measure, Connect;
7 };
```

Listing 5: Activity *Connect* (comp. IMUDRIVER, Fig. 3)

Let us start with the communication between M and Tim. At exactly each period (ensured by the invariant $x \leq Per$ and the guard $x = Per$), Tim transmits, through Boolean sig, a signal to M. M has two locations: *wait* (to wait for the signal) and *manage* (to execute activities, if any). As soon as (an urgency enforced with an eager $\wr$ edge) sig is true, M updates the activities statuses in array run according to the control task instructions ($update(run)$), computes the identity of the next activity to execute ($next(run, i)$), and transits to *manage*.

M and the activities communicate to achieve a sequential execution within task update. If there is still at least one activity to execute in this period ($i \neq m$, that is an activity that was validated by the control task, still did not terminate and still is not executed in this period), M gives it the control ($\Pi := run[i].a$ on the edge $manage \rightarrow manage$). Such activity will then execute until it is terminated or paused, where it computes the identity of the next activity to execute and gives the control back to M (operation op). And so, i moves through run as the control switches between M and the activities to execute until all are paused or terminated ($i = m$). M transits then back to *wait* and awaits the next period.

Now, if an activity has the control (through Π), it is, depending on its status, either (1) executed nominally, by taking a starting edge (in E^S, *Definition 1*, *e.g.* $ether_{Connect}$ to $start_{Connect}$) or resuming after a pause (*e.g.* $get_{Measure\ pause}$ to $get_{Measure}$ in E^R) or (2) interrupted by taking an $\wr$ interruption edge (in E^I) to the *stop* location (*e.g.* $ether_{Connect}$ to $stop_{Connect}$). The execution ends with a termination (taking an edge in E^T, *e.g.* $get_{Measure} \rightarrow ether_{Measure}$) or a pause (taking an edge in E^P, *e.g.* $write\_p_{Measure\ exec} \rightarrow get_{Measure\ pause}$). In the former case, the activity acknowledges the control task (not shown here) to update its status (*e.g.* operation $ack(Measure)$).

Finally, each codel (except $ether$) location c (c_{exec} if c is TU) is associated with the invariant $x \leq W_c$, and its outgoing edges are guarded with $x > 0$ to emulate a non-zero execution time inferior to the WCET of codel c (*e.g.* location $start_{Connect}$). Eager edges in E^M (*e.g.* $start_{Measure} \rightarrow start_{Measure\ exec}$) ensure through the guard $\forall c' \in \mu(c) : \neg r\_c'$ that codel c starts executing as soon as none of the codels c' in conflict with c is currently executing (at location c'_{exec}). Similarly, operation $r\_c := false$ on the outgoing edges of locations c_{exec} allow codels in conflict with c to capture the end of the latter's execution (*e.g.* $op_{fe}(write\_p_{Measure})$).

2) DUTA implementation to BIP: At this step, we need to correctly map the DUTA implementation (faithful to the GenoM3 semantics [18], [20]) to BIP. The main difficulty here is the fact that BIP does not use shared variables and rather relies on local variables and connectors. When taking a rendezvous interaction involving n ports p_1 ... p_n, the operations on the edges labeled with such ports are executed one after another in one atomic sequence. For instance, in listing 7, when the only possible interaction in connector rv is taken (line 5), the operations associated with the edges labeled with ports mm (in M), p (in $Measure$) and p (in $Connect$) are executed one after another (in no predefined order). Within this sequence, value-passing operations (using the keywords up and $down$) may be inserted to propagate the changes made to local variables in one component to variables in other components in a typical data flow *à la process calculi*. For readability and space, we present a generic solution where value-passing is implicit (without up and $down$ operations). Such solution entails (i) duplicating shared variables within each DUTA as local ones and (ii) adding rendezvous connectors to propagate any change made to a local variable in one DUTA to the local variables having the same name in all other DUTA.

Let us illustrate with an example. In the DUTA implementation, M and the activities *Measure* and *Connect* share the variables run, i and Π (Figures 5 and 6). We proceed as described in the generic solution above. Firstly, each of the atom components M, *Measure* and *Connect* will have run, i and Π as local variables, that we make sure are equal at the initial global state of the system. For that, we simply initialize each local duplicate variable v (within the "*initial to*" statement of its BIP atom component) of each global variable $v \in U_s$ (in the original DUTA composition) to $\Theta(v)$ (Sect. III-B1). Listing 6 shows how we guarantee i is equal to 0 in all atom components M, $Measure$ and $Connect$ (of

types ma, $measure$ and $connect$, respectively) at the initial global state of the system.

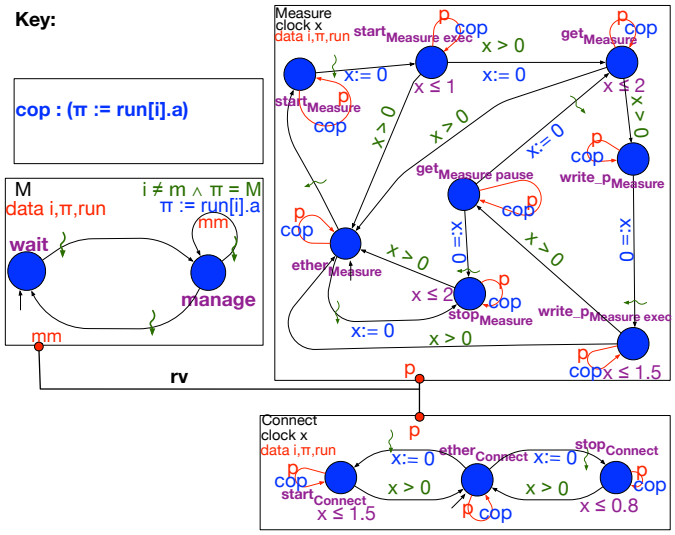

Fig. 7: Partial BIP model of update (IMUDRIVER, Fig. 3).

```
1  atom type ma()
2  data int i
3  ...
4  state start, manage

6  initial to start
7  do {i=0;}
8  ...
9  end

11 atom type measure()
12 data int i
13 ...
14 state ether, start, ...

16 initial to ether
17 do {i=0;}
18 ...
19 end

21 atom type connect()
22 data int i
23 ...
24 state start, ether, stop

26 initial to ether
27 do {i=0;}
28 ...
29 end
```

Listing 6: Correct initialization of i (local) variables

Secondly, side effects of operations involving (originally) shared variables are propagated. For simplicity, we show how this is done for only one operation affecting one shared variable, namely that on the self-loop at *manage* (in M, Fig. 5) affecting Π. Fig. 7 shows a partial BIP-implementable model were all remaining edges are stripped from their shared-variable-dependent guards and operations, as well as their labeling ports, for readability. To capture the said side effects (of the edge $manage \rightarrow manage$ in M) no matter what the current locations of *Measure* and *Connect* are, self-loop edges (in red) are added on all locations in each of these activities atom components. Each of the added edges is associated with the operation cop, which copies the operations the effects of which we want to propagate (those associated with $manage \rightarrow manage$, labeled mm, in M). Finally, each port p is exported to form a rendezvous connector rv with port mm. Now, when the guard on $manage \rightarrow manage$ is true, the only interaction of rv (the rendezvous between mm and each p) is enabled and taken urgently (since one of the involved edges, *i.e.* that labeled with mm, is $\lozenge$), which results in executing the operations on the edges labeled with mm and each p one after another in one atomic sequence. Therefore, taking such interaction, all local Π variables are equal to $run[i].a$ (with run and i guaranteed to be equal in all components using the same initialization and propagation mechanisms).

Consequently, combining the proper initialization (as shown in listing 6 for variables i) and propagation of side effects of each operation (as shown for that on $manage \rightarrow manage$ in Fig. 7), we guarantee that, at any global state of the system, local duplicates (in different atom components) of an originally shared variable (in the underlying DUTA implementation) have equal values. Listing 7 shows how to declare the type of connector rv (Fig. 7) and the instantiation code in BIP, given that ports p and components M, $Measure$ and $Connect$ are previously instantiated from their types (see definition of atom/port types in Sect. II-A).

```
1      connector type rv (Basic mm, Basic p, Basic q)
2      define mm p q
3      end
4      ...
5      connector rv rv(M.mm, Measure.p, Connect.p)
```

Listing 7: Connector rv (Fig. 7) type and instantiation

3) Soundness: The mechanism shown above (correct initialization + side-effect propagation) guarantees a strict equivalence between the DUTA implementation and its BIP counterpart. This may be proven using timed bisimulation in the style of [18] (with the difference that it is a strong bisimulation here). For readability and space, we only give a sketch of such proof, restricted to one component with one execution task.

Hereafter, an *action* is a time progress or discrete transition in the timed transition system giving the semantics of the DUTA implementation or the BIP model (since both are DUTA-based, such semantics are similar). A time action corresponds to an arbitrary evolution of time in the system. A discrete action in the DUTA implementation (resp. BIP model) is, by abuse of notation, an edge in a DUTA (resp. an interaction). Each (edge) action in the DUTA implementation *corresponds* to an (interaction) action in the BIP model: it is the interaction synchronizing the same edge with other edges in the BIP model (in order to propagate side effects on shared variables), and vice versa (the "corresponds to" is symmetric). For instance, in Fig. 7, the (edge) action in the DUTA implementation $manage \rightarrow manage$ (in DUTA M) corresponds to the sole interaction defined by connector rv, and conversely.

Let Φ (resp. Ψ) be the (global) timed transition system representing the semantics of the DUTA implementation (resp. the BIP translation) of some component $Comp$ with one execution task (see the generic definitions in Sect. III-B1a and Sect. III-B1b). Each state of Φ (resp. Ψ) will then be of the form (L, V) (resp. $(\mathcal{L}, \mathcal{V})$) where L (resp. $\mathcal{L}$) contains the current location of each DUTA (resp. atom component) as well as the current value of each variable in the composition, and V (resp. $\mathcal{V}$) the valuations of all clocks. The main difference between the representation of states in Φ and Ψ is the fact that L gives the valuations of global variables in Φ while $\mathcal{L}$ gives those of local variables in Ψ. For instance, if (L_0, V_0) (resp. $(\mathcal{L}_0, \mathcal{V}_0)$) is the initial state of Φ (resp. Ψ), then while L_0 contains the initial valuation of *e.g.* Π, $\mathcal{L}_0$ contains the initial valuation of Π_M, the duplicate of Π local to the manager M and, for each activity A_i, the initial valuation of Π_{A_i}, the

duplicate of Π local to A_i. To simplify the explanation, we group the (local) duplicates of each (originally global) variable u in a set u'. For instance, if we reuse the example of initial states and variable Π, $\mathcal{L}_0$ contains the valuations of variables in the set $\Pi' = \{\Pi_M, \Pi_{A_1}, ..., \Pi_{A_m}\}$.

Now, it is sufficient to say that a state in Φ and a state in Ψ are bisimilar iff (i) the current location of each DUTA (given by L) and its BIP atom component counterpart (given by $\mathcal{L}$) are equal, (ii) the valuation of each local variable in each set u' (given by $\mathcal{L}$) is equal to the valuation of the global variable u (given by L) and (iii) the valuation of each clock in each DUTA (given by V) is equal to the valuation of the clock of the corresponding BIP atom component (given by $\mathcal{V}$).

Thanks to the proper initialization mechanism (as exemplified in listing 6), and the fact that all clocks start at zero, we may easily see that (L_0, V_0) and $(\mathcal{L}_0, \mathcal{V}_0)$ are bisimilar. Now, we need to ensure that for any pair of bisimilar states (L, V) in Φ and $(\mathcal{L}, \mathcal{V})$ in Ψ, if we take whatever discrete (resp. time progress) action from (L, V) to reach a state (L', V) (resp. (L, V')), taking the corresponding action from $(\mathcal{L}, \mathcal{V})$ would lead to state $(\mathcal{L}', \mathcal{V})$ (resp. $(\mathcal{L}, \mathcal{V}')$) such that (L', V) and $(\mathcal{L}', \mathcal{V})$ (resp. (L, V') and $(\mathcal{L}, \mathcal{V}')$) are bisimilar ($\Psi$ simulates Φ) and vice versa (Φ simulates Ψ).

For discrete actions, thanks to the propagation mechanism (exemplified in Fig. 7), we can easily see that taking any edge in the DUTA implementation and its corresponding interaction in the BIP model, along which we ensure all duplicates of affected shared variables are updated, would result in states where each global variable u (in the DUTA implementation) is equal to each duplicate of u (in the BIP model). That is, if a discrete action (an edge) is taken from a state (L, V) in the DUTA implementation, and its corresponding action (the corresponding BIP interaction) is taken from a state $(\mathcal{L}, \mathcal{V})$ in the BIP model (with (L, V) and $(\mathcal{L}, \mathcal{V})$ bisimilar), to reach, respectively, (L', V) and $(\mathcal{L}', \mathcal{V})$, then the valuation of each variable u given by L' is equal to the valuation of each element of u' given by $\mathcal{L}'$. Additionally, the location of each DUTA after taking the action, given by L', is the same for its BIP atom component counterpart, given by $\mathcal{L}'$, as the translation does not bring any change to this aspect (the edges added for propagating side-effects are self-loops, *e.g.* the red edges in Fig. 7). Which means that L' and $\mathcal{L}'$ respect the bisimilarity condition, and thus (L', V) and $(\mathcal{L}', \mathcal{V})$ are bisimilar.

For time actions, the translation does not affect any timing constraint between the DUTA implementation and the BIP model: (i) clock constraints are not modified and (ii) the added edges to propagate side effect are not $\wr$ and have no clock constraints, which means they have no time-related effect on the interactions in which they take place. We may thus prove with no particular difficulty that taking a time action from (L, V) (in Φ) to reach (L, V'), and the same action (same time progress) from $(\mathcal{L}, \mathcal{V})$ (in Ψ) to reach $(\mathcal{L}, \mathcal{V}')$, would result in (L, V') and $(\mathcal{L}, \mathcal{V}')$ being bisimilar. Thus, combining the proof on the discrete actions and that on the time actions, we conclude that Ψ simulates Φ. Finally, we follow the same steps to prove that Φ simulates Ψ, and thus prove (strong) bisimilarity between Φ and Ψ under both types of actions.

4) Extending to Runtime: At this stage, we have a sound translation from GenoM3 to BIP. However, both DUTA and BIP models are "offline": codels are not executed. Rather, the execution of a codel `c` is emulated as a time progress comprised between zero (excluded) and the WCET of `c` reflected by an invariant and $x > 0$ guards. Moreover, the offline model may be non deterministic: when a codel `c` has more than one successor, there is an edge from location c (c_{exec} if `c` is TU) to each location c' mapping a codel `c'` successor of `c` in the GenoM3 specification. Both of these aspects, retrieved from *Definition 1* and explained in Sect. III-B1c, can also be seen in the BIP model (see *e.g.* location get in Fig. 7). Yet, the models we aim to generate will be run, on the real robot, by the BIP Engine (Sect.II-A1). We may then execute the real C code attached to codels which, we recall, we refer to simply as executing codels (Sect. II-B). It follows that we need to extend the model to take into account such concrete execution.

First, we need to remove non-determinism. When a GenoM3 activity is executed, any non-determinism within is resolved: when a codel has more than one successor, its concrete execution returns the codel to execute next (among the successors defined by the transitions in the GenoM3 specification). For any codel `c` with more than one successor, we need thus an intermediate location c_{test} to decide which codel to execute next according to the return value of the execution of `c`. If `c` is TS (resp. TU), the edges from location c (resp. c_{exec}) to each location c' (of each successor `c'`) in the offline model are then replaced by one edge from c (resp. c_{exec}) to c_{test}, on which codel `c` is executed, plus $\wr$ edges, guarded with an equality between the return value of the execution and `c'`, between c_{test} and each c'. Second, we need to replace the "emulation" of execution time by a timing constraint on the concrete execution of the codel. Since the concrete execution is done on an edge/interaction rather than in a location, we use the BIP construct $resume()$ to define such constraint.

Let us illustrate how this is done for activity *Measure* (Fig. 7). Without details on other operations and guards (previously explained for the DUTA implementation and its BIP counterpart), and with a simplified version of codels names (without activities subscripts), listing 8 shows the (extended model) of the header of the BIP atom component for activity *Measure* as well as the behavior of the TS codel `get`. The extension for the latter is comprised between lines 16 and 30. Notice how, to remove non-determinism, intermediary location get_{test} is added. On the edge $get \rightarrow get_{test}$ (lines 16 to 20), the concrete execution of codel get takes place (line 19). Then, once get_{test} is reached, it is urgently left to reach either get_{pause} or $ether$, depending on the return value of the concrete execution (lines 22 to 30). Dually, there is no more an invariant at location get and a guard $x > 0$ on its outgoing edge(s) (Fig. 7). Instead, the outgoing edge of get is $\wr$ (keyword *eager*, to start the execution immediately, line 18) and the $resume(x \leq 2)$ statement (line 20) checks whether the concrete execution of the edge operations (*i.e.* executing codel `get` in this case, line 19) takes no longer than 2 time units, that is the WCET of `get` (we recall that clock x is reset when reaching get according to *Definition 1*). If the timing constraint of $resume()$ is violated,

the Engine detects it (example in Sect. IV).

```
1       /* external data type for codel return */
2       extern data type genom_event
3       ...
4       atom type measure()

6       data genom_event next_codel
7       clock x unit millisecond
8       ...
9       port Basic get_to_get_test()
10      port Basic get_to_ether()
11      port Basic get_to_get_pause()

13      state ether, start, start_exec, start_test, get,
            get_test, get_pause, write_p, write_p_exec,
            write_p_test, stop
14      initial to ether
15      ...
16      on get_to_get_test
17      from get to get_test
18      eager
19      do {next_codel=... } /* call C code of get*/
20      resume (x≤2)

22      on get_test_to_ether()
23      from get_test to ether
24      provided (next_codel == ether)
25      eager

27      on get_test_to_get_pause()
28      from get_test to get_pause
29      provided (next_codel == get_pause)
30      eager
31      ...
32      end
```

Listing 8: Codel execution in BIP (activity *Measure*, Fig. 6)

5) Automatic synthesis: After developing our sound translation and extending it to runtime, we need to automatize it to reach our goal of a sound and automatic translation from GenoM3 to BIP. Automation is very important as it allows to obtain the BIP counterpart from any GenoM3 specification, while sheathing the translation details to the robotic programmer. Listing 9 shows an example of how such automation is achieved using the template mechanism (introduced in Sect. II-B2). It is an excerpt that generates the locations of the BIP atom component from the GenoM3 specification of some activity a, which would, for activity *Measure*, generate line 13 in listing 8. The interpreter outputs everything as is, except what is enclosed in <' '> that it evaluates in Tcl, and in <" "> that it evaluates and outputs the result.

In sum, the listing implements rule (2) of *Definition 1* (to get the locations of the DUTA of a, which remain the same in the BIP model) plus the additional c_{test} locations added for the concrete execution of codels with more than one successor (Sect. III-B4). In order to do so, we need first to know which codels are targeted by a pause transition. This is done from line 1 to 6, where we iterate over each successor y of each codel c of a and add y to the list called p only if the transition c to y is a pause transition (notice that [$a codels] returns all codels of a specified by the user, that is excluding ether, which is sufficient here because we know ether has no successors in the specification, Sect. II-B1). Thus, arriving at line 7, p will contain all the codels targeted by a pause transition in the GenoM3 specification of a (if any). Then, we apply rule (2) of *Definition 1*: we generate the location *ether*, always present in an activity (initial location, line 7), then, from line 8 to 12, we output for each codel c (excluding ether) a location with the same name (line 8), and an additional location c_{exec} (resp. c_{pause}) iff codel c is TU, line 9 (resp. TS targeted by a pause transition, line 10 to 12), with any TU codel c recognized through the non emptiness of the list of all codels in conflict with c, returned by the *mutex* function (line 9). Finally, if codel c has more than one successor (line 13), an additional location c_{test} is generated for concrete execution purposes (Sect. III-B4).

```
1   <'set p [list]'>
2   <'foreach c [$a codels] {'>
3   <'   foreach y [$c yields] {'>
4   <'     if {[$y kind] == "pause" && !($y in $p)} {lappend p
          $y}
5   <'   }'>
6   <'}'>
7   state ether
8   <'foreach c [$a codels] {'>, <"[$c name]">
9   <'   if {[llength [$c mutex]]} {'>, <"[$c name]">_exec<'}'>
10  <'   } else {
11  <'     if {$c in $p} {'>, <"[$c name]">_pause<'}
12  <'   }'>
13  <'   if {[llength [$c yields]] > 1} {'>, <"[$c
         name]">_test<'}'>
14  <'}'>
```

Listing 9: Automatic generation of locations (for activity a)

IV. Experiments and Discussion

We generate automatically the BIP model of our autonomous navigation case study (Sect. II-B3), using the template developed in Sect. III. We add, to the generated model, a monitor to verify a crucial timed property, and properly react to its possible violation online. The experiments, publicly available as videos (see *Artefacts* below), are carried out in simulation and on the real Robotnik platform.

A. Properties of interest

In laser-based navigation, failure to use updated LRF data during motion may lead to collision with obstacles which could damage the robot or injure humans. An important property to verify is thus "always getting new laser data in a bounded amount of time when the robot is moving". That is, each time port **Laser** (LaserDriver (Fig. 3, Sect. II-B3)) is written, it will be rewritten before a *timeout* occurs, which means that there must be a maximum amount of time separating any two successive writes on the port. This is a *bounded response* property, the violation of which means that there is a serious problem such as a starvation phenomenon (a codel is waiting forever to get access to the port it writes) or a sensor defect. The robot must thus urgently stop moving and abandon its mission. We also visualize the violation of timing constraints extracted from the specification, mainly the WCET of codels.

B. RV with BIP

After we generate the runtime BIP model, we augment it with a monitor to verify the bounded response property. First, we create a BIP atom *monitor* which verifies online the correctness of the property (listing 10). Starting to move is captured via port *go*. Port *scan* corresponds to the event of writing (the GenoM3) port **Laser** within the constant *timeout*. Port *report*

corresponds to detecting the violation of the property. Finally, port *finish* is triggered when the motion ends (goal reached, invalid, unreachable). It follows that location *idle* (resp. *busy*) corresponds to the robot at rest (resp. at motion), that is no (respect. one) instance of activity *GotoPosition* (NAVIGATION, Fig. 3) is being executed. Notice the non-determinism at *busy* when $x \leq timeout$ (lines 15 to 24) with no real impact on the desired behavior (if *scan* is triggered first when both *scan* and *finish* are possible, *finish* will follow immediately anyway).

```
1  atom type monitor()

3  clock x unit millisecond
4  export port Basic scan(),
       report(), go(),
       finish()

6  state idle, busy

8  initial to idle

10 on go
11 from idle to busy
12 eager
13 do {x=0;}

15 on scan
16 from busy to busy
17 provided (x<= timeout)
18 eager
19 do {x=0;}

21 on finish
22 from busy to idle
23 provided (x<= timeout)
24 eager

26 on report
27 from busy to idle
28 provided (x> timeout)
29 eager

31 end
```

Listing 10: The monitor atom type

Second, we need to create connectors that link ports automatically generated in the model with the ports of the atom *monitor* so they correspond to the wanted events as explained above. We make sure this "connection" between the monitor and the generated model is *non invasive*, that is the monitor does not modify (only observes) the events of the underlying model unless the priority is violated (at which point it intervenes to stop the robot). We will see how this works for ports *scan* and *report*, for instance. The event that *scan* corresponds to is writing the port **Laser**. We need thus a connector that involves both *scan* and the connector within the compound *laserdriver* (generated from the GenoM3 component LASERDRIVER, Fig. 3) that corresponds to writing the GenoM3 port **Laser**. This connector is defined as *Write_Laser*. Now, we create a broadcast connector involving both parties as follows:

```
connector br2 Scan_OK (LaserDriver.Write_Laser, Monitor.scan)
```

Where *br2* is a broadcast connector type with two ports (the first is the sender), *Monitor* an instance of atom *monitor* and *LaserDriver* an instance of *laserdriver*. Using a broadcast ensures the monitor's "non-invasive" feature described above: writing the laser must remain possible even when the robot is not moving, as allowed by the GenoM3 model. When the robot is moving, the maximal interaction (involving both the sender and the receiver) is guaranteed by the BIP Engine.

Now, port *report* in the monitor must correspond to the action to take if the property is violated, *i.e.* to urgently stop the robot. To do so, we couple triggering *report* with the generation of a request to activity *Stop* (component SAFETYPILOT). This is because activity *Stop* interrupts activity *StopIfObstacle* in the same component, the codel `stop` of which writes a null speed to its **Cmd** (GenoM3) port (which makes the robot stop moving when applied by ROBOTDRIVER, Fig. 3, Sect. II-B3). The port triggering the behavior following a *Stop* request in the compound *safetypilot* (generated automatically from the GenoM3 component SAFETYPILOT) is defined as *Rq_Stop*. We need thus to create a rendezvous connector involving *report* (from the monitor) and *Rq_Stop*:

```
connector sync2 Scan_Fail (SafetyPilot.Rq_Stop, Monitor.report)
```

Where *sync2* is a two-port rendezvous connector type and *SafetyPilot* an instance of *safetypilot*. Alternatively, we may send a request *Stop* in component NAVIGATION, which will interrupt activity *GotoPosition* and eventually entail writing a null speed to the **Cmd** (GenoM3) port of SAFETYPILOT.

The BIP model (including the monitor) is now ready for execution and RV (including proper reaction to the possible violation of the timed property of interest). We set the timeout to 100 ms, which is the period of POTENTIALFIELD (in charge of potential-field navigation). The robot fulfills its missions correctly. We inject then some delays: we (i) create an activity *SetDelay* in the GenoM3 component LASERDRIVER which uses a *usleep()* function in order to delay writings to port **Laser**, then (ii) visualize how the monitor intervenes to stop the robot quickly. We may see within the execution trace that the Engine forces taking connector *Scan_Fail*, which results in executing codel `stop` of activity *StopIfObstacle* (SAFETYPILOT), and therefore a zero speed is sent to the controller as shown in the listing below.

```
[BIP ENGINE]: state #165351: 1 interaction:
[BIP ENGINE]: [0] ROOT.Scan_Fail: SafetyPilot.Rq_Stop()
     Monitor.report() ] 26s591ms324us108ns, +INFTY ]
[BIP ENGINE]: →choose [0] ROOT.Scan_Fail:
     SafetyPilot.Rq_Stop() Monitor.report() at global time
     26s591ms324us109ns
...
[GenoM3] SafetyPilot Calling
     SafetyPilot_activity_StopIfObstacle_stop codel.
[GenoM3] SafetyPilot Exiting
     SafetyPilot_activity_StopIfObstacle_stop codel with
     ::SafetyPilot::ether.
```

While tuning the timeout, we realize that a constraint as small as 40 ms is too strict as the monitor stops the robot too often. This observation allowed us to reconsider the 40 ms period that we give to *SafetyPilot* (which also relies on the LRF).

The BIP Engine allows us also to further tune codels WCETs, as it issues a warning whenever a $resume()$ constraint (Sect. III-B4) is violated. Below is a warning issued by the Engine pointing out that a $resume$ within component *init* in the compound instance *PotentialField* (mapping activity *init* in the GenoM3 component POTENTIALFIELD) is violated. The messages given by GenoM3 help localizing where the violation occurred (in this example, the WCET of codel `start`).

```
[GenoM3] PotentialField Calling
     PotentialField_activity_Init_start codel.
[GenoM3] PotentialField Exiting
     PotentialField_activity_Init_start codel with
     ::PotentialField::ether.
[BIP ENGINE]: WARNING: state #903017 and global time
     1min3s230ms653us976ns: violation of the following
     timing constraint ROOT.PotentialField.init:
[BIP ENGINE]: ROOT.PotentialField.Init resume [ -INFTY,
     1min3s229ms530us282ns ]
```

Artefacts: Link [1] directs to three videos of our experiments. One video shows the full automatic generation process. The other two show the RV: one using the Gazebo simulator and

the other on the Robotnik platform. In the former (resp. the latter), the BIP Engine interrupts activity *StopIfObstacle* in SAFETYPILOT (resp. *GotoPosition* in NAVIGATION) in order to stop the robot (by executing codel `stop` of *StopIfObstacle*, resp. *GotoPosition*) following injected delays through *SetDelay* requests. A README file is provided for further details.

C. Discussion

We manage to verify online a crucial bounded response property and adequately react to its violation, as well as to monitor timing constraints related to concrete execution of codels, on the real robotic platform. Execution and RV with the BIP Engine offer a rigorous and efficient alternative to exhaustive techniques such as model checking. Indeed, the formal models obtained from our case study are too large to scale with tools such as TINA and UPPAAL, as we confirm when we generate them using the templates developed in [17] and [20]. To give an idea on the size of formal models of our case study, the automatically generated BIP model is over 13 thousand lines (excluding codels C code) and its compilation takes over 20 minutes on a modern computer.

However, the BIP Engine induces, in this application, up to 10% overhead (processor load), which creates delays that may play a role in reaching the timeout and thus violating the bounded response property. Such overhead also prevents us from applying the approach in this paper to robotic applications running at high frequencies such as the drone navigation presented in [14]. Another issue is that the monitor is added manually to the generated BIP model, which threatens the usability of our approach by a regular robotic engineer. Some future directions to solve the above issues are given in Sect VI.

V. RELATED WORK

A. RV in Robotics

The literature of RV of functional robotic components is relatively rich. For instance, the Java PathExplorer tool [24] is used with the NASA robot K9 case study. Important LTL and concurrency-related properties (such as deadlock freedom) are verified at runtime. ROSRV [25], an RV environment for ROS-based robotic systems, is another related work. It adds a monitoring layer on top of the ROS components to restrict the execution to scenarios satisfying security and safety properties. A more recent example is found in [30], where past-time LTL (available since [29]) is used to formulate properties of a collision avoidance algorithm, which are then checked online by monitors based on (untimed) FSM models. *Fault Detection, Isolation and Recovery* (FDIR) approaches are also related works when RV is employed to perform fault detection. In particular, BIP has been used to develop FDIR components to detect and recover from time-related faults in aerospace robotics [33], [32].

Globally, RV in robotics suffers from two major limitations. Firstly, it is mostly restricted to temporal and safety properties (no verification of timed properties), which is the case in all the related work above (except the FDIR ones). This problem is not intrinsic to RV, as RV theories for timed properties have significantly progressed in the last decade [7], [35]. Rather, the application of such theories in the robotic domain is timid as of today. Secondly, as emphasized in Sect. I, robotic functional components are mostly specified in non formal frameworks. In all the related work above, the authors generate the (formal) monitors that they link to (non formal) specifications, which raises a number of concerns. For instance, it is difficult to reason on the trustworthiness of the overall approach (*i.e.* to know at what point we can trust the non-formal specifications, and consequently trust the whole system). Moreover, this heterogeneity leads to performance issues that are more perceptible than those that we report here where the model (including the monitor), the executer and the verifier are all in BIP. As an example, the authors of [24] notice a slowdown by an order of magnitude (compared to 10% of overhead in our case) due to the fact that the monitors and the logic Engine (to check the properties) are written in two different languages, respectively Java and Maude [9].

Recent works in [10], [13], [11] are a notable exception to the above limitations. However, their approach requires encoding robotic specifications in the formal language P [12], whereas we favor increasing usability by robotic programmers by automatically and transparently bridging a robotic framework (GenoM3) with a timed formal language (BIP).

B. Verification Works Involving GenoM3

Recent works involving GenoM3 include automatic translation and verification of functional components using model checking and SMC [17], [20], [19], [16]. The formal models of the case study we present here are too large to scale with model checking. Furthermore, as we conclude in [20], SMC raises an open *confidence* problem (it is hard to set the probability at which we deem the properties "sufficiently" satisfied).

When it comes to RV, works like [3] use the untimed version of BIP to model and verify functional components written in GenoM2 (an earlier version of GenoM). None of such works escapes the two limitations explained in Sect. V-A: (i) no timed properties could be verified (due to the untimed nature of BIP then) and (ii) it is not possible to verify the soundness of the BIP models (vis-à-vis their robotic counterpart) due to the absence of operational semantics of GenoM2.

C. Our Contribution

The work we present here tackles both limitations of RV in robotics (Sect. V-A): (i) runtime models are formal, with the BIP model faithful to the semantics of the robotic specification and (ii) timed properties are supported, with appropriate actions whenever they are violated. Furthermore, the translation from GenoM3 to BIP is automated. We offer therefore a trustworthy and efficient alternative to model checking for large models, with a support for timed properties. To the best of our knowledge, this is the first work where both execution and RV of timed properties, based on formal foundations, are directly accessible from a mainstream robotic framework.

VI. CONCLUSION

In this paper, we propose a sound and automatic translation from the robotic framework GenoM3 to the real-time formal language BIP. The approach is convenient to robotic programmers with no formal background as it conceals the translation

details and provides the BIP model for any new robotic application with no additional modeling efforts. The BIP model of a real-world case study is generated and executed, using the BIP Engine, on the robotic platform. Orchestrated with a monitor, the generated BIP model is used to verify a crucial timed property (bounded response) at runtime and properly react to its violation, on a model that originally does not scale with model checking.

We give two directions of future work. First, though acceptable compared to that reported in related work, the 10% overhead of the BIP Engine introduces unpredictable delays (Sect. IV-C). For future work, we plan to investigate further the sources of such overhead in order to try and reduce it. Second, the manual encoding of BIP monitors is unsuitable for robotic programmers (Sect. IV-C). In the future, we aim to extend the BIP template to generate, from a GenoM3 specification and a (possibly timed) property, an equivalent BIP model including the property monitor. However, we first need a high-level, preferably robotic-friendly language, where the robotic engineer can express the properties of interest. Once such a language is defined, we may benefit from theoretical results on generating timed automata monitors from property specifications [28], [34].

REFERENCES

[1] Artefacts (videos). Shortened DropBox link https://bit.ly/3i8IXs4.

[2] The Pocolibs middleware https://git.openrobots.org/projects/pocolibs.

[3] Tesnim Abdellatif, Saddek Bensalem, Jacques Combaz, Lavindra De Silva, and Felix Ingrand. Rigorous Design of Robot Software: A Formal Component-Based Approach. *Robotics and Autonomous Systems*, 60(12):1563–1578, 2012.

[4] Tesnim Abdellatif, Jacques Combaz, and Joseph Sifakis. Model-Based Implementation of Real-Time Applications. In *International Conference on Embedded software*, pages 229–238, 2010.

[5] Rajeev Alur. Timed Automata. In *International Conference on Computer-Aided Verification*, pages 8–22. Springer, 1999.

[6] Ananda Basu, Marius Bozga, and Joseph Sifakis. Modeling Heterogeneous Real-Time Components in BIP. In *International Conference on Software Engineering and Formal Methods*, pages 3–12. IEEE, 2006.

[7] Andreas Bauer, Martin Leucker, and Christian Schallhart. Runtime Verification for LTL and TLTL. *ACM Transactions on Software Engineering and Methodology*, 20(4):1–64, 2011.

[8] Sébastien Bornot, Joseph Sifakis, and Stavros Tripakis. Modeling Urgency in Timed Systems. In *International Symposium on Compositionality*, pages 103–129. 1997.

[9] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martı-Oliet, José Meseguer, and José F Quesada. Maude: Specification and Programming in Rewriting Logic. *Theoretical Computer Science*, 285(2):187–243, 2002.

[10] Ankush Desai, Tommaso Dreossi, and Sanjit A Seshia. Combining Model Checking and Runtime Verification for Safe Robotics. In *International Conference on Runtime Verification*, pages 172–189, 2017.

[11] Ankush Desai, Shromona Ghosh, Sanjit A Seshia, Natarajan Shankar, and Ashish Tiwari. SOTER: A Runtime Assurance Framework for Programming Safe Robotics Systems. In *International Conference on Dependable Systems and Networks*, pages 138–150. IEEE, 2019.

[12] Ankush Desai, Vivek Gupta, Ethan Jackson, Shaz Qadeer, Sriram Rajamani, and Damien Zufferey. P: Safe Asynchronous Event-Driven Programming. *ACM SIGPLAN Notices*, 48(6):321–332, 2013.

[13] Ankush Desai, Shaz Qadeer, and Sanjit A Seshia. Programming Safe Robotics Systems: Challenges and Advances. In *International Symposium on Leveraging Applications of Formal Methods*, pages 103–119. Springer, 2018.

[14] Mohammed Foughali. Toward a Correct-and-Scalable Verification of Concurrent Robotic Systems: Insights on Formalisms and Tools. In *International Conference on Application of Concurrency to System Design*, pages 29–38. IEEE, 2017.

[15] Mohammed Foughali. Formal Verification of the Functional Layer of Robotic and Autonomous Systems. *PhD Thesis, INSA Toulouse*, 2018.

[16] Mohammed Foughali. A Two-Step Hybrid Approach for Verifying Real-Time Robotic Systems. In *International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 1–10. IEEE, 2020.

[17] Mohammed Foughali, Bernard Berthomieu, Silvano Dal Zilio, Pierre-Emmanuel Hladik, Félix Ingrand, and Anthony Mallet. Formal Verification of Complex Robotic Systems on Resource-Constrained Platforms. In *International Conference on Formal Methods in Software Engineering*, pages 2–9. ACM, 2018.

[18] Mohammed Foughali, Silvano Dal Zilio, and Félix Ingrand. On the Semantics of the GenoM3 Framework. *Tech. Report, LAAS-CNRS*, 2019.

[19] Mohammed Foughali and Pierre-Emmanuel Hladik. Bridging the Gap between Formal Verification and Schedulability Analysis: The Case of Robotics. *Journal of Systems Architecture*, 111:817–852, 2020.

[20] Mohammed Foughali, Félix Ingrand, and Cristina Seceleanu. Statistical Model Checking of Complex Robotic Systems. In *International SPIN Symposium on Model Checking of Software*, pages 114–134. Springer, 2019.

[21] Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated Planning: Theory and Practice*. Elsevier, 2004.

[22] Sorin Grigorescu, Bogdan Trasnea, Tiberiu Cocias, and Gigel Macesanu. A Survey of Deep Learning Techniques for Autonomous Driving. *Journal of Field Robotics*, 37(3):362–386, 2020.

[23] Matteo Guerra, Denis Efimov, Gang Zheng, and Wilfrid Perruquetti. Avoiding Local Minima in the Potential Field Method Using Input-to-State Stability. *Control Engineering Practice*, 55(C):174–184, 2016.

[24] Klaus Havelund, Grigore Rosu, and Daniel Clancy. Java PathExplorer: A Runtime Verification Tool. *Tech. Report, NASA Ames Research Center*, 2001.

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

[26] Nathan Koenig and Andrew Howard. Design and Use Paradigms for Gazebo, an Open-Source Multi-Robot Simulator. In *International Conference on Intelligent Robots and Systems*, pages 2149–2154. IEEE, 2004.

[27] Hadas Kress-Gazit, Tichakorn Wongpiromsarn, and Ufuk Topcu. Correct, Reactive, High-Level Robot Control. *IEEE Robotics & Automation Magazine*, 18(3):65–74, 2011.

[28] Moez Krichen and Stavros Tripakis. Conformance Testing for Real-Time Systems. *Formal Methods in System Design*, 34(3):238–304, 2009.

[29] Orna Lichtenstein, Amir Pnueli, and Lenore Zuck. The Glory of the Past. In *Workshop on Logic of Programs*, pages 196–218, 1985.

[30] Chenxia Luo, Rui Wang, Yu Jiang, Kang Yang, Yong Guan, Xiaojuan Li, and Zhiping Shi. Runtime Verification of Robots Collision Avoidance Case Study. In *Computer Software and Applications Conference*, pages 204–212. IEEE, 2018.

[31] Anthony Mallet, Cédric Pasteur, Matthieu Herrb, Séverin Lemaignan, and Félix Ingrand. GenoM3: Building Middleware-Independent Robotic Components. In *International Conference on Robotics and Automation*, pages 4627–4632. IEEE, 2010.

[32] Jorge Ocón, Iulia Dragomir, Andrew Coles, Lars Kunze, Robert Marc, Carlos Perez, Thierry Germa, Vincent Bissonnette, Genny Scalise, Mohammed Foughali, Konstantinos Kapellos, Raùl Dominguez, Florian Cordes, Gerhard Paar, and Giulio Reina. ADE: Autonomous DEcision making in very long traverses. In *International Symposium on Artificial Intelligence, Robotics and Automation in Space*, 2020.

[33] Jorge Ocón et al. The ERGO Framework and Its Use in Planetary/Orbital Scenarios. In *International Astronautical Congress*, 2018.

[34] Srinivas Pinisetty, Yliès Falcone, Thierry Jéron, Hervé Marchand, Antoine Rollet, and Omer Nguena Timo. Runtime Enforcement of Timed Properties Revisited. *Formal Methods in System Design*, 45(3):381–422, 2014.

[35] Srinivas Pinisetty, Thierry Jéron, Stavros Tripakis, Yliès Falcone, Hervé Marchand, and Viorel Preoteasa. Predictive Runtime Verification of Timed Properties. *Journal of Systems and Software*, 132:353–365, 2017.

[36] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y Ng. ROS: an Open-Source Robot Operating System. In *International Workshop on Open Source Software*, page 5, 2009.

[37] Richard Volpe, Issa Nesnas, Tara Estlin, Darren Mutz, Richard Petras, and Hari Das. The CLARAty Architecture for Robotic Autonomy. In *IEEE Aerospace Conference*, pages 115 –121, 2001.