Model Checking Real-Time Properties on the Functional Layer of Autonomous Robots

Mohammed Foughali, Bernard Berthomieu, Silvano Dal Zilio, Félix Ingrand, Anthony Mallet

Model Checking Real-Time Properties on the Functional Layer of Autonomous Robots

Mohammed Foughali, Bernard Berthomieu, Silvano Dal Zilio,
Félix Ingrand, and Anthony Mallet

CNRS, LAAS, 7 avenue du Colonel Roche, F-31400 Toulouse, France
Univ de Toulouse, LAAS, F-31400 Toulouse, France

Abstract. Software is an essential part of robotic systems. As robots and autonomous systems are more and more deployed in human environments, we need to use elaborate validation and verification techniques in order to gain a higher level of trust in our systems. This motivates our determination to apply formal verification methods to robotics software. In this paper, we describe our results obtained using model-checking on the functional layer of an autonomous robot. We implement an automatic translation from GenoM, a robotics model-based software engineering framework, to the formal specification language Fiacre. This translation takes into account the semantics of the robotics middleware. TINA, our model-checking toolbox, can be used on the synthesized models to prove real-time properties of the functional modules implementation on the robot. We illustrate our approach using a realistic autonomous navigation example.

1 Introduction

Software is an essential part of robotic systems. As robots and autonomous systems are more and more deployed in human environments (autonomous cars, coworker robots, surgery robotics, etc.) and/or costly exploration missions (extraterrestrial rover, deep space mission, etc.), we need to use more elaborated V&V techniques, in order to gain a higher level of trust in the behavior of such systems. Indeed, the trust we currently put in robotics software mainly rely on testing campaigns, best coding practices, and the choice of sound architecture principles. This does not rise to the level found in many regulated domains, such as the aeronautic or nuclear industries, where formal methods are routinely used to check the most vital parts of systems.

On the other hand, robotics software provides new opportunities to apply formal methods. Indeed, robotics applications are often deployed using model-based software engineering approaches [14] and described as a set of functional modules orchestrated by a robotics middleware. These modules, and their interactions, are amenable to an interpretation into formal models.

Autonomous systems are typically organized along layers [13]. The lower one, the *functional layer*, interacts directly with sensors and actuators and performs the data processing tasks. The *decisional layer* deals with more cognitive activities, such as task

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

planning or monitoring. With respect to V&V, the decisional layer models are often formal and range from planning models (PDDL, ANML, NDDL, etc.) to acting models (TDL, SMach, RMPL, OpenPRS, etc.) or monitoring models (Livingstone, etc.) [22], a fact reflected by a large body of research applying formal V&V approaches to the deliberation functions [1, 16, 21, 29]. In contrast, little has been done to bridge frameworks used to deploy functional level modules with formal methods and their associated V&V tools.

In this paper, we propose to connect a robotics model-based approach (GenoM3) with a formal V&V framework (Fiacre/TINA). We describe how we can automatically synthesize a formal model of robotics functional modules and then use it to prove important behavioral and timed properties of the modules implementation on the robot. We illustrate our presentation with a realistic autonomous navigation example, on which we formally prove properties of interest to the robot programmers.

The paper is organized as follows. After a section on related works focusing on formal verification of the functional level of robotic systems, we introduce (Section 3) the Fiacre formal specification language and the TINA model-checking toolbox. We then describe the GenoM3 framework (Section 4), used to specify functional level modules for robotic systems. In Section 5, we illustrate our robot navigation example as specified in GenoM3. Section 6.1 gives examples on how we map GenoM modules constituents into Fiacre so as to automatically synthesize formal models. Before concluding, Section 7 discusses some examples of properties formally checked on our navigation modules.

2 Related Work

An early work on formal verification in robotics is presented in [19]. It proposes the verification of robotics applications specified in Orccad [30]. Behavioral properties are checked with Mauto after translating Orccad descriptions into the ESTEREL synchronous language [12], with time-related constraints translated into logical events. Timed properties are checked with Kronos, a TCTL model-checker, after translating the specification into Timed-Argos [25], an extension of the synchronous language Argos with delays.

Brat et al. [13] describe an approach to verify autonomous systems with planning, execution and functional layers. They propose a modular verification approach combining compositional techniques (assume-guarantee), static analysis, testing and model checking to assert safety properties. No timing constraints are taken into account.

In [32], the authors attempt to prevent state explosion while model-checking large systems through a compositional approach (inferring properties of the system from the properties of its constituents). They succeed to assert some properties of low-level controllers. Not all properties are amenable to compositional verification, however. They suggest to combine model checking and automated theorem proving to benefit from their respective strengths.

BIP [4], a modeling framework based on automata, is used in the joint verification effort presented in [2]. The functional modules, written in GenoM, of an outdoor robot with two navigation modes, are modeled in BIP. The invariant extractor and SATsolving tool D-Finder [5] is used to check, offline, the absence of deadlocks within the BIP model. Additional safety constraints can be added and automatically translated from logical formulae into BIP. The resulting model is run within the BIP Engine on DALA, an iRobot ATRV (All-Terrain Robotic Vehicle), and the constraints are consequently enforced at runtime. Timing constraints are not considered.

In [20], the MAUVE framework is used to build functional level components for a P3DX mobile robot. The schedulability of the different components is formally verified. Execution scenarios are manually translated into Fiacre [6] models and behavioral properties are asserted on them.

This list is far from being exhaustive. Indeed, we only mentioned methods comparable to ours, omitting for instance those relying on hybrid formal models like [17] or on pure theorem proving as in [31]. Our approach is the closest in spirit to that of [19] and relies on model-checking. Formal models are automatically synthesized and all properties are checked in the same verification framework, including timed properties.

3 Fiacre and TINA

Fiacre [6] is a specification language for describing compositionally both the behavioral and timing aspects of embedded and distributed systems. It has a formal semantics and can be used as an input format for formal verification tools (mainly real-time model-checkers) as well as for simulation purposes.

Fiacre stems from several projects involving industrial and academics partners. Besides the applications described in this paper, Fiacre has been used in a variety of applicative domains, like telecoms, avionics or robotics systems [7, 11, 20, 28]. In this work, we use Fiacre specifications with the model-checking toolbox TINA.

3.1 The Fiacre Language

Fiacre descriptions are made of processes and components, both parametrizable by values, value locations (shared variables) and interaction labels (for communication or synchronization).

Processes describe sequential behaviors; they specify a set of control states and a set of transitions, each expressing a state change by a statement built from deterministic constructs (assignments, conditionals, loops, and sequential composition), nondeterministic constructs (nondeterministic choice and assignments), interaction statements and jump statements. Several examples of Fiacre processes are shown in Section 6.1.

Components describe in a hierarchical manner the architecture of the system; a system is a parallel composition of process or component instances. Components also specify the interactions between the constituting processes or components, and possibly constrain these interactions with timing and/or priority requirements.

Apart from its ability to model priorities and timing constraints (using a dense time model), a distinctive feature of Fiacre is to include a rich set of datatypes: booleans, integers and integer ranges, records, tagged unions, arrays and queues. The language is statically typed, with depth subtyping to handle integer ranges. In terms of process interactions, Fiacre supports both the classical paradigms of shared variables and synchronous message passing à la process calculi. Shared variables and interaction ports are created local to components. Finally, Fiacre provides functions, native or imported. Some introductory material and examples can be found on the Fiacre site (`www.laas.fr/fiacre`).

Semantics: Classically for a timed language, the semantics of a Fiacre description is a timed transition system, that is a transition system with two kinds of transitions: discrete transitions resulting from discrete state changes and continuous transitions resulting from time elapsing. The semantics of a component is the synchronized product of the semantics of its subcomponents, further constrained by time and priority constraints on their interactions, if any.

Verification: Fiacre descriptions can be complemented by declarations of properties. Atomic properties include the states of process instances, predicates on the values of variables and Fiacre events (interactions). The Fiacre observables are boolean combinations of atomic properties. They can be combined to form property patterns in the style of [18]. For checking real-time properties, these patterns are enriched with time constraints [3]. For verification, the real-time patterns are translated by the Fiacre compiler into LTL properties on the Fiacre description instrumented with observers.

As an illustration, this is how a "leadsto within" timed property is handled. The property is written in Fiacre as (*source* **leadsto** *target* **within** $[d1, d2]$), where *source* and *target* are some observables and $[d1, d2]$ is a time interval. The property asserts that along each path some state obeying *target* occurs within a delay in interval $[d1, d2]$ after each state obeying *source*. This property is encoded using a Fiacre process (an observer) given in the listing below; the process is connected with the main Fiacre program through two transition guards on the *source* and *target* observables. With this observer, the property is to show that the state `error` of the observer is unreachable.

```
process LeadsToWithin is
   states idle, start, watch, error
   from idle
      on source; to start
   from start
      wait [d1,d1]; to watch
   from watch
      select
         on target; wait [0,0]; to idle
      unless
         wait ]Δ,...[; to error /* where Δ = d2 − d1 */
      end
```

3.2 The TINA Toolbox

TINA [9] is a toolbox for the analysis and verification of Time Petri nets (possibly) enriched with priorities, stopwatches and/or data processing. It is freely available at `www.laas.fr/tina`.

Time Petri nets: Together with Timed Automata, Time Petri nets [26] (*TPN* for short) are a prominent model for analysis of real-time systems. Time Petri nets enrich Petri nets with time intervals associated with the transitions of the net specifying the possible time delays between last enabledness of these transitions and their activation (or firing in Petri net terminology).

Due to the dense nature of time considered in *TPN*, their state spaces are typically infinite, but finite abstractions of these are available since [8], known as *State Classes*. State Classes provide a finite time-abstracted representation of the behavior of bounded *TPN* preserving their markings and traces. A state class associates a marking of the *TPN* with a system of difference constraints (a DBM) capturing the times in the future at which the transitions enabled at that marking can fire.

Since it preserves markings and traces, the state classes construction is suitable for LTL model-checking. A simple variation of the construction (reducing classes by inclusion) only preserves markings and is typically coarser; it is the method of choice for reachability analysis.

In contrast with the well known zone constructions for Timed Automata, state classes do not capture clock domains for the enabled transitions, but potential firing times in the future (called firing domains). For these reasons, state class constructions are typically coarser than zone constructions preserving the same properties. But zone constructions are also applicable to *TPN* and are indeed necessary to handle some *TPN* extensions like priorities.

A description of available abstraction methods for *TPN* can be found in [9, 10]. TINA offers all constructions discussed in these papers, as well as several constructions relying on discrete time and a number of constructions specific to Petri nets.

Enriching TPN: *TPN* can be conveniently enriched by a number of features enhancing their expressiveness like *priorities* expressing that some transitions should be favored over others when fireable at the same instant, *stopwatches* allowing to encode preemption, or *data-processing* consisting of synchronizing the evolution of the *TPN* with computations on a set of variables in some programming notation. TINA provides state class constructions for such enriched *TPN*.

Verification in TINA: The TINA toolbox provides state space generators and offline model-checkers for LTL and modal μ-calculus. The generators produce compressed representations of state spaces into files. Some classes of properties can also be checked on the fly when building state spaces. When a property reveals false, a counter example scenario is generated as a timed trace and can be replayed in a simulator.

Verification of Fiacre descriptions: For their verification, Fiacre descriptions are translated into enriched *TPN* as defined above by an optimizing compiler. The compiler, `frac`, performs syntax analysis and type checking, then encodes the description into an enriched *TPN* for TINA preserving its semantics. The compilation process includes a model optimization pass that simplifies redundant transitions, removes dead code and abstracts some variables, retaining only those contributing to the state (unlike e.g. those only used as temporaries). This optimization pass helps reduce the size of the state space.

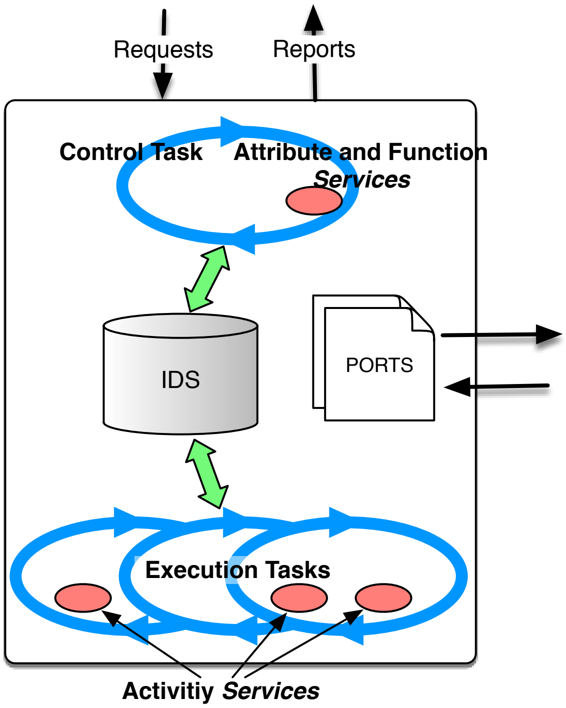


Fig. 1: A generic GenoM3 module.

The `frac` compiler also translates the properties declared in the description into properties in the format supported by the TINA model checkers. Verifications of Fiacre properties are then carried out exactly like verification of TINA models properties; in case of failure, a timed scenario can be computed, corresponding to a Fiacre scenario.

4 GenoM3

Functional modules are the building block of LAAS robot architecture functional level [23]. Each module is in charge of a specific function on the robot, from controlling a low-level driver (e.g. motors), a sensor (e.g. laser, camera, etc), up to more complex functionalities (e.g. motion planning, Simultaneous Localization And Mapping "SLAM", etc). These modules are controlled by a supervisor (e.g. OpenPRS[1], eltcsh[2], etc).

GenoM3 [24] is a tool that parses a specification language for functional modules. It provides a template-based generator to synthesize code, libraries, models, etc, from the specification.

A GenoM3 module (Fig. 1) is specified in the language with the following elements:
- an internal data structure (IDS), shared among the services S.

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

[2] https://www.openrobots.org/wiki/eltclsh

- execution tasks ET_i aperiodic or with a period (p_i); each runs the active activity services associated with it.
- services S, that can be of three different types: *attribute* (to set or get an element of the IDS), *function* (for a quick and simple computation) or more interestingly *activity*.
- ports P, in or out, depending on whether the module reads or writes them.
- a list of exceptions for non nominal executions.

Services can take parameters and return values. Each *Activity* executes in an execution task ET that it specifies. It also defines an automaton that specifies for each state:
- the codel [3] to execute in this state, taking as arguments the elements in and out from the IDS, and the ports in and out it needs for its execution.
- the list of states it yields to.
- the WCET (Worst Case Execution Time) of the codel.

All activities have a `start` state-codel, which is the entry point in the automaton, and an `ether` state which is a sink (terminal) state. At runtime, the codel associated with a state must return the state to which it will transition, or throw an exception. An activity thus terminates (transitions to `ether`) either with an exception or with a nominal end. In both cases, exception or return values are reported to the client that requested the activity. A service may have a `validate` codel to validate the in arguments before it runs. Any service may be incompatible with a list of services that need to be interrupted before it executes (including itself if e.g. at most one instance of the same service may run at any time). Interrupted activities execute directly their `stop` codel, if defined, otherwise they transit to `ether`.

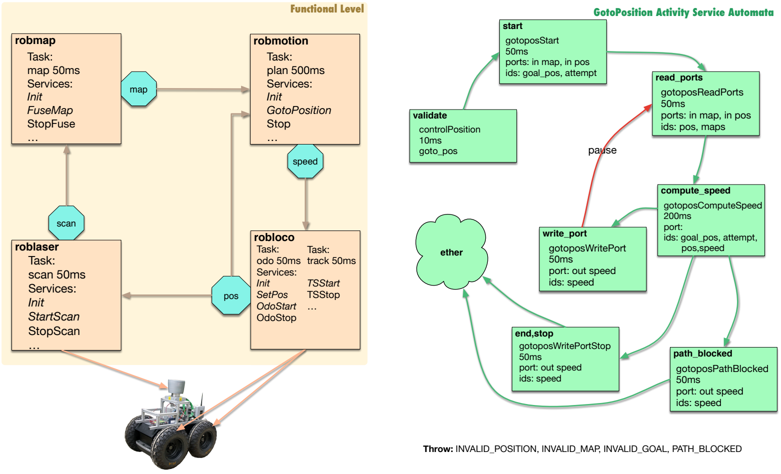
Each module has an implicit aperiodic control task CT, in charge of the module I/O. The CT handles requests from clients as well as the reports upon execution of services. CT is also responsible for executing attributes and functions, and instructing the different ET_i of the activity instances to run or interrupt.

Templates Mechanism The GenoM3 parser builds an abstract syntax tree and converts it into a suitable representation for the scripting language of the template interpreter (TCL). Then, every file of the template is read by GenoM3 and interpreted within this representation. Special markers in the file are detected and their content replaced in a manner similar to how a PHP script is embedded into an HTML page. The scripted code has access to all the information of the module description file. A typical template will consist of regular code, mixed with scripted loops on e.g., services that generate calls to functions of the core libraries. Since the interpreter relies on a complete scripting language, there are virtually no restrictions on what a template can express and synthesize.

There are already templates to synthesize: the module itself for various middleware (e.g., PocoLibs[4], ROS-Com [27], Orocos [15]); client libraries to control the module (e.g., JSON, C, OpenPRS), etc. The template we developed in this work maps the PocoLibs module implementation of any set of modules specified in GenoM3 into their

[3] Codels (code elements) are the programmer implementation of the specified service, broken down to small chunks of C or C++ code.

[4] `https://git.openrobots.org/projects/pocolibs/gollum/index`



(a) The robot navigation with its four GenoM3 modules, their tasks and a partial list of their services (activities are in *italic*)

(b) The GotoPosition activity automaton.

Fig. 2: Modules and an activity service.

timed Fiacre model. From now on, when we refer to a GenoM3 module, we implicitly mean the GenoM3 PocoLibs implementation of the module.

5 Illustrative Example

To illustrate our approach, we introduce a realistic example of a robot navigation implementation (Fig. 2a). This navigation remains generic, in the sense that it could be instantiated with different sensors, motion planners, etc. Yet, the modules, the ports they share, the periods of their internal tasks, the services, their automata and the WCETs associated to their codels are the same as the ones of the real navigation running on our RMP 400[5] robot, Mana.

Fig. 2a presents the four modules in charge of the navigation:

ROBLOCO is in charge of the robot low-level controller. It has a track task (period 50 ms) associated to the activity *TSStart* (TrackSpeedStart, interruptible by the function *TSStop*) that reads data from the **speed** port and sends it to the motor controller. In parallel, one of the odo task (period 50 ms) associated activities, namely *OdoStart* (interruptible by the function *OdoStop*), reads the encoders on the wheels and produces a current position on the **pos** port.

[5] http://rmp.segway.com/tag/rmp400/

ROBLASER is in charge of the laser. It has a scan task (period 50 ms) associated, *inter alia*, to the *StartScan* activity (interruptible by the function *StopScan*). The latter produces, on the port **scan**, the free space in the laser's range tagged with the position where the scan has been made (read on **pos**).

ROBMAP aggregates the successive **scan** data in the **map** port. A fuse task (period 50 ms) and *FuseMap*, one of its activities, perform the computation. The function *FuseStop* interrupts the activity *FuseMap*.

ROBMOTION has one task plan (period 500 ms) which, given a goal position (via the activity *GotoPosition*), computes the appropriate speed to reach it and writes it on **speed**, using the current position (from **pos**), and avoiding obstacles (from **map**). *GotoPosition* interrupts itself, so a new request will cancel the currently running one (if it exists) and force the execution of its `stop` state. Similarly, The *Stop* service (function) interrupts *GotoPosition*.

Each activity introduced above has its own automaton. Fig. 2b presents the automaton of *GotoPosition*. For each state, we define a codel, its WCET, the ports and the elements of the IDS it needs (in and out). Listing 1 presents the GenoM3 specification of this activity.

```
activity GotoPosition (in robloco::position goto_position)
  { validate controlPosition (in goto_position) wcet 10 ms;
    codel <start> gotoposStart(in goto_position, port in pos, port in map,
    ids out goal_pos, ids out attempts) yield read_ports, ether wcet 50 ms;
    codel <read_ports> gotoposReadPorts(ids out pos, ids out explored_map,
    port in pos, port in map) yield compute_speed wcet 50 ms;
    codel <compute_speed> gotoposComputeSpeed(ids in goal_pos, ids in pos,
    ids in explored_map, ids in verbose, /* This codel takes a long time */
    ids out speed, ids inout attempts) /* we make sure no ports are locked */
    yield write_port, end, path_blocked wcet 200 ms;
    codel <write_port> gotoposWritePort(ids in speed, port out speed)
    yield pause::read_ports wcet 50 ms; /* enforces task cycle termination */
    codel <end,stop> gotoposWritePortStop(ids out speed, port out speed)
    yield ether wcet 50 ms;
    codel <path_blocked> gotoposPathBlocked(ids out speed, port out speed)
    yield ether wcet 50 ms;
    interrupts GotoPosition; /* a new instance interrupts a running one */
    task plan; /* the execution task in which the activity will execute */
    throw Invalid_Position, Invalid_Map, Invalid_Goal, Path_Blocked; };
```

Listing 1: The GenoM3 specification of the GotoPosition activity.

Activities are executed in their respective execution tasks. For example, *GotoPosition* executes in the plan task along its period. It begins with executing the state `start` associated codel (`gotoposStart`), and transitioning to the state returned by such an execution (see Section 4). It continues until a transition labeled with *pause* occurs (e.g., Fig. 2b the transition from `write_port` to `read_ports`) or the activity terminates. In either case, the control is given back to the execution task which will then execute the other active instances, if any, or otherwise wait for the next period signal.

For the sake of simplicity, our description of the mechanisms offered by GenoM3 to specify and deploy functional modules remains partial. Still, one can see the complexity raised by running these 4 modules, with 9 threads, 27 services including 10 activities with their respective automata and overall more than 35 codels with their WCET.

The PocoLibs implementation of the module offers a high level of parallelism while preserving shared data access with proper locking. However, it does not offer any guarantee on crucial properties such as schedulability of tasks, boundedness in time of ports updates, proper termination of services, absence of deadlock due to sharing resources, etc.

6 Mapping and Automatic Synthesis

An important step of this work is to automatically synthesize a Fiacre model of any GenoM3 module. All the generic software components (tasks, services, automata, ports, etc) potentially present in GenoM3 modules are formalized into Fiacre. In this section, we analyze through a few illustrative examples how we map some of the GenoM3 module software component into Fiacre processes/components. We then briefly discuss the integration of such a mapping into the translator.

6.1 Mapping

Periodic Execution Tasks Most GenoM3 specifications include periodic execution tasks in charge of executing the activities they manage. We model a periodic execution task with two Fiacre processes. The first one is a simple one-state **timer**. It is in charge of scheduling a **manager** process that manages the execution of the activities. The **timer** (Listing 2–left) assigns, every new time period `PERIOD`, the value `true` to the variable `tick` that it shares with the **manager**. The **manager** starts only when `tick` is true and switches this flag to false. If there are active activities in this task, it transitions to the state `manage` and executes them accordingly (Section 5). It does not transition back to its initial state, `start`, unless all eligible executions in this cycle have ended.

```
process timer (&tick: bool) is
  states start
  from start
     wait [PERIOD,PERIOD];
     tick := true;
     to start
```

```
process Manager (&tick: bool, ...) is
  states start, manage
  from start
     wait [0,0];
     on tick;
     tick := false;
     if (...) /* no active activity */
     then to start
     else to manage end
  from manage
     wait [0,0];
     ... /* execute one active activity */
     if (...) /* no more activities */
     then to start
     else to manage end
```

Listing 2: Fiacre models of an execution task timer and manager (simplified)

State-Codels, WCETs and Concurrency PocoLibs ensures proper locking of the resources the state-codels share (in the IDS for each module and among ports across multiple modules). In this context, state-codels are categorized as either **thread-safe** or **non-thread-safe**. A thread-safe state-codel runs with no condition on resources availability (e.g., it uses no shared resources or uses some of them exclusively) while a non-thread-safe one needs to have all the resources it accesses unlocked so it can execute. A thread-safe state-codel is mapped into a single Fiacre state with every transition out of it associated with the firing interval]0,WCET]. The target states correspond to the yield values of the state-codel (Section 4). In contrast, we map a non-thread-safe state-codel into two Fiacre states (listing 3), the first for waiting and the second for executing. The transition from the first to the second fires providing no conflicting state-codel (i.e. potentially locking at least one of the needed resources) is in its Fiacre executing state.

```
from NTS_WAITING
   wait [0,0];
   on CONFLICT_1 = NOT_RUNNING and CONFLICT_2 = NOT_RUNNING ...;
   NTS := RUNNING;
   to NTS_EXECUTING
from NTS_EXECUTING
   wait ]0,WCET];
   NTS := NOT_RUNNING;
   ... /* possible transitions */
```

Listing 3: Fiacre model of a non-thread safe state-codel (simplified)

Activities A GenoM3 activity is mapped into a Fiacre process. The latter is constituted of the states corresponding to the activity state-codels and the transitions corresponding to their yield values. The task manager (Section 6.1) communicates with the activities to ensure a correct execution of active instances (Section 5). This communication, not shown in our listings for the sake of simplicity, also includes the proper handling of potential interruptions. Listing 4 is a simplified, symbolic, view of the Fiacre process corresponding to the *GotoPosition* activity (Section 5, start state-codel only).

```
process GotoPosition_plan (.../*shared variables*/) is
  states start, start_2, read_ports, ...
  from start_waiting
     wait [0,0];
     on (...) /* wait until the manager allows me to run */
     if (...) /* interruption signal (by manager) */ then to stop_
     /* otherwise, nominal execution: */
     else on (GetGoalPosition = NOT_RUNNING and ... );
     GotoPosition_plan_start := RUNNING;
     to start_executing
     end
  from start_executing
     wait ]0,0.05]; /* the WCET of the Start codel */
     /* back to not running when leaving: */
     GotoPosition_plan_start := NOT_RUNNING;
     select
     /* non determinism. Either to read_ports: */
     to read_ports_waiting
     /* or to ether */
   [] ... /* back to the manager */; to start_waiting /* terminate */
   end
from read_ports_waiting
   ...
```

Listing 4: Extract from the GotoPosition Fiacre process (simplified)

6.2 Automatic Synthesis

For producing Fiacre models from GenoM3 descriptions, we rely on the generic template mechanism provided by the GenoM3 environment (Section 4). Fiacre models are generated fully automatically from unrestricted GenoM3 descriptions.

The mappings from GenoM3 to Fiacre, some of which are discussed in Section 6.1, have been carefully chosen so that the behavior of the generated Fiacre models faithfully represents that of the GenoM3 module. The translation of GenoM3 descriptions into Fiacre gives them a formal semantics, which enables verification of real-time properties as illustrated in the next section. This constitutes the very first formalization of GenoM3 specifications.

7 Experiments and Discussion

In this section, we rely on the automatically generated Fiacre models (Section 6) of our GenoM3 modules (Section 5) to express and assert various temporal/timed properties using, respectively, Fiacre and TINA (Section 3) on each individual module but also on all the four modules together with a realistic navigation scenario. We assume that the targeted robotics platform is real-time, has enough cores to run all the tasks in parallel[6] and that the affinity is set to one core per task. All experiments are carried out on a typical mid-range computer; Intel Core i5 2.7 GHz with 8 GB of RAM.

7.1 Single Module Verification

The Fiacre model of a module cannot be analyzed without embedding it in a “system model”, i.e., we need a “client” to synthesize the possible requests the module may serve. For this, the template automatically produces a Fiacre process able to send any type of request at any given time. Clearly, the Fiacre model of the module and that of its client form a system that will cover all the reachable states the real module may encounter when evolving alone (no interactions through ports).

Schedulability We refer to an execution task as **schedulable** if it never overruns its period. We start with the most complex module, ROBLOCO (Section 5), involving three

[6] The template still provides the user with the possibility to specify their hardware constraints.

tasks running in parallel and a number of services. The modeling choice made in Section 6.1 for periodic tasks permits an easy expression of the property for both execution tasks. E.g. for task **odo**:

```
property sched_odo is always ((robloco/odo_manager/state manage) ⇒
 not (robloco/odo_manager/value tick_odo))
```

The idea is that a period is violated only if a new period tick occurs while an activity is being executed (see Section 6.1). This modeling choice makes it easy to express schedulability properties not with timed properties (implying a larger state space to analyze) but with reachability properties (which do not involve traces). Thus, we can use the coarser TINA construction that does not preserve firing sequences (smaller state space). We end up with a manageable state space (built in about 18 mn) with 10 857 940 classes all obeying the property, for both tasks. We then assert the truth of the same property on the remaining modules execution tasks in less than 1 mn overall.

Safe end Let us consider the module ROBMAP (Section 5). We assert the following safety property: when the module terminates following a *kill* request, all services have properly terminated. This translates in Fiacre to: for all execution paths, if the process *timer* is in state *shutdown*, then the function *running* returns false for all services instances:

```
property safeshut is always ((robmap/timer/state shutdown) ⇒
 not (robmap/manager/value (running (.../*all services instances*/))))
```

Indeed, our unique execution task's timer transits to the state *shutdown* when the module is killed, and the function **running** evaluates to true only if at least one of the services is running or being interrupted. Again, this property is an invariant and does not rely on traces. In our experiment, this property can be checked true by exploring a state class graph with 125 606 classes and 102 512 markings, computed in 4 s.

7.2 Full Perception-Plan-Action Loop Verification

We synthesize the global Fiacre model for the four modules communicating through ports. We then add a Fiacre client defining a full-navigation scenario.

Schedulability of tasks and progress of activities One of the properties we succeeded to verify in Section 7.1 is the tasks schedulability. However, this property is not necessarily preserved when the modules evolve altogether. Indeed, ports, properly synthesized in the Fiacre model of the four modules, constitute another shared resource that codels use concurrently. Since we already saw how to express the schedulability properties in Fiacre, we will skip directly to the results. Interestingly, no task respects its period anymore, except for the task plan of the module ROBMOTION.
These results lead us to further investigation. For the tasks becoming non schedulable, could it be possible that some activities are infinitely blocked while waiting for some ports to be free. Once more, our modeling choices, particularly the ones pertaining to periodic tasks and their associated activities (Section 6.1), allow an easy and quick verification of such a property. Using the Fiacre pattern **leadsto** (Section 3), we simply

check that none of the involved tasks is forever blocked in its state manage. E.g. for the task track:

```
property no_block_track is (navigation/robloco/track_manager/state
    manage) leadsto (navigation/robloco/track_manager/state start)
```

The property holds for all the concerned tasks. Since no activity is infinitely blocked, which would be definitely worrying, the programmer has to decide whether it is critical for this application to have all the tasks schedulable. If yes, they may consider tuning the different periods while watching the effects on the robot performance. Our experiments show that, for instance, doubling the period of the task track renders it schedulable. All results are obtained in less than 40 s overall.

***speed/pos** update bounded in time* One important aspect in our navigation would be the time elapsed between a **pos** update, and the next **pos** update following a **speed** update with all the ports properly updated in-between (**scan**, **map**). We proceed by first checking upper bounds on each part of the whole loop between the subsequent events, using the Fiacre timed pattern **leadsto within** (Section 3), then summing these bounds. We consider all scenarios including a blocked path or a reached goal. The state spaces computation time range from 10 s to 30 s. The result is 1.274s, which is acceptable considering the maximum speed of the robot and its laser range.

Safe stop We extend the client providing the navigation scenario to generate, at any given moment, a *TSStop* (Section 5) request. We again rely on the pattern **leadsto within** (Section 3) to formally prove that the generation of such a request leads to the termination of the running instance of *TSStart* in a maximum duration. We proved that this duration amounts to 72 ms with the model checker succeeding to find a counterexample for the next smaller value (i.e 71 ms). Since the end of *TSStart* (through an interruption) means necessarily sending a null speed to the motor controller, the programmer may deduce critical information from this current setup, e.g., that the robot driving at 2 m/s will advance at least 0.14 m before a full stop. The reachability graph features 1 484 091 classes built in about 3 mn.

8 Conclusion

We formally check important real-time properties on the navigation modules of our RMP 400 robot. Our results compare favorably with previous works. In contrast to [2] and [13], we take all time constraints into account. Also, unlike the related works cited in Section 2 ([2] aside), we provide a fully automatic translation from GenoM3 specifications to the equivalent Fiacre model. Finally, and compared to [19], our experiments tackle examples of a high complexity and a maximum level of parallelism. We succeed to check nontrivial timed properties; no property verification lasted more than 18 mn (including the time needed to generate the reachability graph). These promising results are mainly due to careful modeling choices (Section 6 & 7) and optimized state space generation techniques implemented in TINA.

Formalizing GenoM3 specifications (Section 6) remains the hardest part of our effort. Several PocoLibs/GenoM3 aspects were not trivially expressible in Fiacre. Moreover, avoiding the combinatory explosion of model checking was challenging. Modeling choices were systematically assessed considering, despite their correctness, the ability to express and verify important properties on them. Many of such choices were thus discarded or refined throughout the process. As a consequence, the resulting models for our experiments remain scalable.

An unexpected result from this work is that transforming a GenoM3 specification and its PocoLibs implementation in a language like Fiacre, with a clear formal semantics, forced us to clarify some of the implementation choices and fix bugs.

A limitation of our approach in its current setting is that we need a fine knowledge of the Fiacre model produced by the translator from GenoM3 in order to express properties. For the robot programmers, it would be more convenient to express the properties within GenoM3, in a language they are familiar with. As a next step, the template will include the translation of properties expressed in GenoM3 into Fiacre/TINA properties. As for the verification results, when TINA evaluates the property to false, it would be equally important to automatically interpret the counterexample into what the robot programmers would easily grasp (that is at GenoM3 level), so they can act accordingly.

Last, robotic platforms seldom offer enough cores/processors to run all the tasks/threads of a realistic application in parallel. Thus, we now aim to verify real-time properties while taking into consideration the actual hardware constraints.

References

[1] 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.

[2] 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.

[3] N. Abid, S. Dal Zilio, and D. Le Botlan. A formal framework to specify and verify real–time properties on critical systems. *International Journal of Critical Computer-Based Systems*, 5(1):4–30, 2014.

[4] A. Basu, M. Bozga, and J. Sifakis. Modeling Heterogeneous Real-Time Components in BIP. In *SEFM*, pages 3–12, 2006.

[5] S. Bensalem, M. Bozga, T. H. Nguyen, and J. Sifakis. D-finder: A tool for compositional deadlock detection and verification. In *CAV*, 2009.

[6] 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 *ERTS*, Toulouse, 2008. HAL - CCSD.

[7] B. Berthomieu, S. Dal Zilio, and Ł. Fronc. Model-checking real-time properties of an aircraft landing gear system using fiacre. In *ABZ 2014: The Landing Gear Case Study*, pages 110–125. Springer, 2014.

[8] B. Berthomieu and M. Menasche. An enumerative approach for analyzing time Petri nets. *IFIP Congress Series*, 9:41–46, 1983.

[9] 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*, 42(14):2741–2756, 2004.

[10] B. Berthomieu and F. Vernadat. *State Space Abstractions for Time Petri Nets*. Handbook of Real-Time and Embedded Systems, CRC Press, Boca Raton, FL., U.S.A., 2007.

[11] P.-A. Bourdil, B. Berthomieu, and E. Jenn. Model-checking real-time properties of an auto flight control system function. In *IEEE ISSREW*, 2014.

[12] F. Boussinot and R. de Simone. The ESTEREL Language. In *Proceeding of the IEEE*, pages 1293–1304, September 1991.

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

[14] D. Brugali. Model-Driven Software Engineering in Robotics. *IEEE Robotics and Automation Magazine*, 22(3):155–166, September 2015.

[15] H. Bruyninckx. Open robot control software: the OROCOS project. In *IEEE International Conference on Robotics and Automation*, 2001.

[16] A. Cimatti, M. Roveri, and P. Bertoli. Conformant planning via symbolic model checking and heuristic search. *Artificial Intelligence*, 2004.

[17] E. Dolginova and N. Lynch. Safety verification for automated platoon maneuvers: A case study. In *Hybrid and Real-Time Systems*, pages 154–170. Springer, 1997.

[18] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in property specifications for finite-state verification. In *ICSE*, 1999.

[19] B. Espiau, K. Kapellos, and M. Jourdan. Formal verification in robotics: Why and how ? In *International Symposium of Robotics Research*, 1996.

[20] N. Gobillot, C. Lesire, and D. Doose. A Modeling Framework for Software Architecture Specification and Validation. In *LLNCS: Simulation, Modeling, and Programming for Autonomous Robots*. Springer International Publishing, 2014.

[21] D. Hähnel, W. Burgard, and G. Lakemeyer. GOLEX—bridging the gap between logic (GOLOG) and a real robot. In *KI Advances in Artificial Intelligence*, pages 165–176. Springer, 1998.

[22] F. Ingrand and M. Ghallab. Deliberation for autonomous robots: A survey. *Artificial Intelligence*, 2015.

[23] F. Ingrand, S. Lacroix, S. Lemai-Chenevier, and F. Py. Decisional autonomy of planetary rovers. *Journal of Field Robotics*, 24(7):559–580, 2007.

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

[25] F. Maraninchi. Operational and compositional semantics of synchronous automaton compositions. In *CONCUR*. Springer, 1992.

[26] P. M. Merlin and D. J. Farber. Recoverability of communication protocols: Implications of a theoretical study. *IEEE Trans. Comm.*, 24(9), 1976.

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

[28] S. Rangra and E. Gaudin. SDL to Fiacre translation. *Embedded Real-Time Software and Systems, Toulouse*, 2014.

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

[30] D. Simon and A. Joubert. Orccad: Towards an open robot controller computer aided design system. Technical report, Research Report 1396, INRIA, 1991.

[31] H. H. Täubig, U. Frese, C. Hertzberg, C. Lüth, S. Mohr, E. Vorobev, and D. Walter. Guaranteeing functional safety: design for provability and computer-aided verification. *Autonomous Robots*, 32(3):303–331, December 2011.

[32] T. Wongpiromsarn and R. M. Murray. Formal verification of an autonomous vehicle system. In *Conference on Decision and Control, May*, 2008.