The Journal of Systems & Software 181 (2021) 111033

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

HIPPO: A formal-model execution engine to control and verify critical real-time systems☆

Pierre-Emmanuel Hladik *, Félix Ingrand, Silvano Dal Zilio, Reyyan Tekin

LAAS-CNRS, Université de Toulouse, CNRS, INSA, Toulouse, France

ARTICLE INFO

Article history:
Received 20 November 2020
Received in revised form 13 June 2021
Accepted 16 June 2021
Available online 28 June 2021

Keywords:
Verifiable implementation
Formal toolchain
Robotic case study

ABSTRACT

The design of embedded real-time systems requires specific toolchains to guarantee time constraints and safe behavior. These tools and their artifacts need to be managed in a coherent way all along the design process and need to address timing constraints and execution semantic in a holistic way during the system's modeling, verification, and implementation phases. However, modeling languages used by these tools do not always share a common semantic. This can introduce a dangerous gap between what designers want to express, what is verified and the behavior of the final executable code. In order to address this problem, we propose a new toolchain, called HIPPO, that integrates tools for design, verification and execution built around a common formalism.

Our approach is based on an extension of the FIACRE specification language with runtime features, such as asynchronous function calls and synchronization with events. We formally define the behavior of these additions and describe a compiler to generate both an executable code and a verifiable model from the same high-level specification. The execution of the resulting code is supported by a dedicated execution engine that guarantees real-time behavior and that reduces the semantic gap between high-level models and executable code.

We illustrate our approach with a non-trivial use case: the autonomous navigation of a Segway RMP440 robotic platform. We describe how we derive a HIPPO model from an initial specification of the system based on the robotics programming framework GenoM. We also show how to use the HIPPO runtime to control this robot, and how to use formal verification in order to check critical properties on this system.

1. Introduction

The design of embedded real-time systems requires specific toolchains to guarantee time constraints and safe behavior. These tools, and their artifacts, need to be coherently managed all along the design process and need to address timing constraints and execution semantic in a holistic way during the system's modeling, verification, and implementation phases.

This paper presents such an integrated toolchain, named HIPPO, that focuses on the generation of formally verifiable, real-time applications. More precisely, we focus on the solutions adopted in order to guarantee that the timing constraints expressed in our (formal) model of the system are, beyond verification with model checking, transcribed and enforced in the executable. This is a classical problem, widely discussed in the literature. A difficulty often mentioned in this context is that we should be wary of semantic gap between the models produced by the designer, the models used for verification, and the executable.

To overcome this pitfall, we propose to build our approach around the formal specification language FIACRE (Berthomieu et al., 2008a). This language has several nice features. First, it is rich enough to model the behavioral and timing aspects of concurrent systems and it already comes with abstractions (concurrent processes, ports, etc.) and a rich type system (including records, arrays, fifo queues, etc.). Moreover, FIACRE has a formal semantics and can be used with model-checkers in order to check timed and temporal properties on a given model. Finally, we can reuse several tools that have already been developed around this language, such as code editors or libraries to perform simulations.

Our approach relies on a dedicated *compiler*, called `frac`, that can transform a FIACRE model into a Time Transition Systems (TTS) (Berthomieu et al., 2008a), a low-level representation of the possible synchronizations and state changes in the system. The TTS level plays a role similar to assembly code, where FIACRE is the high-level language, and the behavior of a FIACRE "program" is defined as the semantics of its TTS.

☆ Editor: A. Bertolino.

* Corresponding author.
E-mail addresses: pierre-emmanuel.hladik@laas.fr (P.-E. Hladik), felix.ingrand@laas.fr (F. Ingrand), silvano.dal.zilio@laas.fr (S. Dal Zilio), reyyan.tekin@laas.fr (R. Tekin).

https://doi.org/10.1016/j.jss.2021.111033
0164-1212/

For the Hippo toolchain, we chose to generate code at the TTS level, so the compilation result can stay very close to the semantics of the initial model. An advantage of this choice is that we only need to rely on a simple runtime, that is used to ensure that the control flow of the executable is subsumed into the behavior of the TTS model. As a consequence, we guarantee that the behavior of the generated code, coupled with the Hippo engine, is included in the behavior of the formal model.

Outline and contributions of the paper. The remainder of the paper is structured as follows. Section 2 gives an overview of works relevant to the generation of real-time verifiable executable and analyzes their shortcomings, which we want to address in our approach. Our results are organized along three main parts.

The first part (Section 3) is dedicated to a presentation of Hippo. We give a high-level overview of the Fiacre specification language and describe how Hippo is obtained from Fiacre with the addition of tasks and events. We explain how each of the new constructs can be interpreted inside of Fiacre, which gives a formal definition of these extensions and allow us to apply model-checking tools on a model.

The second part (Sections 4 and 5) is dedicated to the description of the execution engine and its performance. We describe the design principles of the Hippo runtime in Section 4. The code generation and its associated runtime are based on a software design where the control behavior is implemented synchronously and the execution of the functional processes are managed by an asynchronous scheduler. An overview of the structure of the code generator is given as well as the orchestration of the execution engine. A focus is also made on the way scheduling is managed and on different solutions to take it into consideration during the specification and verification phases. This section also includes a discussion on the methods used to increase our confidence in the implementation of the Hippo engine. We report some experimental results on the performance of the engine in Section 5 by studying time overheads of the engine and the effect of model's size on CPU usage.

The last part (Sections 6 and 7) is a detailed description of a complex, real-life case study. We describe how we deploy Hippo and Fiacre along G^{en}oM, a robotics programming framework, and how we apply it to a complex autonomous outdoor robotic platform. We also report on the results obtained with online run-time monitoring and offline verification of this autonomous robot.

We make contributions in each of these three parts. At the language level, we describe an executable specification language that is expressive enough to control complex systems, while retaining the possibility to perform formal verification on the concurrent and real-time behavior of a model. At the execution level, we describe a real-time engine that enforces the predictability of the system and that reduces the semantic gap between a specification and its implementation. Lastly, in our experiments, we give an example of how to use our toolchain to model and formally verify a complete and realistic robotic system. More generally, since the tooling developed for our use case supports all the features of G^{en}oM, we can apply our approach on all the robotic experiments built using G^{en}oM.

2. Motivations and related work

We now examine the motivation for this work, some of the related and relevant works and how we bootstrapped the Hippo toolchain.

2.1. A language-based solution for solving the semantic gap issue

Many high-level languages have been proposed to facilitate the design of real-time embedded systems. For example, there are generic languages like UML with specialized versions such as MARTE (Object Management Group, Inc. (OMG), 2009) or some Domain-Specific Languages such as AADL (Feiler et al., 2006). The typical use of these languages in a design process allows the designer to produce a high-level model that is refined to obtain a detailed model of the system's behavior. This model is then used as an input for verification activities and then coding activities. These high-level languages are seldom formally defined and the verification process usually begins with a translation step in a formalism that allows verification (see Fig. 1). In addition, depending on the property to be checked (schedulability, liveness, buffer size, etc.), it is possible that different abstractions may be required, thus producing multiple models.

Another source of murkiness lies in transformation of the model into executable code, where processes can vary a lot: it can be done entirely manually; semi-automatically (for example by producing a code skeleton); or fully automatically. For example, the design of an embedded system with AADL can use Cheddar (Singhoff et al., 2004), MAST (Harbour et al., 2001), Tina (Berthomieu et al., 2004), etc. to verify properties and Ocarina (Lasnier et al., 2009) can be used to generate code. In this example, there is no guarantee that the execution semantic considered by these different tools are strictly the same.

The main problem with these approaches is that a significant semantic gap can exist between verified models and executed ones. This problem comes from the fact that since the behavior of the high-level model is not formally defined, the transformations cannot be validated and there is no guarantee that the verified behavior is exactly the one that will be executed.

As a matter of proof, we observe that there are few examples, in the literature, of real and fully functional applications where verification and code generation are performed jointly.

Our main motivation in this work is to propose a tool to reduce this semantic gap for real-time embedded applications and show its applicability on a real case study. In particular, beside a description of the Hippo framework, we provide a complete and documented example of complex critical real-time system for which we apply our approach during the design, code generation and verification stages.

2.2. A concern for pragmatism

Another motivation is to follow a practical approach. Obtaining a toolchain that allows modeling, code generation in a faithful way, and checking formal properties presents many difficulties. One of these difficulties lies in proving that the multiple transformations preserve semantics. A possible solution in this case is to write a formally verified translator, using for instance an interactive theorem prover. We decided not to follow this approach for pragmatic reasons.

Our main goal in this work is to show the feasibility of a toolchain based on Fiacre and Tina coupled with a study of its applicability on a real case. In order to gain some trust on the quality of our tools, we choose to use a testing based approach. At the present time, we cannot state that Hippo provides an absolute guarantee of correctness, but we describe a method for testing that the behaviors (traces) observed in a real execution are valid executions in the formal model.

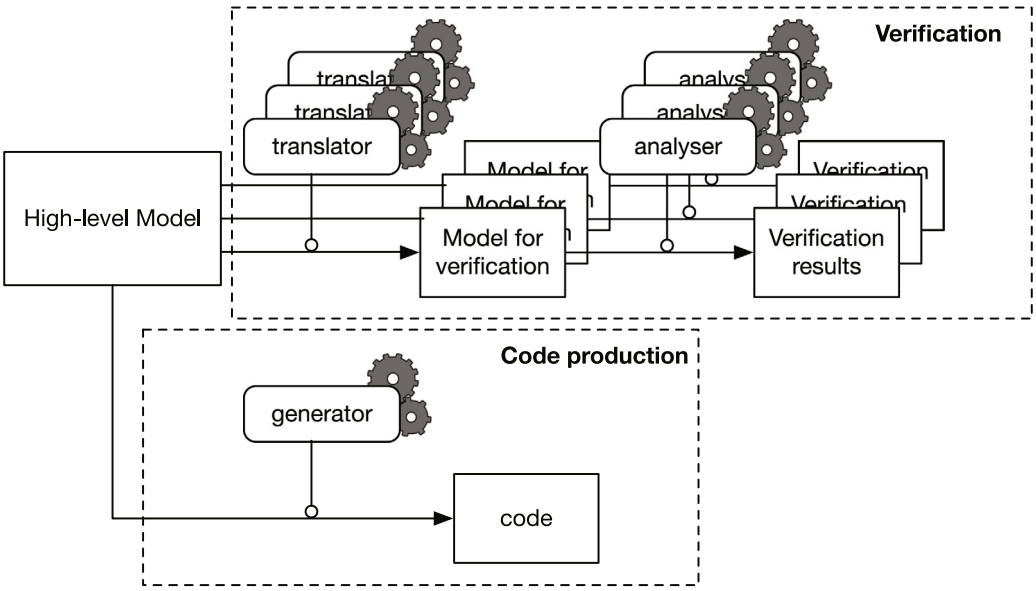


Fig. 1. A schematic representation of a generic design process for real-time embedded systems.

2.3. Related work

There are many examples of work interested in the generation of executable code (an *implementation*) whose behavior is consistent with the model of a system (its *specification*). The list of contributions presented in this section is far from exhaustive and we mostly focus on work that address formal languages, a time-triggered model of computation, and code generation for embedded systems.

Synchronous models. A first body of work is related to models based on a logical (and therefore discrete) notion of time. The most notable example is SCADE, an industrial toolbox based on the synchronous language Lustre (Halbwachs et al., 1991). SCADE is perhaps the best-known example of a software product that proposes a formally defined modeling language; tools to model-check behavior; and tools to generate faithful code. A dedicated and certified compiler can generate C or Ada functions from a SCADE sheet which will execute with the same behavior on an embedded target. On the other hand, the operating assumptions of this approach are quite strict, since they rely on the *synchronous paradigm*, which entails a logical abstraction of time.

In the same category, another well-known toolbox is Simulink, developed by Mathworks. Simulink provides a compiler generating C code for a large number of targets. The code generator is highly configurable and is mainly based on an engine with periodic tasks. The used methods do not guarantee a faithful executable but some extensions exist to connect a subset of Simulink to SCADE (Caspi et al., 2003).

Another approach based on the synchronous paradigm is Prelude (Forget et al., 2009), a data-flow synchronous language with support for multi-periodic systems. The toolchain for Prelude includes a compiler that generates a set of real-time tasks programmed in C with POSIX. This framework was extended in Pagetti et al. (2018) to generate time-predictable code targeting multicore platforms.

Event-based models. Other works rely on an event-based model, such as Ptolemy (Liu and Lee, 2002) developed at UC Berkeley. Ptolemy provides a singular example since its semantics can support a combination of continuous time and synchronous time events (Lee and Sangiovanni-Vincentelli, 1996). Nonetheless, when used as an execution engine, rather than for simulation, Ptolemy relies on an event-triggered programming model where actions are controlled via deadlines and events. This work was the first step in the development of another approach, called Ptides (Derler et al., 2008), based on a discrete-event model that offers a formal semantics to achieve deterministic behavior in both time and value.

Another model sharing similarities with Ptides is Giotto (Henzinger et al., 2003), a language for modeling control systems with periodic activity and data exchange. The semantic of Giotto is based on the Logical Execution Time (LET) assumption, and a compiler can generate an executable that respects this paradigm. The execution engine is based on a simple synchronous virtual machine (Henzinger and Kirsch, 2007) and guarantees the same behavior of the model and the execution; however, the language is not formally defined. Our approach is greatly inspired both by Ptides and Giotto for the choice of a "time-deterministic" model of computation.

A similar motivation can also be found in the design choices behind OASIS (Louise et al., 2011), a framework provided by the CEA LIST to generate an executable based on a time-triggered approach. The temporal information in an OASIS model is directly specified in the code using a dedicated language, called ψC. This language introduces synchronization instants that need to be checked during the execution while execution flow is controlled by an automata. A specific engine is implemented to perform the execution of the automata and to guarantee the temporal constraints, whereas concurrency between tasks is delegated to the operating system. The design principles of OASIS are more focused on dependability and certification issues, rather than on formal verification of properties related to the system's behavior. Nonetheless, this work is interesting in our context since it shows that it is possible to implement a very efficient and portable execution layer based on a time-triggered approach, with very low latency. We apply some of the same ideas in the implementation of Hippo.

Process algebraic approaches. Another interesting set of work is related to the use of "process algebra" for the specification of systems. Indeed, part of the semantics of Fiacre can be traced back to the LCS language of Berthomieu (Berthomieu and Le Sergent, 1994) (one of the designers of Fiacre). LCS is a high level, asynchronous parallel programming language based upon the behavioral paradigms introduced by CSP and CCS. Fiacre retains

some characteristics of LCS, such as a component-based design; a very versatile type system; and the use of "channel-based" synchronization primitives. On the other hand, Fiacre descriptions may be constrained in order to keep the state space of systems finite (for the purposes of model-checking). Another major addition is the possibility to define real-time constraints on the synchronization between processes, using a dense time model, as well as time-outs on events.

Another ancestor in the genealogy of Fiacre is LOTOS (Garavel et al., 2017), a formal specification language that includes concurrent processes, for describing the control part of distributed systems, together with support for rich data structures. A key difference with our approach is that LOTOS does not provide support for expressing timing constraints. Nonetheless, in the absence of time, it is possible to compile a Fiacre specification into LOTOS using a compiler called `flac`. LOTOS models can be formally verified using the CADP toolbox, which also includes the EXEC/CÆSAR tool for generating executable C code (Garavel et al., 2001). This work was extended in Evrard and Lang (2015) to support the generation of systems of distributed tasks, able to synchronize using a multiway rendezvous. Our current implementation of Hippo does not support distribution of code; hence we do not share the same concerns regarding the implementation of synchronization in our engine.

Similarly to Fiacre, the BIP framework (Sifakis, 2005) developed at Verimag is a formal language, and a process algebra, used as the input language in a formal verification tool (RT-DFinder). This framework is particularly interesting in our context since it provides a compiler from BIP specifications into the *BIP Execution Engine*. The BIP language offers a component-based semantic to design concurrent systems that communicate via ports. A model in BIP can be compiled to generate an executable in C++ which, together with the execution engine, enforces real-time constraints. While the initial BIP implementation did not explicitly take time into account, a distributed and real-time implementation of BIP was recently proposed (Dellabani, 2018). The work presented in this article uses the same approach as BIP but exploiting a different set of formal verification tools (Fiacre and Tina). Moreover, the runtime implementation of Hippo guarantees a synchronous behavior coupled with an asynchronous scheduling in order to facilitate the verification, which is not the case of BIP.

Automata-based models. Another related work is CPAL (Cyber–Physical Action Language), a language to model, simulate, verify and program Cyber–Physical Systems (Navet et al., 2016). CPAL is jointly developed at the University of Luxembourg and by the company RTaW since 2011. This language is based on synchronous programming approach and time-triggered languages such as Giotto. The syntax of CPAL provides concepts specific to embedded systems with a formal execution semantics. CPAL also provides a faithful real-time execution engine for embedded systems. To our knowledge, the CPAL language is not formally defined, even if the processes are *Finite State Machines*, and no tools are available to model-check it. However, the proposed scheduling analysis approach is a source of inspiration for Hippo and future extensions. Similarly, the implementation choices for simulation and execution offer interesting leads for future work.

Some studies have also been carried to generate code from timed automata (TA). For example, Amnell et al. in Amnell et al. (2002) proposes a method for generating C code from TA models extended with a notion of real-time tasks that allow them to check the behavior of a model and its schedulability. In the same context, Kristensen et al. (2017) proposed a tool to generate executable code from a deterministic semantic simplification of a given real-time model in Uppaal. To our knowledge, these works have never been integrated into a design process, nor coupled with high-level languages.

2.4. Hippo *toolchain*

Our work takes place in the context of the Tina toolbox and the Fiacre modeling language. Tina (Berthomieu et al., 2004) is a toolbox for the editing and analysis of Petri Nets, Time Petri Nets (TPN) and an extension of Time Petri Nets with data handling and priorities called Time Transition Systems (tts).

A tts is a generalization of a TPN with data variables. Data are managed with expressions that may be associated with transitions: a guard predicate `pre` and an action function `act`. These expressions may refer to a fixed set of variables that form the data set of the tts. For a transition `t` with guards `pre_t` and `act_t`, we have that `t` is enabled in a tts if there are both: (1) enough tokens in the places of its pre-condition; and (2) the predicate `pre_t` is true. When `t` is fired, the marking of the underlying Petri net is changed and the data set is updated by executing the action guard `act_t` .

Fiacre (Berthomieu et al., 2008a) is a mature language, with a long history of deployment in academic and industrial projects. It was designed as a pivot language and an interoperability format (an intermediate format) to simplify the connection between high-level modeling languages, such as AADL or SysML, and model-checking tools inside the Topcased (Berthomieu et al., 2008b) environment. The main purpose of Fiacre is to allow the modeling of the behavioral and timing aspects of the system for formal verification.

Generally speaking, the tool we propose uses Fiacre as an input language but is based on tts for its semantics. Our goal is that every trace played by the engine should be a trace of its tts model. Also, like in tts, we ensure the atomicity of operations: it is not possible to observe a state during the firing of a transition. Coupled with the tools for translating high-level languages to Fiacre, it is thus possible to obtain a complete chain allowing to check the behavior of the system on a model where the semantic gap with the execution is lessened (see Fig. 2).

3. Fiacre extensions

We briefly present the Fiacre language, the proposed H-Fiacre extensions and their semantics.

3.1. *The* Fiacre *language*

A presentation of the Fiacre language is available in Berthomieu et al. (2020). In order to illustrate its main elements (this presentation is not exhaustive and does not show, for example, the subset of Fiacre used to define functions, akin to a first-order functional language), an example is given in Listing 1. This example, based on Carruth and Misra (1996), is taken from the official documentation and models the Fischer protocol which ensures mutual exclusion among N processes using real-time clocks and a shared variable `lock`.

Fiacre is a component-based language of concurrent systems. We briefly describe the features of Fiacre by looking at the code in Listing 1, which defines a system with a single component (`Main`) built from two instances of the same process (`Proc`), with different `id` but sharing a common lock. A Fiacre specification is composed of parallel processes (line 3) communicating via ports and/or shared variables (`lock` line 3). A process describes the behavior of sequential components and is defined by a set of control states (line 4), each associated with an expression built from: classical imperative constructs such as assignments (line 10), conditionals (line 16), while loops, pattern matching, and sequential compositions; synchronization on data-event ports (with n-way synchronizations, $n \geq 2$, and communication of values); and jumps to the next state (lines 7, 11, 14, etc.). Processes can

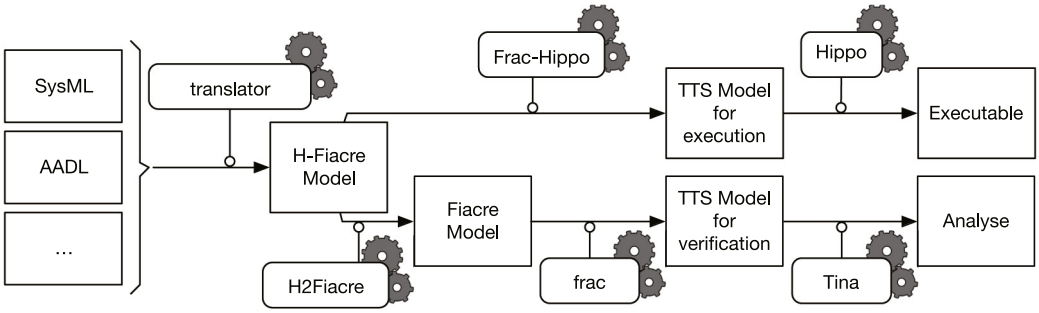


Fig. 2. A schematic representation of the Hippo toolchain.

be composed together into components (lines 25–30), which are also a unit for defining communication ports, priorities between event, and shared variables.

Timing constraints in Fiacre are expressed using its `wait` statement (lines 9 and 13) with an open or closed time interval where bound values are constants in $\mathbb{Q}^+$ (`...` is used to denoted infinity). It means that the control state has to be delayed for a duration within the interval before forwarding.

Priorities can be added between communication events to specify that one event should always occur before another if they happen at the same time.

```
1  type tyEvt is record time : int, id : nat end
2  /* Process */
3  process Proc (pid : id, &lock : lock) is
4     states WaitLock, WaitLock2, SetLock, TestLock,
         CriticalSection
5     from WaitLock
6        on (lock = 0);
7        to WaitLock2
8     from WaitLock2
9        wait [0, 2];
10       lock := pid;
11       to SetLock
12    from SetLock
13       wait ]2, ...[;
14       to TestLock
15    from TestLock
16       if lock = pid then
17         to CriticalSection
18       else
19         to WaitLock
20       end
21    from CriticalSection
22       lock := 0;
23       to WaitLock
24 /* Main component */
25 component Main is
26    var lock : lock := 0
27    par
28       Proc (1, &lock)
29    || Proc (2, &lock)
30    end
```

Listing 1: The Fischer protocol example in Fiacre.

3.2. H-Fiacre: an extension of Fiacre with tasks and events

We want to use Fiacre not only to do verification, but also to generate executable code. Thus, we extend the syntax of Fiacre to take into account tasks and events; this extension is called H-Fiacre. These new operators allow to express, in the model, when to start new computations and how to react to external events.

Functional codes (*i.e.* controller, filter, position estimation, etc.) are embedded into functions that we call *operations*. An operation is basically a C function with input and output data. These operations are called from a H-Fiacre model through a *task*. Each operation has its own task and is scheduled by the operating system (see Section 4 for more details). So, H-Fiacre extends the original language with operators for declaring task activation and termination.

Aside from tasks, we also extend the syntax to declare *external events*, which model events originating from the environment of the system. This is useful, for example, to define how the system should react when an external sensor sends a message or when a signal is emitted from the operating system or the task execution space.

Overall, we extend Fiacre with four new statements: `task`, `start`, `sync`, and `event`.

3.2.1. External task declaration (task)

Tasks are always declared at top-level, like Fiacre functions, processes and components. The syntax for declaring a task is:

```
task t (a1 : ty1, ... , an : tyn) : rty is  c_foo
```

This declares a task, with local name `t`, corresponding to an operation coded by the C function identified as `c_foo`. This identifier is useful during the code generation, when we actually need to link the model with external code. The task in Hippo and the C function should both have the same type; the type of functions with n parameters, of respective types `ty1`, ..., `tyn`, and return type `rty`.

Task activation (start). A task can be activated using the `start` statement, with the name of the task and one expression for each of the required parameters

```
start t (p1, ..., pn)
```

where `p1`, ..., `pn` are the parameters of type `ty1`, ..., `tyn` passed to the function.

Task termination (sync). We can synchronize on the termination of the task `t` and retrieve the returned value using a `sync` statement, as follows:

```
sync t ret
```

The second parameter of a `sync` statement should be a variable of type `rty`. When the statement is executed, `ret` is assigned to the value returned by the "call" to `c_foo`. Since we have records with named fields in Fiacre (a data type similar to `struct` in C) it is easy to define one to return multiple values.

The `start` and `sync` statements are blocking and immediate, *i.e.*, the transitions that represent `start` or `sync` need to be fired as soon as they are enabled. A consequence is that the task cannot be reentrant. Moreover, due to the synchronous implementation of the Hippo runtime, the synchronization of a termination of a task is only done at a global clock tick (see Section 3.3).

3.2.2. External event declaration (event)

An external event is used to model a signal that originates from outside the engine space. An event `e` can possibly carry a tuple of values and is defined using a top-level declaration as follows:

```
event e : ty1 # ... # tyn is c_event
```

where ty1, …, tyn are the types of data bound to the event (or eventually sync if there is no data) and c_event is a symbol matching a C function that catches the event and returns a structured data of type ty1, …, tyn. The # notation is used to be consistent with Fiacre channels where a series of types separated by # are associated with ports transferring several values simultaneously.

To receive and match the values retrieved during a synchronization with an external event, we reuse the syntax for a reception on a regular Fiacre channel:

```
e ? d1, d2, ..., dn;
```

in which d1, . . . ,dn are variables (or possibly patterns) that are assigned with the data retrieved from e.

For the same reason as with tasks, due to the synchronous implementation of the Hippo runtime (see Section 3.3), synchronization on events only occurs at a global clock tick. Likewise, reception on an external event is both blocking and immediate.

3.2.3. Restrictions of H-Fiacre

H-Fiacre does not impose any restrictions on the usage of Fiacre language, except on delays. A first restriction is that we forbid left-open intervals in a wait statement. Moreover, the engine is deterministic and imposes to react as soon as possible. It means that a time interval [a,b] will always "expire" after duration *a*. That is why we should restrict to punctual time constraints, of the form wait [a,a].

3.2.4. A toy example in H-Fiacre

```
1  type tyEvt is record time : int, id : nat end
2  type tyDblEvt is array 2 of tyEvt
3
4  event e : tyEvt is c_click
5  task t (tyDblEvt) : nat is c_print
6
7  process double_event is
8    states wait_first, wait_second, start_print,
       wait_print
9    var tmp : tyDblEvt := [{time=0,id=0}, {time=0,id
       =0}], ret : nat := 0
10   from wait_first
11     e?tmp[0]; // first event, assign tmp[0] value
12     to wait_second
13   from wait_second
14     select
15       wait [200,200];
16       to wait_first
17     []e?tmp[1]; // second event, assign tmp[1] value
18       to start_print
19     end
20   from start_print
21     start t (tmp); // start task t
22     to wait_print
23   from wait_print
24     sync t ret; // wait end of task t
25     tmp := [{time=0,id=0}, {time=0,id=0}];
26     to wait_first
```

Listing 2: An example of model in H-Fiacre: a double events detection.

Listing 2 gives a simple example that illustrates the use of external events and tasks. The purpose of the process double_event is to detect when two occurrences of event e occur less than 200 units of time apart. This can be useful, for example, to implement an anti-rebound function on a safety critical control panel. This event is declared at line 4 and is bound to the C function c_click, which returns a structure with two fields: time and id (line 1).

Task t is defined at line 5 with a two-element table as a parameter. The C function bound to t is called c_print and returns an integer.

The process starts in state wait_first (the first state in its declaration) by waiting for event e, without any constraints on the time that it should wait (line 11). If and when the event occurs, the process transitions to state wait_second where one of two things can happen (declared using a select statement that models a non-deterministic choice between several continuations, separated by []). Either 200 units of time elapse without any event occurring (line 15), or a second e occurs (line 17).

We assume that function c_click returns the date at which the click event occurs (in field time of the record of type tyEvt). Therefore if we reach state start_print then we have recorded a pair of events in variable tmp (lines 11 and 17). This information can be used in function c_print to log the exact delay between the two occurrences of e (line 21). Then the system waits until the end of task t (line 24) before restarting with its initial behavior.

3.3. Semantics of a task and an event

The semantics of our new statements can be formally expressed in Fiacre, and thus defines a transformation to rewrite a H-Fiacre specification into a "pure" Fiacre model.

This rewriting process consists in replacing tasks and events with concurrent processes that model their behavior. Synchronization methods on task activation and termination (start and sync) as well as event arrival are modeled by ports. These ports then carry the data exchanged between the Fiacre model and external tasks and events.

3.3.1. Semantics of an external task

Listing 3 gives the template for the translation into a plain Fiacre model of a H-Fiacre task declared as task t (p:tyIn) : tyOut is c_foo. We suppose that the task t is called *n* times in the model (*i.e.* the statements start, respectively sync, are used *n* times for the task t) then we use an indice *i* to distinguish these calls.

A task t is modeled by a new Fiacre process. For each statements start t (p), respectively sync t ret, in the H-Fiacre model, a port t_activate_i is created, resp. t_terminated_i (see lines 3 and 4). They are used to model the activation, resp. the termination, of the task and to pass parameters, resp. the return value. The *i*th statement start t (p) in the H-Fiacre model is replaced by a port emission t_activate_i!p and sync t ret with t_terminated_i?ret.

The behavior of the task is modeled by different states that express the life-cycle of a task. A state (line 8) synchronizes the task with its activation through one port t_activate_i and copies the parameter values into a local variable param. The select statement ensures that at most one instance of the task can run at any time.

The running state (line 15) calls the functional C code to compute the return value. Note that for the Fiacre model, the call to a function is assumed to be in null time, so the execution time is represented by an interval of numbers (line 18) between its best and worst-case response time. In order to model the real behavior of a task, it is necessary to model the scheduler of the system. Here, we do not model this scheduler and instead represent it by the best and worst-case response times (and not the execution time) (see next Section for more details).

A terminating state (line 25) signals the end of the task and returns the result through port.

An additional state (line 20) is added to represent the time-triggered behavior of the Hippo runtime (see Section 4). The termination signals, t_terminated_i (lines 25–29), need to be

synchronized with the clock of the HIPPO runtime. To do this, a synchronization is added through the port `t_SyncGlobal` (line 22). The global clock process is described in the next section.

```
1  process p_task_t [
2      t_SyncGlobal : none,
3      t_activate_1, t_activate_2, ..., t_activate_n
        : tyIn,
4      t_terminated_1, t_activate_2, ... t_activate_n
        : tyOut
5    ] is
6    states waiting, running, synchronizing,
       terminating
7    var param : tyIn, ret : tyOut
8    from waiting
9      select
10         t_activate_1?param; to running
11     []  t_activate_2?param; to running
12     ...
13     []  t_activate_n?param; to running
14     end
15   from running
16     /* The computational function is called */
17     ret := c_foo(param);
18     wait[$bcrt, $wcrt];  /* simulate the WCRT */
19     to synchronizing
20   from synchronizing
21     /* Synchronization with the global tick */
22     t_SyncGlobal;
23     to terminating
24   from terminating
25     select /* The return value is written */
26         t_terminated_1 ! ret; to waiting
27     []  t_terminated_2 ! ret; to waiting
28     ...
29     []  t_terminated_n ! ret; to waiting
30     end
```

Listing 3: Template to implement a H-FIACRE task "`task t (p:tyIn):tyOut is c_foo`" in FIACRE.

3.3.2. Global clock process

Listing 4 shows the process to synchronize all tasks of a H-FIACRE model on a global clock. We assume that the set of tasks is `t1`, ..., `tm`. Each task has its own synchronization port, for example `t1_SyncGlobal` for task `t1`. The port `nop` is introduced to allow the clock to progress if no synchronization is required and priorities are added (line 16) between `nop` and all other ports to assure that task synchronization always happens before `nop`. This model guarantees that the termination of a task is only signaled at a tick.

```
1  process global_clock [
2      t1_SyncGlobal, t2_SyncGlobal, ..., tm_SyncGlobal
        , nop : none
3    ] is
4    states timer, tick
5    from timer
6      wait[1,1];
7      to tick
8    from tick
9      select
10         nop; to timer
11     []  t1_SyncGlobal; to tick
12     ...
13     []  tm_SyncGlobal; to tick
14     end
15 ...
16 priority
17   t1_SyncGlobal > nop
18   ...
19   tm_SyncGlobal > nop
```

Listing 4: Global clock template to synchronize a set of tasks.

3.3.3. Semantics for an external event

The Listing 5 shows the formal definition in FIACRE for the behavior of an event declared as `event e : tyEvt is c_foo`. This rewriting is very close to that of a task. An event is synchronized to the global clock (line 13) and uses the port `e_happened_i` (indice *i* is used to denote multiple synchronization points) to synchronize the occurrence of the event with the main behavior (lines 16 to 21).

Note that an event in H-FIACRE is similar to a port in FIACRE and that the syntax of the statement `e?d1, . . . , dn` to model an event is exactly the same as a synchronization with a port.

The real difficulty with modeling an event lies in modeling a realistic timed pattern of the event occurrence, *e.g.*, is it a periodic event or a sporadic event with an inter-arrival time? This is a common problem in modeling and is not addressed here. Several studies (Tanguy et al., 2014; Abdellatif et al., 2013) tackle this problem and can be used as a basis to extend H-FIACRE in the future. For the example exposed in Listing 5, we suppose that an event can rehappen at the earliest after 30 units of time and at worst 1000 units.

```
1  process p_event_e [
2      e_SyncGlobal : none,
3      e_happened_1, e_happened_2, ..., e_happened_n :
        tyEvt
4    ] is
5    states waiting, synchronizing, posting
6    var tmp : tyEvt
7    from waiting
8      /* timed pattern to produce event */
9      wait [30, 1000];
10     tmp := any;
11     to synchronizing
12   from synchronizing
13     e_SyncGlobal;
14     to posting
15   from posting
16     select
17         e_happened_0!tmp; to waiting
18     []  e_happened_1!tmp; to waiting
19     ...
20     []  e_happened_n!tmp; to waiting
21     end
```

Listing 5: Template to implemente a H-FIACRE event "`event e : tyEvt is c_foo`" in FIACRE.

4. Execution engine

The execution engine is a critical component in our approach. We present its principle and the specific code which is generated for a particular H-FIACRE model to run with HIPPO. We also describe the implementation of the engine and its threads, as well as the underlying scheduling hypothesis. Some experimental measures of HIPPO performance are given in the next Section.

4.1. Principles of the execution engine

HIPPO tackles the problem of generating code whose behavior is consistent with a model of the system, meaning that every sequence of observable events during an execution should correspond to an acceptable sequence of events (a *trace*) for the model. For a HIPPO execution, these events are transitions fired and time delays at the TTS level. It means that a H-FIACRE model is compiled into a TTS extended with task and event manager and the engine executes the TTS.

The central idea of the execution engine is to separate the execution of the TTS from the execution of functional processes, *i.e.* tasks and events. The control flow of the tasks is caught by the operating system while the TTS execution is done by a time-triggered engine. This means that the HIPPO runtime is based on

a globally asynchronous and locally synchronous approach and that the TTS evolves via a sequence of atomic actions, indexed by a global logical clock, while the functional part is asynchronously executed inside concurrent tasks. Hence we can identify two separate and distinct execution spaces: (1) the TTS *engine space*, devoted to actions that change the state of the H-FIACRE model; and (2) the *task execution space*.

We depict this model in Fig. 3, where we make explicit that events related to the TTS engine are always synchronized on the same clock. At each tick, actions in the TTS are performed until a time blocking situation or an external event waiting. In our context, *time blocking* events correspond to situations in which the system has to wait for an internal event, coming from the TTS engine space. On the opposite, *external events* originate from the task space, such as events generated by a sensor or the termination of a task for instance. Note that, at certain global logical clock ticks, the TTS engine does nothing. This is simply due to the timed events that have not yet expired, *e.g.*, the transition in the TTS that cannot be fired due to its clock race condition.

In Fig. 3, we also make obvious the fact that tasks can be executed concurrently (or even on separate processors). Actually, we are not concerned with the way execution threads in the task space are managed by the underlying operating system scheduler. This is irrelevant from the TTS engine point of view, since only the activation dates of the threads are controlled by the runtime. In the same way, we only observe the termination of a task at a behavior engine tick, even if the task terminates earlier from the operating system point of view.

To summarize, the HIPPO runtime implements an engine that performs actions on the TTS model in a synchronous way, whereas the flow control of tasks is delegated to the operating systems in an asynchronous way. This clear "separation of concerns" helps us enforce a time deterministic model of computation.

4.2. Code generator

The code generator produces C code from a H-FIACRE specification. During this step the H-FIACRE is transformed into TTS that preserves the semantics of the original model. Basically, the TTS is obtained after type checking, propagation of constants, and after all possible synchronization patterns being statically resolved.

In practice, the code generator is written in Standard ML and shares most of its code (and more than 90% of the front-end code) with the tool used to compile a FIACRE specification into a TTS suitable for model-checking. The main difference between the "verification" and the "runtime execution" TTS formats comes from the presence of additional external code in H-FIACRE (such as the code of external tasks).

In HIPPO, two types of C files are used to code a TTS: a file that describes the discrete state and the transitions of the system (which is actually a Time Petri net); and a source file that contains a set of functions describing the guards (conditions that need to be true) and actions (updates which are applied on the variables) for all the transitions.

Basically, HIPPO executes a binary compiled version of the "runtime" TTS obtained from the model and extended with the C code from the operations and the external event handlers (see Fig. 4). The engine is a lightweight middleware that schedules operations in the TTS engine space. As a result, we obtain a self-contained executable file that can be optimized depending on the target architecture and operating system.

For the toy example presented in Section 3.2.4, the TTS is composed of 4 places, 5 transitions, 2 external elements (1 task and 1 event), 4 guard functions and 4 update functions. Two specific data structures have also been generated to manage `tyEvt` and `tyDblEvt` types.

4.3. Engine implementation

The implementation of the HIPPO runtime is presently based on Linux (ideally with PREEMPT_RT) and uses the POSIX services of the operating system with SCHED_FIFO scheduler (similar to a fixed priority scheduler). However, several design choices were made to easily port the runtime to different operating systems. In particular, the used native services were deliberately limited and general to ensure that they would be available on the majority of existing real-time operating systems (RTOS), *e.g.*, management services for tasks, mutexes and alarms. Currently, a beta version is also available for Xenomai 3.0.6.

All data are static, meaning no memory allocation occurs during execution. This design choice was made to reduce execution time and to keep the opportunity to port the runtime on RTOS for microcontroller, such as FreeRTOS or RTOS from the AUTOSAR family.

All tasks and events have their own thread which is suspended until the TTS engine resumes them. The data used as parameter for task are read at `start` and data produced by a task are written when the `sync` call is performed. Hence, a copy of the data is done and the execution is only on a local data. This mechanism guarantees the reentrancy of the task execution. Note that it does not assure reentrancy if the functions executed by the thread are not reentrant. This execution model is similar to the Acquisition Execution Restitution (AER) proposed in Durrieu et al. (2014) where execution is decoupled from data access.

Note that the data that are only used at the functional level, *i.e.* data that are not used in the behavior model, such as the output value of a control loop computed by an operation, are not represented in the H-FIACRE model. This means that it is up to the programmer to plan the data exchanges at the level of the operations. In particular, the designer has to ensure that functions are thread-safe and that the concurrent access to shared data is correct and consistent.

The TTS engine has its own thread, with the highest priority. A periodic alarm updates logical time of the engine and calls the TTS engine if a task terminates, an event arrives, or a waiting delay expires. During a run of the engine, the TTS is updated until a blocking state (due to a delay or the waiting of an event) is reached. The state of the TTS is updated using the action functions (for the data part) and the firing rules of the associated Time Petri Net (for the discrete part).

The source code of HIPPO and all scripts for tests and experiments are available on GitLab at https://gitlab.laas.fr/pehladik/hippo.

4.4. Scheduling model

The scheduling of the tasks is not represented in a H-FIACRE model and is delegated to the engine. The current implementation uses the SCHED_FIFO scheduler of Linux and all threads that managed tasks have the same priority (threads for events have a highest priority). The SCHED_FIFO scheduler is equivalent to a multiprocessor, global, fixed-priorities scheduling algorithm with a FIFO rule for tasks with the same priority.

Formally verifying the scheduling behavior of the system could be tedious, because of the difficulty of effectively modeling preemptive systems using a realtime model-checker. If the scheduler is non-preemptive, it is relatively simple to explicitly model the scheduler in FIACRE. In Appendix we give an example of model for the FIFO scheduler used in the actual implementation.

In the case of a preemptive scheduler, different approaches are possible. A first one is to develop a model-specific scheduling analysis to compute the best and worst-case response time and to use them, as mentioned in Section 3.3.1, in the FIACRE model

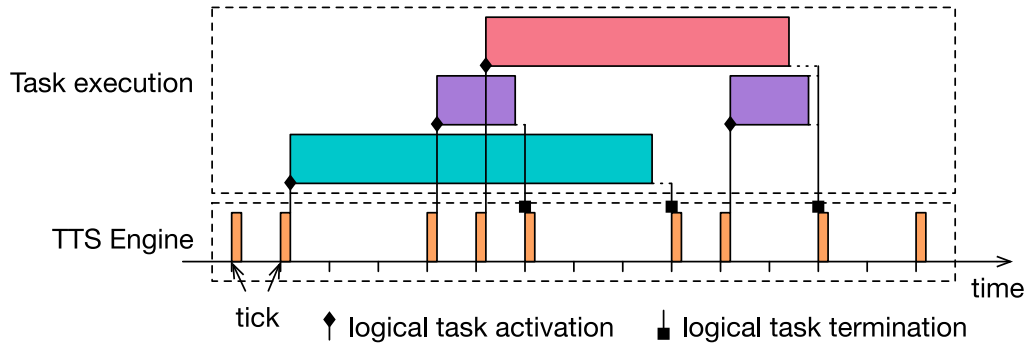


Fig. 3. A schematic representation of the execution control flow of HIPPO.

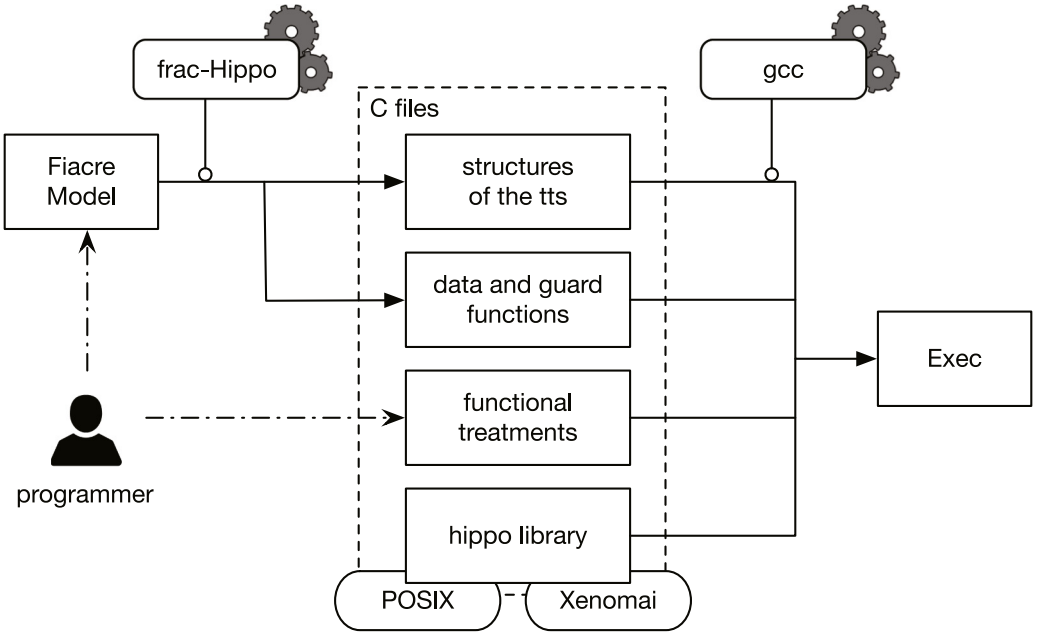


Fig. 4. Tools for code generation and compilation.

to proceed to the verification. By definition, this response time takes into account the scheduling aftermaths. The difficulty with this method is to have a tight method to compute response times. By default, it is possible to estimate an upper bound of the response time using response time analysis methods assuming that all tasks are independent. This default calculation is particularly pessimistic and it is preferable to use scheduling analysis with more advanced task models such as transactions, DAG, etc.

Another approach to consider for scheduling is to combine activation patterns with scheduling analysis. This method is used to restrict the behavior of tasks to models for which scheduling analysis methods exist. For example, the Ravenscar profile (Burns, 1999) defined this kind of patterns. The Logical Execution Time (LET) assumption introduced with GIOTTO (Henzinger et al., 2003) is also based on the same idea. To illustrate our point, we will examine the LET approach in more detail.

Using LET, the system designer specifies the logical execution time of each task, that is, the duration between the instant at which the task is activated and the instant at which the task provides its outputs. When the LET expires, the outputs are made visible for other tasks. Fig. 5 shows examples of LET for three tasks. Each task has its own LET and can be executed at anytime during this interval. The schedulability analysis for this model can be done by a simple utilization test (Henzinger et al., 2002; Hladik, 2018).

To model any system with the LET assumption in H-FIACRE, the designer needs to follow a pattern, especially to call tasks. Listing 6 shows the pattern to use `start` and `sync` under LET. The input data and activation of the task `t` is triggered at line 3. The LET assumption is modeled by the value of the `wait` statement in line 5 (this value is the duration of the LET delay). The result of the operation is read in line 8 and its value captured in `res`. In FIACRE, a value is only updated during a transition, so that the value of `res` is made visible, *i.e.*, the outputs are provided, only when the transition `to next` (line 8) is fired. So, by adding this pattern for each task activation and termination, a LET scheduling analysis can be done.

```
1 process pExample is
2     ...
3       start t (argOp);/* activate the task */
      to wait_deadline
4     from wait_deadline
5       wait[$LET, $LET]; /* LET delay */
6       to read_result
7     from read_result
8       sync t res; /* synchronize the end of the task
        with the LET delay */      to next    ...
```

Listing 6: Pattern to model a task `t(a1:ty1):rty` with a Logical Execution Time.

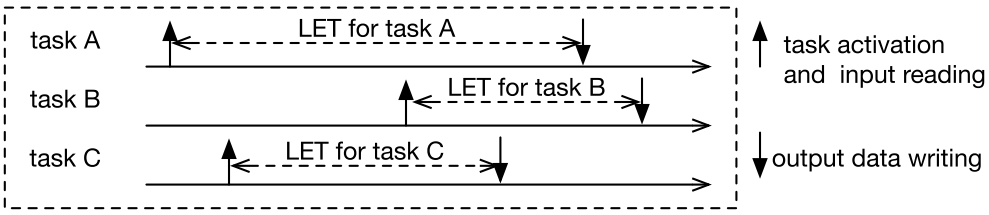


Fig. 5. Timing diagram for three tasks under the LET assumption.

Note that the hypothesis of a multiprocessor architecture (or not) has no impact on the approach. From an execution point of view, this is completely transparent because the scheduling is entirely delegated to the operating system. However, as we will see in the performance evaluation part (see Section 5), it would be possible to optimize the execution of the engine by dedicating a core to it. As far as verification is concerned, either the approach abstracts the architecture (by delegating the scheduling analysis to an estimation of the response times), which also makes the multiprocessor assumption irrelevant, or it must be considered in the scheduler model (which is done for example in Appendix) at the risk of increasing the state space explosion.

4.5. Implementation validation

The translation from H-Fiacre to the C code executable has not been formally proven or certified yet. However, to increase our confidence in our implementation, we paid particular attention to the quality and readability of the engine code. More importantly, we wrote automatic tests to compare traces obtained from actual Hippo executions with the traces of the formal model. To do this, we use the tool `play` from the Tina toolbox which is a stepper simulator that allows to simulate a tts model. In this context, a *simulation* is a series of states separated by delay transitions or discrete transitions. A textual format `scn` exists to record the transitions of a simulation. Our tests consist in checking that a trace in `scn` format generated by Hippo can also be observed in the Fiacre model simulated with `play`. So, as shown in Fig. 6, we can generate a trace in a `scn` format with Hippo and play this trace with the tts model generated from the same H-Fiacre model. A test is valid when the trace can be simulated by the tts, otherwise the analyzer returns the name of the first conflicting transition.

To test our implementation, we use multiple models generated from our use cases or selected from our catalogue of Fiacre and Tina examples. Twelve tests (of 30 to 200 lines models) were systematically applied to each code evolution, more than 100 generated models were tested and four complete use cases were released. The current code is 100% reliable on the basis of tests we have run. All tests are available on the Gitlab repository.

4.6. Semantic distance between the execution and verification models

Our verification strategy relies on an interpretation of H-Fiacre into Fiacre, see Fig. 2. The value of the properties that we check on the formal model also depend on the *faithfulness* of our execution engine; meaning that execution traces observed in the execution should have an equivalent in the formal model. (This is the purpose of the property-based testing approach described in the previous section.) As a consequence, we over-approximate the set of possible executions in the system. The same applies for the set of reachable states.

In this work—and more precisely in our case studies—we propose an approach where we synthesize both the *execution* and *verification models* (the implementation and the specification) from the same high-level description of the system. This solution fits nicely with our pragmatic approach, since it is easier in this case to tailor the model generation framework. For instance, one can imagine generating more constrained models for the environment, by restricting some classes of failures, or by abstracting whole parts of a system. In this case, we use the same interpretation of H-Fiacre constructs described in Section 3.3, and therefore still deal with an over-approximation of the behavior.

Our choice to merely over-approximate the behavior has a direct impact on the usefulness of verification. In practice, it means that we can only utilize verification to check safety properties: that the system cannot reach an unsafe state, like with a mutual-exclusion property for example. Likewise, with timed properties, we can only rely on the computation of upper-bounds for worst-case execution (or traversal) times. These are the kind of properties that we check in our use case (see Section 6).

The distance between the specification and implementation models is actually not that big in practice. One main difference is the fact that our execution engine has a deterministic firing policy: "transitions" are totally ordered and we always fire the "smallest transition" first. As a result, a deadlock in the implementation model should always be witnessed in the specification. On the opposite, the specification model is non-deterministic. This discrepancy could be solved by adding priorities, but this could have an adverse effect on the state explosion problem. A more subtle difference lies in the interpretation of H-Fiacre tasks (see Listing 3) and the fact that our engine will always react "as soon as possible" (see the discussion in Section 3.2.3). A possible choice to reduce the semantic gap in this case could be to fix the duration of each task (a task execution time will always be equal to its worst-case), but this may be too restrictive.

Another legitimate question is the level of trust that can be placed on our toolchain. At present, it is not possible to state that Hippo provides an absolute guarantee of correctness, meaning that the specification and the implementation models have equivalent observational semantics. To prove such result with a very high degree of confidence would require defining the operational semantics of our execution engine very precisely; with the same level of details than when proving the correctness of a compiler. We would also need to rewrite each language transformation using a formal framework and to prove the correctness of each transformation. This could be the subject of future works, but it is not part of our priorities yet. Indeed, these tasks are very time-consuming and their results are often "brittle", meaning that proofs can break and are not easy to adapt when we decide to change some aspects of the semantics or the encodings.

A mitigating solution would be to extend the tts model with support for tasks and events (resulting in a H-tts) and to prove a full abstraction result between tts and H-tts, which should be easier. We could also adapt our model-checker to directly accept H-tts models, internalizing the behavior of the execution engine into the semantics of the model. In practice, our approach is closer in spirit to that of tools qualification. As such, it is still useful to catch specification problems early in the design of a system.

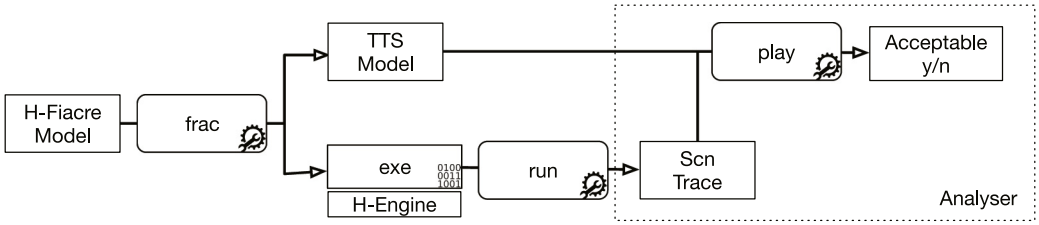


Fig. 6. Scheme of the acceptance test for traces.

5. Empirical analysis of the HIPPO engine

This section presents the performance of HIPPO for different size models. We have provided two kinds of measurements, which are the CPU usage and the time spent in the HIPPO engine. The experiments were performed on a computer with an Intel i5-8265U (1.6 GHz) processor, with eight cores and 8GB of RAM memory. The system runs the Ubuntu 18.04 distribution, using the PREEMPT_RT patched kernel 5.4.47-rt28. In addition, the highest priority is given to the HIPPO engine process, using the POSIX methods of `sched.h`. The measurements were carried out using LTTng, an open source tracing framework for Linux (The LTTng Project, 2020).

5.1. Benchmark

We propose a set of synthetic benchmarks used in our experiments (an example of a more realistic application is given in Section 6.) We should use models of the form $P_{m,n}$ built as the composition of m periodic processes. Each process calls its own and private task n times in a period. In our case, the tasks are C functions that compute some unoptimized and arithmetical operations. Listing 7 shows an example of a periodic process, p0, with two task calls ($n = 2$). After being activated (line 12 by its periodic clock (lines 3 to 9), process p0 calls its task (t0) a first time (line 14), then waits for the return value (line 15). The process returns to its idle state after calling t0 a second time. Note that the number of places and transitions in the TTS of a benchmark $P_{m,n}$ is proportional to m and n.

```
1  task t0 (a1 : ty1) : rty is  c_t0
2  ...
3  process periodic_clock_0 [S_1 : none] is     states
       a, b, o
4      from o
5          wait [0,0];  to a /* offset */
6      from a
7          wait [3,3]; to b /* period */
8      from b
9          S_1; to a process p0 [S_1 : none] is
10     states a, b, c, d, e
11     var ret : nat := 0
12     from a          S_1; to b
13     from b
14         start t0 (1); to c /* start task t1*/
       from c
15         sync t0 ret; to d /* wait end of task t1 */
       from d
16         start t0 (1); to e
17     from e
18         sync t0 ret; to a
```

Listing 7: Example of a p0 periodic process with two task calls

5.2. CPU usage

This section reports on the CPU usage of the HIPPO engine. The CPU usage is defined as the rate of CPU time spent in the HIPPO engine by the total time of an execution. For a HIPPO execution with a tick frequency F_{tick} and a given function $R[i]$ that returns the response time of the TTS engine, *i.e.* the part of the HIPPO engine that executes the TTS, activated at the ith tick ($i \in 1..n$), we define the CPU usage as :

$$U = \frac{F_{tick}}{n} \sum_{i=0}^{n} R[i]$$

For the experiment we have run HIPPO models with 2 tasks per process, as described previously. HIPPO engine's frequency is set to 1 kHz, so every 1 ms HIPPO updates the running model and goes on.

For an example that corresponds to a realistic application with 20 processes (equivalent to 140 transitions in the TTS model), we measure a CPU charge (on one processor) of 13.2%.

More extensive experiments are shown in Fig. 7(a). We can see, that the CPU usage increases linearly with the number of tasks (number of transitions). This is easily explained by the fact that the engine has to handle a larger number of transitions at each execution. A second experiment was conducted by significantly increasing the number of processes (see Fig. 7(b)). For this experiment, the Turbo Boost of the hardware architecture was used, bringing the processor frequency to 3.9 GHz. We observe the same behavior as before, but with greater variability.

These experiments show that we can envisage running HIPPO with hundreds of parallel tasks on a modern embedded architecture. The main limitation is the processing speed. On the other hand, we can predict the expected performance based on the size (in number of transitions) of the TTS and the charge of the tasks.

5.3. Time overheads of the TTS engine

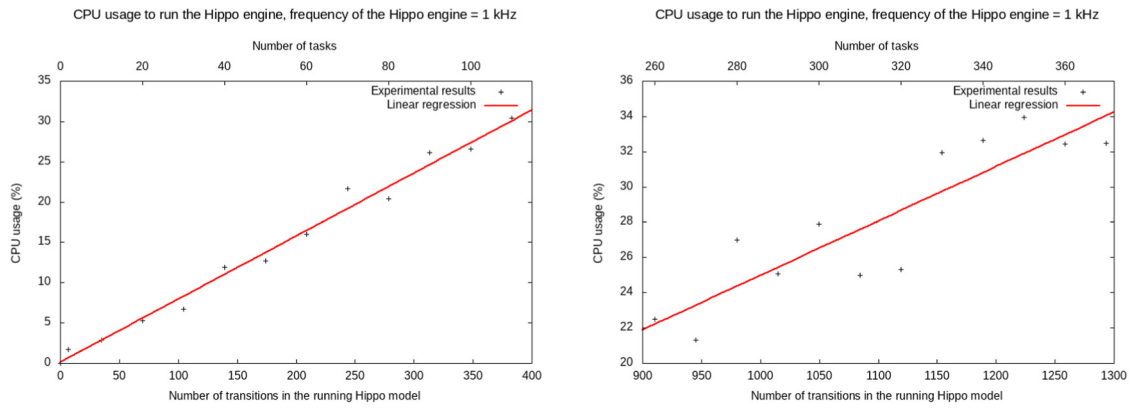
For our next experiment, we look at the "time overhead" of HIPPO by studying the time spent in one turn of the engine. Like in the previous experiments, we set the frequency to 1 kHz; meaning that the global tick (one turn of the engine) is at 1 millisecond. So, to successfully run the model, HIPPO must complete all its operations in less than 1 ms.

For the simple example with 20 processes (equivalent to 140 transitions in the TTS model), we measure an average time spent in one turn of HIPPO engine of 0.13 ms with a best-time of 5.5 µs (the engine has no transition to fire) and a worst-time of 0.26 ms.

Fig. 8 shows the distribution of the time spent in one turn of the HIPPO engine for three models: a small model with 40 HIPPO tasks (20 processes, 2 tasks per process, 140 transitions); an intermediate model with 400 HIPPO tasks (100 processes, 4 tasks per process, 1100 transitions); and another one with 800 HIPPO tasks (200 processes, 4 tasks per process, 2200 transitions). None of the experiments use the Turbo Boost.

For the 400-tasks model, the median equals to 0.32 ms and the interquartile range equals to 0.06 ms, 95% of the values are lower than 0.4 ms and the worst-case is 0.98 ms. So, for this execution there is no *tick miss*, *i.e.* all the HIPPO engine turns were executed in less than 1 ms.

For the 800-tasks model, the median equals to 0.7 ms and 95% of the values are lower than 0.97 ms. However, 3% of the



(a) Without TurboBoost (b) With TurboBoost

Fig. 7. CPU usage as a function of number of Hippo tasks.

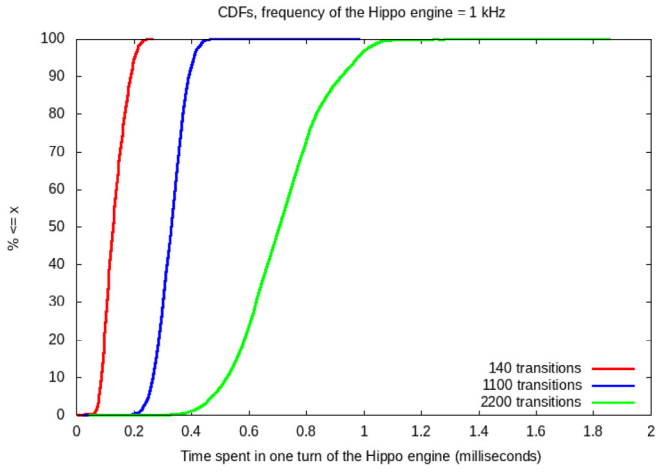


Fig. 8. Time spent in the Hippo engine.

values are greater than 1 ms, with a worst-case of 1.85 ms (note that we do not use Turbo Boost here). Indeed, the number of operations performed by Hippo depends on the running model. For a given model, a big number of operations to perform during a tick can involve an overrun of the tts engine. This limitation was expected and the choice of hardware must be made with full knowledge of the facts. The system with 800-tasks is an extreme case, not representative of a real application (see the use case in Section 6 for a realistic example). In addition, the operating modes of the systems naturally exclude a behavior depending on the operating phases, which normally limits the number of transitions to be evaluated at each engine tick (which is not the case for the models in our benchmark).

An interesting feature of our architecture is that we can increase the predictability of Hippo by dedicating a processor solely to the engine. This option can easily be added in the implementation of Hippo by assigning an affinity to the thread.

6. Case study: a software controller for the mobile robot minnie

Fiacre is first and foremost an intermediary language, defined with the goal to ease the interoperability between formal verification tools and high-level (component-based) specification languages.

The robotic group at LAAS has used for years such a specification language, called G^{en}oM, to program and deploy components for their robot functional architecture. In this section, we show how we use G^{en}oM to generate both a H-Fiacre (execution) model and a Fiacre (verification) model. The former is used with Hippo to produce the runtime controller software of a real robot called Minnie to perform runtime monitoring of critical assertions and take appropriate corrective actions. The latter is used with model checking tools to verify several interesting properties on the system. In the following, we should simply use the terms Hippo/Tina models to refer to them, or sometimes use the qualifiers online/offline models.

G^{en}oM (GENerator Of Modules) (Mallet et al., 2010) is a tool to specify and implement robotic *functional components* also called *modules* (see the nine modules on Fig. 9). These modules provide *services* in charge of functionalities that may range from simple low-level driver control (e.g. the velodyne or imu modules to respectively control a Velodyne HLV32 or an XSens IMU) to more integrated computations (e.g. pom for localization with an Unscented Kalman Filter, or PotentialField for navigation). G^{en}oM proposes a language to completely specify the functional components down to (but not including) the C/C++ functions (also called *codels*) that implement the services computation steps.

6.1. G^{en}oM and the Minnie RMP440 Robot

To illustrate how G^{en}oM is used, we present a complete navigation experiment for Minnie, an RMP440 LE robot (see Fig. 9). Minnie is not an autonomous car, nevertheless, it shares a lot of common sensors and effectors with one: an XSens MTi IMU, a KVH DSP-5000 fiber optic gyro, a Novatel GPS, all connected through serial/USB lines and a HDL-32E Velodyne lidar (on an ethernet UDP interface). The goal of this section is not to discuss the overall localization/navigation implemented on Minnie, but to give a reasonable idea of the overall complexity entailed by a non-trivial robotic experiment.[1]

The RMP440 platform comes with a low-level controller (accessed through an ethernet interface), which allows controlling the robot with a speed (x-linear and z-angular) command, and returns the platform wheel odometry. The platform also includes a Nuvis 5306RT i7-6700 CPU with 16 GiB RAM and a 256 GiB SSD drive, running Ubuntu 18.04. In case of emergency, a human operator can take control of the robot using a wireless joystick

[1] The complete code of the Minnie experiment is available at https://redmine.laas.fr/projects/minnie.

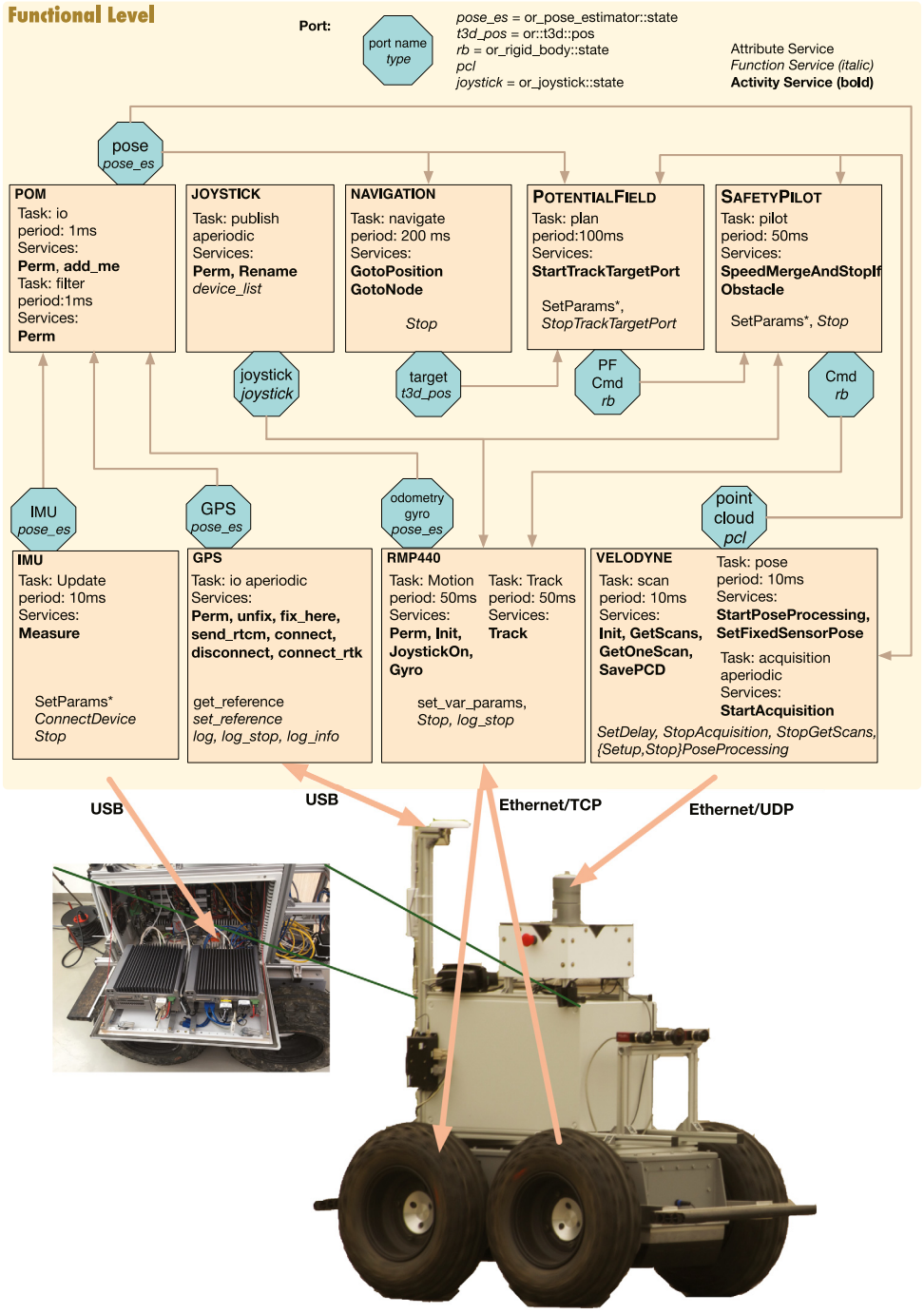


Fig. 9. Architecture of the Minnie RMP440 experiment.

communicating with the robot via a USB dongle. Commands emanating from the joystick should take precedence over commands from the robot controller.

All the hardware components of Minnie are controlled through their respective G^{en}oM modules[2] (depicted with boxes, like GPS) which produce shared data in ports (depicted with octagons). Links in the diagram describe which modules read from which ports. Fig. 9 lists, inside each module, the *execution tasks* they include, their *activity services*, the *ports*' name and the data type they hold. We can understand the basic behavior of the robot by looking at the tasks and services implemented in each of these modules, and the exchange of information between them.

Module POM uses an Unscented Kalman Filter (UKF) to merge pose estimations from GPS, IMU and RMP440 (gyro and odometry) and to provide the position of the platform in the **pose** port. Module NAVIGATION offers services to navigate in a graph of positions in a topological map of the environment and produces in a port, the next **target** to navigate to. This port is used as the goal to reach by POTENTIALFIELD which produces a speed reference in port **PF Cmd**, while avoiding obstacles found in the **point cloud** port using a Potential Field method inspired from Guerra et al. (2016) (the points in the cloud are collected in an *occupancy grid* which is then used to provide obstacles position in the local map).

[2] The gyro is managed inside the RMP440 module.

The speed reference is then read by SafetyPilot which, as last resort, checks in **point cloud** that no obstacles is too close to the robot, and stops the robot if needed. It also considers the data in port **joystick** and uses it as a speed command producer if the proper joystick buttons are pushed (which is a way to gain control back on the robot platform in case something goes astray while navigating). The final speed produced, written in **Cmd**, is then read by rmp440 (if it is executing the *Track* service), which pushes it to the low-level controller of the robot. Last, rmp440 also has a *JoystickOn* service (incompatible with *Track*) which computes a speed command and send it to the wheels controller.

6.2. GenoM *Specification*

All nine modules in Fig. 9 are an instance of a generic GenoM component presented in Fig. 10. Hence a module is a unit composed of a *control task*, a set of *execution tasks*, and a set of *services*. Concerning the use of data, each module also includes an *Internal Data Structure* (IDS) and may expose/read a set of *ports*:

Control Task: A component always has an implicit cyclic *control task* that manages the control flow by processing *requests* and sending *reports* (from/to external clients); it also executes *control services*, and activates/interrupts *activity services* in *execution tasks*.

Execution Task(s): Aside from the *control task*, whose reactivity must remain high, one may need one or more cyclic *execution tasks*, aperiodic or periodic, in charge of longer computations needed by *activity services* (e.g. velodyne has three execution tasks: `scan` and `pose` running at 100 Hz, and `acquisition` aperiodic).

Services: The core algorithms needed by the component are encapsulated within *services*. *Services* are associated to *requests* (with the same name). The algorithm executed by these services may require a *short* computation time or a *long* one. *Short* services are known as *control services* and are directly executed by the *control task*. *Control services* are in charge of quick computations and may be *attributes* (setters/getters of the *IDS fields*) or *functions* (in *italic* on Fig. 9). *Longer* services are known as *activities* (in **bold** in Fig. 9) and they are executed by *execution tasks* (e.g. velodyne `scan` task has three *activities services*, *Init*, *GetScans* and *GetOneScans*).

Activity Automaton and Codels: *Activities* are long-running services. They are modeled with an automaton that breaks down the computation into different *states* (see an example in the lower right part of Fig. 10). Each state is associated with a *codel*, which specifies a C or C++ function (top right part of Fig. 10). The execution of that *codel* leads to (yields) the next state in the automaton, to execute immediately, or in the next period if this next state name is prefixed with `pause` (see for instance the declaration in Listing 8, line 26).

IDS: A local *internal data structure* is provided for all the *services* to share parameters, computed values or state variables of the component. A codel which needs to access (*in* or *out*) fields from the IDS must specify them in its argument list. GenoM will ensure proper mutual-exclusion when accessing these fields during computation.

Ports: They specify the shared data, *in* and *out*, the component needs or produces from/for other components (octagons on Fig. 9). Access to ports is also specified in the *codels* arguments list and is properly handled/locked with respect to the middleware.

The GenoM language fully specifies the shared **ports** (the green octagons in Fig. 9) between components (in and out), as well as the shared variables in a component, and the periodic tasks (i.e. threads) in which the *services* run. For each *service*, one defines the arguments (in and out), and the automata specifying the steps to follow to execute the `codels`, as well as their arguments.

To further illustrate the GenoM specification of the Minnie robot, Listing 8 presents the *GetScans* activity service of the velodyne module. Note the automata specification, which is also presented in Fig. 11.

```
1  activity GetScans(
2      in double firstAngle = :"First angle of the scan (degrees)",
3      in double lastAngle  = :"Last angle of the scan (degrees)",
4      in double period     = :"Time in between two scans",
5      in double timeout    = :"Timeout used when stamping packets"
6     )
7  {
8   doc  "Acquire periodically velodyne sensor full scans";
9   task scan;
10
11  validate GetScansValidate(
12        in firstAngle,
13        in lastAngle,
14        in period);
15
16  codel <start> GetScansStart(in acquisition_params)
17          yield copy_packets;
18  codel <copy_packets> GetOneScanCopyPackets(
19        in acquisition_params,
20        inout scan_buffer) // get packets from acquisition buffer
21          yield stamp_packets;
22  codel <stamp_packets> GetOneScanStampPackets(
23        in acquisition_params, // stamp packets
24        inout pose_data,
25        in timeout) // with the proper  pose
26          yield pause::stamp_packets, build_scan; // pause:: if
        pose not available
27  codel <build_scan> GetOneScanBuildScan(
28        in acquisition_params,
29        in firstAngle,
30        in lastAngle) // build scan repositioning
31          yield end; // individual packet in the first pose
32  codel <end> GetOneScanEnd(
33        in acquisition_params,
34        port out point_cloud,
35        inout usec_delay) //publish the scan in point_cloud port
36          yield wait;  //  usec_delay is for fault injection
37  codel <wait> GetScansWait(in period) // wait next user defined
        scan period
38          yield pause::wait, copy_packets; // then loop back
39
40  interrupts GetOneScan, SavePCD, GetScans;
41 };
```

Listing 8: The GenoM specification of the *GetScans* activity (executing in the `scan` task of the velodyne module). See the resulting automata Fig. 11.

Overall, the Minnie experiment includes: 9 modules, 9 ports, 24 tasks, 38 activity services (with automata), 41 function services, 43 attribute services, 170 codels over 14k loc (lines of codes) and their respective WCET. The synthesized GenoM modules amount to 200k loc to which one must add external libraries (middleware, PCL, Euler, etc.).

From a specification point of view, GenoM has a clear semantics of what should be done and how it should be properly implemented. This generic component implementation is thus instantiated for each specific component specification using a template mechanism.

6.3. GenoM *Templates*

GenoM alone, just parses and builds an internal representation of `.gen` specification files. To produce output, GenoM has to be called along with a *template*. The templates are the real building blocks used to synthesize source code files adapted to the current

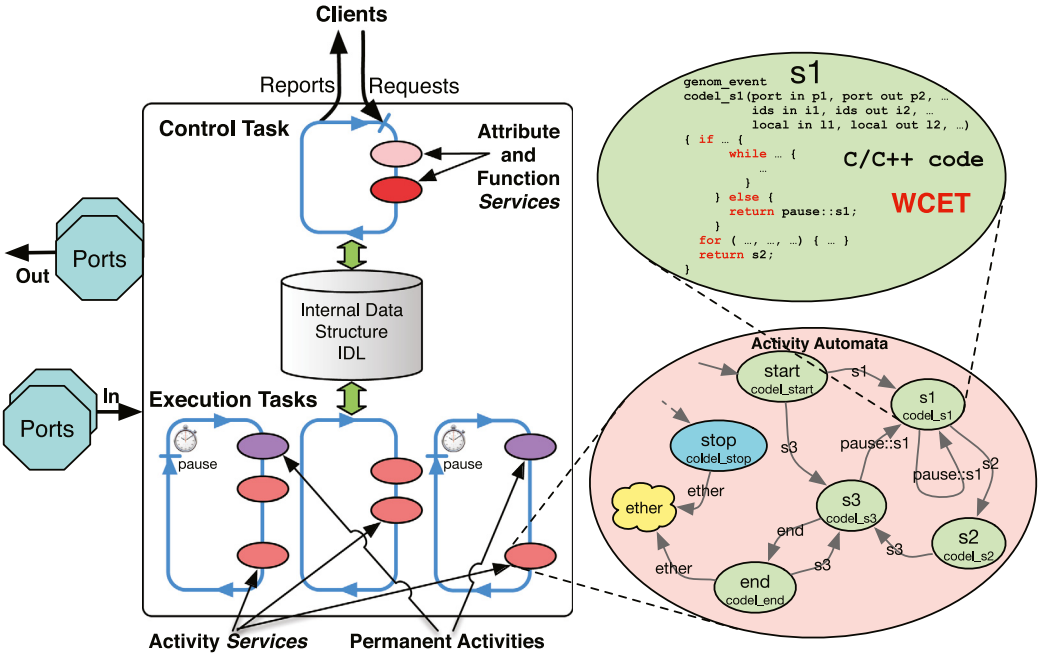


Fig. 10. A G^{en}oM generic functional component (module).

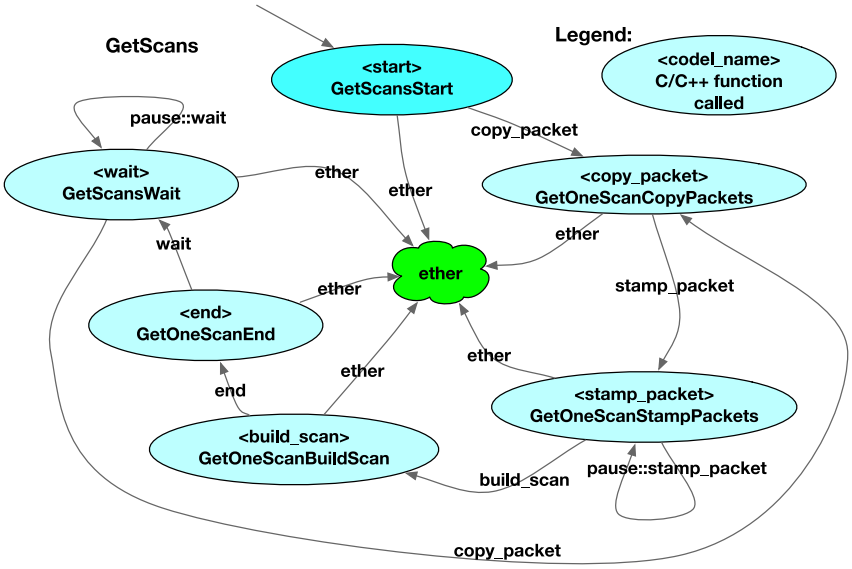


Fig. 11. Finite-state machine of the *GetScans* activity (Listing 8). Each state is labeled with its name between <> and its codel (the C/C++ function called). Transitions are labeled with the name of the next state (i.e. the returned value from the originating state codel).

specification. These codes can be the sources implementing the component itself, or client libraries to interact with the component, or, as we will see in Section 6.4, formal model of the component implementation.

A G^{en}oM template is a set of text files that include Tcl code (Ousterhout and Jones, 2008), whose evaluation in the context of a G^{en}oM call on a specification file will produce the target of this particular template. The target can be as simple as one file with the list of the names of the services specified in the module (in which case the template file will just include a Tcl loop over all services and print their name), or it can be the C code which controls the execution of an activity automaton, or which implements the module itself using the ROS-Com middleware.

The template mechanism was initially introduced to deal with the *middleware independency* problem (Mallet et al., 2010). Indeed, the specifications presented above do not subsume any specific middleware. Different templates are provided to automatically synthesize the components for different middleware which are then linked to the codels library for the considered module (see the workflow on Fig. 12).

A template, when called by G^{en}oM on a given module specification, has access to all the information contained in the specification file such as service names and types, ports and IDS fields needed by each codel, execution tasks periods, activities automata, etc. Through the template interpreter (using Tcl syntax), one specifies what they need the template to synthesize. For instance, Listing 9 shows an excerpt of a template code and Listing 10 the C code it produces when called together with the NAVIGATION component specification file. The interpreter evaluates anything enclosed in *markers* <’ ’> without output, while on the code between <’”’>, variables and commands substitution is performed and the result is output in the destination

file, together with the text outside of the markers. For example, |<'foreach s [$component services] '> ... <"[$s name]"> ... <">| iterates over the list of services of the component, contained in the |$component| variable; while |<"[$s name]">| is replaced by the name of the service contained in the |$s| variable bound by the foreach statement.

```
void genom_<"\$comp">_activity_report(
      struct genom_component_data *self,
      struct genom_activity *a)
{
 switch(a->sid) {
   case -1:
      return; /* permanent activity reports nothing */
<'foreach s [\$component services] {'>
   case <"\$comp">_<"[\$s name]">_RqstId:
      genom_<"\$comp">_<"[\$s name]">_activity_report(
         self,
         (struct genom_<"\$comp">_<"[\$s name]">_activity *)a);
      return;
<'}'>
 }
```

Listing 9: A simple template code snippet. Note the mix of Tcl code (between <' '> and <" ">) and the targeted C code.

```
void genom_Navigation_activity_report(
      struct genom_component_data *self,
      struct genom_activity *a)
{
 switch(a->sid) {
   case -1:
      return; /* permanent activity reports nothing */
   case Navigation_connect_port_RqstId:
      genom_Navigation_connect_port_activity_report(
         self,
         (struct genom_Navigation_connect_port_activity *)a);
      return;
   ...
   case Navigation_GotoPosition_RqstId:
      genom_Navigation_GotoPosition_activity_report(
         self,
         (struct genom_Navigation_GotoPosition_activity *)a);
      return;
   case Navigation_GotoNode_RqstId:
      genom_Navigation_GotoNode_activity_report(
         self,
         (struct genom_Navigation_GotoNode_activity *)a);
      return;
 }
```

Listing 10: Excerpt of the synthesized C code for the PocoLibs NAVIGATION component corresponding to the template in Listing 9. Note how the C code is synthesized for all the services of the component.

As shown on Fig. 12, there are already templates to synthesize: the component implementation for two middleware: PocoLibs.[3] and ROS-Com (Quigley et al., 2009) There are pros and cons to use one or the other, for example for port communication mechanism, PocoLibs uses shared memory while ROS implements it with publish/subscribe over sockets. Note that the only source code provided by the programmer is the `.gen` component specification, and the `.c/.c++` component codels. Everything else is automatically synthesized by the templates. There exist other templates to produce client libraries to control the component (*e.g.* JSON, C, OpenPRS), stubs for the initial codel definition, etc.

6.4. The GenoM toolchain for verification and code generation

The FIACRE template presented and deployed in this paper is not the first implementation of a transformation from GenoM to FIACRE. A first experiment was performed in Foughali (2018), but was mostly a *proof of concept* and remained at a too abstract level to lead to safe execution on critical systems (e.g. UAVs). Even on less critical robots, such as Minnie, the interleaving of service automata execution was not properly handled and led to suboptimal reaction time. Based on this first experience, new templates have been implemented such that we can now derive better bounds on the reaction time of the system.

As Fig. 13 shows, the template mechanism used to synthesize the GenoM modules from their specifications and codels can also synthesize both the FIACRE verification model (which can then be used with the TINA toolbox) and the H-FIACRE runtime model (which can be compiled and linked to the codel library to produce an executable module). These models contain FIACRE processes implementing the algorithms of all the internal components of a GenoM module as presented Fig. 10. These FIACRE processes, similar to the ones on Listings 1 and 2, model each and every algorithm of the *Control Task*, the *Execution Tasks* and their respective *Timer*, the handling of the *Services* within the Control Task, and the execution of the *Activity Services* with their codel *Automata*, etc. A careful examination of these FIACRE processes show that they indeed replicate the algorithms of the Pocolibs or ROS-Com version of the module. All these FIACRE processes are then composed in parallel, sharing FIACRE ports and shared variables (Section 3.1), to produce one large FIACRE component modeling one or more GenoM modules. As for the *codels*, they are encapsulated using H-FIACRE *task* and called using *start/sync* (Section 3.2.1). Communication with the PocoLibs Mbox (or the ROS CallBackQueue) is modeled with *event* ports (Section 3.2.2).

Note that our workflow slightly differs from the more generic one presented in Section 2.4 and illustrated on Fig. 2 (where the FIACRE model is obtained from the H-FIACRE one). This choice was made to simplify the implementation because the FIACRE GenoM template file is in fact the same (the two models share 95% of their code). It is when we call the GenoM command on this template, that a flag (`-tina` or `-hippo`) is used to synthesize one model or the other.[4] Moreover, the GenoM versatile template mechanism allows us a more fine grained control on the produced model with for example varying level of abstraction for the FIACRE model.

The time constraints used in both models come from temporal information found in the modules (for instance the period of tasks) and from the Worst-Case Execution Time (WCET) of the codels. At the moment, the WCET are obtained by running the regular modules with GenoM embedded profiling tool: `profundis`.

The difference between the synthesized verification model in FIACRE and the HIPPO executable model in H-FIACRE are minimal:

- Codels execution is really carried out in the HIPPO models (with `start/sync`), but is modeled as a time delay in the interval $[0, wcet]$ (or $[wcet, wcet]$) in the TINA model.
- Non deterministic choices (e.g. codels returned values, used for activities automata transition, or control codels success/exception) are handled with H-FIACRE tasks `start/sync` and tests/case on the codel returned values in the H-FIACRE models, but with a simple nondeterministic choice operation (a `select`) between all possible returned values in the TINA model.
- The TINA model must include a *client* FIACRE process to model the behavior of the environment (i.e. the requests sent to the controller). On the opposite, the HIPPO model is simply "linked with the real world" using H-FIACRE *event ports* and task executions. In this case, the H-FIACRE event port handles the mechanism which receives new requests (PocoLibs Mbox or the ROS CallBackQueue).

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

[4] The FIACRE GenoM template is available here: https://redmine.laas.fr/projects/genom3-fiacre-template/gollum.

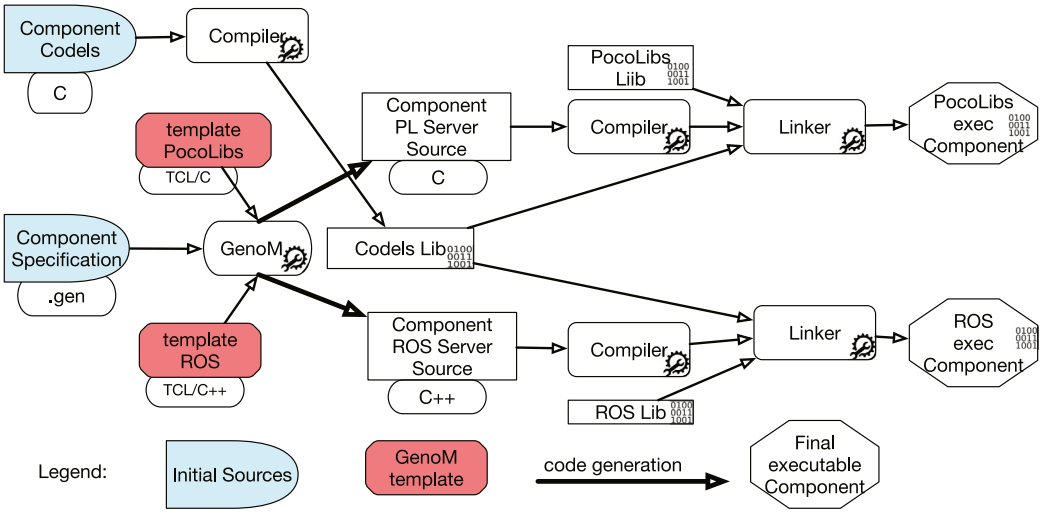


Fig. 12. Toolchain with the regular PocoLibs and ROS template.

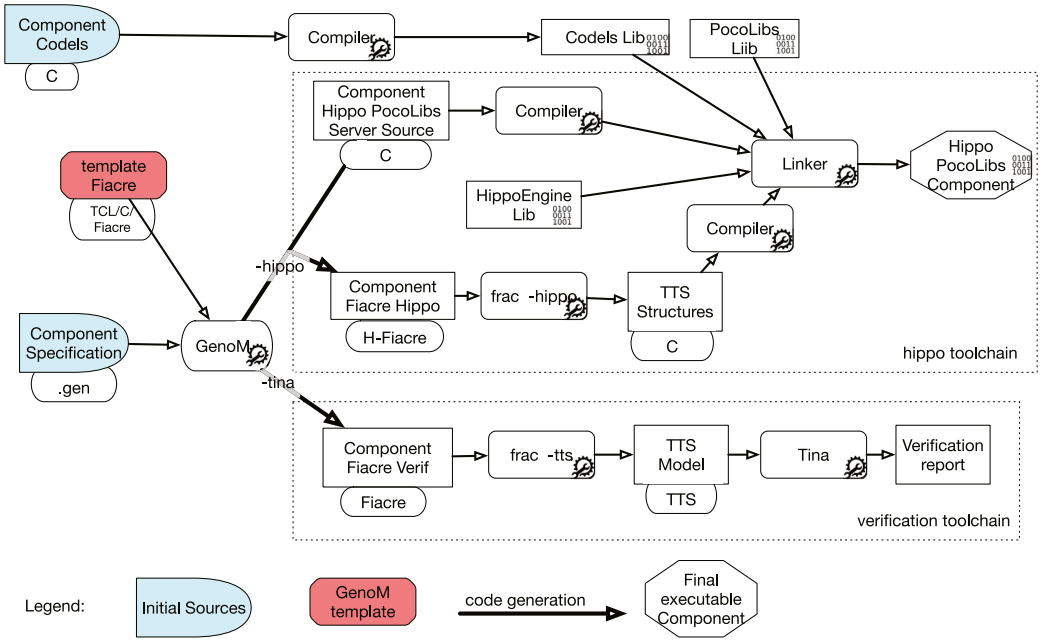


Fig. 13. Toolchain with the Fiacre template to produce the `-tina` and the `-hippo` versions (only the PocoLibs version of the Hippo branch is presented here).

Even though this is not a formal proof, the fact that the online and offline models are synthesized from the same template and only minimally differ gives a very strong argument to support that our models have the proper semantics. It increases our confidence that both models are observationally equivalent and close to the modules produced with the existing PocoLibs or ROS-Com template.

6.5. Comparison with previous experimentations with GenoM

We experimented with other V&V templates in previous works (transformations from GenoM to the input language of other V&V frameworks), namely BIP (Abdellatif et al., 2012; Foughali et al., 2020) and Uppaal/Uppaal-smc (Foughali et al., 2019). However, none of these works reached the level of fidelity achieved with our current Fiacre template. We give a high-level evaluation of our past experiences in Table 1. We compare three different target frameworks: the current Fiacre, RT BIP, and Uppaal. In each case, we score the fidelity of our results in three different categories. Offline is for models used for formal verification or simulation purposes (the equivalent of the `-tina` version in our work). Online is for generated, executable code (similar to our `-hippo` version). We consider two different cases here, that correspond to two different "robotic middleware": PocoLibs (Herrb, 1992) and ROS-Com (Quigley et al., 2009). The two formal models of the PocoLibs version and the ROS-Com are identical, except for the part which models the GenoM **port** communication mechanism. PocoLibs implements it with shared memory (with locking) while ROS-Com uses publish/subscribe over sockets.

The H-Fiacre modules together with a Hippo engine at 10 kHz have an execution trace completely equivalent to the regular PocoLibs or ROS-Com modules. Of course, this does not qualify as a formal proof of equivalence, but from a roboticist point of view, the fact that such a complex rover experiment behaves the same with Hippo than the regular modules is clearly encouraging. This confidence is increased by the fact that the very same model can be used with Tina (See Section 7).

Table 1
Existing formal framework templates for GenoM. The +, ++ and +++ correspond to our own subjective evaluation of the applicability of the approach and the fidelity of the synthesized formal model to GenoM. – indicates that the tool needs more development to converge in producing meaningful and useful results.

Formal Frameworks	Offline	Online PocoLibs	Online ROS-Com
Fiacre (Berthomieu et al., 2008a)	Tina (Dal Zilio et al., 2015) +++	Hippo (Hladik, 2020) +++	Hippo ++
RT BIP (Socci et al., 2013)	RT D-Finder (Ben Rayana et al., 2016) –	RT BIP Engine (Abdellatif et al., 2010)++	RT BIP Engine +
Uppaal (Behrmann et al., 2004) Uppaal-smc (David et al., 2015)	Uppaal ++	N/A	N/A

Still, writing these templates is tedious. It requires a very good knowledge of the GenoM specification and implementation, and of course a good knowledge of the formal frameworks used. But an interesting side effect is that writing the formal version of a synthesized implementation (e.g. the Pocolibs implementation of the module) requires to also clarify the specification and/or the implementation when they are subject to ambiguities. This is a win/win strategy, the GenoM designers/programmers are invited to clarify the semantics of the tool and, in exchange, we are able to properly and formally model it.

7. Case study: Online control and offline verification results

We report here the results obtained on the Minnie use case presented in Section 6, to which we apply the online and offline tools presented in Sections 4.2 and 4.5. We start with the results obtained online while using Hippo and the H-Fiacre model. Next, we focus on the use of offline verification tools from the Tina toolchain.

7.1. Controlling and monitoring Minnie with Hippo

We have been able to synthesize automatically a Hippo model from the 9 components in the GenoM specification of Minnie. The resulting Hippo model is 35 852 lines of H-Fiacre code, with 230 Fiacre processes, 197 Hippo tasks, 9 event ports, 441 external functions, and 1760 transitions in the TTS.[5] It is linked with the codels library and Hippo runs the whole experiment at 10 kHz in one process. The load on the CPU remains acceptable, and no noticeable slowdown is observed (5%–10% more than the sum of all regular GenoM components load).

The advantage of running Hippo instead of the regular PocoLibs or ROS-Com module is to monitor online some critical properties, a first step toward runtime verification. Here is a list of the ones checked by default and already included in the synthesized model.

Task period overshoot: Periodic execution tasks are specified to run within a given period, if for some reasons, their period is not respected, the Hippo model will report the number of cycles they have overshot. If this happens too often and or with a large number of reported cycles, there is probably something wrong in the design and the specification or the hardware need to be modified.

WCET overshoot: WCET are obtained by profiling the regular PocoLibs module on the same setup. Yet, they can sometimes be exceeded, in which case the Hippo model will report the number of ticks by which it overshot its specified value. This properties is also a runtime verification that these WCET values, also used for offline verification (e.g. schedulability), are realistic.

Possible Uninitialized Port Read: When controlling a multi modules experiment, the Hippo engine checks that no codel will use a port with the `in` direction before a codel with an `out` direction has already been called. If this is the case, most likely the value read in the port is not semantically "correct".[6]

We can also define additional monitors that go beyond these default properties. We give an example from the Minnie experiment in Listing 11. In this example, we monitor the time spent between two updates to the **point cloud** port of the velodyne. If the port is not refreshed for more than 200 ms (2000 cycles at 10 kHz), the monitor triggers an emergency stop of the robot. This is achieved by forcing a transition to the *stop* state of the *Track* activity in the rmp440 module.

```
process Velodyne_Scans_rmp440_Track_Stopper(
  &scan_updated:bool, // Shared with the GetScans service.
  &TrackTask_activities: Activities_rmp440_TrackTask_Array,
  Track_index: act_inst_rmp440_TrackTask_index_type) is

   states monitor_start, monitor_wait, monitor_error

   from monitor_start
      on (scan_updated); // monitor_start scan_updated
      scan_updated := false;
      to monitor_wait

   from monitor_wait
      select
         wait [2000,2000]; // 200ms at 10 kHz = 2000 tick
         to monitor_error  // monitor_wait 200ms elapsed
      []
         on (scan_updated); //scan_updated before 200ms
         scan_updated := false;
         to monitor_wait
      end

   from monitor_error
      if (TrackTask_activities[Track_index].status =
     ACT_RUN_FCR) then // Track running?
         // Emergency stop
         TrackTask_activities[Track_index].stop := true
      end;
      to monitor_start
```

Listing 11: Example of user-defined monitor for module velodyne.

An emergency stop is a safety-critical action. Therefore we would like to compute a bound (a worst-case response time) on the time that could elapse between sending a request to stop, and the actual start of this action. By looking at the specification of the rmp440 module, we find that stopping the *Track* activity executes a codel, `stopTrack`, that immediately sets `linear.x` and `angular.z` speeds at respectively 0 m/s and 0 rad/s (this stops the robot very abruptly, without a regular deceleration). A careful examination of multiple traces shows that the robot typically stops within 17 ms to 35 ms after detecting the problem, which is consistent with the TrackTask task period of 50 ms (so on average the `stopTrack` will be executed after 25 ms). Section 7.5 presents a more rigorous evaluation of this response time, using formal verification on the offline model.

[5] Code is available here: https://redmine.laas.fr/projects/minnie/gollum.

[6] This type of error often occurs upon startup of the experiment where all the modules are starting at once, and subtle race condition can lead to these situations.

7.2. Verification

Since we have a formal model for the modules in Minnie, it is also possible to check its behavior using the tools available in TINA: *play* to simulate the model; *selt* and *sift* to model-check properties; *plan* to find possible firing schedules times from an execution sequence; etc. This verification step allows the designer to check specific properties such as schedulability, the reachability of (or better impossibility to reach) particular state and maximum response time between two states. But what is also interesting during this phase is that while checking a property, the designer may also discover inappropriate behaviors that are not directly expressed in the property. For example, by checking the maximum delay for taking an action, the designer may discover an execution sequence that leads to the immediate realization of this action when the system starts up. Here, the property is verified but it permits to identify an inappropriate behavior. Thus, the verification stage can be considered both as a means of proving the good behavior of the system and as a means of debugging it.

As mentioned previously, to proceed to the verification of Minnie, one needs to provide a client which sends requests and receives replies, otherwise the model only starts the permanent services (if any). These requests are dispatched to the proper modules for "execution" and replies are received accordingly. On the complete (offline/verification) model generated from Minnie, we are not able to explore the complete state set of the system (with a limit of 16 GB of RAM). Yet, we can perform complete verification on one of the components.

The verification of a safety invariant is straightforward. It is enough to express the property that we expect to be true on each reachable state as a Boolean combination of atomic properties. Then the property can be checked on the fly with the *sift* tool. *sift* enumerates the reachable states of the system, stopping if the invariant is false, in which case it returns a counter-example that can be used to compute an execution trace explaining how to reproduce the error. In the cases where we are able to generate the whole state space, we can use one of the model-checkers included in TINA, called *selt*, to prove more complex properties (properties than can be expressed as formulas in Linear Temporal Logic, LTL).

We have used this mechanism to check several properties on the Minnie use case. We now give three different examples.

7.3. Schedulability

We can check that a periodic task, in a module, will always finish its execution before its next activation. To this end, it is enough to check that the FIACRE process modeling the $G^{en}{}_{o}M$ execution task can never reach its `overshoot` state (this state is the same one used in the HIPPO version to detect overshoot at runtime, see Section 7.1). This is an example of safety property. So for the VELODYNE module, which includes only two periodic tasks (`velodyne_scan` and `velodyne_pose`), it is enough to check an invariant of the form: $\neg$(velodyne_scan_overshoot $\vee$ velodyne_pose_overshoot)

Our model also includes a specific mechanism for dealing with CPU cores. We can fix a maximal number of available cores, with the constraint that two codels cannot share the same core at the same time. Even if we cannot generate the whole state space for the model, `sift` was able to find scheduling errors when using only 3 cores with VELODYNE. This led us to change and optimize the codels for the VELODYNE to solve the problem.

Table 2
Complexity of checking mutual exclusion between services.

Scenario	*JoystickOn* then *Track*	*Track* then *JoystickOn*
Time	16 min	10 h
#classes	42,714,945	832,778,752
#markings	5,817,082	44,533,432

7.4. Mutual exclusion

The RMP440 module is critical, since it commands the speed of the wheels, and needs a careful verification. When running the *Track* service, it grabs the speed **Cmd** from SAFETYPILOT, and when running the *JoystickOn* service, it computes a speed from JOYSTICK. These two services are declared as interrupting each other: they should never run together.

We are able to check the property in two symmetrical scenarios (expressed using different models of the client), considering the RMP440 module alone (i.e. without inclusion of other modules): a scenario where a *JoystickOn* request is sent, shortly followed by a *Track* request; and the other way around. We are able to prove that our invariant is true. This is the worst-case since it means that we have to explore the whole state space. We give some information on the complexity of the problem in Table 2. In this context, a marking is a particular set of states and values for all the processes and variables in the system. A class is a state extended with timing information on the enabled transitions (therefore we can have several classes with the same marking).

7.5. Delay to stop

The last property we check is related to the HIPPO monitor presented in Section 7.1, Listing 11. The problem here is to compute, offline, the Worst-Case Response Time (WCRT) between an interrupt from the *Track* activity, and the end of the execution of the `stopTrack` codel. This is an example of quantitative property that can be checked by adding a monitor to the model. (Listing 12 gives the code for this monitor.) Indeed, it is possible to reach state `robot_NOT_stopped` in process `rmp440_Track_Stopper` if and only if the timeout used in state `wait_delay` (141 ms in this case, see line 19) is less than or equal to the WCRT. Hence, to compute the right value, it is enough to try different values for the timeout.

We used this approach to compute a theoretical WCRT value (141 ms) with a precision of 1 ms. This value is much higher than the one measured during our tests with the real robot. On the other hand, with our approach, it is possible to generate a scenario corresponding to this worst-case. An analysis of the counter-example computed by TINA shows that this scenario is indeed possible in the real system. This scenario corresponds to an extreme situation where we added twice the running time (WCET) of a slow codel (43 ms), conflicting with the codel in charge of stopping the robot.

Yet, the theoretical WCRT is still "reasonable"; even at 6 m/s, Minnie will travel at most 85 cm before pulling the brakes. Also, while this scenario is very unlikely, the value of 141 ms should be the one chosen when performing a safety analysis, or in case we want to certify our system.

Overall, the automatic synthesis for such a complex robotic experiment of a complete formal model which can be both used for offline and online verification is rather encouraging. It shows that some non trivial critical properties can be checked beforehand, even at design time; and that some specifications can be translated into online monitor which will formally enforce them. Last, but not the least, the deployment of both models also provides a positive feedback on the tool itself and its semantics, but also on the specific architecture needed to run a particular experiment (number of cores needed, proper initialization sequence, etc.).

```
process rmp440_Track_Stopper(
   &track_started:bool,
   &track_stopped:bool,
   &TrackTask_activities: Activities_rmp440_TrackTask_Array,
   Track_index: act_inst_rmp440_TrackTask_index_type) is

   states wait_started, wait_stop, wait_delay, finished,
     robot_stopped, robot_NOT_stopped

   from wait_started
      wait [0,0];
      on (track_started); // wait the Track service has
     started
      to wait_stop

   from wait_stop // (no wait) can stop anytime
      TrackTask_activities[Track_index].stop := true;
      to wait_delay

   from wait_delay
      wait [141,141]; //The response time value we want to
     measure
      to finished

   from finished
      wait [0,0];
      if (track_stopped)  then
         //The robot has been stopped before the delay
         to robot_stopped
      else
         //The robot has not been fully stopped yet
         to robot_NOT_stopped
      end
```

Listing 12: A Fiacre monitor used to measure a response time with `sift` (the `track_started` and `track_stopped` booleans are set by the *Track* activity Fiacre process).

8. Conclusion

We describe a language and a compiler, called Hippo, able to generate executable code from its formal model. This tool is based on an extension of the formal language Fiacre with new operators for activating and waiting on the result of external tasks. Our implementation follows a synchronous principle for the behavior engine and uses a more flexible, asynchronous model for tasks scheduling. We evaluated the performance of applications generated with Hippo and measured the overhead of our execution engine, with both synthetic benchmarks and a robotic use case. Our results are promising and reasonable for a real usage.

We make several contributions beyond the implementation of this approach. First, we show how to interpret the semantics of Hippo in plain Fiacre, which means that we are still able to check temporal properties on this new, "runtime oriented" language. Next, we show the effectiveness of this approach by reporting our experience with a non-trivial use case; a mobile robot navigation application derived from a high-level specification written in the $G^{en}{}_{o}M$ framework. This specification has been translated into Hippo to allow the automatic generation of an executable which fully controls the robot in place of the regular $G^{en}{}_{o}M$ synthesized module. We also discuss how this executable can be enhanced in order to enforce critical safety properties at runtime. Using the same template, the specification can also be used to synthesize a verification model which can be analyzed offline, strengthening the confidence we put in the application.

We have also identified some of the limitations of our approach that we would like to address in future works. Concerning formal verification, it is not possible to state that Hippo produces faithful code, meaning that the online and offline models have equivalent observational semantics. Instead, we focus on checking that the behavior of the (Hippo) implementation is included in the behavior of its (Fiacre) specification. We benefit in this case from the compositionality of our encodings. Our experiments also expose the limits of using formal verification due to the state-explosion problem. We are able to prove many safety invariants on our most complex models, but we sometimes need to abstract some of the behavior (for instance by limiting the ability for a component to randomly fail) or to limit the state space of the system by focusing on particular "scenarios" (see our experiments in Section 7.4). We plan to continue our investigations on these issues. For example, we are trying to better take into account the deterministic aspects of Hippo (which would reduce interleavings) as well as to take into account the different "modes" of the system (nominal mode, failure, etc.). In addition, there are perspectives on the development of specific design tools (debugging, simulator) based on model-checking methods.

To conclude, we would like to stress that the most innovative parts of our contributions are the result of confronting viewpoints and objectives that originate from the diverse fields of expertise of the authors: real-time system design and analysis; execution control and planning; formal verification;…This is definitely an asset in this context. This is evidenced, for example, in our design choices for the scheduling model of our engine, which is in part motivated by formal verification—because we want to avoid complex preemptive behaviors that could make verification harder—but is also influenced by the kind of system we target. Another example can be seen in our use of runtime monitoring, where we use the same high-level model to generate both an observer for the verification model (to check a timed property with model-checking) and a runtime monitor for the execution code (to check the same property at runtime). This also helps explain why each aspect of our design may appear as the result of a trade-offs: our scheduling strategy prioritizes predictability but it may not be optimal; we are good in terms of verification but our toolchain is not formally certified; we target the control part of robotic systems but we do not address other interesting problems, such as planning or control at the acting level. Each of these aspects naturally leads to possible improvements and is the subject of possible future works.

CRediT authorship contribution statement

Pierre-Emmanuel Hladik: Conceptualization, Methodology, Software, Validation, Investigation, Writing - original draft, Writing - review & editing, Visualization. **Félix Ingrand:** Conceptualization, Methodology, Software, Validation, Investigation, Writing - original draft, Writing - review & editing, Visualization. **Silvano Dal Zilio:** Conceptualization, Methodology, Validation, Investigation, Writing - review & editing. **Reyyan Tekin:** Software, Validation, Investigation, Visualization.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

The authors would like to thank the anonymous reviewers, as well as Bernard Berthomieu, Mohamed Oussama Ben Salem and Mohammed Foughali for their careful reading and insightful suggestions that helped improve the quality of this paper. This work has been supported by the European Union's Horizon 2020 research and innovation programme under grant agreement No. 825619 (AI4EU) and the Artificial and Natural Intelligence Toulouse Institute - Institut 3iA (ANITI) under grant agreement No: ANR-19-PI3A-0004. We also acknowledge the support of the RTRA STAE project Daedalus, under grant No: R051-L00-T00.

Appendix. FIACRE model of a FIFO scheduler

A SMP FIFO scheduler can be simply modeled with a queue. The proposed model is inspired from Foughali et al. (2018). The scheduler is coded by a process (line 8) with two shared data (see Listing 13 for an example). The `ready_list` (line 47) is a queue (native type of FIACRE) used as the classical ready list in a scheduler and it is used to stack the tasks that are ready to be executed. The `launch` (line 48) variable is an array of booleans that states if a task can start its execution or not. When a task is activated, its `id` is queued in the `ready_list` (line 28) and a test is done to know if a processor is available to execute the new task (line 13). If it is possible, the `launch` is updated (line 15). Then, the task waits that a processor is available to continue its execution by checking the status of `launch` (line 31). When a task terminates its execution, a processor is free (line 37) and if the `ready_list` queue is not empty the first task is resumed by changing its status in `launch` (line 13).

Remark that we use the best-case execution time and the worst-case execution to represent the execution time of the task (line 36). The translation from a HIPPO model to a FIACRE model with a FIFO scheduler is implemented and available on gitlab.

```
1  const nbOfProcessors : nat is 2
2  const nbOfTasks : nat is 120
3  type fifo is queue nbOfTasks of 0..(nbOfTasks-1)
4  type start_tab is array nbOfTasks of bool
5  ...
6  process scheduler (
7      &ready_list: fifo,
8      &launch: start_tab,
9      &unused_proc: nat) is
10   states exec
11   from exec
12     on (not (empty ready_list)) and
13         (unused_proc > 0));
14     unused_proc:= unused_proc-1;
15     launch [first ready_list]:= true;
16     ready_list := dequeue ready_list;
17     wait [0,0]; to exec
18
19 process p_task [SyncG : none, t_a : ty1, t_t : tyOut
       ] (
20     id : 0..nbOfTasks-1,
21     &unused_proc : nat,
22     &ready_list : fifo,
23     &launch : start_tab) is
24   states waiting, sched_activate, sched_resume,
       running, sched_terminate, synchronizing,
       terminating
25   var param : tyIn, ret : tyOut
26   from waiting
27     t_a?param; to sched_activate
28   from sched_activate /* task activation: scheduler
       call */
29     ready_list := enqueue(ready_list, id);
30     wait [0,0]; to sched_resume
31   from sched_resume
32     on launch[id]; launch[id] := false;
33     wait [0,0]; to running
34   from running
35     ret := c_foo(param); wait [$bcet, $wcet];
36     to sched_terminate
37   from sched_terminate  /* task termination:
       scheduler call */
38     unused_proc := unused_proc + 1; wait [0,0]; to
       synchronizing
39   from synchronizing
40     SyncG; to terminating
41   from terminating
42     t_t! ret; to waiting
43 ...
44
45 component Main is
46     var
47         ready_list : fifo := {||},
48         launch : start_tab := [false,..., false],
49         unused_proc : nat := nbOfProcessors
50     ...
51
52     par
53     ...
54     ||  scheduler (&ready_list, &launch, &
       unused_proc)
55     ||  p_task [SG, t_a , t_t](1, &unused_proc, &
       ready_list, &launch)
56     ...
57     end
```

Listing 13: An example of FIFO scheduler in FIACRE.

References

Abdellatif, T., Bensalem, S., Combaz, J., De Silva, L., Ingrand, F., 2012. Rigorous design of robot software: A formal component-based approach. Robot. Auton. Syst. 60 (12), http://dx.doi.org/10.1016/j.robot.2012.09.005.

Abdellatif, T., Combaz, J., Sifakis, J., 2010. Model-based implementation of real-time applications. In: International Conference on Embedded Software. URL http://dl.acm.org/citation.cfm?id=1879052.

Abdellatif, T., Combaz, J., Sifakis, J., 2013. Rigorous implementation of real-time systems – from theory to application. Math. Struct. Comput. Sci. 23 (4), http://dx.doi.org/10.1017/S096012951200028X.

Amnell, T., Fersman, E., Pettersson, P., Sun, H., Yi, W., 2002. Code synthesis for timed automata. Nord. J. Comput. 9 (4), http://dx.doi.org/10.5555/779110.779112.

Behrmann, G., David, A., Larsen, K.G., 2004. A tutorial on Uppaal 4.0. In: Formal Methods for the Design of Real-Time Systems. Springer, Berlin, Heidelberg, pp. 200–236. http://dx.doi.org/10.1007/978-3-540-30080-9_7.

Ben Rayana, S., Bozga, M., Bensalem, S., Combaz, J., 2016. RTD-finder - a tool for compositional verification of real-time component-based systems. In: Chechik, M., Raskin, J.-F. (Eds.), Tools and Algorithms for the Construction and Analysis of Systems. Springer, Berlin, Heidelberg, pp. 394–406. http://dx.doi.org/10.1007/978-3-662-49674-9_23.

Berthomieu, B., Bodeveix, J.-P., Farail, P., Filali, M., Garavel, H., Gaufillet, P., Lang, F., Vernadat, F., 2008a. Fiacre: an intermediate language for model verification in the topcased environment. In: Proc. of the Embedded Real-Time Software. ERTS, URL https://hal.archives-ouvertes.fr/inria-00262442.

Berthomieu, B., Dal Zilio, S., Vernadat, F., 2020. A FIACRE V3.0 primer. [online].

Berthomieu, B., Garavel, H., Lang, F., Vernadat, F., 2008b. Verifying dynamic properties of industrial critical systems using TOPCASED/FIACRE. ERCIM News (75), URL http://ercim-news.ercim.eu/verifying-dynamic-properties-of-industrial-critical-systems-using-topcasedfiacre.

Berthomieu, B., Le Sergent, T., 1994. Programming with behaviors in an ML framework—the syntax and semantics of LCS. In: Proc. of the 5th European Symposium on Programming. ESOP, http://dx.doi.org/10.1007/3-540-57880-3_6.

Berthomieu, B., Ribet, P.-O., Vernadat, F., 2004. The tool TINA – Construction of abstract state spaces for Petri nets and time Petri nets. Int. J. Prod. Res. 42 (14), http://dx.doi.org/10.1080/00207540412331312688.

Burns, A., 1999. The ravenscar profile. Ada Lett. XIX (4), http://dx.doi.org/10.1145/340396.340450.

Carruth, J.A., Misra, J., 1996. Proof of a real-time mutual-exclusion algorithm. Parallel Process. Lett. 6 (2), http://dx.doi.org/10.1142/S012962649600025X.

Caspi, P., Curic, A., Maignan, A., Sofronis, C., Tripakis, S., Niebert, P., 2003. From simulink to SCADE/Lustre to TTA: a layered approach for distributed embedded applications. In: Proc. of the 2003 ACM SIGPLAN Conference on Language, Compiler, and Tool for Embedded Systems. http://dx.doi.org/10.1145/780732.780754.

Dal Zilio, S., Berthomieu, B., Le Botlan, D., 2015. Latency analysis of an aerial video tracking system using fiacre and tina. In: FMTV Verification Challenge of WATERS 2015. LAAS-VERTICS, URL http://arxiv.org/abs/1509.06506v1.

David, A., Larsen, K.G., Legay, A., Mikučionis, M., Poulsen, D.B., 2015. UPPAAL SMC tutorial. Int. J. Softw. Tools Technol. Transf. 1–19. http://dx.doi.org/10.1007/s10009-014-0361-y.

Dellabani, M., 2018. Formal Methods for Distributed Real-Time Systems (Ph.D. thesis). Université Grenoble Alpes.

Derler, P., Lee, E., Matic, S., 2008. Simulation and implementation of the PTIDES programming model. In: Proc. of the 12th IEEE/ACM International Symposium on Distributed Simulation and Real-Time Applications. DS-RT, http://dx.doi.org/10.1109/DS-RT.2008.51.

Durrieu, G., Faugère, M., Girbal, S., Gracia Pérez, D., Pagetti, C., Puffitsch, W., 2014. Predictable flight management system implementation on a multicore processor. In: Proc. of the Embedded Real Time Software and Systems. ERTS, URL https://hal.archives-ouvertes.fr/hal-01121700/.

Evrard, H., Lang, F., 2015. Automatic distributed code generation from formal models of asynchronous concurrent processes. In: Proc. of the 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing. http://dx.doi.org/10.1109/PDP.2015.96.

Feiler, P., Gluch, D., Hudak, J., 2006. The Architecture Analysis & Design Language (AADL): An Introduction. http://dx.doi.org/10.21236/ada455842.

Forget, J., Boniol, F., Lesens, D., Pagetti, C., 2009. Implementing multi-periodic critical systems: from design to code generation. In: FM Workshop on Formal Methods for Aerospace. URL https://arxiv.org/abs/1003.2871.

Foughali, M., 2018. Formal Verification of the Functional Layer of Robotic and Autonomous Systems (Ph.D. thesis). LAAS/CNRS.

Foughali, M., Bensalem, S., Combaz, J., Ingrand, F., 2020. Runtime verification of timed properties in autonomous robots. In: Proc. of the 18th ACM-IEEE International Conference on Formal Methods and Models for System Design. MEMOCODE, http://dx.doi.org/10.1109/MEMOCODE51338.2020.9315156.

Foughali, M., Berthomieu, B., Dal Zilio, S., Hladik, P.-E., Ingrand, F., Mallet, A., 2018. Formal verification of complex robotic systems on resource-constrained platforms. In: Proc. of the 2018 ACM/IEEE Conference on Formal Methods in Software Engineering. FormaliSE, http://dx.doi.org/10.1145/3193992.3193996.

Foughali, M., Ingrand, F., Seceleanu, C., 2019. Statistical model checking of complex robotic systems. In: International SPIN Symposium on Model Checking of Software. http://dx.doi.org/10.1007/978-3-030-30923-7_7.

Garavel, H., Lang, F., Serwe, W., 2017. From LOTOS to LNT. In: ModelEd, TestEd, TrustEd. Springer, pp. 3–26. http://dx.doi.org/10.1007/978-3-319-68270-9_1.

Garavel, H., Viho, C., Zendri, M., 2001. System design of a CC-NUMA multiprocessor architecture using formal specification, model-checking, co-simulation, and test generation. Int. J. Softw. Tools Technol. Transf. 3 (3), 314–331. http://dx.doi.org/10.1007/s100090100044.

Guerra, M., Efimov, D., Zheng, G., Perruquetti, W., 2016. Avoiding local minima in the potential field method using input-to-state stability. Control Eng. Pract. 55 (C), 174–184. http://dx.doi.org/10.1016/j.conengprac.2016.07.008.

Halbwachs, N., Caspi, P., Raymond, P., Pilaud, D., 1991. The synchronous dataflow programming language Lustre. Proc. IEEE 79 (9), http://dx.doi.org/10.1109/5.97300.

Harbour, M.G., García, J.J.G., Gutiérrez, J.C.P., Moyano, J.M.D., 2001. MAST: Modeling and analysis suite for real time applications. In: Proc. of the 13th Euromicro Conference on Real-Time Systems. ECRTS, http://dx.doi.org/10.1109/EMRTS.2001.934015.

Henzinger, T., Horowitz, B., Kirsch, C., 2003. Giotto: a time-triggered language for embedded programming. Proc. IEEE 91 (1), http://dx.doi.org/10.1109/JPROC.2002.805825.

Henzinger, T., Kirsch, C., 2007. The embedded machine : Predictable, portable real-time code. ACM Trans. Program. Lang. Syst. 29 (6), http://dx.doi.org/10.1145/1286821.1286824.

Henzinger, T.A., Kirsch, C.M., Majumdar, R., Matic, S., 2002. Time-safety checking for embedded programs. In: Proc. of the Second International Conference on Embedded Software. EMSOFT, http://dx.doi.org/10.1007/3-540-45828-X_7.

Herrb, M., 1992. Pocolibs: POsix COmmunication LIbrary. Tech. Rep., LAAS-CNRS, URL https://git.openrobots.org/projects/pocolibs/gollum/index.

Hladik, P.-E., 2018. A brute-force schedulability analysis for formal model under logical execution time assumption. In: Poc. of the 33rd ACM/SIGAPP Symposium on Applied Computing. http://dx.doi.org/10.1145/3167132.3167199.

Hladik, P.-E., 2020. Hippo. Tech. Rep., LAAS-CNRS, URL https://redmine.laas.fr/projects/genom3-fiacre-template/gollum/hippo.

Kristensen, J., Mejlhilm, A., Pedersen, S., 2017. Automatic Translation from UPPAAL to C. Tech. Rep., Department of Computer Science, Aalborg University.

Lasnier, G., Zalila, B., Pautet, L., Hugues, J., 2009. Ocarina : An environment for AADL models analysis and automatic code generation for high integrity applications. In: Proc. of the Reliable Software Technologies – Ada-Europe 2009. http://dx.doi.org/10.1007/978-3-642-01924-1_17.

Lee, E., Sangiovanni-Vincentelli, A., 1996. The tagged signal model-a preliminary version of a denotational framework for comparing models of computation. Tech. Rep., Department of Electrical Engineering and Computer Science, University of California, URL https://ptolemy.berkeley.edu/papers/96/denotational/denotationalERL.pdf.

Liu, J., Lee, E., 2002. Timed multitasking for real-time embedded software. IEEE Control Syst. Mag. 23, http://dx.doi.org/10.1109/MCS.2003.1172830.

Louise, S., Lemerre, M., Aussagues, C., David, V., 2011. The OASIS kernel: A framework for high dependability real-time systems. In: Proc. of the 13th IEEE International Symposium on High-Assurance Systems Engineering. HASE, http://dx.doi.org/10.1109/HASE.2011.38.

Mallet, A., Pasteur, C., Herrb, M., Lemaignan, S., Ingrand, F., 2010. Genom3: Building middleware-independent robotic components. In: IEEE International Conference on Robotics and Automation. pp. 4627–4632. http://dx.doi.org/10.1109/ROBOT.2010.5509539.

Navet, N., Fejoz, L., Havet, L., Sebastian, A., 2016. Lean model-driven development through model-interpretation: the CPAL design flow. In: Proc. of the 8th European Congress on Embedded Real-Time Software and Systems. ERTS, https://hal.archives-ouvertes.fr/hal-01289494/.

Object Management Group, Inc. (OMG), 2009. UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems. OMG Document Number: formal/2009-11-02, URL https://www.omg.org/spec/MARTE/1.0/PDF.

Ousterhout, J.K., Jones, K., 2008. TCL and the TK Toolkit (Addison-Wesley Professional Computing), second ed. Addison-Wesley Publishing Company, USA.

Pagetti, C., Forget, J., Falk, H., Oehlert, D., Luppold, A., 2018. Automated generation of time-predictable executables on multicore. In: Proc. of the Real-Time Networks and Systems. RTNTS, http://dx.doi.org/10.1145/3273905.3273907.

Quigley, M., Gerkey, B., Conley, K., Faust, J., Foote, T., Leibs, J., Berger, E., Wheeler, R., Ng, A.Y., 2009. ROS: an open-source Robot Operating System. In: IEEE International Conference on Robotics and Automation.

Sifakis, J., 2005. A framework for component-based construction. In: Proc of the 3rd IEEE International Conference on Software Engineering and Formal Methods. SEFM, http://dx.doi.org/10.1109/SEFM.2005.3.

Singhoff, F., Legrand, J., Nana, L., Marcé, L., 2004. Cheddar a flexible real time scheduling framework. In: Proc. of the 2004 Annual ACM SIGAda International Conference on Ada: The Engineering of Correct and Reliable Software for Real-Time & Distributed Systems using Ada and Related Technologies. SIGAda, http://dx.doi.org/10.1145/1032297.1032298.

Socci, D., Poplavko, P., Bensalem, S., Bozga, M., 2013. Modeling mixed-critical systems in real-time BIP. In: 1st Workshop on Real-Time Mixed Criticality Systems. URL https://hal.archives-ouvertes.fr/hal-00867465/.

Tanguy, J., Béchennec, J.-L., Briday, M., Roux, O.H., 2014. Reactive embedded device driver synthesis using logical timed models. In: Proc. of the 4th International Conference on Simulation and Modeling Methodologies, Technologies and Applications. SIMULTECH, http://dx.doi.org/10.5220/0005040101630169.

The LTTng Project, 2020. LTTng website. URL https://lttng.org. (Accessed 27 October 2020).

Pierre-Emmanuel Hladik is an Assistant Professor at the National Institute of Applied Sciences (INSA) of Toulouse and he carries out his research activity at the Laboratory of Architecture and Analysis of Systems (LAAS-CNRS) in the VERTICS team. He teaches in the field of autonomous embedded systems. His research interests deal with the modeling and verification of constraint concurrent systems, and in particular scheduling and methods to design safe systems.

Félix Ingrand (M) is a tenured researcher at CNRS. After his Ph.D. from University of Grenoble (1987), he spent four years at SRI International (Menlo park, CA) where he worked on procedural reasoning. He joined the Robotics and Artificial Intelligence Group at CNRS/LAAS in 1991. His work deals with architecture for autonomous systems with an emphasis on the decisional aspect. He has done some work on procedural reasoning, as well as on action planning and plans execution control. Recently, he worked on extending the LAAS Architecture toward formal approaches to perform validation, verification, and correct controller synthesis. This work is being conducted in various past and current projects: AMAES (ANR), MARAE (FNRAE), GOAC (ESA), CPSE Labs (H2020), AI4EU (H2020), AIPlan4EU (H2020).

Silvano Dal Zilio is a CNRS researcher at LAAS, in Toulouse, France. His research interests include the formal verification of concurrent and distributed systems; methods and tools for checking the safety of critical embedded systems; and concurrency semantics. He focuses on new verification methods and tools for checking critical systems having strong temporal and timing requirements. Silvano leads the VERTICS group, which develops the TINA model-checking toolbox and the FIACRE formal specification language.

Reyyan Tekin is a research intern at ENS-PSL University and Inria. He received his B.Sc. degree from Lille University in 2019 and he is currently pursuing a M.Sc. degree at Toulouse III - Paul Sabatier University. His research interests include real-time systems, embedded systems and programming languages.