Hippo: A Formal-Model Execution Engine to Control and Verify Critical Real-Time Systems

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

► To cite this version:

Pierre-Emmanuel Hladik, Félix Ingrand, Silvano Dal Zilio, Reyyan Tekin. Hippo: A Formal-Model Execution Engine to Control and Verify Critical Real-Time Systems. 2020. hal-03017661v1

HAL Id: hal-03017661
https://laas.hal.science/hal-03017661v1

Preprint submitted on 21 Nov 2020 (v1), last revised 15 Jun 2021 (v4)

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

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

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
firstname.lastname@laas.fr

November 20, 2020

Abstract

The design of embedded real-time systems requires specific toolchains to guarantee time constraints and safe behavior. These tools 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 implementation built around a common formalism.

Our formalism 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 obtain a HIPPO model from an initial specification of the system based on the robotics programming framework GenoM. We illustrate our approach by describing how the HIPPO runtime is used to control this robot, but also how we can use formal verification 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 need to be managed in a coherent way all along the design process and need to address timing constraints and execution semantic in an holistic way during the system's modeling, verification, and implementation phases.

This paper presents such an integrated toolchain, named HIPPO, and focuses especially on tools to carry out real-time executable that can be formally verified. 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, and 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 has 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) model [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 a FIACRE model is semantically equivalent to its TTS.

For the HIPPO toolchain, we chose to generate a code as close as possible to the TTS level. As a result, the code generated after compilation is very close to the semantics of the initial model. Moreover, we only need to rely on a simple runtime that is used to ensure that the control flow of the executable is identical, in every detail, to the behavior of the TTS model. Therefore, the generated code, coupled with the HIPPO engine, has the same semantics as the initial specification.

Outline. 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 analyze their shortcommings which we want to address in our approach.

Section 3 presents the extensions to the FIACRE language that allow us to describe HIPPO models. In particular, we describe how we extend the description of the functional and real-time behavior with the addition of tasks and events. A formal definition of these extensions is also provided to allow the verification of the model.

In Section 4, we describe the design principles of the HIPPO runtime. 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. In this section, we also explain the methods used to increase our confidence in the implementation and show some performance results.

Before concluding, Section 5 presents a case study : how we deploy Hippo and Fiacre along G$^{en}_{o}$M to control and verify the autonomous navigation of an outdoor robotic platform.

2 Motivation and Related Work

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

2.1 Motivation

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 Language 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, producing thus multiple models.

On the other hand, the process of transforming the model into executable code can be entirely done manually, semi-automatically by producing for example 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 this approach 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.

We also note that in the literature there are few examples of real and fully functional applications where verification and code generation is performed jointly. Another of our motivations is therefore to provide the community with a complete and documented ap-

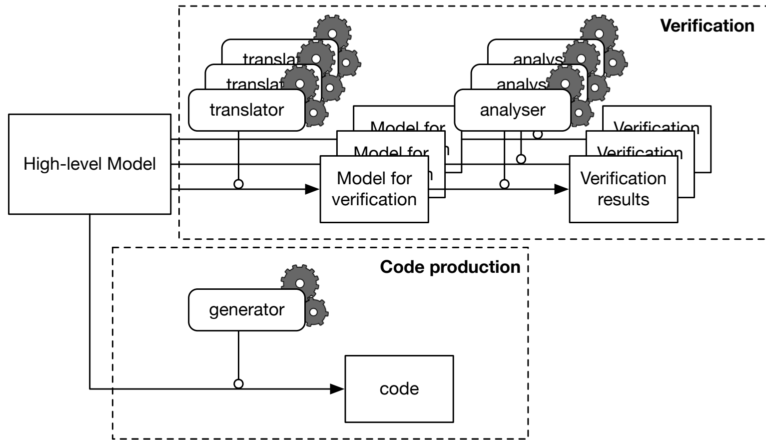


Figure 1: A schematic representation of a generic design process for real-time embedded systems.

plication of a complex critical real-time system for which it is possible to use design, code generation and verification methods.

The work presented in this paper is part of this context and aims to propose a tool to reduce this semantic gap for real-time embedded applications and show its usability on a real case study.

2.2 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.

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].

Other work 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 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.

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 of the 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.

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, there is a now a distributed and real-time implementation of BIP [Dellabani, 2018].

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.* [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.

2.3 Hippo Toolchain

As already mentioned, the work presented in this paper proposes a complete toolchain to verify and execute real-time applications while minimizing the semantic gap. To do this, we are working in the environment 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 the semantics of the TTS produced by this language. Thus the engine plays exactly the same trace as its model in TTS. This choice has been made to guarantee the behavior as easily as possible and to ensure the atomicity of the operations. 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 as close as possible to the execution (see Fig. 2).

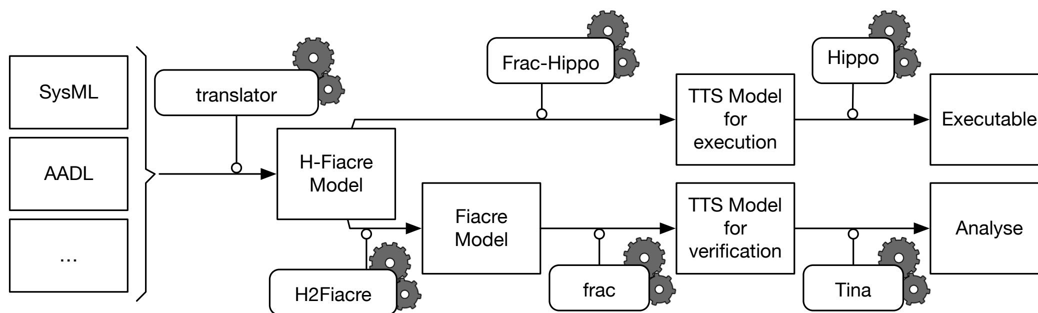


Figure 2: A schematic representation of the HIPPO toolchain.

3 Fiacre extensions

We now 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 first-order functional language included in FIACRE), 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 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 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), of the form `wait` $[a, b]$, which represents the possible delays during which an event can occur (where a, b are constants in $\mathbb{Q}^+$). The intended meaning is that the control state needs to wait a duration between a and b before forwarding.

Priorities can be added between communication events to specify that one event should always occur before another.

```
1  type tyEvt is record time : int, id : nat end
2  /* Processes */
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.* controler, 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 an H-FIACRE model through a *task*. Each operation has its own task and is scheduled by the operating system (see Sect. 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 Sect. 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 catchs 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 transfering 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 than with tasks, due to the synchronous implementation of the HIPPO runtime (see Sect. 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 Restriction of H-FIACRE

H-FIACRE does not impose any restrictions on the usage of FIACRE language, except on delays. For our purpose, we will require every time interval in a `wait [a,b]` statement to be punctual, meaning that $a = b$. This restriction ensures that the behavior of the timing system is deterministic[1].

3.2.4 A toy example in H-FIACRE

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 transition 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).

[1]It is possible to have an interval with different bounds, but during the execution the engine reacts as soon as possible and fires the waiting transition with its lower bound. The only restriction is to have a left-half-open interval.

```
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]; /* wait first event */
12     to wait_second
13   from wait_second
14     select
15       wait [200,200];
16       to start_print
17     []e?tmp[1];  /* wait second event */
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.

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 Semantic of a Task and an Event

The behavior of our new statements can be formally expressed in Fiacre, and so it is possible to define a transformation to rewrite an 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 Semantic of an external task

Listing 3 gives the template for the translation of an (executable) H-Fiacre into a plain Fiacre process. Each task `t_j` is modeled by a single Fiacre process. For each `start t_j`, respectively `sync t_j`, in the H-Fiacre model a port `t_activate_j_i` is created,

resp. t_terminated_j_i (see line 1). They are used to model the activation, resp. the termination, of the task and to pass parameters, resp. the return value.

The behavior of the task is modeled by different states that express the life-cycle of a task. A state (line 4) synchronizes the task with its activation through one port t_activate_j_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 11) calls the functional C code to compute the return value. Note that for the FIACRE model the call of a function is assumed to be in null time, so the execution time is represented by an interval of numbers (line 13) 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 response times (and not the execution time) (see next Section for more details).

A terminating state (line 20) signals the end of the task and returns the result through port.

```
1  process p_task_t_j [SyncGlobal_j : none, {t_activate_j_i | i } : ty1, {t_terminated_j_i
       | i } :  tyOut] is
2    states waiting, running, synchronizing, terminating
3    var param : tyIn, ret : tyOut
4    from waiting
5      select
6          t_activate_j_0?param; to running
7      []  t_activate_j_1?param; to running
8        ...
9      []  t_activate_j_n?param; to running
10     end
11   from running
12     ret := c_foo(param); /* The computational function is called */
13     wait[$bcrt, $wcrt];  /* simulate the WCRT */
14     ret := any;
15     to synchronizing
16   from synchronizing
17     SyncGlobal_j; /* Synchronization with the global tick */
18     to terminating
19   from terminating
20     select /* The return value are written */
21         t_terminated_j_0! ret; to waiting
22     []  t_terminated_j_1 ! ret; to waiting
23       ...
24     []  t_terminated_j_n ! ret; to waiting
25     end
```

Listing 3: Template to translate a task in FIACRE.

An additional state (line 16) is added to the model to represent the time-triggered behavior of the HIPPO runtime (see Section 4). The termination signals, t_terminated_j_i (lines 20-24), need to be synchronized with the clock of the HIPPO runtime. To do this, a new process global_clock is added (see Listing 4) with a port synchronization SyncGlobal_j. The port N is only introduced to allow the clock to progress if no synchronization is required and a priority is added between SyncGlobal_j and N to assure that

SyncGlobal_j always happens if a synchronization is needed. This model guarantees that the termination of a task is only signaled at a tick.

The statement start t param can be simply replaced by a port emission t_activate_j_i!param and the sync t ret with t_terminated_j_i?ret.

```
1  process global_clock [{SyncGlobal_j | j in 1..#tasks }, N : none] is
2    states timer, tick
3    from timer
4      wait[1,1];
5      to tick
6    from tick
7      select
8          N; to timer
9      []  SyncGlobal_1; to tick
10     ...
11     []  SyncGlobal_j; to tick
12     end
```

Listing 4: Global clock template.

3.3.2 Semantics for an external event

The Listing 5 shows the formal definition in Fiacre for the behavior of an event. This rewriting is very close to that of a task. An event is synchronized to the global clock (line 9) and uses the port e_happened_k_i to synchronize the occurrence of the event with the main behavior (lines 12 to 17).

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_k [SyncGlobal_k : none, {e_happened_k_i | i} : tyEvt] is
2    states waiting, synchronizing, posting
3    var tmp : tyEvt
4    from waiting
5      wait [30, 1000]; /* timed pattern to produce event */
6      tmp := any;
7      to synchronizing
8    from synchronizing
9      SyncGlobal_k;
10     to posting
11   from posting
12     select
13         e_happened_0!tmp; to waiting
14     []  e_happened_1!tmp; to waiting
15       ...
16     []  e_happened_n!tmp; to waiting
17     end
```

Listing 5: Event behavior 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. The implementation of the engine and its threads, as well as the underlying scheduling hypothesis are introduced and then we conclude this section with encouraging implementation and experimental results.

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 an 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 corresponds 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.

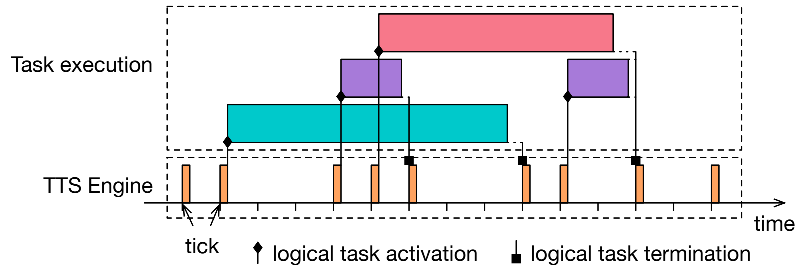


Figure 3: A schematic representation of the execution control flow of Hippo.

In Figure 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 semantic 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 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 extern code in H-Fiacre (such as the code of external tasks).

In Hippo, two type 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 events 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.

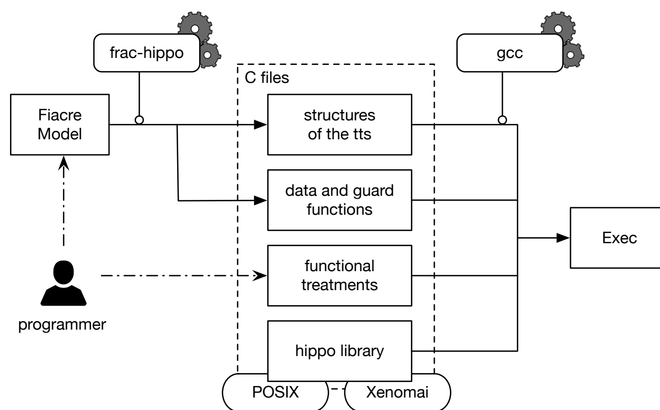


Figure 4: Tools for code generation and compilation.

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, *i.e.*, no memory allocation occurs during the execution. This design choice was made to reduce the time execution and to port the runtime on a microcontroller with a micro-OS such as FreeRTOS or 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 A 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 Sect. 3.3.1, in the Fiacre model 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.

An other 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. Figure 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].

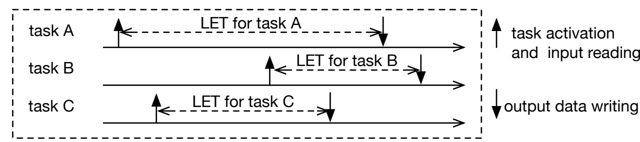


Figure 5: Timing diagram for three tasks under the LET assumption.

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 5. The LET assumption is modeled by the value of the `wait` statement in line 8 (this value is the duration of the LET delay). The result of the operation is read in line 11 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 12) is fired. So, by adding this pattern for each task activation and termination, a LET scheduling analysis can be done.

```
1    task t (a1 : ty1) : rty is  c_foo
2   [...]
3   process pExample is
4   [...]
5     start t (argOp);/* activate the task */
6     to wait_deadline
7   from wait_deadline
8     wait[$LET, $LET]; /* LET delay */
9     to read_result
10  from read_result
11    sync t res; /* synchronize the end of the task with the LET delay */
12    to next
13  [...]
```

Listing 6: Pattern to model a task `t` with a Logical Execution Time.

4.5 Implementation validation

The translation from H-FIACRE to the C code executable has not been formally proved 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 the 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 analyser 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.

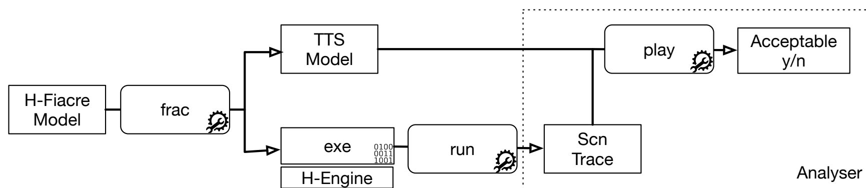


Figure 6: Scheme of the acceptance test for traces.

4.6 Experimental measures

This section presents the performances of HIPPO for different sizes models. We have provided two kinds of measurements, which are the CPU usage and the time spent in the HIPPO engine. The experiments were made in a computer with an Intel i5-8265U (1.6GHz) 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 the LTTng, an open source tracing framework for Linux [The LTTng Project].

4.6.1 Benchmark

We propose a set of synthetic benchmarks used in our experiments. (We give an example of a more realistic application in Sect. 5.) 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 15) by its periodic clock (lines 3 to 10), process p0 calls its task (t0) a first time (line 18), then waits for the return value (line 20). 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
4     states a, b, o
5     from o
6       wait [0,0];  to a /* offset */
7     from a
8       wait [3,3]; to b /* period */
9     from b
10      S_1; to a
11
12 process p0 [S_1 : none] is
13    states a, b, c, d, e
14    var ret : nat := 0
15    from a
16      S_1; to b
17    from b
18      start t0 (1); to c /* start task t1*/
19    from c
20      sync t0 ret; to d /* wait end of task t1 */
21    from d
22      start t0 (1); to e
23    from e
24      sync t0 ret; to a
```

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

4.6.2 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 millisecond HIPPO updates the running model and goes on.

For an example that correspond 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 Figure 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 Figure 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 than 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.

4.6.3 Time consumption of the TTS engine

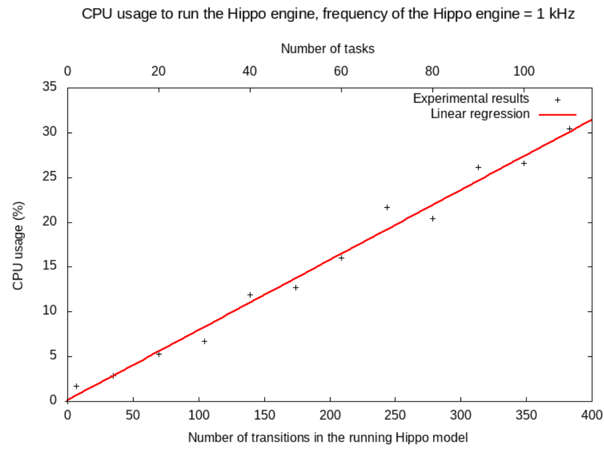
For our next experiment, we look at the "time consumption" of HIPPO by studying the time spent in one turn of the engine. Like in the previous experiments, we set the frequency to 1 kHz; meaning that the global tick (one turn of the engine) is at 1 millisecond. So, to successfully run the model, HIPPO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

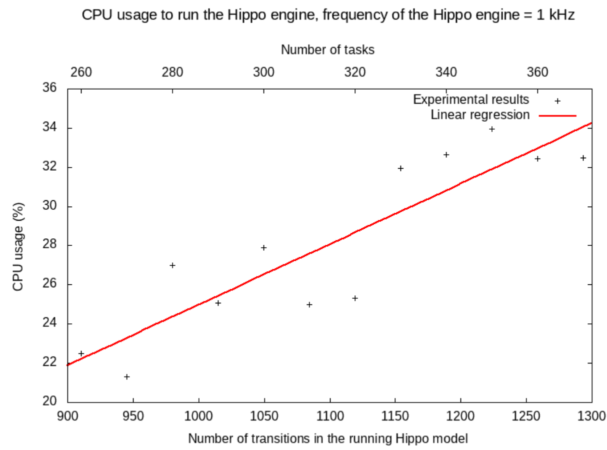
Figure 8 shows the distribution of the time spent in one turn of the HIPPO engine for three models : a small model with 40 HIPPO tasks (20 processes, 2 tasks per process, 140 transitions); an intermediate model with 400 HIPPO tasks (100 processes, 4 tasks per process,1100 transitions); and another one 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 interquatile 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 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 particularly huge and is not representative of a real application



(a) Without TurboBoost



(b) With TurboBoost

Figure 7: CPU usage as a function of number of Hippo tasks.

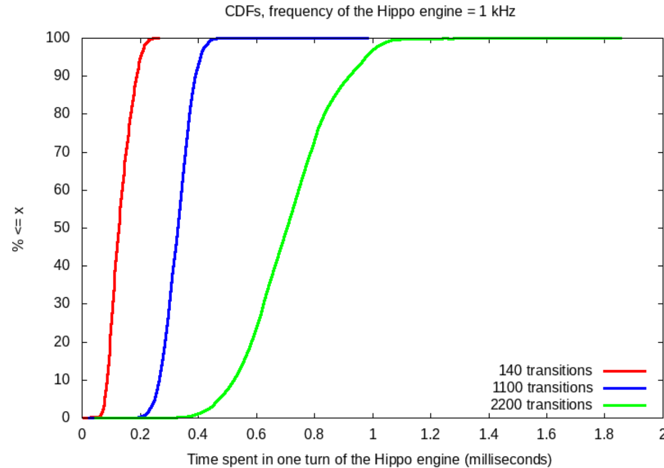


Figure 8: Time spent in the HIPPO engine

(see the use case in Sect. 5). 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 (that is not the case for benchmark's models).

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.

5 Case Study: A Software Controller for the Mobile Robot Minnie

FIACRE is a (French) acronym that stands for "Intermediate Format for Distributed, Embedded Component Architectures". As an intermediary language, its goal is 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 use G^{en}oM to generate an H-FIACRE model, and then HIPPO to produce the runtime controller software of a real robot called Minnie. This controller is also used to perform runtime monitoring of critical assertions and take appropriate corrective actions. G^{en}oM also synthesizes the regular FIACRE model to check several interesting properties on the system.

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

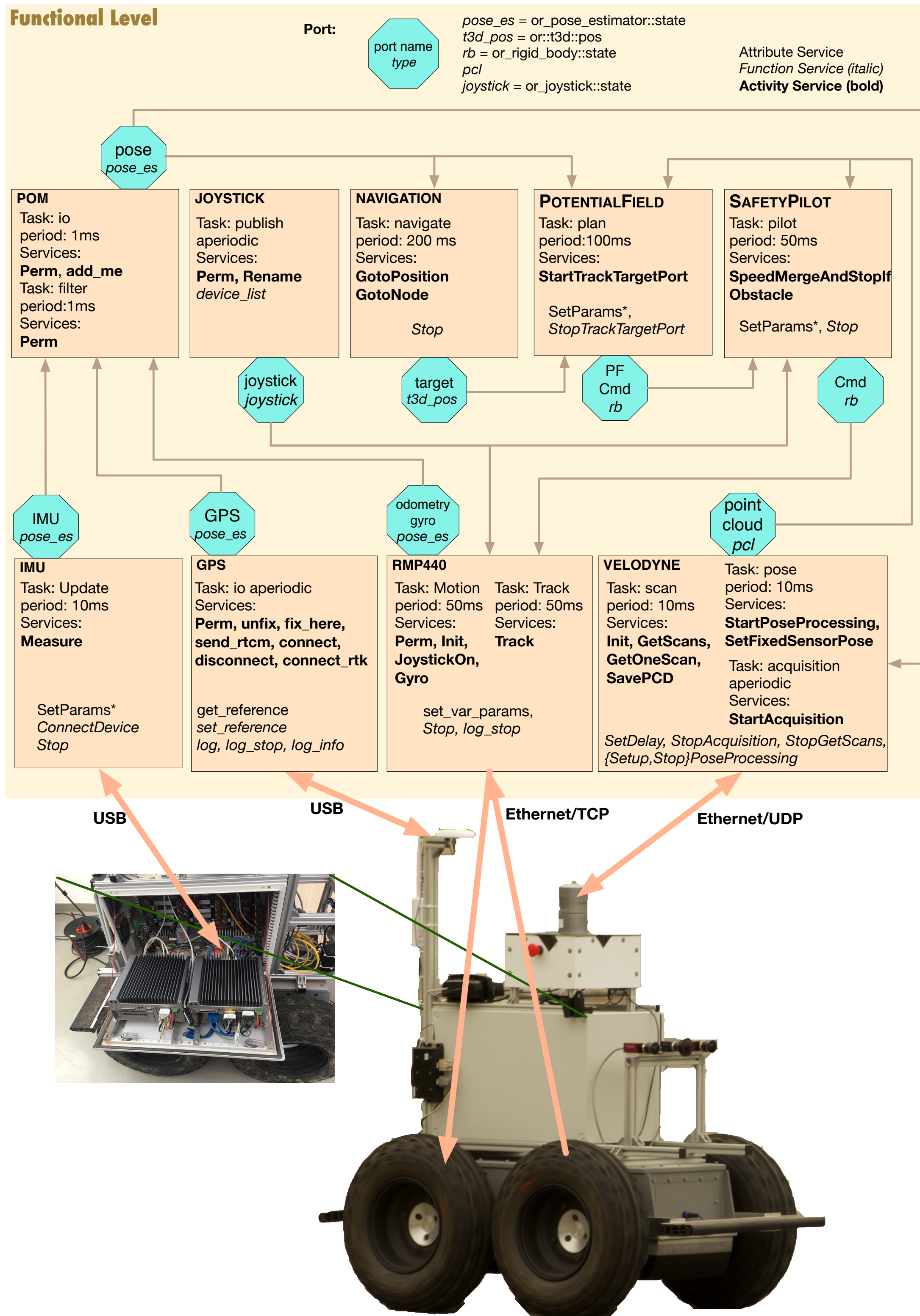


Figure 9: Architecture of the Minnie RMP440 experiment.

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). GenoM 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.

5.1 GenoM and the Minnie RMP440 robot

To illustrate how GenoM 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 an HDL-32E Velodyne lidar (on an ethernet UDP interface).

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 Gb RAM and a 256 Go SSD drive, running Ubuntu 18.04. In case of emergency, a human operator can take control of the robot using a wireless joystick 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 GenoM 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). 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

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

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.

The goal of this section is not to discuss the overall localisation/navigation implemented on Minnie, but to give a reasonable idea of the overall complexity entailed by a non-trivial robotic experiment. The complete code of the Minnie experiment is available at https://redmine.laas.fr/projects/minnie.

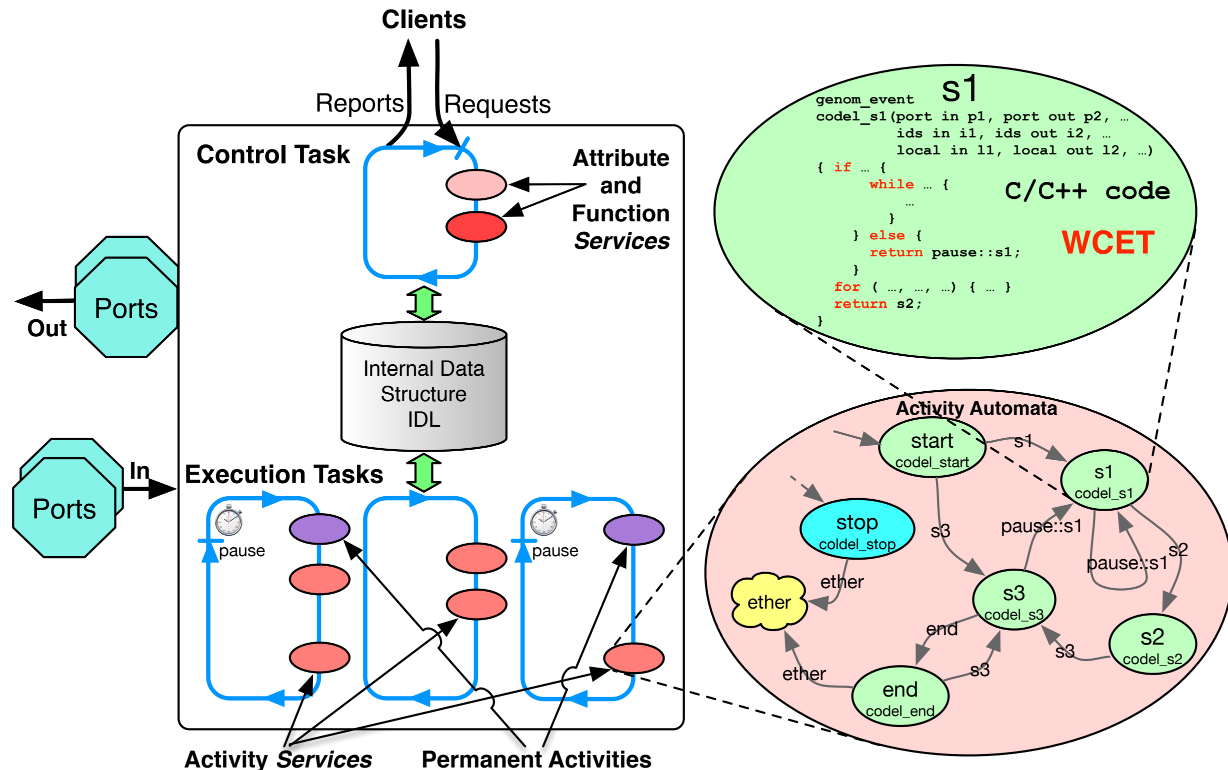


Figure 10: A G$^{en}_{o}$M generic functional component (module).

5.1.1 G$^{en}_{o}$M specification

All nine modules in Fig. 9 are an instance of a generic G$^{en}_{o}$M 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 runs *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 modelled 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 (yield) 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 19).

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.

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.

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.

```
activity GetScans(
     in double firstAngle = :"First angle of the scan (in degrees)",
     in double lastAngle  = :"Last angle of the scan (in degrees)",
     in double period     = :"Time in between two scans",
     in double timeout    = :"Timeout used when stamping packets")
{
 doc  "Acquire full scans from the velodyne sensor periodically";
 task scan;

 validate GetScansValidate(in firstAngle, in lastAngle, in period);

 codel <start> GetScansStart(in acquisition_params)
             yield copy_packets;
 codel <copy_packets> GetOneScanCopyPackets(in acquisition_params,
                 inout scan_buffer) // get packets from acquisition buffer
             yield stamp_packets;
 codel <stamp_packets> GetOneScanStampPackets(in acquisition_params, // stamp packets
                 inout pose_data, in timeout) // with the proper  pose
             yield pause::stamp_packets, build_scan; // pause:: if pose not available
 codel <build_scan> GetOneScanBuildScan(in acquisition_params,
                 in firstAngle, in lastAngle) // build scan repositioning
             yield end;              // individual packet in the first pose.
 codel <end> GetOneScanEnd(in acquisition_params,
                 port out point_cloud, inout usec_delay) //publish the scan in the
             yield wait;  // point_cloud port. usec_delay is for fault injection.
 codel <wait> GetScansWait(in period) // wait next user defined scan period
             yield pause::wait, copy_packets; // then loop back.

 interrupts GetOneScan, SavePCD, GetScans;
};
```

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

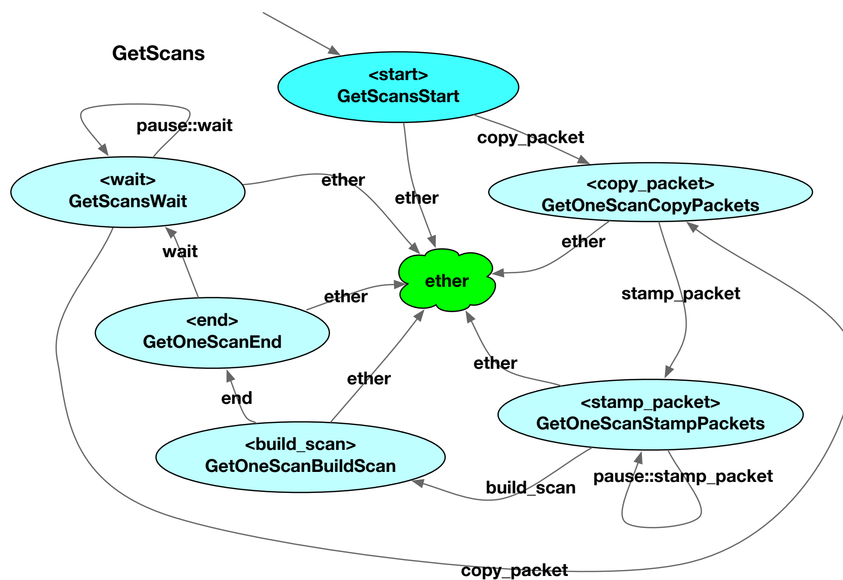


Figure 11: Finite-state machine of the *GetScans* activity (Listing 8)

5.1.2 GenoM templates

A GenoM template is a set of text files that include Tcl code, whose evaluation in the context of a GenoM 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 name of the services specified in the module (in which case the template file will just include a 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. Templates are the building blocks of any output of GenoM.

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.

A template, when called by GenoM on a given module specification, has access to all the information contained in the specification file such as services names and types, ports and IDS fields needed by each codel, execution tasks periods, etc. Through the template interpreter (using Tcl syntax), one specifies what they need the template to synthesize.

There are already templates to synthesize: the component implementation for various middleware (*e.g.* PocoLibs[3], ROS-Com [Quigley et al., 2009]); client libraries to control the component (*e.g.* JSON, C, OpenPRS), etc. Among the available middleware, we rather focus on PocoLibs as it is the most suitable for real-time applications (notably UAVs). Yet, its implementation, as efficient as it can be, cannot guarantee crucial properties, for this we need formal verification.

5.1.3 The GenoM toolchain for verification and code generation

The template mechanism used to synthesize the GenoM modules from their specifications and codels can also synthesizes 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), see Fig. 12. In fact, the Fiacre GenoM template file is really the same, we choose to synthesize one model or the other with a flag (`-tina` or `-hippo`) when calling the GenoM command on this template.

The time constraints used in the Fiacre model come both 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 profiling tool: `profundis`.

Our current template 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 services automata execution was not properly handled and lead to suboptimal reaction time. Another

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

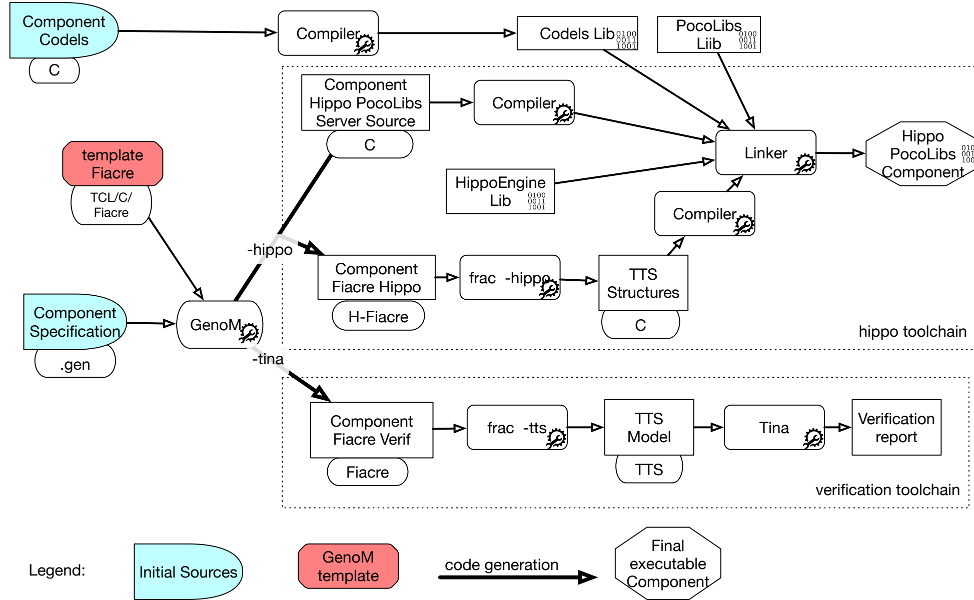


Figure 12: Toolchain with the Fiacre template (which can produce the `-tina` and the `-hippo` versions).

side-effect of this new template is that we can now derive better bounds on the reaction time of the system.

We also experimented with other V&V templates in previous works (transformations from G$^{en}_{o}$M to the input language of other real-time model-checkers), namely BIP [Abdellatif et al., 2012] 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 H-FIACRE modules 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.

In fact, the difference between the synthesized verification model and the HIPPO executable model are minimal:

- Codels execution is really carried on in the HIPPO models (with `start/sync`), but

is modeled with a time interval of [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 FIACRE tasks `start/sync` and tests in the HIPPO models, but with `select []* end` in the offline TINA model.
- The TINA model must include a *client* component to model the behavior of the environment (i.e. the requests received by the controller). On the opposite, the HIPPO model is simply "linked with the real world" using HIPPO *event ports* and task executions. In this case, the event port handles the mechanism which receives new requests (PocoLibs Mbox or the ROS CallBackQueue).

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 existing templates that directly target PocoLibs or ROS-Com server.

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., 2006] UPPAAL-SMC [David et al., 2015]	UPPAAL ++	N/A	N/A

Table 1: Existing formal framework templates for GenoM. The +, ++ and +++ correspond to our own subjective evaluation of the usability 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.

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.

5.2 Controlling and monitoring Minnie with HIPPO

We have been able to synthesize automatically an 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 1 760 transitions in the TTS.[4] It is linked with the codels library and HIPPO runs the whole experiment at 10kHz in one process. The load on the CPU remain acceptable, and no noticeable slowdown is observed (5-10% more than the sum of all regular G$^{en}_{o}$M components load).

The advantage of running HIPPO instead of the regular PocoLibs or ROS-Com server 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 than 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"[5].

We can also define additional monitors that go beyond these default properties. We give an example from the Minnie experiment in Listing 9. In this example, we monitor the time spent between two updates to port **point cloud** of the VELODYNE. If the port is not refreshed for more than 200 ms, 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.

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 actually 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 0 (which stops the robot very abruptly, without a regular deceleration). A careful examination of multiple traces shows that the robot typically stops 17–35 ms after detecting the problem, which is consistant with the TrackTask task period of 50 ms (so on average the `stopTrack` will be executed after 25 ms). Section 5.6 presents a more rigorous evaluation of this response time, using formal verification on the offline model.

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

[5] 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.

```
process Velodyne_Scans_rmp440_Track_Stopper(
      &scan_updated:bool, //boolean 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 elapsed
      scan_updated := false;
      to monitor_wait
   end

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

Listing 9: Example of user-defined monitor for module VELODYNE.

5.3 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 (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 need 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 states as a Boolean combination of atomic properties. Then the property can be checked on the fly with the *sift* tool. *sift* enumerates the reachable state 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.

5.4 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 a (Fiacre) task can never reach its `overshoot` state (this state is the same used in the Hippo version to detect overshoot at runtime, see Section 5.2). 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:

$$\neg(\mathsf{velodyne_scan_overshoot} \vee \mathsf{velodyne_pose_overshoot})$$

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 a scheduling errors when using only 3 cores with velodyne. This led us to change and optimise the codels for the velodyne to solve the problem.

5.5 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).

Scenario	*JoystickOn* then *Track*	*Track* then *JoystickOn*
Time	16 min.	10 h.
#classes	4,2714,945	832,778,752
#markings	5,817,082	44,533,432

Table 2: Complexity of checking mutual exclusion between services

5.6 Delay to stop

The last property we check is related to the HIPPO monitor presented in Sect. 5.2, Listing 9. 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 10 gives a 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 17) is less or equal to the WCRT. Hence, to compute the right value, it is enough to try different values for the timeout.

```
1  process rmp440_Track_Stopper(&track_started:bool, &track_stopped:bool,
2         &TrackTask_activities: Activities_rmp440_TrackTask_Array,
3         Track_index: act_inst_rmp440_TrackTask_index_type) is
4
5  states wait_started, wait_stop, wait_delay, finished, robot_stopped, robot_NOT_stopped
6
7  from wait_started
8      wait [0,0];
9      on (track_started); // wait the Track service has started
10     to wait_stop
11
12 from wait_stop  // (no wait) can stop anytime
13     TrackTask_activities[Track_index].stop := true;
14     to wait_delay
15
16 from wait_delay
17    wait [141,141]; //<--- This is the response time value we want to measure
18    to finished
19
20 from finished
21    wait [0,0];
22    if (track_stopped)  then
23       to robot_stopped  //The robot has been stopped before the delay
24    else
25       to robot_NOT_stopped //The robot has not been fully stopped yet
26    end
```

Listing 10: 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).

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 $6\,\mathrm{m\,s^{-1}}$, 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).

6 Conclusion

We describe a language and a compiler, called HIPPO, able to generate executable code that has the same semantics than 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 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 GenoM 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 GenoM 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.

The performance of the applications generated with HIPPO is also evaluated and the overhead seems reasonable for a real usage.

We have several ideas for future works. For example, we discuss the problem of schedulability analysis, which should be further explored in subsequent works. It would also be interesting to port our engine on embedded and low-resource systems, particularly on micro-controller architectures. Another interesting research direction will be to add more

support for expressing user-level properties; for example towards runtime verification by integrating a property language in our toolchain. We also want to add support for defining runtime monitors that could run alongside an application.

Acknowledgements

This work has been supported by the European Union's Horizon 2020 research and innovation programme under grant agreement No. 825619 (AI4EU).

The authors would like to thank Dr. Bernard Berthomieu and Dr. Mohamed Oussama Ben Salem for their careful readings and insightful suggestions that helped to improve the quality of this paper.

A 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 6) with two shared data (see Listing 11 for an example). The `ready_list` (line 36) 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 37) 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 21) and a test is done to know if a processor is available to execute the new task (line 9). If it is possible, the `launch` is updated (line 11). Then, the task waits that a processor is available to continue its execution by checking the status of `launch` (line 23). When a task terminates its execution, a processor is free (line 27) and if the `ready_list` queue is not empty the first task is resumed by changing its status in `launch` (line 9).

Remark that we use the best-case execution time and the worst-case execution to represent the execution time of the task (line 26). The translation from a Hippo model to a Fiacre model with a FIFO scheduler is implemented and available on the 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 (&ready_list: fifo, &launch: start_tab, &unused_proc: nat) is
7   states exec
8   from exec
9     on (not (empty ready_list)) and (unused_proc > 0);
10    unused_proc:= unused_proc-1;
11    launch [first ready_list]:= true;
12    ready_list := dequeue ready_list;
13    wait [0,0]; to exec
14
15 process p_task [SyncG : none, t_a : ty1, t_t :  tyOut] (id : 0..nbOfTasks-1, &
      unused_proc : nat, &ready_list : fifo, &launch : start_tab) is
16   states waiting, sched_activate, sched_resume, running, sched_terminate, synchronizing
      , terminating
17   var param : tyIn, ret : tyOut
18
19   from waiting
20       t_a?param; to sched_activate
21   from sched_activate /* task activation: scheduler call */
22     ready_list := enqueue(ready_list, id); wait [0,0]; to sched_resume
23   from sched_resume
24     on launch[id]; launch[id] := false; wait [0,0];  to running
25   from running
26     ret := c_foo(param); wait [$bcet, $wcet]; to sched_terminate
27   from sched_terminate  /* task termination: scheduler call */
28     unused_proc := unused_proc + 1; wait [0,0]; to synchronizing
29   from synchronizing
30     SyncG;  to terminating
31   from terminating
32     t_t! ret; to waiting
33 ...
34 component Main is
35   var
36     ready_list : fifo := {||},
37     launch : start_tab := [false,..., false],
38     unused_proc : nat := nbOfProcessors
39   ...
40   par
41   ...
42     ||  scheduler (&ready_list, &launch, &unused_proc)
43     ||  p_task [SG, t_a , t_t](1, &unused_proc, &ready_list, &launch)
44   ...
45   end
```

Listing 11: An example of FIFO scheduler in Fiacre.

References

T. Abdellatif, J. Combaz, and J. Sifakis. Model-Based Implementation of Real-Time Applications. In *International Conference on Embedded Software*, 2010. URL http://dl.acm.org/citation.cfm?id=1879052.

T. Abdellatif, S. Bensalem, J. Combaz, L. De Silva, and F. Ingrand. Rigorous design of robot software: A formal component-based approach. *Robotics and Autonomous Systems*, 60(12), 2012. doi: 10.1016/j.robot.2012.09.005.

T. Abdellatif, J. Combaz, and J. Sifakis. Rigorous implementation of real-time systems – from theory to application. *Mathematical Structures in Computer Science*, 23(4), 2013. doi: 10.1017/S096012951200028X.

T. Amnell, E. Fersman, P. Pettersson, H. Sun, and W. Yi. Code synthesis for timed automata. *Nord. J. Comput.*, 9(4), 2002. URL https://dl.acm.org/doi/10.5555/779110.779112.

G. Behrmann, A. David, and K. G. Larsen. A Tutorial on Uppaal 4.0. Technical report, Department of Computer Science, Aalborg University, Denmark, Nov. 2006. URL https://www.it.uu.se/research/group/darts/papers/texts/new-tutorial.pdf.

S. Ben Rayana, M. Bozga, S. Bensalem, and J. Combaz. RTD-Finder - A Tool for Compositional Verification of Real-Time Component-Based Systems. In *TACAS*, 2016. URL http://link.springer.com/chapter/10.1007/978-3-662-49674-9_23.

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

B. Berthomieu, P.-O. Ribet, and F. Vernadat. The tool TINA – construction of abstract state spaces for Petri nets and time Petri nets. *International Journal of Production Research*, 42(14), 2004. doi: 10.1080/00207540412331312688.

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

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

B. Berthomieu, S. Dal Zilio, and F. Vernadat. A fiacre v3.0 primer, 2020. URL http://projects.laas.fr/fiacre/doc/primer.pdf.

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

J. A. Carruth and J. Misra. Proof of a real-time mutual-exclusion algorithm. *Parallel Processing Letters*, 6(2), 1996. doi: 10.1142/S012962649600025X.

P. Caspi, A. Curic, A. Maignan, C. Sofronis, S. Tripakis, and P. Niebert. 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*, 2003. doi: 10.1145/780732.780754.

S. Dal Zilio, B. Berthomieu, and D. Le Botlan. Latency Analysis of an Aerial Video Tracking System Using Fiacre and Tina. In *FMTV verification challenge of WATERS 2015*. LAAS-VERTICS, Sept. 2015. URL http://arxiv.org/abs/1509.06506v1.

A. David, K. G. Larsen, A. Legay, M. Mikučionis, and D. B. Poulsen. UPPAAL SMC tutorial. *International Journal on Software Tools for Technology Transfer*, pages 1–19, Apr. 2015. doi: 10.1007/s10009-014-0361-y.

M. Dellabani. *Formal methods for distributed real-time systems*. PhD thesis, Université Grenoble Alpes, 2018.

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

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

P. Feiler, D. Gluch, and J. Hudak. The architecture analysis & design language (aadl): An introduction. 2006. doi: 10.21236/ada455842.

M. Foughali. *Formal Verification of the Functional Layer of Robotic and Autonomous Systems*. PhD thesis, LAAS/CNRS, Dec. 2018.

M. Foughali, B. Berthomieu, S. Dal Zilio, P.-E. Hladik, F. Ingrand, and A. Mallet. 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)*, 2018. doi: 10.1145/3193992.3193996.

M. Foughali, F. Ingrand, and C. Seceleanu. Statistical Model Checking of Complex Robotic Systems. In *International SPIN Symposium on Model Checking of Software*, 2019.

M. Guerra, D. Efimov, G. Zheng, and W. Perruquetti. Avoiding local minima in the potential field method using input-to-state stability. *Control Engineering Practice*, 55 (C), Oct. 2016. doi: 10.1016/j.conengprac.2016.07.008.

N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language Lustre. *Proceedings of the IEEE*, 79(9), 1991. doi: 10.1109/5.97300.

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

T. Henzinger and C. Kirsch. The embedded machine : Predictable, portable real- time code. *ACM Transactions on Programming Languages and Systems*, 29(6), 2007. doi: 1286821.1286824.

T. Henzinger, B. Horowitz, and C. Kirsch. Giotto: a time-triggered language for embedded programming. *Proceedings of the IEEE*, 91(1), 2003. doi: 10.1109/JPROC.2002.805825.

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

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

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

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

J. Kristensen, A. Mejlhilm, and S. Pedersen. Automatic translation from uppaal to C. Technical report, Department of Computer Science, Aalborg University, 2017.

G. Lasnier, B. Zalila, L. Pautet, and J. Hugues. 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*, 2009. doi: 10.1007/978-3-642-01924-1_17.

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

J. Liu and E. Lee. Timed multitasking for real-time embedded software. *IEEE Control Systems Magazine*, 23, 2002. doi: 10.1109/MCS.2003.1172830.

S. Louise, M. Lemerre, C. Aussagues, and V. David. 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)*, 2011. doi: 10.1109/HASE.2011.38.

A. Mallet, C. Pasteur, M. Herrb, S. Lemaignan, and F. Ingrand. GenoM3: Building middleware-independent robotic components. In *Proc. of the IEEE International Conference on Robotics and Automation (ICRA)*, 2010. doi: 10.1109/ROBOT.2010.5509539.

N. Navet, L. Fejoz, L. Havet, and A. Sebastian. 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)*, 2016. URL https://hal.archives-ouvertes.fr/hal-01289494/.

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

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

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

F. Singhoff, J. Legrand, L. Nana, and L. Marcé. 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)*, 2004. doi: 10.1145/1032297.1032298.

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

J. Tanguy, J.-L. Béchennec, M. Briday, and O. H. Roux. 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)*, 2014. doi: 10.5220/0005040101630169.

The LTTng Project. LTTng website (accessed: 27.10.2020). URL https://lttng.org.