A formal toolchain for offline and run-time verification of robotic systems
Silvano Dal Zilio, Pierre-Emmanuel Hladik, Félix Ingrand, Anthony Mallet

To cite this version:

HAL Id: hal-03683044
https://hal.laas.fr/hal-03683044v2
Submitted on 17 Oct 2022

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.
A formal toolchain for offline and run-time verification of robotic systems*

Silvano Dal Zilio¹, Pierre-Emmanuel Hladik², Félix Ingrand¹ and Anthony Mallet¹†
¹LAAS-CNRS, Université de Toulouse, CNRS, INSA, Toulouse, France
²Nantes Université, École Centrale Nantes, CNRS, LS2N, Nantes, France

Abstract

Validation and Verification (V&V) of autonomous robotic system software is becoming a critical issue. Among the V&V techniques at our disposal, formal approaches are among the most rigorous and trustworthy ones. Yet, the level of skills and knowledge required to use and deploy formal methods is usually quite high and rare. In this paper, we describe an approach that starts from a regular, but rigorous, framework to specify and deploy robotic software components, which can also automatically synthesize a formal model of these components.

We describe how we can execute the resulting formal model, in place of a traditional implementation, and show how this provides the opportunity to add powerful monitoring and runtime verification capabilities to a system, e.g., to prevent collisions, or trigger an emergency landing. Since the runtime used to execute formal models is specifically designed to be faithful to their semantics, every execution (in the implementation) can be mapped to a trace in the specification. As a result, we can also prove many interesting properties offline, using model-checking techniques. We give several examples, such as properties about schedulability, worst-case traversal time, or mutual exclusion.

We believe that having a consistent workflow, from an initial specification of our system, down to a formal, executable specification is a major advance in robotics and opens the way for verification of functional components of autonomous robots and beyond. We illustrate this claim by describing a complete example based on a genuine drone flight controller.

*There is an extended version of this paper (including the Minnie RMP440 experiment) available here: https://www.dropbox.com/s/aivkpbl1v8xcmw45/ras_2022_extended.pdf.

This work has been supported by the Artificial and Natural Intelligence Toulouse Institute - Institut 3iA (ANITI) under grant agreement No: ANR-19-PI3A-0004 and by the European Union’s Horizon 2020 research and innovation programme under grant agreement No. 101016442 (AIPlan4EU).

†Authors in alphabetical order.
Contents

1 Introduction and motivation 3

2 State of the art and proposed approach 3
   2.1 Relevant surveys, general formalisms and approaches 4
   2.2 ROS related approaches 5
   2.3 Decisional Functionalities, Acting and Skills 6
   2.4 Specific formal framework for robotic systems 6
   2.5 Domain Specific Language (DSL) leaning toward formal specification 7
   2.6 Proposed approach and background work 7

3 A framework for robotic functional components specification: GenoM 8
   3.1 Overview 9
   3.2 Specification language 11
   3.3 Codels definition 15
   3.4 Main Algorithms 15
   3.5 Executable Components Synthesis 17
   3.6 GenoM template mechanism 19

4 A formal framework for offline and run-time verification: FIACRE 20
   4.1 A common formal language: The FIACRE Language 20
   4.2 Hippo an engine to execute H-FIACRE model 24
   4.3 FIACRE offline verification tools suite 24

5 A GenoM template to synthesize a FIACRE model 26
   5.1 The FIACRE/H-FIACRE GenoM toolchain 26
   5.2 Mapping the GenoM internal algorithms in FIACRE and H-FIACRE 27
   5.3 Offline verification 28
   5.4 Runtime verification 30

6 An example: a UAV controller 30
   6.1 A quadcopter flight controller 30
   6.2 Offline verification 32
      6.2.1 Schedulability of components 32
      6.2.2 Maximum bounded time between events 33
      6.2.3 Proper services exclusion 33
      6.2.4 State explosion problem 34
   6.3 Runtime verification and monitoring 35

7 Discussion, Future work and Conclusion 36

A Synthesized FIACRE models 45
1 Introduction and motivation

A large part of the development of Autonomous Robots is dedicated to the software which controls these systems. With robots now present in many aspects of our daily life (e.g., drones, autonomous cars, driverless shuttles) one can easily foresee what could be the consequence of an unsatisfied safety specification in the software. These software applications are usually organized along an architecture, and developed with specific frameworks, Domain Specific Languages (DSL), and middleware. Being able to validate and verify parts of these control software is now becoming critical to gain confidence in the system and to enable certification. In this context, deploying formal methods for verification would be of significant help.

On one hand, roboticists already have experience with formal methods when it comes to high-level models, such as those used in task planning for example. Apart from components resulting from deep learning methods, many decisional functionalities [Ingrand and Ghallab, 2017] are implemented in components deploying some kind of models (planning models, diagnosis models, etc.) and rely on some automatic search to find a solution. On the other hand, most of the functional components of the robots in charge of sensors and effectors, as well as the low-level processing for localization, trajectory control, motion planning, etc. are seldom developed with formal models. Often, they rely on a common middleware and some DSL which eases their integration, but their implementation remains mostly unverified, except from being tested in some situations.

There are some tools and frameworks which by design provide formal verification (we give some examples in the state of the art section), but they often remain too complex to be used by robot programmers. In this paper, we propose to rely on an existing robotic DSL, called GenoM [Mallet et al., 2010], developed by some of the authors, that is used for the specification and deployment of robotic functional components. On the other end of our toolchain, we rely on a formal framework for time-critical systems built around the FIACRE [Berthomieu et al., 2008] specification language. We show that we can automatically synthesize the formal model from the original robotic specification. This formal model can then be used to verify (offline) formal properties of the implementation and to monitor and enforce (online) safety properties.

The paper is organized as follows. Section 2 presents a state of the art in using formal verification focused on robotic applications. We then introduce the GenoM framework (section 3) with its specification language along its algorithms, illustrated on a simple robotic application, and its template mechanism which enables code synthesis for the component final implementation. In section 4, we briefly present the formal framework on which our approach relies, followed by section 5 which describes how the principal algorithm implementing a GenoM component get translated to FIACRE, and what type of offline and online verification can be done. In section 6, we illustrate our approach on a real-world complex and complete example, a drone flight controller with stringent real-time requirements, and we show how we can prove offline some safety properties and monitor more complex ones online. We discuss our approach, its limits, possible future works and conclude in section 7.

2 State of the art and proposed approach

There are already numerous surveys related to the formal validation and verification of autonomous systems. We try to focus on robotic systems, for which an implementation on a real robot has been done, possibly with feedback from researchers in robotics. Moreover, robots are complex systems with many constraints and feature diverse programs of diverse kinds (e.g., control, decisional, learning). We favor approaches which address this richness, without oversimplifying the problem.
The following subsections may appear somehow unrelated, but our goal here is to shed some light on various aspects of the state of the art which we believe are of interest to roboticists and still relevant to the V&V community. We start with section 2.1 which introduces formal V&V with a quick overview of surveys and approaches. Section 2.2 is dedicated to ROS, by far the most popular software development framework in robotics, and the numerous works proposing some software V&V. Section 2.3 step back from the low-level robotic software components and survey some V&V issues related to decisional components. In section 2.4 we present some formal frameworks which have been explicitly designed or modified to address robotic applications, while section 2.5 considers robotic DSL which include some formalisms usable in a V&V context. Last, we conclude this section with our proposed approach (section 2.6).

2.1 Relevant surveys, general formalisms and approaches

Until recently, most surveys in using formal methods for V&V of software [D’Silva et al., 2008, Woodcock et al., 2009, Bjørner and Havelund, 2014] were general and did not even mention robots or autonomous systems. This has changed in the last few years, and there is now an increasing number of works related to formal validation and verification of autonomous robots, e.g., drones, autonomous cars, humanoid robots. This is probably a sign of a growing interest in this field as well as a need to sort out and map the large number of papers and research topics.

The authors in [Luckcuck et al., 2019] make an impressive survey of formal specification and verification approaches (model-checking, theorem proving, runtime verification and formal synthesis) for autonomous robots. In a follow-up article [Luckcuck, 2021], one of the authors proposes some general recipes that can be applied during the design of such systems: verifying executive decisions; controlling machine learning; verifying and enforcing safety claims; verifying tasks and missions; and logical barriers. These recipes do not all translate in the proposed formalism, yet they clarify where the efforts should be focused. In [Fisher et al., 2021] the authors address the certification aspect, which may involve formal models and approaches. Still the survey is quite extensive and is illustrated with seven use cases. They analyze the situation by considering autonomous systems along a three layers architecture (reactions, rules and principles) which has some similarity with the classical three layers robotic architecture [Bonasso et al., 1998, Alami et al., 1998, Kortenkamp and Simmons, 2008].

Formalisms  Most of the surveys above agree that different types of formalism can be deployed in robotics. Some models are grounded in simple yet powerful primitives. Automata and state machines [Bohren and Cousins, 2010, Verma et al., 2006, Li et al., 2018] are often put forward as they easily capture the various states of the subsystems and their transitions. Petri nets [Costelha and Lima, 2012, Lesire and Pommereau, 2018] are also often used, as they easily model coordination, together with their time extension, e.g., time Petri nets [Berthomieu and Diaz, 1991]. Time is also at the core of Timed Automata [Alur and Dill, 1994], a well-known formal model on which tools like UPPAAL [Bengtsson et al., 1995] and Kronos [Yovine, 1997] are based, and that is also used in the real-time implementation of the BIP framework [Socci et al., 2013]. Other models are provided as languages defined at a higher level of abstraction, such as synchronous languages, but can be translated to mathematical or logical representations. We can also mention other formalisms such as temporal logic [Kress-Gazit et al., 2011], situation calculus [Levesque et al., 1997, Claßen et al., 2012], as well as interval temporal logic [Allen, 1981], which have all been employed in use cases in robotics. Finally, we should not forget methods often related to control theory and geared towards hybrid systems [Tomlin et al., 2003].
**V&V Approaches**  Among the various techniques available to the V&V community, the most popular ones found in robotic application fall into three main categories:

*State exploration and model checking.* These approaches [Clarke, 2008], given an initial state and a transition function, make an offline exploration of the reachable states of the system. It is sometimes possible to use a symbolic approach to explore the search space without the need to explicitly enumerate all the states, but most approaches suffer from a state explosion problem and often face scalability issues. A related but incomplete approach is *Statistical model checking* (SMC) which samples the reachable state using a probability model of the transition function, and thus evaluates properties to be verified with a resulting probability.

*Logical inference.* These approaches [Bensalem et al., 2010] are also offline, but work by building an over approximation of the set of reachable states as a logical statement (e.g., obtained by combining invariants of components). These approaches can potentially address the state explosion problem, but they also have their own limitations. For instance, the logical invariants they build can be too general, or too loosely fit, to check the property we are interested in.

*Runtime verification.* While the two previous approaches are offline and are mostly used at design and specification time, *Runtime verification* [Leucker and Schallhart, 2009] is an online approach where the state transition model is given to an engine that monitors and checks properties and model’s consistency on the fly. This approach requires specification of what needs to be done when a property is violated, but enables verification of models that would not scale, nor fit, with the previous approaches.

### 2.2 ROS related approaches

ROS [Quigley et al., 2009] is the most popular framework to develop and deploy robotic applications and software. It relies on a simple yet powerful mechanism to allow programs (called *nodes*) to exchange and share data (*topics*) through a publish-subscribe mechanism. Therefore, whoever develops a method that can support ROS communication mechanisms and primitives can apply it to thousands of robotic applications at once. Indeed, there are many works which rely on this topic publish-subscribe mechanism. ROSRV [Huang et al., 2014] is inserted in between publishers and subscribers, and checks the data exchanged. ROS Monitoring [Ferrando et al., 2020] improves ROSRV design by remapping ROS topics. Moreover, it can also be used offline (by checking runtime logs) and uses an oracle to check properties expressed in various temporal logic. HAROS (High Assurance ROS) [Santos et al., 2021] mostly does static code analysis from which it can extract run-time architecture constructs (check the publish-subscribe). Moreover, they introduce a plugin mechanism and list some plugins (e.g., to do model checking) which can be used within HAROS. Declarative Robot Safety (DeRoS) is a DSL [Sorin et al., 2016] to describe monitors which implement safety rules based on ROS topics values. RTAMT4ROS [Ničković and Yamaguchi, 2020] is about integrating Signal Temporal Logic within ROS. SOTER [Shivakumar et al., 2020] introduces, on top of ROS, an event-based runtime assurance monitoring framework programmed in P [Desai et al., 2013] (which is closer to a skill language).

Overall, most of these approaches rely on the ROS node external interface: topics, services and actions. They spy, study, monitor and evaluate the topics values exchanged between nodes, but the nodes themselves remain black boxes. Indeed, ROS itself does not require the programmer to specify how a node should be implemented. Yet, there are some efforts to formally model the ROS communication layer itself [Halder et al., 2017], or to verify some simple properties [Come et al., 2018, Meng

---

1It also provides *services* (Remote Process Call between nodes) and *actions* (for more complex, asynchronous service calls).
et al., 2015, Wong and Kress-Gazit, 2017], but overall, the lack of ROS language to further specify nodes beyond topics, services and actions makes it rather difficult to verify anything.

A solution to this problem may come from higher level specification languages, which map to ROS communication mechanism. In [Bardaro et al., 2018], the authors propose to model the robot software in AADL and then synthesize code in ROS. Another proposal is described in [Kai et al., 2017], where the authors describe a dedicated Architecture Description Language for robotics, called MontiArcAutomaton, together with a model-driven engineering approach for generating ROS code. Due to the lack of ROS formalism, these approaches introduce a semantic gap, i.e., there may be a difference in semantics between the behavior described by the high-level models and the final execution on the system.

2.3 Decisional Functionalities, Acting and Skills

If we consider decisional functionalities onboard a robot, planning is deployed with models of how actions can be used to reach a goal state while acting uses declarative operational models (skills) of how to execute an action.

Most embedded task planning approaches rely on models (Planning Description Language (PDDL) [Ghallab et al., 1998], New Domain Description Language (NDDL) [Frank and Jónsson, 2003], Action Notation Modeling Language (ANML) [Smith et al., 2008], chronicle [Ghallab, 1996], etc.) and the way they solve the problem has strong similarities with model checking (exploring the reachable states of the system in search of a goal state) [Bensalem et al., 2014] or constraints satisfaction (finding consistent values of state variables with respect to given constraints).

Unlike planning, the verification of acting models usually requires more work. We can take the example of the ROS framework, where the control of ROS nodes can be delegated to technologies such as hierarchical concurrent state machines: SMACH [Bohren and Cousins, 2010] or Behavior Trees [Marzinotto et al., 2014]. Even if these two acting frameworks do not directly map to a formal model, they point to the right direction to study them. We can find many more examples of verification for acting models in the literature: in [Kovalchuk et al., 2021] the authors consider probabilistic time automata (UPPAAL SMC) to model acting skills; ASPiC [Lesire and Pommereau, 2018] is an acting system based on the composition of skill Petri nets and behavior trees [Albore et al., 2021]; NuSMV has been used in [Simmons and Pecheur, 2000] to check Task Description Language (TDL) based acting components; RML [Williams and Ingham, 2003] and Proteus [McClelland et al., 2021] have been used for acting and monitoring; RAFCON [Brunner et al., 2016] provides mechanisms to model and execute plans produced by high-level planners. Finally, several works have studied the use of the logic-based action language Golog [Levesque et al., 1997, Claßen et al., 2012, Eckstein and Steinbauer, 2020], and the associated execution system Golex [Hähnel et al., 1998], as a newer acting framework.

2.4 Specific formal framework for robotic systems

Another class of work focuses on specific formalisms which by design allow to formally specify a robotic system. Some general formalisms, such as the “synchronous approach” [Benveniste and Berry, 1991, Benveniste et al., 2003] have been instantiated in several languages (e.g., Lustre, Esterel, Signal) and commercial frameworks. ORCCAD [Simon et al., 2006] proposes to deploy robotic systems programmed in the Maestro language then translated to Esterel [Boussinot and de Simone, 1991]. This framework has been used to successfully program a complete AUV (Autonomous Underwater
Vehicle) and prove some properties on the system. Yet, the Maestro language was probably too abstract for robotic system programmers and remained seldom used.

RoboChart [Cavalcanti, 2017, Miyazawa et al., 2017, Ribeiro et al., 2017, Cavalcanti et al., 2021] proposes an interesting approach where the programmer explicitly models the robotic application in a formal framework based on timed communicating sequential processes. The model is provided by the programmer who must have some familiarity with formal models and languages.

In [Figat and Zieliński, 2022, 2020], the authors propose a six layers architecture with a meta-model using hierarchical Petri net. A robotic system can then be specified and modeled with proper parametrization of the metamodel. The resulting hierarchical Petri net can then be used to check formal properties and synthesize controllers.

Overall all these frameworks are rather powerful and convincing when it comes to formal verification, yet, their adoption by the robotics community remains a serious challenge, and one could fear that the entry ticket to master these frameworks may be too high for regular robot programmers.

2.5 Domain Specific Language (DSL) leaning toward formal specification

There exist numerous frameworks to develop and deploy robotic software [Nordmann et al., 2016, Brugali, 2015]. Some offer specification languages, or rely on well-known specification frameworks such as the UML profile MARTE [Brugali, 2018]. Some others, as in [Brugali, 2021], just provide tools and API libraries to ease integration of different components or, as OROCOS [Bruyninckx, 2001], focuses on real-time control of robots. In [Dhouib et al., 2012, Yakymets et al., 2013], a robotic DSL, RobotML, associated to the platform Papyrus (an Eclipse UML tool) is presented to facilitate the development of robotic applications. Another example is Smartsoft [Schlegel et al., 2009, Schlegel, 2021], which provides a framework to also specify the complete architecture of a robotic system, while [Lotz et al., 2016] provides a metamodel to separate user programmer concerns and system integrators issues. These tools greatly ease the overall architecture and analysis of the system, but they seldom connect to a formal framework supporting V&V.

In a similar way, MAUVE [Gobillot et al., 2016, Doose et al., 2017] is an interesting framework that extracts temporal information from runs to verify behavior with an Linear Temporal Logic (LTL) checker, and performs runtime verification of temporal properties. Similarly, Drona [Desai et al., 2017] allows model-checking and runtime verification using Signal Temporal Logic.

2.6 Proposed approach and background work

In the survey [Bjørner and Havelund, 2014], the authors write:

“We will argue that we are moving towards a point of singularity, where specification and programming will be done within the same language and verification tooling framework. This will help break down the barrier for programmers to write specifications.”

Similarly, in [Nordmann et al., 2016] the authors survey robotic DSL, and they argue that:

“both communities should foster collaboration in order to make formal methods more practicable and accepted in robotic software development and to make DS(M)L approaches more well-founded in theory to foster work in the field of model validation and verification.”

Our approach follows the same philosophy. We advocate that one way to painlessly introduce formal V&V in robotic components is to rely on existing robotic specification languages and frameworks, and offer some automatic translation to formal models. For this, we need to ensure that the
Some earlier ideas pertaining to this work were initially introduced in [Foughali et al., 2016], a conference paper, too limited in space to present the overall approach in detail. As an early work, the synthesized formal model was more a proof of concept than an exact truthful implementation of the real components. The illustrated example of the paper was based on toy G\textsuperscript{en}\textsubscript{b}M components unable to control a real robot. Still the results, despite being limited and simplified, were encouraging enough to completely rewrite the model from scratch in 2019 with a code equivalence in mind and adding the runtime extensions to execute the complete model on the real platform. Similarly, we also had an implementation able to synthesize a simplified model in BIP, but the invariants extracted from our models by the verification tool RTD-Finder [Ben Rayana et al., 2016] remained too general to be used. The BIP Engine could run this partial model on a mobile robot [Foughali et al., 2021], but was too slow and incomplete to run a complex flight controller on a UAV (Unmanned Aerial Vehicle) at 10 kHz.

Still, parts of the work in this present paper have been introduced in [Ingrand, 2021, Hladik et al., 2021]. The former mostly focuses on the overall approach and the exploratory work we also did with BIP and UPPAAL. The latter focuses on the HIPPO engine, and the implementation of the FIACRE extension to make it an embedded language, able to react to events and execute external code. It also shows how we used the FIACRE template and its first implementation as a runtime verification framework for a mobile robot.

While [Hladik et al., 2021] focuses on the description of the formal languages and is intended to the runtime verification community, the present paper focuses more on the robotic software engineering aspects, with the objective of stressing the applicability of the overall approach for real applications and to show how properties can be verified offline with model checking or enforced online by a runtime controller. As such, it presents novel and unpublished contributions to: i) the G\textsuperscript{en}\textsubscript{b}M specification language, the semantics of the component implementation and the underlying algorithms along a simple yet realistic example, ii) the synthesis of a complete and trustful formal model of the implementation and iii) a complex example (an UAV flight controller) for which we perform runtime verification at 10 kHz.

With respect to the state of the art, we summarize our approach along the following original lines and features:

- We focus on the functional components of robotic systems.
- These components are fully specified with a DSL by robot programmers (not experts in formal models) and the component implementation is automatically synthesized from these specifications.
- The formal model is also automatically synthesized from these specifications and is consistent with the semantics of the component implementation.
- The synthesized formal model can be used both offline for verification with model-checking and online for runtime verification and properties enforcement.

### 3 A framework for robotic functional components specification: G\textsuperscript{en}\textsubscript{b}M

G\textsuperscript{en}\textsubscript{b}M started as a versatile “software engineering” tool [Fleury et al., 1997] and, thanks to its clear
3.1 Overview

Programming autonomous robotic systems is a complex task and often relies on architectural principles and tools. In most robotic software architectures, one distinguishes the decisional layer in charge of decisional functions (planning, acting, monitoring, etc.) [Ingrand and Ghallab, 2017], and the functional layer in charge of sensors, effectors and functional activities such as localization, mapping, motion planning, trajectory following, obstacle avoidance, etc. GENOM (GENerator Of Modules) is a tool to specify and implement modular software components for a robotic functional layer. Each component is a specialization of the generic architecture shown in Fig. 1. To illustrate it, we introduce the example on Fig. 2, along its complete specification in section 3.2.

![Figure 1: Architecture of a generic GENOM functional component.](image-url)

As shown on Fig. 1, a component is an executable program containing the following items:

**IDS:** An Internal Data Structure is a collection of typed data, private to the component and not visible in its public interface. It allows to share parameters, computed values or state variables between codels (see below) of the component implementation, much as global variables would do. Component entities that need to read or write members of the IDS must specify them individually in their argument list, so that a proper synchronization between different threads of the component can be achieved automatically on behalf of the user, typically by using system mutex and locking primitives.
CT_robot

**IDS:**
- speed, patrol_speed
- threshold
- x, y, width, height

**Task:**
- track 10ms

**Services:**
- SetThreshold
- SetPatrolSpeed
- StopTracking
- ColorTrack

Figure 2: CT_robot: (ColorTracker_robot) a simple robot tracking a color.

**Ports:** They describe structured data that are part of the component public interface, represented by octagons in Fig. 1 and 2. Ports are named resources, accessed in an atomic way, so that contained data are always consistent. Data can be either exported by the component (out port) or asynchronously imported from other components (in port). Component entities accessing those data declare them in their argument list, so that automatic handling with respect to the underlying middleware can be achieved on behalf of the user.

**Services:** The second part of the component public interface is a set of services that are procedures taking arguments and returning values. They represent the core functionalities of the component and implement the user algorithms. Depending on their characteristics, services fall into three subcategories that are part of their definition: attribute, function and activity.

**Attributes** are accessors for the IDS members that need to be exposed publicly without further algorithmic processing but possibly filtered by a validation procedure. This is typically used for setting or getting the value of a parameter or a group of parameters used by other algorithms implemented by other services. We consider that attributes services have zero processing time in practice (or take a negligible amount of time).

**Functions** (in italic on Fig. 2) are used to implement simple procedures that are meant to produce a result immediately and not remain in a running state once their execution is done. For this reason, they are typically invoked synchronously and are not expected to linger.

**Activities** (in bold on Fig. 2) implement complex procedures that are meant to run for a long time. They usually do not provide a result directly but rather as a side effect of their execution, by performing some actions through connected hardware, continuously executing an algorithm on incoming data retrieved by input ports or producing data exported to output ports. They are typically invoked asynchronously and modify the component state. Because of their long-lasting nature, activities are interruptible, at the client request, or as a specified side effect of another service starting.
Requests, Reports, Exceptions: Components provide a remote procedure call interface that enables external clients to request the execution of services at any time. This is implemented via a request-reply protocol specific to the middleware and transparent to the user. In addition to any data computed and returned as a result of the service execution, the reports always contain the status of the execution: in case of failure or anomalous execution, an exception is raised. Exceptions are user-defined types, like any other data type, that describe precisely the nature of the execution error and allows error recovery in the client invoking the failing service.

Control Task: It is an internal unit of execution (typically a thread) that manages in an infinite loop the control flow of a component by processing incoming requests made by external clients, triggering the execution of the subroutines associated with the requests and sending corresponding reports once the execution is complete. The control task is directly in charge of executing the user code associated with attributes and functions services, while it delegates the execution of activities to execution tasks (see below). All components have exactly one control task.

Execution Task(s): They are also a cyclic internal unit of execution. They execute the user code associated with activities that they are in charge of and are driven by the control task. At the user choice, they can be periodic or run freely, depending on the algorithms that they are to run. They can also optionally execute a particular activity, named permanent activity, during their whole lifecycle instead of just as the result of a user request. A component can have from zero to many execution tasks.

Codels: A codel is specified with: i) the name of a C function (implementing C or C++ code), ii) the arguments to pass when calling it (from or to the IDS, in or out ports data and service input parameters or output values) and iii) a list of allowed return values. The returned values are used by the activity automata, defined in the next paragraph.

Activity Automaton: Activities are modeled with an automaton that breaks down the processing into different states (see an example in the lower right part of Fig. 1). Each state is associated with a codel (top right part of Fig. 1). The execution of that codel leads to (yields) the next state in the automaton, to execute immediately, or in the next period if this next state name is prefixed with pause (e.g., $s1$ leads to $\text{pause::s1}$ or $s2$).

IDS and Ports are data structures, while the other elements are mostly algorithms with a generic implementation. External accesses, i.e., Ports and requests-replies are handled by the selected middleware at code generation time.

3.2 Specification language

We use $\text{GenoM}$ both as the name of our overall framework and as the name of the specification language used to instantiate all the primitives and notions defined in the previous section. This language is specified by a formal grammar [Mallet, 2013] and parsed by a classical LALR parser (namely flex and bison). To avoid confusion, we also use the term “.gen language” for the specification language and “.gen files” for the actual files containing the description specification of a component. .gen specifications are plain text files, much like source code.

To better understand the specification language and the implementation of a component as described above, we now examine in more detail the CT_robot (ColorTracker_robot) example in Fig. 2. This small robot is mounted with a camera, and we program it to provide a service to follow a colored “object” by keeping it centered in the image. We assume that the image processing is performed with
some simple OpenCV [Bradski, 2000] primitives (that return the \(x, y\) position of the barycenter of the tracked color in the image), and the robot is controlled with linear speed along \(x\) (forward) and angular speed around \(z\) (upward).

The component provides a number of services, in particular a \texttt{ColorTrack} activity service in charge of tracking the color, passed as argument, in the image and producing the speeds to keep the color centered. When the robot loses the tracked color, it uses a predefined patrolling speed (e.g., making circles) until it finds it again.

We now examine and explain in detail the specification of this component. The prologue part declares the component name and provides some self-explanatory fields.

```plaintext
1 component CT_robot {
  2 version "1.0";
  3 email "felix@laas.fr";
  4 lang "c";
  5 doc "A simple component for a robot tracking a colored object in an image.";
  6 codels-require "opencv, cv_bridge";
  7 exception bad_image_port, bad_cmd_port, bad_color, opencv_error, e_mem;
```

The \texttt{codels-require} section (line 6) specifies that these packages (opencv and cv_bridge) are to be linked with the final component. \texttt{exception} (line 7) defines the possible unexpected failures or anomalous execution of services returned as part of the requests reports. They may refer to data types defined in a similar manner as shown below, but not presented here for brevity.

The following section defines a \texttt{cmd_s} structured type to hold a speed \((\text{vx}, \text{wz})\), a \texttt{color} structure, holding \texttt{rgb}, and the Internal Data Structure (IDS) with an instance of those types and a few other simple members. The language used for data type description is a subset of the OMG IDL specification [OMG, 2018].

```plaintext
8 struct cmd_s { // The internal speed struct declaration
  9   double vx; // linear speed along x (x-forward)
 10   double wz; // angular speed around z (z-up)
 11 };
12
13 struct color {
 14   unsigned short r,g,b; // 0-255 each
 15 };
16
17 ids {
  18   cmd_s speed; // Internally computed speed
  19   cmd_s patrol_speed; // Patrolling speed
 20   long x,y; // Position of the "center" of the tracked color in the image
 21   long width,height; // Size of the image
 22   long threshold; // acceptable color difference
 23 };
```

The IDS contains data (e.g., speed and patrol_speed) which can be read and written by all the services defined in the component. GenbM components synthesized implementations manage data access and proper synchronization of the IDS fields among the threads of execution.

The next section defines \texttt{ports}.

```plaintext
/* ------------------ Ports Definition ---------------------------------- */
port in sensor::Image ImagePort {
  doc "The port ImagePort containing the image from the camera."
};

port out geometry::Twist CmdPort { // CmdPort is the speed command port
  doc "The port CmdPort in which we put the speed at which we drive the robot."
};
```

Ports contain data of the specified type (not described here for brevity). \texttt{Out} ports are produced by the component (e.g., \texttt{CmdPort} of type geometry::Twist to get the robot moving), \texttt{in} ports are
read by the component (e.g., `ImagePort` of type `sensor::Image` produced by another component controlling the camera).\footnote{In this particular example, we use the ROS template, thus the two ports are mapped to ROS topics.} Independently of the middleware used, GenoM components synthesized implementations handle the access to ports with proper mutual exclusion (if needed).

The `task` section defines one periodic task with a specified period (e.g., 10 ms for `track`). To specify a task without a period, i.e., looping continuously, the `period` keyword would simply be omitted.

```plaintext
/* ------------------- Tasks Definition ---------------------------------- */
task track {
  period 10 ms; // fast, but we only process the image when it is new.
codel <start> InitIDS(port out CmdPort, ids out speed) yield ether;
codel <stop> CleanIDS(port out CmdPort) yield ether;
}
```

The task is in charge of running the activity instances specifying it as their task. It is defined by a name (`track`) and an optional period (10ms). It may also optionally define a permanent activity that behaves like a service activity but not associated to a user request and without any input or output parameter. A permanent activity is instead started as soon as the task is created and stopped right before the task is destroyed. This can be used to initialize the component or perform any required background job. The declaration of this activity uses an execution automata in the exact same way as activity services and is described in more detail in the next paragraph. In this particular example, the permanent activity is defined by `start` (line 35) and `stop` (line 36) codels. The `start` codel is declared as yielding only to the reserved keyword `ether`: this indicates the absence of a yielded state instead of an actual state and thus the termination of the activity once the `start` codel returns\footnote{“ether” can be understood as a synonym for “void” and a historical reference to the non existing aether substance.}. The `start` codel thus runs once during the component startup and the permanent activity of the task terminates right after (but the task keeps looping executing service activities). Similarly, the `stop` codel runs just once upon component termination, much like a destructor, to do some cleanups.

The last section specifies the services (Attribute, Function and Activity).

```plaintext
/* ------------------- Attribute Services Definition ------------------- */
attribute SetThreshold(in threshold);
attribute SetPatrolSpeed(in patrol_speed);

/* ------------------- Function Services Definition ------------------- */
function StopTracking() {
  doc "Stop the tracking."
  codel ReportStop(); // This codel does not do much... just here as an example.
  interrupts ColorTrack; // When StopTracking is called, this will force the transition to the
    // stop codel in the ColorTrack activity automata
}

/* ------------------- Activity Services Definition ------------------- */
activity ColorTrack (in color tracked_color) // the color we want to track in the image
  doc "Produce a twist so the robot follow a color in an image"
  task track; // The task in which ColorTrack instance will execute
  validate ValidateColor(local in tracked_color); // validate the values passed as argument

  // Automata syntax
  // codel <state> c_function({{ids|port|local}? {in|out|inout} arg_k,*})
  // yield {pause::}?<state_i>*{, {pause::}?<state_j>}{*;
  // - ids/port/local is optional if arg_k name is not ambiguous,
  // - start, stop are predefined states, ether is the reserved keyword for the 'void' (or terminal) state,
  // - yield pause::state means the transition will be effective in the next task cycle.

codel <start> GetImageFindCenter(port in ImagePort, local in tracked_color,
  ids in threshold, ids out x, ids out y,
  ids out width, ids out height)
  yield pause::start, // no new image, wait next cycle of the track task
  Lost, // lost the color
  CompCmd, // found the color, compute the speed
```

```
ether; // in case of error.
codel <Lost> ComputeSpeedWhenLost(ids out speed, ids in patrol_speed)
yield PubCmd;
codel <CompCmd> ComputeSpeed(ids in x, ids in y, ids in width, ids in height, ids in speed)
yield PubCmd;
codel <PubCmd> PublishSpeed(ids in speed, port out CmdPort)
yield pause::start, // Loop back at the search in the next cycle
erher; // in case of error.
codel <stop> StopRobot(ids out speed, port out CmdPort) // stop is a predefined state
yield ether; // ColorTrack execution will jump to this state when the
//service is interrupted
after SetPatrolSpeed; // we can only call this service after SetPatrolSpeed
throw bad_cmd_port, bad_image_port, // Possible errors in the codels.
opencv_error, bad_color; // Any will force execution to ether
interrupts ColorTrack; // Only one ColorTrack service running at a time
};

Services are started when the corresponding execution request is received. They can take arguments (e.g., tracked_color (line 51) in the ColorTrack activity) and return values (none in this example). They can specify a validate codel (line 54), which is executed by the control task and is used to check the validity of the arguments of the service. In the example above, the validate codel (tracked_color) checks that the argument has proper value (0-255 for each rgb field). They can also have an after (resp. before) field to specify the services which need to be executed before (resp. after). For example, the after line 80 specifies that ColorTrack can only be called after SetPatrolSpeed has been successfully executed.

Attribute services (line 39) are setter or getter of particular IDS members (or a subset of members). These services can then be used to get the values of the designated members or set them to new values passed as arguments. The two attributes services defined in lines 39 and 40 are used to set the threshold and patrol_speed fields of the IDS.

Function services (line 42) are services with a simple codel (e.g., ReportStop(); in the StopTracking line 43). Note the interrupts field which specifies that calling this service interrupts the ColorTrack activity service, if running.

Activity Services (line 50) are the most complex services. They declare (line 53) the task in which they run (at the task’s period) and an automata specifying the different steps to take to perform the service. The sequence of codel/state declarations below is drawn Fig. 3. The start state (line 63) is the entry point of the automata and the ether state is the virtual termination pseudo-state (normal termination or as the result of raising an exception). For each state, a C function (codel) to call is specified (e.g., GetImageFindCenter for the start state) as well as the arguments to be passed (ports, IDS fields or service argument if any) and the possible next state(s) in the automata that the codel may yield to (in the sense “lead to”). If the state name is prefixed with pause:: (lines 66 and 75) then this transition is delayed until the next period of the task. A stop state (line 77)

Figure 3: ColorTrack Activity Automata.
can be specified to indicate which codel to execute when the service is interrupted by the user or by another service (via the interrupt keyword). For example if the StopTracking service defined above is called, and the ColorTrack service is running, it will be interrupted and its control will be passed to the stop state of its automata which will then execute the StopRobot C function. Note that ColorTrack also interrupts itself, which means that only one instance, the last one, can be running at once. Last, the throw field specifies the possible exceptions (among the ones declared in the prologue) the ColorTrack activity can return.

The CT_ROBOT component is rather simple. We will see a more complex and realistic experiment (deploying five GenoM components) in Section 6. Moreover, the interested reader can check [Hladik et al., 2021] for another complex experiment where this approach has been deployed with nine GenoM components.

### 3.3 Codels definition

To complete the specification of a GenoM component and enable the synthesis of the final executable, one also needs to define the codels declared in the .gen specification. Defining a codel consists in implementing the corresponding C function that processes its parameters and returns one of the allowed states to transition to at runtime (yield keyword).

The listing below presents an example of such C function.

```c
/** Codel ComputeSpeed of activity ColorTrack.
 * Triggered by CT_robot_CmdCmd.
 * Yields to CT_robot_PubCmd.
 */
genom_event ComputeSpeed(int32_t x, int32_t y, int32_t width, int32_t height,
    CT_robot_cmd_s *cmd, const genom_context self)
{
    float cmd_x_pixel_value= 5.0 / width; // 5 rad/s for the entire width
    float cmd_y_pixel_value= 5.0 / height; // 5 m/s for the entire height
    cmd->wz = - ((x - width/2) * cmd_x_pixel_value);
    cmd->vx = - ((y - height/2) * cmd_y_pixel_value);
    return CT_robot_PubCmd;
}
```

GenoM synthesizes and updates the comments and the function argument list with the proper type as defined in the .gen file (line 72 of the specification listing page 14). In the listing above, the seven first lines are synthesized, the programmer can then focus on the algorithm to deploy. The returned value must be among the ones in the Yields comment (line 3) (consistent with the ones specified line 73 in the .gen file).

The other codels for this particular component must also be defined (GetImageFindCenter, ComputeSpeedWhenLost, PublishSpeed, StopRobot and ReportStop) but are not shown here for brevity.

At this point, the developer has written a specification file, i.e., the .gen file, and a file with the codels implementation. No additional code writing is necessary and everything else needed to obtain the final component executable is automatically synthesized. The synthesis of the component itself and associated tools and libraries is automatically performed (Sect. 3.5) using a text template with substitutions mechanism (Sect. 3.6). Before describing the template mechanism, we shall examine the main algorithms used to implement a component instance.

### 3.4 Main Algorithms

This section broadly presents the main algorithms involved in GenoM component synthesis. This may seem rather detailed, but it is necessary to synthesize the formal model equivalent to these algorithms which define the operational semantics of GenoM. Two algorithms are important to present: the control task (Algo 1) with the function CTServ which executes the control task part of a service, and the execution task (Algo 2) with the function Invoke which executes the codel automata for the activities. They have been simplified to keep them...
short and to the point. For example, the arguments of the requests, the returned values and the port management are not presented here.

Algorithm 1: Control task

```
while not shutdown do
    newEvent ← WaitAndGetEvent(); // an incoming request or a signal from an ET
    if newEvent.type = request then // newEvent is a request
        Srv ← newEvent.Rqst; // Which service for this request?
        CTServ(Srv); // This will call the CT part of the service
    end
    forall ET ∈ ExecutionTasks do // For all ET execution tasks
        forall Act ∈ ET.activities[] do // For all activities (an array of the activities associated with this task).
            if Act.status = ETHER then // Activity is finished, report it
                Report(Act.reply);
                Act.status ← VOID;
            end
        end
    end
end
```

For Algo 1 and Algo 2, we consider the list `ExecutionTasks` which contains the execution tasks of the component. An execution task `ET` is a data structure including the fields: `periodic` (a boolean), `period`, `sched ∈ {CONT, DONE}`, and activities (an array of the activities associated with this task). An activity `Act` is a data structure including the fields: `status ∈ {INIT, RUN, STOP, ETHER, VOID}`, `stop` (a boolean), `pause` (a boolean), `state ∈ automata states, e.g., start, stop, ether, etc.}, i.e., the current state of the automata.

Function CTServ(Service)

```
Input: Service Srv
if CheckBeforeAfter(Srv) and (not Srv.validate or Call(Srv.validate) = ok) then
    if Srv.type = attribute then // Attribute service setter/getter
        TakeResource(IDS.field);
        Srv.reply ← SetOrGet(IDS.field);
        GiveResource(IDS.field);
    end
    if Srv.type = function and Srv.codel then // Function service codel
        TakeResource(Srv.codel);
        Srv.reply ← Call(Srv.codel);
        GiveResource(Srv.codel);
    end
    forall ET ∈ ExecutionTasks do
        forall Act ∈ ET.activities[] do
            if Act ∈ Srv.interrupt then Act.stop ← T; // Srv inter. Act
        end
    end
    if Srv.type = function or Srv.type = attribute then // Send the reply
        Report(Srv.reply);
    else // Activity is transferred to its execution task
        Srv.task.activities[Srv].status ← INIT;
    end
else // before/after or validate codel are not OK
    Report(error(Srv));
end
```

The control task algorithm, presented in Algo 1, is an infinite loop, with two parts. The first part begins by waiting for an event. If the event is a new request, then the Function CTServ is called. Note that we do not specify how requests are being passed to the control task `WaitAndGetEvent`, nor how `Report` is implemented. This will be handled by the selected middleware. The function CTServ checks before/after constraints, validate codel and according to the type, executes setter or getter, function codels and then interrupt services as needed.
For function and attribute services, it reports their execution, and for activity services, it dispatches them in the proper execution task. The second part of the loop in control task algorithm checks over all activities in all execution tasks, to report their results if any has terminated \((\text{status} = \text{ETHER})\).

Algorithm 2: Execution task

Data: \(\text{ET}\) is an execution task

while not shutdown do
  if \(\text{ET}\.periodic\) and \(\text{ET}.sched \neq \text{CONT}\) then wait next \(\text{ET}.period\); // periodic task
  newLoop ← \(\text{ET}.sched \neq \text{CONT}\);
  forall \(\text{Act} \in \text{ET}.activities\) do
    switch \(\text{Act}.status\) do
      case \(\text{INIT}\) do
        \(\text{Act}.status ← \text{RUN}; \text{Act}.state ← \text{start}; \text{Act}.pause ← F\);
        break;
      end
      case \(\text{RUN}\) do
        if \(\text{Act}.stop\) then // Activity is interrupted
          \(\text{Act}.status ← \text{STOP}; \text{Act}.state ← \text{stop}; \text{Act}.stop ← F; \text{Act}.pause ← F\);
        end
        // No break... Fallthrough
      end
      case \(\text{STOP}\) do
        if \(\text{Act}.pause\) and newLoop then \(\text{Act}.pause ← F\);
        break;
      end
    end
  end
  \(\text{ET}.sched ← \text{DONE}; \text{newLoop ← F}\);
  forall \(\text{Act} \in \text{ET}.activities\) do
    if \(\text{Act}.pause\) or \(\text{Act}.status \neq \text{RUN and Act}.status \neq \text{STOP}\) then continue;
    \(\text{Act}.status ← \text{Invoke(Act)}; // Call the codel associated to the current state\)
    if \(\text{Act}.status = \text{RUN and not Act}.pause\) then \(\text{ET}.sched ← \text{CONT}\);
  end
end

The execution task algorithm is presented in Algo 2. It is also an infinite loop, scheduled at its period if one was specified. The first part checks and sets for all activities their \(\text{status}\), \(\text{state}\) and \(\text{pause}\). The second part loops over all activities again and executes one step of the automata for all activities with proper \(\text{status}\) and \(\text{pause}\) values. The invoke function calls the proper codel and sets \(\text{state}\), \(\text{pause}\) and returns \(\text{status}\) accordingly.

The \text{Invoke} function is defined for a specific activity service automata. Indeed, the code will depend on the number of states in the automata and the transitions. Still the algorithm sketches the main idea. We have a specific treatment for the \(\text{stop}\) state, which means that the activity has been interrupted. Otherwise, the processing is the same for all other states: execute the codel, if the value returned is prefixed with \(\text{pause}:\) then we set the \(\text{pause}\) flag as to inform the \(\text{ET}\) that the execution should continue with the next activity within the current period.

3.5 Executable Components Synthesis

\text{GenoM} was initially developed to synthesize the code which implements the component itself. The synthesis follows the workflow described on Fig. 4. Given a .gen specification file and the codels declared in the specification (blue boxes), we synthesize, according to the template selected by the user (red boxes), the components for either ROS [Quigley et al., 2009] or pocolibs [Herrb, 1992] middleware. Both versions implement the algorithms presented in section 3.4 and behave exactly the same. The two flavors only differ in the way that the ports and the requests/replies are implemented, but internally there is no difference. Regarding \text{GenoM} ports, the pocolibs template uses shared memory while the ROS one implements them with ROS Topics and publish-subscribe over sockets. The request-reply handling in pocolibs relies on the pocolibs Mbox mechanism. With ROS, a specific CallBackQueue is used; \text{GenoM} attribute and function services are handled as ROS
Function Invoke(Activity)

(this function depends on the automata, we present here a generic version)

Input: Activity Act
Output: Status

if Act.state = stop then // The activity has been interrupted
    if Act.codels[stop] then // If the codel stop exists, we call it
        TakeResource(Act.codels[stop]):
        Act.state ← Call(Act.codels[stop]):
        GiveResource(Act.codels[stop]):
    else // No stop codel, we are done
        Act.state ← ether:
    end
else if Act.state = x then // We are in state x
    TakeResource(Act.codels[x]):
    Act.state ← Call(Act.codels[x]):
    GiveResource(Act.codels[x]):
    .... // As many states as needed for the considered automata
else if Act.state = y then // We are in state y
    TakeResource(Act.codels[y]):
    Act.state ← Call(Act.codels[y]):
    GiveResource(Act.codels[y]):
end
if Act.state matches pause::z then // Execution returned a pause state
    Act.pause ← T:
    Act.state ← z:
else if Act.state = ether then // The automata execution is finished
    Signal(): // Wakeup the control task
    return ETHER;
else // Automata execution is not done yet
    return RUN;
end

Services; and GenoM activity services are implemented using ROS Actions.

Figure 4: Toolchain with the regular pocolibs and ROS template.
3.6 \texttt{GenoM} template mechanism

When called alone, the \texttt{GenoM} program just parses and builds an internal representation of the \texttt{.gen} specification files. To produce output, \texttt{GenoM} has to be called along with a template. The templates are the real building blocks used to synthesize source code files adapted to the current \texttt{.gen} specification. These codes can be the sources implementing the component itself, or client libraries to interact with the component, or, as we will see in Section 5.2, formal models of the component implementation.

A \texttt{GenoM} template is a set of text files that contain embedded Tcl expressions and statements whose evaluation by the \texttt{GenoM} tool produces output. When applied to a specific \texttt{.gen} file, this produces the target of this particular template for this particular \texttt{.gen} specification. The target can be anything, but it usually contains source code that implements a part of a component (client communication library, execution of an activity automaton, data serialization...) for a specific middleware.

The template mechanism was initially introduced to deal with the middleware independence problem [Mallet et al., 2010]. Indeed, the specifications presented above do not assume any specific middleware. In short, the components are specified in a generic way using \texttt{GenoM} and different templates are then used to automatically synthesize the components for different middleware, which are then linked to the codels library for the considered component.

A template, when called by \texttt{GenoM} on a given component 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. The interpreter evaluates as Tcl code anything enclosed in markers `<' '>'` without output, while on the code between `<'' ''>`, Tcl variables and commands substitution is performed and the result is output in the destination file, together with the text outside of these markers.

For example:

```tcl
foreach s [\$component services] {'
"[\$s name]"
'}
```

iterates over the list of services of the component, contained in the \$component variable; while `<"[\$s name]">` is replaced by the name of the service contained in the \$s variable bound by the \texttt{foreach} statement, and will produce this list:

SetThreshold
SetPatrolSpeed
StopTracking
ColorTrack

A more complex example is shown below (excerpted from the ROS template). It shows an excerpt of a template code which produces C++ code for tasks and services. Note the Tcl loops over all the execution tasks and over all the services of the component. Note also the Tcl test to check if a service is an “activity” (or a “function”/“attribute”).

```c
/* spawn tasks */
foreach t [\$component tasks] {'
cids.tasks.<"[\$t name]">._spawn(&cids, <"[\$t name]">, _task);
if (!cids.tasks.<"[\$t name]">._spawned) goto shutdown;
'}
/* advertise services */
genom_log_info("advertising services");
foreach s [\$component services] {'
if {[\$s kind] ne "activity"} {'
cids.services.<"[\$s name]"> = cids.control.node->advertiseService(
    <"[\$s name]">, &genom_component_data::<"[\$s name]">_rqstcb, &cids);
} else {'
cids.control.node->setParam(<"[\$s name]">/status_list_timeout", 0);
cids.services.<"[\$s name]"> = new actionlib::ActionServer::<"[\$component name]">_genom::action_<"[\$s name]">(_,
    <cids.control.node, <"[\$s name]">>,
    boost::bind(&genom_component_data::<"[\$s name]">_rqstcb, &cids, _1),
    boost::bind(&genom_component_data::<"[\$s name]">_intercb, &cids, _1),
    19
```
The resulting C++ code produced when called together with the CT_ROBOT component specification file is below. Note how the C++ code is synthesized for one execution task (track) and then all the services of the component declared in the .gen file.

```cpp
#include "cids/services.h"

The resulting C++ code produced when called together with the CT_ROBOT component specification file is below. Note how the C++ code is synthesized for one execution task (track) and then all the services of the component declared in the .gen file.

```cpp
#include "cids/services.h"

```cpp
/* spawn tasks */
cids.tasks.track_.spawn(&cids, "track", CT_robot_track_task);
if (!cids.tasks.track_.spawned) goto shutdown;
/* advertise services */

```cpp
/* spawn tasks */
cids.tasks.track_.spawn(&cids, "track", CT_robot_track_task);
if (!cids.tasks.track_.spawned) goto shutdown;
/* advertise services */

```cpp
genom_log_info("advertising services");
cids.services.SetThreshold_ = cids.control.node->advertiseService("SetThreshold", &genom_component_data::SetThreshold_rqstcb, &cids);
cids.services.SetPatrolSpeed_ = cids.control.node->advertiseService("SetPatrolSpeed", &genom_component_data::SetPatrolSpeed_rqstcb, &cids);
cids.services.StopTracking_ = cids.control.node->advertiseService("StopTracking", &genom_component_data::StopTracking_rqstcb, &cids);
cids.control.node->setParam("ColorTrack/status_list_timeout", 0);
cids.services.ColorTrack_ =
new actionlib::ActionServer<::CT_robot_genom::action_ColorTrack >(
  cids.control.node, "ColorTrack",
  boost::bind(&genom_component_data::ColorTrack_rqstcb, &cids, _1),
  boost::bind(&genom_component_data::ColorTrack_intercb, &cids, _1),
  false);
cids.services.ColorTrack_->start();

Different templates have been developed and are available to synthesize, e.g., the component implementation for various middleware (pocoLibs, ROS...)(Section 3.5), client libraries to control the component (e.g., Tcl, Python, C, Matlab, OpenPRS).

4 A formal framework for offline and run-time verification: FIACRE

We describe the FIACRE specification language, and its associated toolset, which is used in all our formal verification activities. The goal of this section is to give just enough details to understand the basic constructs and capabilities of the language, and some insight to how we can use FIACRE to capture the finest details of a G68U6M implementation. You can find a more precise description of the language in [Berthomieu et al., 2020], and how it has been recently extended for runtime verification in [Hladik et al., 2021].

FIACRE is a component-based language for describing the behavior of concurrent, communicating systems with real-time constraints. It borrows many design ideas from process algebras and offers a syntax that is very close to conventional programming languages. For instance, it supports functions, the declaration of typed variables, basic control structures such as loops and conditional statements. This language has been designed to propose abstractions that are easy to use by someone who is not an expert in formal languages. While the language was initially designed only for model-checking purposes, we have recently proposed an extension, named H-FIACRE, that can be used to generate and execute code.

4.1 A common formal language: The FIACRE Language

We use a simple program (Listing 1) to illustrate some of the constructs of the language. The complete code is available on Gitlab\(^5\). The example models a system for tracking a blue hockey puck on a table (see Fig. 5). Four ArUco [Garrido-Jurado et al., 2014] markers are used to delimit the space in which the puck moves. The camera takes new pictures with a period of 17 ms, which translates to a refresh rate of about 60 frames per second. Processing relies on three C functions: capture to capture the image, markers to detect the markers, and compute_center to calculate a transformation matrix to position the search area of the puck. As the camera is fixed, it is not necessary to recalculate the position of the markers for each image.

\(^5\)https://gitlab.laas.fr/hippo/fiacre_puck.
Listing 1: H-FiACRE model, puck.fcr, for the puck tracking system.

```
// Part 1: Declaration of types, events and tasks

type ty_key is union f | other end // union can be used for enumeration types

task t_capture() : nat is c_capture // each task is connected to a C function

// Part 2: Description of the behavior

// Part 3: Instanciation of the system
```

```
component main is
   var available : bool := false, markersFound : bool := false
   port Activation : none in [0,0]
   par * in
      p_periodicClock [Activation]
      | p_capture [Activation](available)
      | p_computeCenter (available, markersFound)
      | p_findMarkers (markersFound)
   end

main
```
The detection is carried out when we receive a request from the user (defined as a keypress event on the f key), after which we locate the puck when a captured image is available, and we can detect the markers in this image.

Figure 5: Image from the tracking Hockey puck system.

Our model for this program (Listing 1) has three parts: (i) the declaration of types, events and tasks used in the system (lines 4–9); (ii) the processes describing the behavior of each function and their coordination (lines 14–45); (iii) and the composition of the principal component of the system (called main, lines 50–61).

The code in Listing 1 is actually an example of H-FIACRE specification. H-FIACRE is an extension of FIACRE that adds the capability to make asynchronous calls to concurrent subroutines with the operator start (e.g., line 23 to call t.capture); to wait for a return value with a sync (e.g., line 26 for the end of t.capture); and to declare external events. The first two additions are useful to describe function calls, what we call task. A H-FIACRE task is basically a C function with input and output data that can be executed concurrently. The prototype of each task has to be declared in the first part of a H-FIACRE model (lines 6-8).

Likewise, external events are useful to model events originating from the environment of the system. They can be used to define how the system should react when an external sensor sends a value, when a signal is emitted from the operating system or a message is received on a socket. An event has to be declared in the first part of the model (e.g., line 9 for the event e.keyboard used in our example to react to a keypress) and is linked to a C function. Synchronization on event ports uses the ? operator to match a received value (line 42).

All the other elements presented in the following are part of the language FIACRE (and of H-FIACRE by extension). So, in addition to tasks and events, it is possible to declare complex types in the first part of the model. For example, line 4 type ty_key is declared as an enumeration of constant elements.

The behavior of the system is described in the second part of the model. For this, we use concurrent processes (e.g., lines 14, 19, 31 and 39). A process describes the behavior of sequential components and is defined by a set of control states (e.g., line 20 for the process p.capture). The behavior of each process is defined by a state transition system, where transitions are expressed using statements built from classical imperative constructs: assignments (line 26), conditionals (line 43), pattern matching, sequential compositions, etc.; but also operators that are usually found in process algebras, such as synchronization on data-event ports (with n-way synchronizations, and the exchange of values) and non deterministic choice (with the select operator line 25). Finally, a to operator indicates the target state of a transition (e.g., lines 16, 17, 22).

In our model, the search for markers after pressing the f key is modeled by the process p.findMarkers (line 39). This process has three states (s.fm.a, s.fm.b and s.fm.c, line 40) and a local variable v.key of type ty_key (line 41). The first state is blocking while waiting for the external event e.keyboard (line 42). The next

---

In this section only, the term “component” means a FIACRE component, not to be confused with a G6%M component.
state is determined according to the value \( v_{\text{key}} \) returned by the event (line 43). If the value is equal to \( f \) then one passes in the state \( s_{\text{fm}_b} \), otherwise one returns in the state of waiting. Once in the state \( s_{\text{fm}_b} \) (line 44), we launch the execution of the task \( t_{\text{findmarker}} \) then we pass to the state \( s_{\text{fm}_c} \). This last state is blocked until task \( t_{\text{findmarker}} \) finishes and returns to the state \( s_{\text{fm}_a} \) (line 45).

One of the originality of FiACRE is its ability to describe timing constraints. For instance, a wait statement (see lines 16 and 27) imposes minimal and maximal bounds on the delay at which a transition can be taken. An example is given in the definition of process \( p_{\text{periodicClock}} \), whose behavior is to reach state \( s_{\text{pc}_b} \) every 17 ms (line 16). While we use a wait statement to encode a periodic task activation in this case, the same mechanism can be used to model timeouts, watchdogs or the execution time of a computation.

A component can also declare synchronization ports in its interface (e.g., the port \( \text{Activation} \) line 14 for process \( p_{\text{periodicClock}} \)) and shared variables (e.g., \( \text{available} \) line 19 for process \( p_{\text{capture}} \)). The ports and shared variable can be used to exchange data or synchronize processes’ behavior. For example, the process \( p_{\text{periodicClock}} \) is synchronized with the process \( p_{\text{capture}} \) through the port \( \text{Activation} \) (lines 17 and 22).

We depict in Fig. 6 (left) the software architecture of our example, extracted from Listing 1, that describes the ports, variables and events through which the processes interact together.

The composition of the whole model is done in the third part of the program, where instances of processes are composed in parallel (lines 53 to 58). This is also the place where ports and shared data are instantiated in order to connect processes together, e.g., the port \( \text{Activation} \) declared in line 52 connects the \( p_{\text{periodicClock}} \) and \( p_{\text{capture}} \) processes (lines 54 and 55). The instantiation of the shared data is done in this part, for example \( \text{available} \) and \( \text{markersFound} \) to test if an image is available and if the markers were found (line 51). It is possible to associate a time interval to a synchronization on a port (line 52). Here the interval \([0,0]\) means that the synchronization between process \( p_{\text{capture}} \) over \( \text{Activation} \) is immediate.

Remark that the FiACRE code clarifies some of the ambiguities in our informal specification and adds further details. For instance, that the keyboard events are read in the \( p_{\text{findMarkers}} \) process, or that we detect the case where an image capture lasts longer than 15 ms (line 27). However, to keep things simple, we intentionally choose to use a toy example: error handling is not implemented, user interaction is minimal, etc. Nonetheless, the specification is still executable and verifiable. Note that H-FiACRE is quite low-level, hence the use of a more structured and domain-oriented language, such as \( \text{G}^{\text{en}}\text{M} \), is necessary to deal with real-life use cases. The only purpose of the example provided here is to introduce the main elements of the language and to show how they are used, it does not pretend to be a real implementation of the tracking system.

Figure 6: Architecture (left) and Petri net model (right) extracted from the \( \text{puck.fcr} \) model.
4.2 HIPPO an engine to execute H-FIACRE model

H-FIACRE specifications can be compiled into executable C code using a framework called HIPPO. The code generated from a H-FIACRE specification is faithful to the original semantics, meaning that we preserve the behavior of the model. The model is automatically transformed into C code and linked with the H-FIACRE task C functions. An execution engine, part of HIPPO, is used to orchestrate the system execution. This engine controls the behavior of the system and schedules the execution of the operations. Each task of the H-FIACRE specification is scheduled in its own thread by the operating system on which the execution engine is based (i.e., RT-thread in the current implementation). Its implementation is currently based on Linux services and offers soft real-time performance. A complete and more detailed description of HIPPO is available in [Hladik et al., 2021].

4.3 FIACRE offline verification tools suite

We can take advantage of the level of details available in the model to check behavioral properties on the system. FIACRE is one of the input languages of the TINA verification toolbox\footnote{https://projects.laas.fr/tina/}. As such, we can use several model-checking, state space generation and simulation tools. At the heart of many of these tools there is a translation from FIACRE to Time Transition Systems (TTS), an extension of time Petri nets where we can associate data processing functions with net transitions. The compilation of FIACRE models into TTS is done with a tool called FRAC which is part of the toolchain described in Sect. 5. The number of places and transitions in the TTS gives an idea of the architectural complexity of a model, whereas the number of possible reachable states is an indicator of its computational complexity.

We display in Fig. 6 (right) the Petri net obtained from the puck tracking system. Places are represented by circles and transitions are depicted as squares. Intervals near transitions are the temporal condition to be fired (if there is no interval, it means that the transition has no time constraint). We can identify the four processes that constitute the main component and their timing constraints. We also observe the interaction (a shared transition) between the periodic clock and image capture processes over the Activation port. Each transition in the TTS is associated with expressions from the FIACRE model. For instance, transition capture1, from states s_c_b to s_c_c of p_capture, corresponds to line 23 in Listing 1. Likewise, transitions find0 and find1 correspond to the two branches of the conditional in line 43.

We can check several behavioral properties in our example. For instance, whether it is possible to timeout during an image capture (if state s_c_d in p_capture is reachable); or whether it is possible to try and compute the position of the puck in an image without markers (a situation that should not occur). We study these two properties below. We also give some more elaborate examples in Sect. 6.

The verification model must capture the complete behavior of the system, including timing information about the H-FIACRE tasks, external events, the coordination between components, and the functioning of the HIPPO execution engine. It is therefore necessary to precisely capture the behavior of tasks and events in a “pure” FIACRE model. We assume with HIPPO that tasks are executed on an architecture with a First-In First-Out (FIFO) scheduler (the Linux implementation uses SCHED_FIFO with the same priority for all operation-related tasks). Modeling this type of scheduling is simple in FIACRE since the language offers a queue-like structure with operators to manage the addition or removal of an element. Moreover, for a multiprocessor architecture, the allocation of H-FIACRE tasks on a processor is also modeled at this level. A synchronous mechanism for data and threads control flow is also implemented to enhance the predictability and the confidence in the FIACRE model. The equivalence between the two models is not formally proven, but tests have been conducted to compare the execution traces (from H-FIACRE) to the FIACRE model traces (no differences were found). For more details, interested readers can refer to the article [Hladik et al., 2021] and more specifically Section 4.6. The result is a very precise formal model, going as far as the scheduling policies used in the actual implementation.

The modelling of tasks relies on information about the execution times of each function. Timing information can be retrieved using WCET (Worst-Case Execution Time [Wilhelm et al., 2008]) computation tools or, by adding a safety margin, using execution times measured on each function in isolation. In our experiments, we obtained the following intervals for the computation times of the function (in ms): $[2.5, 4]$ for c_capture,
for c_compute_center and \([3,19]\) for c_marker. Note that with this approach the behavior of functions and their data are not analyzed, only their temporal behavior is taken into account.

We can mimic an event (in our case e_keyboard) using a mock-up FiACRE process. The process models the behavior of the environment. In the example below (Listing 2), we consider that we can have sporadic events with an interarrival time in \([1,100]\) (we only need to replace the synchronization on the event e_keyboard, in line 42, with a synchronization on the port ep_keyboard).

Listing 2: An example of a template for an external event.

```plaintext
1 process p_event_e_keyboard [ep_keyboard : ty_key] is
2 states sstart, postingstart
3 var ret : ty_key := other
4
5 from sstart
6 wait [1, 100]; /* a non deterministic delay is added */
7 select /* several behaviors are possible: */
8 ret := f; to postingstart /* either the f key is pressed */
9 [] ret := other; to postingstart /* nor another key is pressed*/
10 end
11 from postingstart ep_keyboard!ret; to sstart
```

We report the results achieved with the FiACRE specifications obtained from the H-FiACRE model of Listing 1 after adding the behavior of the functions, the environment, and the HIPPO runtime. The first problem is to check if process p_capture can be in state d. Note that even though function c_capture has a worst-case execution time of 4 ms, the FIFO scheduling may significantly delay its execution.

This is a simple example of reachability property, whose complexity is commensurate with the number of reachable states in the system. In our case, since we take into account time constraints, we need to record both the discrete state of the system (the current state of each process and the value of shared variables), but also information about the firing delays of possible transitions. This is what we call a class. For a model with a two cores processor, we obtain a result of 318 346 classes (corresponding to 8 190 “discrete states”) and 1 034 051 transitions. This state space was computed in 6 s on a 2 GHz Intel Core i5 computer. The same system, modelling on a single core processor, gives 885 080 classes and 2 846 924 transitions, for a computation time of 17 s.

We find that state d cannot be reached when the hardware had at least two cores. On the other hand, with a single core, we find a possible scenario (an execution trace). This scenario corresponds to the case where t_capture is activated just after a keypress event (e_keyboard) which activates the t_findmarker task. If there is only one core, the t_capture task must wait for the t_findmarker task to finish.

We also find that, with the current specification, process p_computeCenter can try to locate the puck (reach its state b) while process p_findMarkers has not found the markers yet (it is in state c). This is because variable markersFound is not properly reset and, therefore, we can have a race condition between the c.capture and c_marker functions. This is a typical example of concurrency problems which are known to be hard to detect. The aforementioned gitlab repository contains a new version of the specification, puck_mutex.fcr, that corrects the problem.

Properties that need to be checked should be expressed as formulas in Linear Temporal Logic (LTL). This step requires knowledge of LTL and how to use the TINA toolbox. This requires an expertise that can be quickly acquired without knowing the underlying theories. If a property is found false, a counterexample is produced in the form of a trace. It is necessary to analyze this trace in order to isolate the problem. In this case, a good knowledge of the system is necessary to identify and explain the root cause. While a large part of the verification process is automated, the main difficulty remains the analysis of the traces to correct the detected errors. This last activity can be tedious and requires a strong knowledge of the system. Our current plan to alleviate this difficulty is to improve our tooling to better “debug traces”.

25
5 A GenoM template to synthesize a Fiacre model

We present in this section how we use the two frameworks (GenoM and Fiacre) together. The core idea is: based on the GenoM specification of a component and the semantics of its pocolib and ROS implementations, we synthesize Fiacre/H-Fiacre models equivalent to these implementations. As for any GenoM output, this is done with a template which can then be instantiated for any component specification.

We first introduce the overall toolchain/workflow, and then describe in more detail the contents of the models synthesized with the Fiacre template and how GenoM internal algorithms get translated to Fiacre/H-Fiacre models. We then show the type of formal properties one can check offline or monitor at runtime.

5.1 The Fiacre/H-Fiacre GenoM toolchain

Fig. 7 shows the overall toolchain. The template mechanism used to synthesize the GenoM components from their specifications and codels can also synthesize both the Fiacre verification model (which can then be used with the TINA toolbox for the verification, Fig. 7 bottom part) and the H-Fiacre runtime model (which can be compiled and linked to the codel library to produce an executable component, Fig. 7 top part). The Fiacre and H-Fiacre models only slightly differ in the codels call/management, hence, we use the same Fiacre GenoM template file to synthesize both. When we call the GenoM command on this template, a flag (-tina or -hippo) is used to choose one or the other. Moreover, the GenoM versatile template mechanism allows us a more fine-grained control on the produced model with, for example, varying level of abstraction for the Fiacre model.

The Fiacre GenoM template produces models that contain Fiacre processes implementing the algorithms of all the internal parts of a GenoM component as presented in Fig. 1 (Section 5.2 details how this implemen-
tation is done). All the FIACRE processes are then composed in parallel to produce one FIACRE component modelling one or more G\textsuperscript{en}\textsubscript{b}M components. The FIACRE compiler can be used to check that the generated code is syntactically correct. Since the language is strongly and statically typed, this first analysis performs several additional semantic checks. For instance, we prove that every synchronization is well-formed and we can also issue warnings when we detect dead-code or incomplete pattern matches.

Most of these FIACRE processes include time information which comes both from temporal information found in the components specification (for instance the period of execution tasks) and from the Worst-Case Execution Time (WCET) of the codels. At the moment, the WCET are obtained by running the regular pocolibs or ROS components with G\textsuperscript{en}\textsubscript{b}M embedded profiling tool: profundis. This temporal information is thus an input for the FIACRE G\textsuperscript{en}\textsubscript{b}M template (see the Codels WCET green box on Fig.7).

Writing the FIACRE G\textsuperscript{en}\textsubscript{b}M template requires a very good knowledge of the G\textsuperscript{en}\textsubscript{b}M specification and implementation, and a good knowledge of the targeted formal frameworks. But an interesting side effect is that writing the formal version of a synthesized implementation (e.g., the Pocolibs implementation of the component) requires to sometimes check and clarify how the specification is implemented in the algorithms. For example, a priori, nothing was said in the specification about in which order the various execution tasks (and their permanent activity) are started, so the formal model did not enforce a particular order. In the drone experiment, this leads to some awkward behaviors at startup. After checking the real implementation, we found out that they are started in the order they are declared in the .gen file, and we fixed the formal model which then exhibited the expected startup behavior. This is a win-win strategy, the G\textsuperscript{en}\textsubscript{b}M designers are thus invited to clarify the semantics of the tool and, at the same time, the tool is then properly and formally modeled.

Last, we stress again the fact that once the template is written, all G\textsuperscript{en}\textsubscript{b}M components can easily deploy it. In fact, many G\textsuperscript{en}\textsubscript{b}M users are not even aware that the tool comes with a “free” formal model which can be used and deployed at no extra “cost”.

5.2 Mapping the G\textsuperscript{en}\textsubscript{b}M internal algorithms in FIACRE and H-FIACRE

In this section, we detail the main translation performed by the FIACRE G\textsuperscript{en}\textsubscript{b}M template. As shown on Fig. 1, and presented in section 3.4, G\textsuperscript{en}\textsubscript{b}M components are organized along a number of concurrent algorithms to implement the Control Task (Algo. 1 p. 16) with its handling of the Services (call to Function CTServ p. 16), the Execution Tasks (Algo.2 p. 17) and their respective Timer, and the execution of the Activity Services with their codel Automata (Algo. Invoke p. 18). In the regular pocolibs or ROS templates, these algorithms are implemented in C or C++. With the FIACRE template, these algorithms are implemented as FIACRE processes. All these FIACRE processes are then composed in parallel, connected with FIACRE ports and shared variables (Sect. 4.1) to produce one large FIACRE component modelling one or more G\textsuperscript{en}\textsubscript{b}M components. As for the codels, they are all implemented using H-FIACRE task/operation extensions (Sect. 4.1) and event ports (Sect. 4.1) are used to model pocolibs Mbox or the ROS CallBackQueue.

The FIACRE processes synthesized in the FIACRE and H-FIACRE models are direct mappings of the algorithms listed in Section 3.4:

**Control task.** See Algo. 1 p. 16 and the H-FIACRE model on Listing 7, page 45 for the CT\_ROBOT component control task. This process manages the event ports by which the G\textsuperscript{en}\textsubscript{b}M component receives new requests. In the H-FIACRE version, the requests come from the “robot supervisor” (acting skills) while in the FIACRE version they come from a client process which will emit requests on a shared port (this process simulates acting skills). For each request received, the process passes the control to the proper service CT (see next item). Upon return, if the execution did not fail, and the request is an activity, it is passed to the proper execution task. Otherwise (i.e., it is not an activity request) it is reported to the caller. Then the process checks all the running activities for each execution task to check the ones that terminated, in which case it reports their execution status to the caller.

**Services CT (within the control task).** See Function CTServ p. 16 and the H-FIACRE model on Listing 8, page 46 for the control task handling of the ColorTrack service from the CT\_ROBOT component. This process handles validate codel (see an example line 54 page 13), precedence constraints (all after services, (line 80, same page) have executed once and returned and none of the before did), it interrupts
incompatible activities, set or get IDS value, and possibly execute one codel for function services (line 46). This process is “activated” with a variable shared with the control task, and the control is passed back to the control task when it is done.

**Execution task timer.** This is a process which just flips a tick at the specified period (see Listing 12, page 50 for the timer of the track task of CT_ROBOT component).

**Execution task.** See Algo. 2 p. 17, and H-FIACRE model on Listing 9, page 47 for the track execution task of CT_ROBOT component. When a periodic task is synchronized on the shared tick boolean handled by its timer, then it checks the status of all its potential activities, and then, for all the one runnable passes the control to the corresponding activity services for one codel step execution. Upon return, the result of all the activities called in this loop are assessed and execution proceeds accordingly.

**Activity services & permanent services (and their automata).** See Algo. Invoke p. 18 and the H-FIACRE model on Listing 11, page 49 for the execution of the ColorTrack service in the track execution task from the CT_ROBOT component. This process, according to the current state of its automata, calls (start) the codels (a H-FIACRE task), waits (sync) for its return, and then gives the control back to the execution task. It is “activated” with a variable shared with the execution task, and the control is passed back to the execution task when it is done.

The FIACRE model generated for verification and the H-FIACRE executable model really executing the codels are the same with respect to the algorithms they encode, and only slightly differ due to the nature of how they are used (model checking and runtime verification):

- Codels execution is really carried out in the H-FIACRE models (with start/sync, see Section 4.1), but are modeled as a time delay wait[0,wcet] (or [wcet,wcet]) in the FIACRE model.
- Non deterministic choices (e.g., codels returned values or control codels success/exception) are handled with H-FIACRE tasks start/sync and tests/case on the codel returned values in the H-FIACRE models, but with a simple non deterministic choice operation (select) between all possible values in the FIACRE model.
- The FIACRE model must include a client process with an out port to send requests and an in ports to receive replies, connected to the corresponding control task regular ports. The H-FIACRE model is simply “linked with the real world” using H-FIACRE event ports (Section 4.1) and tasks. In this case, the event port handles the mechanism which receives new external requests from the pocolibs Mbox or the ROS CallBackQueue.

Apart from these minor operational differences, both models are exactly the same. Listing 10 and 11 in Appendix show the difference between the H-FIACRE and FIACRE version for codels call.

Moreover, our experiments show that execution observed with a H-FIACRE components (running on the HIPPO engine) are comparable to those observed with regular pocolibs or ROS components in several testing scenarios. Meaning that, in our experiments, we see robots behave in a comparable way, with the same communications between components, and with similar delays in all the implementations. To give but one example, in our UAV use case, all implementations behaved in the same expected manner when we execute a complex navigation and then trigger interrupts or inject an unmonitored fault. 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 system behaves the same with HIPPO than the regular components do is clearly encouraging.

### 5.3 Offline verification

We can model-check and simulate models with the various tools available in the TINA toolbox. As mentioned before, one needs to provide a client which sends requests and receives replies, otherwise, the model would only “start” the permanent services if any and then sit waiting for new client requests (note that with permanent activities with complex automata, one can already have a significant number of reachable states).

---

Listing 3: Generic client automatically synthesized for the CT_ROBOT component.

```plaintext
process client_CT_robot [request_CT_robot: out request_CT_robot_type, // ports with the Control Task process
reply_CT_robot: in reply_CT_robot_type] is

states start_, finish

var CT_robot_rq: request_CT_robot_type,
CT_robot_rp: reply_CT_robot_type,
CT_robot_rqid: CT_ROBOT_RID,
grid: nat := 0

from start_ // at any time, either parse a reply from the Control Task if any available
select // or send one of the possible requests to the Control Task
reply_CT_robot?CT_robot_rp // Any reply from the Control Task on the reply port?
[[]] // or send a request among the following
CT_robot_rq := { rqid = CT_ROBOT_Set_Patrol_Speed_RID, grid = grid};
request_CT_robot!CT_robot_rq; // emit the request on the request port.
grid := grid + 1
[[]] // or
... // removed similar code for brevity
CT_robot_rq := { rqid = CT_ROBOT_Stop_RID, grid = grid};
request_CT_robot!CT_robot_rq; // emit the request on the request port.
grid := grid + 1
[[]] // or
CT_robot_rq := { rqid = CT_ROBOT_ColorTrack_RID, grid = grid};
request_CT_robot!CT_robot_rq; // emit the request on the request port.
grid := grid + 1
end; // and do it again, for ever.
to start_
```

As shown on Listing 3, we can provide a default client (in this example for, the CT_ROBOT) which is connected to the component control task through two ports (one for request, and one for reply) whose model is to either send any of the possible requests at any time, or to get any reply available, hence covering all the reachable states of the component. This may seem awkward at first sight, but this is the way model checking works in general. One wants to explore all the possibly reachable states of the system, and this client offers just that: any request and reply at any time, forever. One can hardly be more general than that. In reality, such an exploration is often untractable and the users are usually more interested in a more specific client with a particular sequence of requests corresponding to the nominal use of the analyzed component. These requests will be dispatched to the proper components for “execution” and replies will be received accordingly. Unfortunately, even if we can synthesize the complete model with any components, we cannot always synthesize the complete reachable state set of the entire system. Yet, as introduced in [Foughali et al., 2016] on a simpler model, we can check interesting specific properties on individual components:

**Schedulability** checks that G^\text{P}^n M tasks finish before their next period by taking into account the WCETs of codels, the number of cores and the scheduling policy. The synthesized model already includes an overshoot state which is reached when the execution takes more than the value specified in the execution task timer (this state is also used in the runtime version to detect overshoots at runtime). So, it is straightforward to ask the model checker if it can find a sequence leading to this particular state for any of the execution tasks.

**Reachability** checks that a particular state is always reached. For example, that a given request eventually will terminate, or that following an interrupt, the running “interrupted” activity will indeed terminate.

**Maximum time for a transition** by programming a FiACRE observer, one can set a transition with a select with two branches, a wait[bound,bound] with the maximum value to test, and a test on the given condition (e.g., the robot is stopped). If we show there can never be an execution taking the wait branch, we know the condition always becomes true within this bound (see an example in Section 6.2.2).

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

But what is also interesting during this phase is that while checking a property, the programmer may also discover inappropriate behaviors that lead to the said property. For example, by checking the schedulability of a task, the designer may discover that it is not satisfied because of a particular execution sequence he did not think of. Here, the property may not be verified but with the provided counterexamples, it identifies an unexpected behavior. Thus, the verification stage can be considered both as a mean of proving the good behavior of the system and as a mean to debug it.

5.4 Runtime verification

The advantage of running HIPPO instead of the regular pcolonbs or R0S component is to monitor online some critical properties, a first step toward runtime verification. Here is a list of the properties checked by default and already included in the automatically synthesized model.

Task period overshoot: Periodic execution tasks are specified to run at 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: as seen above, the WCET are obtained by profiling the regular pcolonbs (or R0S) component on the same setup using the profundis tool. 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 property is also a runtime verification that these WCET values, also used for offline verification (e.g., schedulability), are realistic.

Uninitialized Port Read: When controlling a multi components experiment, the HIPPO engine checks that no codel will use a port with the in direction before a codel with an out direction has already been called. If this is the case, most likely the value read in the port is not semantically “correct”\textsuperscript{10}.

As we will see in section 6.3, we can also define additional specific monitors that go far beyond these default properties, and can handle specific requirements for a particular experiment.

6 An example: a UAV controller

We have seen in section 3.2 how to specify the simple CT_ROBOT component with GenM. The use case presented here is a quadcopter flight controller. It is more complex and involves multiple GenM components. There is another complete use case presented in [Hladik et al., 2021]\textsuperscript{11}, which involves an outdoor robot navigating in a previously unknown environment. We refer the reader to this second use case to get a better idea of the diversity of robotic applications we can address with our approach.

6.1 A quadcopter flight controller

A critical aspect of the quadcopter flight controller experiment is that the motor control of such system requires rather fast processing, and most components and their periodic tasks (for localization, propellers velocity control, etc.) are running at 1 kHz. Moreover, all components are running on board the UAV, with limited processing capabilities. Even if this is not uncommon, it illustrates the usability of GenM for stringent real-time controllers and how our V&V online approach remains applicable within these requirements.

\textsuperscript{10}This type of error often occurs upon startup of the experiment where all the components are starting at once, and subtle race conditions can lead to these situations.

\textsuperscript{11}This paper focusses on the HIPPO engine, and only the RMP440 experiment had been fully deployed then.
Figure 8: Architecture of the Drone experiment.
As shown on Fig. 8, components are depicted with boxes, like NHFC, which produce shared data in ports (depicted with adjacent octagons like cmd velocity). Links between components and ports describe which components read from which ports (e.g., NHFC reads the state port from POM). We list, inside each component, the execution tasks they include (in NHFC: respectively main and wo), their period (1 ms and 10 ms respectively) or ap if aperiodic, their activity services (in bold), as well as the attributes and function services of the component (in italic). The ports are labelled with their name and the data type they hold. Examining the overall architecture, one usually gets an idea of the behavior of the robot by checking the tasks and services implemented in each of these components, and the exchanged information between them through ports.

We now briefly present the five components of this complete UAV experiment for which the control is done at the motor level (i.e., we control the velocity of each single propeller). OPTITRACK is the component which produces the position of the drone as perceived by the Optitrack motion capture system in the mocap port (when flying outdoors, this component is replaced with an RTK GPS). MANEUVER is a trajectory planner for aerial robots. Given a position (with the goto request) or a sequence of waypoints (with the waypoint request), and given the current position in the port state, it outputs at 200 Hz the desired state port containing the intermediate position to reach. NHFC (Near-Hovering Flight Controller) implements a basic stabilization controller, as described in [Spica et al., 2013]. Given the ports state (current position) and the actual velocity of the propellers, it produces at 1 kHz the cmd velocity to apply to each propeller to reach the desired state (desired position). ROTORCRAFT is the low-level hardware controller, which applies at 1 kHz the cmd velocity to each propeller motor, and produces the actual velocity, the accelerations, and angular velocities in imu and the current magnetic heading in mag. Component POM (position manager) uses an Unscented Kalman Filter (UKF) to merge pose estimations from OPTITRACK (port mocap), and ROTORCRAFT (imu and mag) to provide the position of the platform in the state port at 1 kHz.\footnote{All the code of this experiment is available here: https://redmine.laas.fr/projects/drone-v-v/gollum/index.}

Overall, the Quadcopter experiment includes: 5 components, 7 ports, (9 + 5) tasks, 25 activity services (with automata), 38 function services, 50 attribute services, 141 codels and their WCET, amounting to 12k lines of C/C++ code (loc). The synthesized code by G\textsuperscript{es}M, to produce the PocoLibs components, amounts to 123k loc. All components run on an Intel NUC8i7INH CPU with a regular Ubuntu 18.04. Note that we also use a simulator based on Gazebo which simulates the drone flight when given the propeller velocity and produces simulated actual velocities, imu and mag, as well as optitrack mocap data.\footnote{The code of this simulator is available in robotpkg: http://robotpkg.openrobots.org/, in the simulation/mrsim-gazebo and simulation/mrsim-gazebo directories. A complete install procedure is given here: https://redmine.laas.fr/projects/drone-v-v/gollum/install.}

We also synthesize the Fiacre/H-Fiacre model which is 34 000 lines of Fiacre, including 198 Fiacre processes and 225 instances, 193 H-Fiacre tasks (start/sync) and 5 H-Fiacre event ports. The TTS obtained from the specification has 2 057 places and 3 136 transitions.

### 6.2 Offline verification

We are able to synthesize the complete Fiacre offline verification model of the UAV experiment with a simple yet meaningful client (i.e., initialization sequence and a goto request sent to MANEUVER). Yet, despite our efforts with various abstractions, we are not able to synthesize the complete reachable state of the entire system. So, in general, we cannot show that a particular property is always satisfied, but we can still show that some unwanted states may be reached, in which case, we can still work on the implementation to try to fix them. We can also limit the scope of our analysis to one component alone or two components at best. In any case, for complex critical applications, proving offline properties is valuable both as a process, as it sometimes explicitly shows some unexpected execution traces, and as the result which can be formally guaranteed.

#### 6.2.1 Schedulability of components

If we cannot show that all components are schedulable all together, we can still show that they are schedulable one by one for a given number of cores. This is not enough to prove the schedulability of the whole system. However, this result allows us to have some confidence in the system’s ability to meet deadlines. Indeed,
the system has more cores than components (eight cores for five components) which allows to guarantee the schedulability of the system if the components are independent, i.e., there is no synchronization between the elements of the different components. In our case, the components are not independent because they share data via the ports. This data sharing uses lock mechanisms to avoid data inconsistencies. However, there is no precedence between the components (i.e., the progress of a component does not depend on the execution of an element of another component). This interdependence between the components is therefore a simple temporal overhead on the reads and writes on the ports. It would be possible to estimate these overheads (see [Brandenburg and Anderson, 2010] for instance) and then to inject them into the WCET of the codels during the analysis of the components. This work has not been done and should be carried out in a systematic way. Let us note however that safety margins have already been integrated into the WCETs and that the time to write and read data is often low compared to the time to compute functions. For all these reasons, we can have some confidence in the schedulability of the system without proving it.

Moreover, studying the schedulability of each component one by one can also reveal some design or algorithmic problems, which can lead to code refactoring/correction. For example, while analyzing the schedulability of MANEUVER exec task, we found a potential sequence of the automata execution whose unpause sequence could lead to exceeding the period. Similarly, analyzing POM, the measured WCET for the exec codel of the filter permanent activity state is 1.1 ms which is clearly slightly above the limit to guarantee schedulability. Increasing the period to 2 ms made filter schedulable, but not the io task. Increasing it to 3 ms made them both schedulable. Yet, relying on average execution time, both tasks are schedulable at 1 kHz. For NHFC, the wo task is schedulable, but not the main task. Indeed, the control codel of its permanent task is 0.53 ms is too large within 1 ms period. Last, Rotorcraft has a variable in its ids which is shared between the two task comm and main which leads to problems which had to be addressed by reorganizing how these two tasks can share a variable without interlocking each other for too long (by adding a variable in the IDS and a state in the automata to copy the value).

6.2.2 Maximum bounded time between events

Another class of interesting property to prove is the maximum time between two particular events. For example, how long does it take, at worst, in the Rotorcraft component, between a client sending a stop request and the call to the mk_servo_stop codel from the servo activities which effectively stops the propeller motors. As shown on Listing 4, we can add to the synthesized Fiacre model of Rotorcraft an observer component which checks when a stop request is issued and then waits a given number of ticks (in our case a tick equal 0.1 ms). If, using model checking, for all possible executions of the rest of the component, we always reach motors_stopped, it means that the mk_servo_stop codel has been executed, and it will always take less than 5.6 ms to command the motors to stop after a stop request has been issued.

At first, we could not find a bound on the maximum value. After analyzing the counterexamples given by sift, given that the model checker searches a solution for any scheduling policy, we found out that there is a possible starvation due to the comm task mk_recv codel which can keep freeing/locking a resource needed by mk_stop. But, considering that in the real implementation, resources are handled with a FIFO ticket lock, we implemented it in the Fiacre model and found the maximum value to be 5.6 ms.

6.2.3 Proper services exclusion

As seen in section 3.2, services can interrupt other services. For example, in the Maneuver component goto and take_off interrupt each other (the programmer does not want them to run at the same time). We are able to check that this specification is properly implemented in the Fiacre models. For this particular example, we create two symmetrical scenarios, one where a take_off request is sent, followed by a goto request; and the other way around. We prove that our invariant holds, meaning that both services never run together (of course, this can then be proven for any pairs of mutually interrupting services).

We give some performance results about these verification tasks in Table 1. All computations were performed on a 3.5 GHz Intel Xeon computer with 32 GB of RAM. 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).
Listing 4: Example of user-defined event observer for component ROTORCRAFT.

```plaintext
process rotorcraft_Servo_Stopper(&stop_sent:bool, &propellers_stopped:bool) is

states wait_started, wait_delay, finished, finished_in_time, elapsed_delay

from wait_started
wait [0,0];
on (stop_sent); // stop sent by the client to the control task
to wait_delay

from wait_delay
wait [56,56]; // This is the response time we want to measure 5.6 ms
to finished // at 55 ticks elapsed_delay can be reached, at 56, it cannot.

from finished
wait [0,0];
if (propellers_stopped) then
to finished_in_time // The propeller have been stopped.
else
to elapsed_delay // The flag has not been set, so the propellers are still spinnig.
end // If this state can be reached, it means that the waited value is not large enough.
```

Table 1: Complexity of checking mutual exclusion between two activity services from MANEUVER

<table>
<thead>
<tr>
<th>Scenario</th>
<th>take_off then goto</th>
<th>goto then take_off</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td>52 sec</td>
<td>61 sec</td>
</tr>
<tr>
<td>#classes</td>
<td>3,025,904</td>
<td>2,965,966</td>
</tr>
<tr>
<td>#markings</td>
<td>909,177</td>
<td>893,075</td>
</tr>
</tbody>
</table>

6.2.4 State explosion problem

Since we rely on model-checking techniques, our approach can suffer from the so-called state explosion problem [Clarke et al., 2012]. We can identify some factors that impact the performance of our offline verification step.

A main source of complexity, when dealing with concurrent system, stems from the interleaving of independent actions. Models generated from $\mathcal{G}^{\text{en}}\mathcal{M}$ seem to be rather immune to this problem. Indeed, the behavior of components is mostly deterministic and has very few choice points. For instance, $\mathcal{G}^{\text{en}}\mathcal{M}$ enforces a precise order for the initialization of components, which helps control the problem. To give a general idea, the state space for the MANEUVER component (in the second scenario section 6.2.3) has about 8 million transitions, when there are only roughly 3 million states. Another source of nondeterminism, which is the main cause of problems in our case, are interactions with the environments, such as concurrent calls to services. When necessary, we can deal with this issue by limiting the capabilities of the environment, or by working with more constrained scenarios.

Apart from combinatorial complexity, we also need to consider the static, architectural complexity of our models. Indeed, model-checkers are memory-bound applications. Therefore, performance is adversely affected by the size needed to encode each state. For instance, the FIACRE model for the MANEUVER use case has 37 data variables, many of them storing arrays with non-trivial data structures, which amounts to about 600 bytes of memory for a single state. This is quite a lot more than with the models we generally handle when benchmarking model-checkers. Despite this, the peak memory usage observed in each scenario is only 1 GB, with only half of it dedicated to data values, and the other half for storing timing information and transitions. Hence the size of our model does not seem to be a limiting factor in our case. This result is achieved by using state compression techniques, such as sharing common data values between different states. We also provide a compressed, binary format for storing the state space of a model. With this format, we can store the resulting state space in an 18 MB file, instead of 1 GB in main memory.
Another limiting factor relates to the possibility, or lack thereof, to apply optimizations or to use more efficient verification methods. Our models are quite complex, with a heavy use of time constraints, priorities, and complex datatypes. As a consequence, it is not possible to use many classical optimizations, such as partial-order or symmetry reductions techniques (even though many of them are implemented in TINA). This also prevents the use of many symbolic model-checking methods. In this case, also, the situation is not entirely hopeless. We can often use on-the-fly techniques when we are looking for a particular counter-examples, which spares us the need to keep the whole state space in memory. On the other hand, when checking invariants, we can sometimes take advantage of methods to over-approximate the reachable states of the system. However, finding the right combination of methods and abstractions is a problem that requires the skills of formal verification experts. Finally, we can also rely on parallel and distributed techniques to speed up the verification of our models.

6.3 Runtime verification and monitoring

The H-FIACRE model of the 5 components, when linked with the codels library and the HIPPO engine running at 10 kHz, can fly the UAV without noticeable performance issues. We can check and monitor the generic properties presented in section 5.4 (logged and reported to the user): execution task period overshoot, codel WCET overshoot and uninitialized port read which are properly addressed. Most of the time, this is due to some transient situations which are not critical (e.g., missing a task period once every hundred cycles, does not have any visible effect on the drone, while missing 50 cycles in a row is critical).

On top of these generic properties, we also add some H-FIACRE code to monitor critical situations and take corrective (or emergency) actions. For example, we add two specifications which involve different components, hence which cannot be checked on each component separately.

- The first specification is to prevent the Rotorcraft stop service from being executed (hence stopping the servo activity if active) if the drone state position computed by POM is more than 30 cm above ground. This is to avoid the fall of the drone which would damage it.
- The second specification is when experimenting outside new control laws, to prevent the drone from flying too high and hitting the ceiling or the net of our flying arena. Again, if the state position computed by POM has an altitude higher than 10 m, than a priori, something is wrong and the drone should make an emergency landing.

Listing 5 implements the first specification. When a stop is received, by NHFC control task, this transition gets activated. But if the altitude_above_30cm is true, the request is de facto disallowed and reported as such without any action. Otherwise, execution proceeds as expected.

Listing 5: Excerpt from the FIACRE process which manages the stop service from the NHFC component. The altitude_above_30cm boolean is set by the POM filter task permanent activity when it updates the state port of the drone.

```plaintext
process CT_SERV_rotorcraft_stop (&control_turn: Rotorcraft_RID, ..., &altitude_above_30cm: bool) is

states start_, ..., report_exception

var genom_event: qued_genom_event := genom_ok_fcr

from start_

wait [0,0]; // rotorcraft: control task part of the stop request
on (control_turn = Rotorcraft_stop_RID); // this becomes true when a stop request is received by the CT
if (altitude_above_30cm) then // drone too high (>30cm) to stop servo without risk of damage.
genom_event := genom_disallowed_fcr; // the request is not allowed
to report_exception // and is reported as an exception to its client
end;
...
```
To test our first monitor, we position the drone at different altitudes and test that the \textit{ROTORCRAFT stop} is properly discarded by the \textsc{HiPPO} engine to prevent a crash of the drone.

The monitor on Listing 6 continuously checks, at \(10\,\text{kHz}\), if the \texttt{altitude\_above\_10m} flag becomes true. It then waits \(1\,\text{s}\) to confirm this was not a spurious value, and then another second before the \(2\,\text{s}\) have elapsed. If not, in the \texttt{monitor\_error} state, we build an internal \texttt{take\_off 0} request and send it to \texttt{MANEUVER} control task. This will force the drone to land gracefully.

Listing 6: Example of a user-defined monitor for component \textsc{NHFC}, to force an emergency landing when \texttt{POM} reports an altitude \(> 10\,\text{m}\).

```plaintext
process altitude_quad_lander [internal_request: out request\_maneuver\_type]
  (\&altitude\_above\_10m: bool, \&nhfc\_main\_override\_state: quad\_genom\_event,
   \&nhfc\_main\_activities: Activities\_nhfc\_main\_Array, main\_perm\_index: act\_inst\_nhfc\_main\_index\_type) is

states monitor\_start, monitor\_wait, monitor\_arm, monitor\_check, emergency\_landing

var internal\_rq: request\_maneuver\_type

from monitor\_start
  on (not altitude\_above\_10m); // from a safe state to monitor\_arm

from monitor\_arm
  on (altitude\_above\_10m); // WARNING, getting above 10 meters to monitor\_wait

from monitor\_wait
  wait [10000,10000]; // Wait one second to make sure this was not a spurious glitch to monitor\_check

from monitor\_check
  select /// if we stay above for one more second
    wait [10000,10000]; // 1s at 10 kHz = 10000 tick to emergency\_landing
    []
    on (not altitude\_above\_10m); // we are back below 10 to monitor\_arm
  end

from emergency\_landing // Emergency landing
  internal\_rq:=make\_maneuver\_take\_off\_internal\_request(0); // build a take\_off 0 request
  internal\_request!internal\_rq; // launch it to monitor\_start
```

To test this monitor we intentionally make the drone fly above \(10\,\text{m}\) and indeed, after \(2\,\text{s}\) it lands gracefully.

One may argue that we could add such monitoring code within the C/C++ code of the components. Indeed, still, there is no guarantee that the “safety” code will be invoked in time, and corrective actions will be taken as expected. While running a \textsc{HiPPO} formal model of the component, you can guarantee that the formal condition will be checked at \(10\,\text{kHz}\), and that the transition starting the corrective action will be activated right away in the same tick. In fact, we have shown in section 6.2.2 that we can give a bound on the response time of the system with an observer. Last, we see that the \textsc{HiPPO} model covers all the components, so it is easy to implement monitors which involve more than one component (e.g., \texttt{POM}, \texttt{ROTORCRAFT}, and \texttt{MANEUVER}).

7 Discussion, Future work and Conclusion

Our approach is designed for robotic software programmers, in the sense that it relies on a software engineering tool, \textsc{GenM}, developed and used by roboticists to specify and produce the functional components of robots. This specification tool goes beyond classical robotic frameworks such as \textsc{ROS}—which remains at the coordination level by focusing on topics, services and actions—as it also specifies how the component is implemented. We
also focus on performances and automation. \(G^{\text{\%}}\)M can be used to deploy components whose period and reaction time is in the order of the millisecond, and it relies on a template mechanism to automatically produce the component itself for various middleware.

We use the same mechanism to automatically synthesize a formal model of the implementation. Our approach uses the Fi\text{\textsc{acre}} language, which models dynamic timed processes using state machines, guarded transitions and ports. This alleviates the need for a robotic software engineer to master all these formal concepts. We make use of all the possibilities given by the Fi\text{\textsc{acre}} language. We use Ti\text{\textsc{na}}’s model checking tools for checking properties offline, at design time. We also use the Hi\text{\textsc{ppo}} framework for (online) runtime verification and for implementing reactive recovery functions. We have had experiences with other formal tools/frameworks, such as BIP, UPPAAL, UPPAAL-SMC... [Ingrand, 2021] yet, the Fi\text{\textsc{acre}} toolchain is the most advanced one and is observationally similar to the regular component poco\text{\textsc{libs}} or ROS implementation. Model checking has some limits, but still, the result we get on non-trivial examples are encouraging. Among our most notable results is the fact that we can perform runtime verification onboard a drone, using Hi\text{\textsc{ppo}}, at 10 kHz on a NUC, without visible impact on the flying behavior.

Another indirect benefit of our approach is that model checking properties returns a “counter example” when a property is invalidated. Considering the level of detail of the model we synthesize, the analysis of such counter-example and how we got it often shows an unexpected, at first sight, path of execution. Often, we had to go back to the \(G^{\text{\%}}\)M implementer and discuss with her/him if a particular behavior, discovered by Ti\text{\textsc{na}} within the \(G^{\text{\%}}\)M algorithms now implemented in Fi\text{\textsc{acre}}, was expected or not. Clearly, the power of model checking casts a new light on \(G^{\text{\%}}\)M internal algorithms and helps us refine and clarify the semantics of these algorithms.

A question that frequently occurs is how to “prove” the equivalence between a \(G^{\text{\%}}\)M specification and its semantics as provided by the formal models we synthesize? We cannot, in the same way one could not do it with the semantics of \(G^{\text{\%}}\)M obtained from its compiled algorithms, i.e., the behavior of its regular implementation in C/C++. Yet, \(G^{\text{\%}}\)M has been used for years, thanks to debugging and testing. After years of development, the observed behavior of \(G^{\text{\%}}\)M regular components is consistent with what the \(G^{\text{\%}}\)M programmers had in mind, and when it is not, they check the generic code and fix it. Assume now that we consider the H-Fi\text{\textsc{acre}} version to be the reference one. In fact, assume that \(G^{\text{\%}}\)M is now a tool to synthesize a formal model to run on robots. Currently, and as far as we can observe, it behaves exactly like the regular C/C++ version, and consider that from now on, we run this one in all our experiments. When we will observe a discrepancy or a bug in a \(G^{\text{\%}}\)M component, we will fix the H-Fi\text{\textsc{acre}} version (forget about the C/C++ version). How would that be different from what we did for years? At least we can see two major advantages. First, we run a formal model of the whole experiment (codel apart) hence a much better guarantee on added monitoring and runtime. Second, the verification model, which, apart from H-Fi\text{\textsc{acre}} event ports and tasks, is the same than the one run by Hi\text{\textsc{ppo}} would also gain from the fixes made to the H-Fi\text{\textsc{acre}} version as it is deployed (keep in mind they both come from the same template files). This verification model would also be still available for analysis and model checking, hence reinforcing the trust and dependability of the deployed components. So, even if we cannot prove the equivalence between \(G^{\text{\%}}\)M specifications along its semantics, and the formal models we synthesize, there is a lot to gain to consider the formal models both to run the components and to verify it, in particular if the runtime version does the job previously done by the regular C/C++ version.

Yet, our approach has some limitations. We do not go beyond (or inside) codels (i.e., \(G^{\text{\%}}\)M activity automata transition code), but we could rely on tools such as Frama-C used in [Pollien et al., 2021], or Isabelle used in [Täubig et al., 2011] to verify this small pieces of C/C++ code. Still, the arguments of the codels are part of the model, so we know which fields of the IDS or ports are used in the codel and how (in, out or inout), but their value is not part of the model. For offline V\&V, that would be of limited use as Fi\text{\textsc{acre}} does not support floating point values, and model checking has trouble scaling with such type. But for runtime V\&V it is straightforward to build the proper Fi\text{\textsc{acre}} functions to evaluate a particular condition based on the argument value (in fact, this is what we do in the examples section 6.3 when checking the altitude of the drone). Another limitation of the Hi\text{\textsc{ppo}} engine is that the current implementation executes on a single processor (but codels and other subroutines can execute concurrently, on multi-core processors). Distributing the model over multiple CPUs is theoretically possible, but the added communication latency will probably decrease the performances of the Hi\text{\textsc{ppo}} engine and make modeling for verification more complex.

Robotic software developers interested in using V\&V should check which architectures and frameworks...
are better suited for their needs. In section 2.4, we mentioned tools which clearly address these needs, but often at a cost of learning formal frameworks. In section 2.2, we show that nodes developed with ROS alone are not structured enough to provide a path toward verification. The best for robotic software developers is probably for now to rely on DSL (section 2.5), which could be (or already are, for some of them) connected to some formal frameworks and bring verification to their systems. The cleaner and stricter the DSL operational semantics is, the best one can expect from their connection to V&V tools. Moreover, it the chosen DSL and framework provide some automatic synthesis of the models (offline and online) as we do in our approach, the better. It will lower the cost of the entry ticket for verification and then validation of robotic systems.

As for future work, beyond the usual tasks, such as to improve the interface or the usability of the tool, we intend to move “upward” in the decisional functionalities of our robots and plan to implement an acting component using a similar method, i.e., harness an acting component using some skill languages which could be automatically translated to a formal model. There are some encouraging works in this direction from our colleagues [Lesire et al., 2020]. Indeed, the acting level is the one controlling, commanding, supervising the various functional components, so having a formal model which could encompass both levels would be great as we could then model some real skills execution and check properties considering the scenario on how the components are used together. Last, most robotic system activities are performed with some humans involved, as such, modelling the interactions with human in these skills, and taking them into account in the verification process needs to be addressed.

With robotic systems being deployed on a large scale among the general public, software developers should consider formal verification as a mean to prove and certify that the programs they write behave as specified. Thanks to a rigorous specification language—already used for a long time to deploy functional components—we show that we can harness a formal framework in order to perform both offline verification and runtime monitoring. One of the key aspects of the approach is to rely on an automatic synthesis of the formal model, equivalent to the implementation of the component. One primary goal is to lower the entry barriers to formal software verification, so roboticists can de facto use verification tools without being experts in these formalisms and methods. The approach has already been deployed on a number of realistic and large experiments. We present an original example of a UAV flight controller in Sect. 6. The interested reader can also find the description of a mobile rover navigation system in [Hladik et al., 2021]. Our experience is that, despite the size and complexity of the resulting formal models, we can prove critical property offline and can use the generated code in practice. Moreover, we can fly the UAV with the formal model running on board, a very demanding application, while monitoring critical properties online at 10 kHz without noticeable performance issues. These are very promising results which contribute to a more robust and verifiable autonomy of robotic systems.

References


A Synthesized FiACRE models

We present in this appendix some of the synthesized H-FiACRE/FiACRE models for the CT_robot component.14

Listing 7 shows an excerpt from the H-FiACRE model synthesized for the control task Algorithm 1 page 16 for the CT_robot component.

Listing 7: Control Task in H-FiACRE.

```plaintext
process CtrlTask_CT_robot (&control_activity: activity_CT_robot,
&TA: Activities_CT_robot_track_Array, &run_map: CT_robot_Run_Map_Type,
&track_runnable: bool, &control_turn: CT_ROBOT_RID, &schedule_cntrl_task:bool) is
5
states loop_back, get_request, wait_until_done, activity, error,
process_exec_task_track, process_exec_task_, process_pending_activities

var genom_event:CT_robot_genom_event := genom_ok, rq: request_CT_robot_type,
it: act_inst_CT_robot_track_index_type, mess:bool, control_rqst:bool

from loop_back
on (control_turn = CT_ROBOT_NONE); // Control Task has the running token
to get_request

from get_request // CtrlTask going to sleep
select //either we get a new request, or an Execution Task wakes us up
CT_robot_Manage_Mbox?rq; // event port where we get new requests
to activity // CtrlTask got a message

[]
on (schedule_cntrl_task); // CtrlTask has been woken up by an ExctnTask.
schedule_cntrl_task:=false;
to process_pending_activities
end

from activity
...
// control_activity fields are initialized for this instance
control_turn := rq.rqid; // running token is passed to the process handling this request
to wait_until_done

from wait_until_done
on (control_turn = CT_ROBOT_NONE ); // wait for the running token to be passed back to us
to process_pending_activities // Processing the execution task activities (track)

from process_pending_activities
it := 0;
to process_exec_task_track

from process_exec_task_track // we processes all track activities.
if (TA[it].status = INIT) then
if (not TA[it].start_) then // CtrlTask setting ExctnTask track runnable to true.
track_runnable := true;
TA[it].start_:=true
end
end;
if (TA[it].status = ETHER) then // we are done for this activity
if (not (TA[it].sid = CT_ROBOT_NONE)) then //this is not the perm activity
if (TA[it].state = CT_robot_ether) then //CtrlTask processes track activities, updating run_map.
run_map[CT_robot_run_map_index_from_track_activity(TA[it])]:=true
else // CtrlTask processes track activities, handle exception.
CT_robot_handle_exception(TA[it], false)
else // CtrlTask processes track activities, activity report.
CT_robot_activity_report(TA[it])
end;
```

14Some variables have been renamed to make the code more compact, and most comments have been added to explain the model and are not in the original models that are available here: https://redmine.laas.fr/projects/ct_robot/repository/ct_robot/revisions/master/show/vv.
TA[it].status := VOID
end; // VOID, RUN, STOP do nothing
if (it = Nb_act_inst_CT_robot_track-1) then to process_exec_task_track
else
  it := it + 1;
  to process_exec_task_track
end

from process_exec_task_track // we are done will checking all the execution task activities
to loop_back

Listing 8 shows an excerpt from the H-FiACRE model synthesized for the Function CTServ page 16 for the ColorTrack activity.

Listing 8: H-FiACRE of a service (control task part).

process CtrlTask_SERV_CT_robotColorTrack (&control_activity: activity_CT_robot,
&TA: Activities_CT_robot_track_Array, &control_turn: CT_ROBOT_RID,
&track_running_codel: 0..max_number_mutex_codels_CT_robot-1,
&track_runnable: bool, &run_map: CT_robot_Run_Map_Type,
&control_running_codel: 0..max_number_mutex_codels_CT_robot-1) is

states start_, check_run_maps, disallowed, validate, validate_sync, inter, inter_task_,
inter_task_track, wait_track_interrupted_ether, next_track_activity, report_exception,
finish, report_or_transfer

var it: act_inst_CT_robot_track_index_type, ffa_reply_track: ffa_reply_CT_robot_track,
genom_event: CT_robot_genom_event := genom_ok

from start_ // ColorTrack waiting the running token
on (control_turn = CT_ROBOT_ColorTrack_RID); // from the control task
to check_run_maps

from check_run_maps // corresponds to CheckBeforeAfter
if (not run_map[CT_ROBOT_SetPatrolSpeed]) then to disallowed end; // Check SetPatrolSpeed

to validate // has run.

from disallowed
control_activity.state := CT_robot_disallowed();
CT_robot_ColorTrack_activity_report(control_activity, true);
to finish

from validate // corresponds to the call to the validate codel
control_running_codel := ValidateColor; // The following start calls
start CT_robot_ColorTrack_validate_task(control_activity); // the validate codel
to validate_sync

from validate_sync
sync CT_robot_ColorTrack_validate_task genom_event; // wait the end of the validate codel
control_running_codel := 0; // no codel running
if (genom_event = genom_ok) then to inter // CtrlTask ColorTrack validate OK
else to report_exception end // CtrlTask ColorTrack validate NOT OK

from inter // interrupt initialize the loop over all activities of track
it := 0;
to inter_task_track

from inter_task_track // interrupt all the activities which need to be interrupted
if (not (TA[it].status = VOID) and not (TA[it].status = ETHER)) then
  if ((TA[it].sid = CT_ROBOT_ColorTrack_RID)) then // CtrlTask
    TA[it].stop := true; // ColorTrack interrupts a running
    to wait_track_interrupted_ether // activity in track.
  end
end
end;
to next_track_activity
from wait_track_interrupted_ether // will wait until the interrupted activity has finished
    on (TA[it].status = ETHER); // CtrlTask ColorTrack interrupted
    to next_track_activity // service in track has now reached ETHER.

from next_track_activity // next activity
    if (it = Nb_act_inst_CT_robot_track-1) then to inter_task_ else it := it + 1;
    to inter_task_track end

from inter_task_ // done interrupting
    to report_or_transfer

from report_or_transfer
    control_activity.pause := false; // we reset it
    ffa_reply_track := find_free_activity_CT_robot_track(TA, CT_ROBOT_ColorTrack_RID);
    it := ffa_reply_track.index; // ColorTrack is transferred to its ExctnTask track.
    genom_event := CT_robot_send_ir(control_activity);
    ... // removed code for brevity, TA[it] fields get initialized
to finish

from report_exception // CtrlTask ColorTrack report NOT OK.
    control_activity.state := genom_event; // to transmit the error
    CT_robot_ColorTrack_activity_report(control_activity, true);
to finish

from finish // CtrlTask ColorTrack for all ET, setting runnable to true.
    track_runnable := true;
    control_turn := CT_ROBOT_NONE; // Give the running token back to the control task
to start_

Listing 9 shows an excerpt from the H-FIACRE model synthesized for the Execution task Algorithm 2
page 17 for the track task.

Listing 9: Execution task in H-FIACRE.

process ExctnTask_CT_robot_track (&tick_track : nat, &track_runnable : bool,
    &track_turn : act_inst_CT_robot_track_turn_type,
    &TA : Activities_CT_robot_track_Array, &schedule_cntrl_task : bool) is

    states start_, epilogue, parse, exec_prepare, wait_next_cycle, each_activity, next_activity,
        wait_activity

    var it: act_inst_CT_robot_track_index_type, i: act_inst_CT_robot_track_index_type,
        status: activity_status, sched: SCHED_Type := [false, false, true, false] //SCHED_CONT true

    from start_
        if (not sched[SCHED_CONT]) then // ExctnTask track, setting runnable to true.
            track_runnable := true;
            to wait_next_cycle
        else track_runnable := false end; // ExctnTask track, setting runnable to false.
        to parse

    from wait_next_cycle // ExctnTask track waiting next period.
        on (tick_track); // the timer process sets tick_track at the period declared for track
        tick_track := 0; // to inform the timer to rearm
        to parse

    from parse // ExctnTask track parsing activities status.
        sched := [false, false, false, false];
        foreach it do // code equivalent to the switch in Execution task algo
            if (TA[it].status = INIT) then
                if (TA[it].start_ and track_runnable) then
                    TA[it].status := RUN; TA[it].state := CT_robot_start;
                    TA[it].start_ := false; TA[it].pause := false
                end;
                end;
            if (TA[it].stop then
                if (TA[it].sid = CT_ROBOT_NONE) then /* permanent activity */
                    TA[it].status := STOP; TA[it].state := CT_robot_stop;
TA[it].stop := false; TA[it].pause := false
else
  TA[it].status := ETHER; TA[it].stop := false;
  TA[it].state := genom_interrupted;
  CT_robot_genom_interrupt(TA); sched[SCHED_DONE] := true
end

elsif (TA[it].status = RUN) then
  if TA[it].stop then
    TA[it].status := STOP; TA[it].state := CT_robot_stop;
    TA[it].stop := false; TA[it].pause := false
  end;
  if (TA[it].pause and track_runnable) then TA[it].pause := false end
elsif (TA[it].status = STOP) then
  if (TA[it].pause and track_runnable) then TA[it].pause := false end
end

// ExctnTask track, setting runnable to false.
track_runnable := false;
to exec_prepare
from exec_prepare
i := 0; // ExctnTask track executing activities.
to each_activity
from each_activity
  if ((not (TA[i].status = RUN)) and (not (TA[i].status = STOP))) then
    to next_activity // ExctnTask track activity not RUN and not STOP
    end;
    if (TA[i].pause) then to next_activity end; // we move to the next one
    track_turn := i; // ExctnTask track gives the running token to the activity
    to wait_activity
  from wait_activity
    on (track_turn = Nb_act_inst_CT_robot_track); // ExctnTask getting running token back
    case TA[i].status of
    RUN -> // ExctnTask track activity returned RUN.
      if (not TA[i].pause) then sched[SCHED_CONT] := true; sched[SCHED_WAKE] := true end
    | ETHER -> // ExctnTask track activity returned ETHER
      sched[SCHED_DONE] := true; sched[SCHED_WAKE] := true;
      schedule_cntrl_task := true
    | any -> // ExctnTask track ***ERROR*** unexpected activity status.
      null
    end;
    sched[SCHED_ONE] := true;
to next_activity
  from next_activity
    if (i = Nb_act_inst_CT_robot_track-1) then to epilogue
    else i := i + 1; to each_activity end
from epilogue
  if (sched[SCHED_WAKE]) then track_runnable := true end;
  if (sched[SCHED_DONE]) then schedule_cntrl_task := true end;
to start_

Listing 10 presents an excerpt from the H-FlACRE model synthesized for the Function Invoke page 18 for the ColorTrack activity.
states CT_robot_start, CT_robot_start_sync, CT_robot_start_dispatch,
    // for code brevity, removed similar code for states stop, Lost, CompCmd and PubCmd
exception

var state: CT_robot_genom_event := genom_ok
from CT_robot_start
15
on (track_turn = cti); // Activity ColorTrack is getting control in state CT_robot_start.
    if (TA[cti].state = CT_robot_stop) then to CT_robot_stop end;
    track_running_codel := GetImageFindCenter; // Activity ColorTrack calling codel GetImageFindCenter.
    start CT_robot_codel_service_ColorTrack_start_task(TA[cti]);
to CT_robot_start_sync

from CT_robot_start_sync_ // waiting GetImageFindCenter is done
    sync CT_robot_codel_service_ColorTrack_start_task_start state; // Activity ColorTrack returned
    track_running_codel := 0; // from codel GetImageFindCenter.
to CT_robot_start_dispatch_

from CT_robot_start_dispatch_
if (state = CT_robot_Lost or state = CT_robot_CompCmd) then
    TA[cti].state := state; TA[cti].status := RUN; // Activity ColorTrack NOT done for
    track_turn := Nb_act_inst_CT_robot_track; // this cycle
    if (state = CT_robot_Lost) then to CT_robot_Lost end;
    if (state = CT_robot_CompCmd) then to CT_robot_CompCmd end;
else
    if (state = CT_robot_pause_start) then
        state := CT_robot_start; TA[cti].state := state; TA[cti].status := RUN;
        TA[cti].pause := true; // Activity ColorTrack DONE for
        track_turn := Nb_act_inst_CT_robot_track; // this cycle
        if (state = CT_robot_start) then to CT_robot_start end;
    else
        if (state = CT_robot_ether) then // Activity ColorTrack DONE
            TA[cti].state := CT_robot_ether; TA[cti].status := ETHER;
            track_turn := Nb_act_inst_CT_robot_track; //running token passed back
            to CT_robot_start
        else to exception end // Activity ColorTrack exception.
end
... // removed code similar to start for stop, Lost, CompCmd and PubCmd for code brevity
...
from exception // activity colortrack throws an exception
if ((not (state = CT_robot_bad_cmd_port)) and
... // removed all other exceptions for code brevity
    (not (state = CT_robot_bad_color))) then
    CT_robot_colortrack_genom_bad_transition(TA[cti]);
    state := genom_bad_transition
end;
    TA[cti].status := ether;
    TA[cti].state := state;
    track_turn := nb_act_inst_CT_robot_track; // return control to the execution task
to CT_robot_start

Listing 11 shows the differences between the H-FIACRE (Listing 10, between line 14 and line 25) and FIACRE models synthesized for the start state of the ColorTrack activity. The H-FIACRE code of the CT_robot_start state adds the start codel CT_robot_codel_service_ColorTrack_start_task call (line 19), while the FIACRE code of the CT_robot_start_sync_ state has a select (line 12) with four distinct outcomes, instead of the sync waiting the codel return.

Listing 11: H-FIACRE/FIACRE differences.
if (TA[cti].state = CT_robot_stop) then to CT_robot_stop end;
5
on ((not (control_running_codel = genom_CT_robot_SetThreshold_controlcb))):
track_running_codel := GetImageFindCenter;
to CT_robot_start_sync_

from CT_robot_start_sync_ // the main difference between Fiacre and H-Fiacre model is here
10 wait [0,6]; // The wait simulates the time the codel GetImageFindCenter may take 0.006s
track_running_codel := 0; // instead of getting the real next state from execution,
select // the model checker can choose among the possible next state
    state := CT_robot_pause_start
[]
    state := CT_robot_Lost
[]
    state := CT_robot_CompCmd
[]
    state := CT_robot_ether
20 end;
to CT_robot_start_dispatch_

Listing 12 presents the H-FIACRE model synthesized for the track task timer process. The wait[10,10]
corresponds to 10 ms (the period of track).

Listing 12: Execution task timer in H-FIACRE

process Timer_CT_robot_track (&tick_track: nat) is

states wait_once, rearm

5 from wait_once
wait [10,10]; // track timer elapsed once after exactly 10 ms.
tick_track := 1; // this will wakeup the execution task
to rearm

10 from rearm
    on (tick_track = 0); // the execution task will reset it when starting
to wait_once