Verification of Autonomous Robots: A Roboticist's Bottom-Up Approach

Félix Ingrand

Verification of Autonomous Robots: A Roboticist's Bottom-Up Approach*

Félix Ingrand

Abstract Autonomous robots may be one day allowed to fly or to drive around in large numbers, but this will require their makers and programmers to show that the most critical parts of their software are robust and reliable. Moreover, autonomous robots embed onboard deliberation functions. This is what makes them autonomous but opens for new challenges. There are many approaches to consider for the V&V of AR software, e.g. write high-level specifications and derive them in correct implementations, deploy and develop new or modified V&V formalisms to program robotics components, etc. One should note that learned models aside, most models used in deliberation functions are already amenable to formal V&V. Thus, we rather focus on the functional level components or modules and propose an approach that relies on an existing robotics specification and implementation framework ($\mathsf{G^{en}_{o}M}$), in which we harness existing well known formal V&V frameworks (UPPAAL, BIP, FIACRE-TINA). $\mathsf{G^{en}_{o}M}$ was originally developed by roboticists and software engineers, who wanted to clearly and precisely specify how a reusable, portable, middleware independent, functional component should be specified and implemented. As a result, $\mathsf{G^{en}_{o}M}$ has a rigorous specification, a clear semantics of the implementation and it provides a template mechanism to synthesize code that opens the door to automatic formal-model synthesis and formal V&V (offline and online). This bottom-up approach, which starts from components implementation, is more modest than the top-down ones which aim at a larger and more global view of the problem. Yet, it gives encouraging results on real implementations on which one can build

Félix Ingrand
LAAS/CNRS, University of Toulouse, Toulouse, France, e-mail: felix@laas.fr, http://homepages.laas.fr/felix

* This work has been supported by the European Union's Horizon 2020 research and innovation programme under grant agreement No. 825619 (AI4EU) and the Artificial and Natural Intelligence Toulouse Institute - Institut 3iA (ANITI) under grant agreement No: ANR-19-PI3A-0004.

more complex high-level properties to be then verified and validated offline but also with online monitors.

1 Introduction

Validation and Verification (V&V) of Autonomous Systems (AS)[2] software is not a "new problem". More than 20 years ago, a seminal work (Espiau et al, 1996) started a study on how to guarantee robustness, safety and overall dependability of software. Yet, for several reasons, the robotic and AI communities have mostly been focussed on other problems with respect to safe dependable autonomous robots. Meanwhile, there are other fields where bugs and errors can lead to catastrophic events (e.g. aeronautic, nuclear industry, rail transportation) where there is already a large corpus of research, and also successfully deployed tools and frameworks (Woodcock et al, 2009), whose goal is to improve the trust we can put in the software controlling these complex, although not autonomous, systems.

The fast and recent developments of autonomously driving cars have put the spotlight on the dramatic consequences of unverified software. Unfortunately, there is no doubt that autonomous vehicles will cause deadly accidents. They will only become "acceptable" if the carmakers have deployed all reasonably applicable and available techniques to ensure trust and robustness, and if as a result of these techniques, they outperform a regular human driver by one or two orders of magnitude. The decision to deploy these techniques may come from the carmakers themselves, as a commercial argument to safety, or as an incentive from car insurances, or they may come from government certification agencies representing the general public concerns about safety (as this is already the case for aeronautic, railway, etc). In any case, despite the somewhat human-biased argument that we are "all" good drivers, autonomous cars will probably prevail if 5, 10 or 20 years from now, statistics show that they are indeed safer than human driven cars.

One original aspect to consider with respect to AS is that, unlike most critical systems in other domains, AS exhibit and use deliberative functions (e.g. planning, acting, monitoring, etc) (Ingrand and Ghallab, 2017). If one considers the models used by these deliberative functions, some are explicitly written by humans, while others are learned (Argall et al, 2009; Kober et al, 2013). Similarly for functional components, some also use learned models. We will see that if the explicit models are amenable to formal verification, the learned ones pose a new challenge to the V&V community.

For now, most of the trust we put in the AS Software (ASS) is acquired through test (Sotiropoulos et al, 2017; Koopman and Wagner, 2016). Moreover, there are a number of "good" practices, architectures design, soft-

[2] We consider here autonomous systems at large, i.e. including autonomous robots, autonomous vehicles, drones, cyber physical systems, etc.

ware development methodologies, model-based techniques (Brugali, 2015; Mühlbacher et al, 2016), and specification tools that all contribute to establish this trust. Still, formal V&V, when applicable, has the potential to bring a level of confidence unreachable by other practices.

We also need to focus on approaches addressing realistic applications with real implementations and experiments. We already have reached the point where AS with tens of sensors and effectors, executing millions of line of code, running tens of programs on multiple CPUs, are being deployed. The time where one could illustrate an approach with an example, over simplifying reality, and limited added value to the field has passed. Proving that a Lego Mindstorm will not crash in the wall, thanks e.g. to its LTL generated controller, is not quite the same problem than showing that an autonomous car is not going to do the same.

Moreover, the dependability of the software should be considered as a whole, looking at the overall complete system. When it comes to V&V, unless you take adapted protective containment measures between components, the overall system will be, at best, as strong as your weakest link or component. The way the components are being organized, communicate and share resources in the architecture is as critical as the components themselves.

There are some recent valuable surveys available, each with their own reading grid and focus. Their coverage of the decisional components may be limited (Luckcuck et al, 2018), or they are limited to the decisional components (Seshia et al, 2016), or they present a larger safety picture (Guiochet et al, 2017), (Fisher, 2021, chap. 7), but no formal V&V. Still, they are a good source of information in this fast-growing field.

Last, our perspective is definitely from a roboticist's point of view. First, we want to rely, as much as possible, on automatic synthesis of models and code. Second, we are aiming at proving properties which are "useful" for the ASS programmer. Can we guarantee that the plan produced by the planner is safe and that it will be properly executed by the acting system? Can the CPU resources available on the robots guarantee that all components will run fast enough? Can we guarantee that the robot will stop in time when an obstacle has been properly detected, that the initialization sequence will not deadlock, etc? Overall, starting from some real ASS implementations, what is the current status with respect to V&V and how can we improve it?

This chapter is organized as follows: Section 2 presents the V&V models and techniques that we think are relevant to ASS, while Sect. 3 reviews the various situations with respect to availability of formal models in an AS. We then present in Sect. 3.6 some of the robotic software specification frameworks which could be transformed towards a formal model. Section 4 introduces $\mathsf{G^{en}_{\circ}M}$, a tool to specify and deploy software functional component, and its template mechanism. A complex robotics example using $\mathsf{G^{en}_{\circ}M}$, is introduced in Sect. 5, followed by Sect. 6 which presents the four formal frameworks for which $\mathsf{G^{en}_{\circ}M}$ can synthesize a model. Section 7 illustrates some of the V&V

results we obtain both offline and online. Section 8 concludes the chapter and presents some possible future topics of research.

2 Formal Models and V&V

Our goal here is not to survey such a large field of research. We point the reader to (D'Silva et al, 2008; Woodcock et al, 2009; Bjørner and Havelund, 2014) for overviews of formal methods in software development[3]. Formal methods use mathematical or logical models to analyse and verify programs. The key point here is that these models can then be used rigorously to prove properties of the modelled program. Of course, formal methods can cover various parts of the program life cycle, from the specifications down to the code. Here, we mostly focus on approaches that are close to the deployed and running code.

2.1 Models and Methods

There are many formal models available, and none of them covers all the needs. 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 widely used in UPPAAL, Kronos and more recently in the latest BIP version. Other models are provided as languages defined at a higher level of abstraction, such as the synchronous system family, but can be translated to mathematical or logical representation. Such a category also includes temporal logic (Kress-Gazit et al, 2011), situation calculus (Levesque et al, 1997; Claßen et al, 2012) as well as interval temporal logic also deployed on robots in various components. There are also methods geared towards hybrid systems (Tomlin et al, 2003).

From a roboticist's point of view, we should consider the models and methods that seem the most appropriate to represent the type of behavior we have to model and to prove the type of properties we want to check.

[3] Note that none of these three recent surveys mention robots nor AS.

2.2 V&V Approaches

Among the various techniques available to the V&V community, we focused on four different families.

State exploration and model checking. These approaches, given an initial state and a transition function, explore offline the reachable state of the system. The search space can sometimes be studied without being completely built (e.g. by finding a finite set of equivalence classes), but most often, these approaches suffer from state explosion and often face scalability issues.

Statistical model checking (SMC). These approaches, instead of exploring the complete state space, sample it using a probability model of the transition function, and thus evaluate properties to be verified with a resulting probability. Indeed, there are many properties which are desirable, but not required 100% of the time. For example, if your drone flight controller, running at 1 kHz, looses one cycle every one hundred cycles, the consequences are probably not as dramatic as if it looses one cycle every other cycle. So SMC approaches allow one to "explore" states space whose size blows the regular state exploration techniques.

Logical inference. These approaches works offline by building an overapproximation of set of reachable states as a logical statement (e.g. obtained by combining invariants of components). So the set is explored by checking logical properties, not its individual element. If these approaches can potentially address the state explosion, which breaks model checking, they face another problem as the logical invariants can be too general and too loosely fit the real reachable states set.

Last, *runtime verification* is more an online approach where the state transition model is given to an engine that monitors and checks properties and consistency of the model on the fly. This approach requires specification of what needs to be done when a property is violated, but allow verification of models that would not scale, nor fit with the three previous approaches.

We shall now consider the current availability of these models and V&V approach in ASS.

3 Autonomous System Software and Formal Models

With respect to the availability of formal models within ASS, here we examine the situation overall (i.e. the architecture), but also the various type of software components and how they rely (or not) to formal models: programmed with formal models (knowingly or not); learned models; no model and some models.

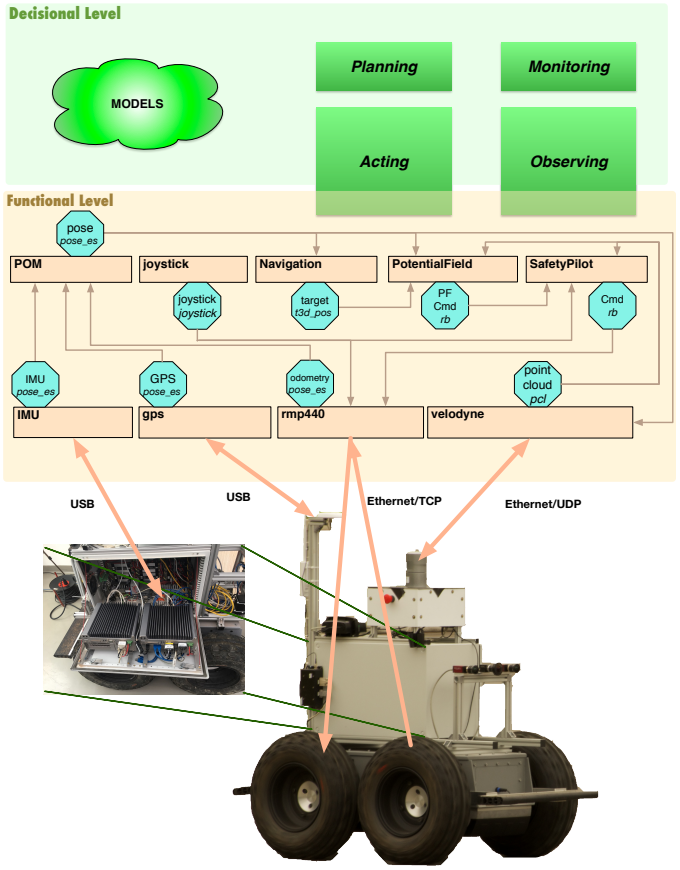

Fig. 1 Overall Architecture of an Autonomous Robot (RMP440).

3.1 Software Architecture

ASS needs to be organized along a particular chosen architecture framework. See (Kortenkamp and Simmons, 2008) for a survey of the different architectures available for AS. Some architectures help and ease the V&V of the overall system, by precisely defining how the layers are organized and how the components interact. To keep things simple with respect to architecture, we consider two levels: the Decisional Level and the Functional Level (see Fig. 1). As a result, we often find two types of software components:

- those performing one of the deliberative functions, that often rely on models (e.g. planning actions, fault detection models, acting skills) that are then used to explore the problem space and find a solution (e.g. a planner combines actions models to find a plan, an fault detection component

monitors the state of the system to find discrepancies with its model, etc), and
- those performing some data and information processing to solve a problem or provide a service through a regular algorithm implementing the function (image processing, motion planning, etc).

In the first category, algorithms involve heuristic searches in a space that cannot be reasonably explicitly computed, while the second make the search for the solution explicitly through deterministic algorithms. The algorithm in the first category may take an apriori unbounded amount of time, while the ones in the second usually have a predictable global computation time. The software in the first category often rely on high-level abstraction models, while the second are often programmed in classical programming languages.

As a result of this dichotomy, the V&V of these different components may rely on different approaches. We now examine the different situations with respect to the availability of models suitable for V&V.

3.2 Directly Programming with Formal Models

We consider here systems or software components that are programmed "directly" using some well established formal frameworks, which may be used in other domains.

For example, the "synchronous approach" (Benveniste and Berry, 1991; Benveniste et al, 2003) has been instantiated in a number of languages (e.g. Lustre, Esterel, Signal, etc) and commercial frameworks. It has been deployed in critical domains such as, but not limited to, aeronautic (Scade, Lustre and Esterel) and electronic (Signal). Esterel (Boussinot and de Simone, 1991) is used in (Simon et al, 2006), which presents a framework (Orccad) to deploy robotics systems programmed in the Maestro language (then translated in Esterel) to specify *Robot Tasks* and *Robot Procedures*. This was used to successfully program a complete AUV, and prove some properties on the system. Yet, the Maestro language was probably too abstract for robotic system programmers, which did not adopt it.

Similarly, there are decisional components that are programmed using situation calculus. Such a formalism has been used in Golog (for planning)(Levesque et al, 1997; Claßen et al, 2012) and Golex (Hähnel et al, 1998) (for acting) within a museum guide robot. However, the language is quite cumbersome, and has hardly been used for other deployments.

There are numerous other formalims available in the V&V community (e.g. BIP, LTL, UPPAAL, etc) which could have been used to program directly some robot components, but apart from toy examples, or localized and limited functionalities, this was not attempted. Similarly, there are formal tools to help the specification and the analysis of ASS (e.g. ALTARICA (Cassez et al, 2004)), but they are not easily linked to the real code running on the robot.

RoboChart (Cavalcanti, 2017; Miyazawa et al, 2017; Ribeiro et al, 2017), (Cavalcanti, 2021, chap. 9) proposes an interesting approach where the programmer explicitly models the robotics application in a formal framework based on timed communicating sequential process. So the model is provided by the programmer who must have some familiarity with formal models and languages. In (Hierons, 2021, chap. 11), they extend this framework to automatically generate test cases, while (Woodcock, 2021, chap. 13) proposes an extension to model uncertainty using probability.

Overall, in this category we find some interesting attempts to bridge the gap between ASS and V&V, but clearly, the languages proposed and the required knowledge and skills to make good use of them is a challenge robotic programmers must overcome.

3.3 "Hidden" Formal Models

In this category we consider the software components that are programmed using some formalisms on their own, but that have not been explicitly used with V&V methods.

For example, PDDL models widely used in automated *planning*, have all the right features to be considered formal models. Even if PDDL expressiveness can be somewhat limited when it comes to real robotics application, the semantic is clear and unequivocal, and most heuristic searches used to find plans, are correct (if not complete, nor optimal). Similarly, there are other planning formalisms (ANML, HTN, TLPlan, TalPLannner, situation calculus, temporal interval logic, etc) (Ingrand and Ghallab, 2017) that all fall in the same category. They were all designed for planning but they have all the right features to be used for V&V. Some works, e.g. (Abdeddaim et al, 2007; Bensalem et al, 2014), explicitly study the link between V&V and planning & scheduling.

Similarly, if we consider the *acting* component, which is more concerned in the operational model (skill) of how to execute an action (as opposed to planning, which is more concerned with a model of how to use it) there are also many works. ASPiC (Lesire and Pommereau, 2018) is an acting system based on the composition of *skill Petri nets*. (Simmons and Pecheur, 2000) shows that TDL based acting components can be verified using NuSMV. RMPL (Williams and Ingham, 2003) (which relies on an Esterel-like language) is also used for acting and monitoring. RAFCON (Brunner et al, 2016) also provides mechanisms to model and execute plans produced by high-level planners. Similarly, in (Pelliccione, 2021, chap. 12) the authors propose to program the acting part of the robotics system with a DSL, relying on LTL, which expresses various control structures and event-handling templates.

So there are many components (mostly deliberative within the decisional layer) that rely on some models that can be linked or transformed to some

of the formal models used for V&V. Such transformation is seldom used, but is an option that could be easily activated if needed.

3.4 Learned Models

Models acquired through learning are of a different kind. They are popular because they successfully tackle problems that resisted analytical modelling and solving. In an AS, one can learn a skill for *acting* (e.g. using reinforcement learning (Kober et al, 2013) or DBN (Infantes et al, 2010)), an action model for *planning*, (e.g. using MDP), or a perception classifier (e.g. using deep learning and convolutional neural network), but from a V&V point of view, these learned models are mostly black boxes.[4]

Still there are some attempts to improve the dependability of learned models. In (Amodei et al, 2016), the authors identify five design pitfalls that can lead to "negative side effects" and they propose some guidelines as to prevent them. Unfortunately, none of them rely on formal methods, and if we can expect better models following them, there is no guarantee that false-positive will not slip in.

Cicala et al (2016) present three areas of ASS where they propose an automatic approach to do V&V. One of the area is Safe Reinforcement Learning for which they propose to deploy probabilistic model checking on discrete-time model Markov chains. Similarly Seshia et al (2016) identify five challenges, to formally verify system that use AI and Machine Learning: Environment Modeling; Formal Specification; Modeling Systems that Learn; Computational Engines for Training, Testing; and Verification and Correct-by-Construction Intelligent Systems. But overall, researchers are just starting to look at these issues, and the inclusion, or not, of learned models in ASS will depend on their success.

For now, we think that these models must be confined to non-critical components, and if not, their results should be merged, combined and checked for consistency with other results before being considered as input in a critical decisional process.

An interesting work is proposed by (Feth et al, 2018) where they learn a model to help identify situations which require a higher level of awareness of the system.

[4] Note that if learning itself is considered a deliberative function, the learned models, can be used within different components, functional or decisional.

3.5 No Model

In many situations, the code and the programs are written following some hopefully good programming practices, but overall, there are no model of what it does. The code is the model. Still, there are a number of tools that make thorough checking of the code with static analysis and even some invariants extraction (D'Silva et al, 2008). Even more, if formal V&V is really required, one can deploy an approach such as the one presented in (Täubig et al, 2011), which requires one to annotate all the functions in the program with logical preconditions, assertions and effects, which will then be checked and inferred by the formal tool (Isabelle). This is a rather tedious process, and can only be done by people that are both familiar with the formal frameworks used *and* that understand the algorithms being implemented. Nevertheless, the results are very encouraging.

3.6 Some "Specification" Models

In this category, we consider all the components that are somehow specified with some languages, or model-driven frameworks that are not formal. For example, most robotic domain-specific language DSL are seldom formal. One often uses the term "semi formal" in the sense that they have a clear syntax, but their semantics is often ambiguous, which prevents them from being directly fed to some V&V engine.

There exist numerous framework to develop and deploy robotic software (Nordmann et al, 2016; Brugali, 2015). Some offer specification languages, or rely on well-known specification framework (e.g. UML, AADL, etc). Some just focus on providing tools and API libraries to ease integration of different components (Brugali, 2021, chap. 1). Orocos (Bruyninckx, 2001) focuses on real-time control of robots. (Dhouib et al, 2012; Yakymets et al, 2013) presents RobotML a robotic domain-specific language (Papyrus) and toolchain (based on Eclipse Modeling Project) that facilitates the development of robotics applications. Smartsoft (Schlegel et al, 2009), (Schlegel, 2021, chap. 3) provides a framework to also specify the complete architecture of a robotic system, while (Lotz et al, 2016) provide a meta-model to separate user programmer concerns, and system integrators issues. But if these tools greatly ease the overall architecture and analysis of the system, they remain short of connecting to a formal model.

Despite its success, ROS provides little support when it comes to ease V&V of the software with formal models. There are some efforts to model its communication layer (Halder et al, 2017), or to verify some simple properties (Come et al, 2018; Meng et al, 2015; Wong and Kress-Gazit, 2017), but overall, the lack of structure required to write ROS nodes makes it rather difficult to extract anything worth verifying. Bardaro et al (2018) propose

to model the robot software in AADL and then synthesize code in ROS. There are also systems that build some "run-time" verification of properties on the top of ROS (Huang et al, 2014; Sorin et al, 2016), but they hardly rely on ROS itself. Kai et al (2017) propose model transformation using MontiArcAutomaton from a high-level specification down to ROS code. Last we should point out that the SMACH (Bohren and Cousins, 2010) component can be used to control ROS nodes with state machine models.

MAUVE (Gobillot et al, 2016; Doose et al, 2017) is an interesting framework that allows to extract temporal information from runs and verify them with an LTL checker, but can also perform some runtime verification of temporal properties. Similarly, (Desai et al, 2017) proposes Drona, to perform model checking and runtime verification to check and enforce properties on a model using a Signal Temporal Logic.

3.7 Discussion

Overall, the situation is not completely hopeless. Many components (in particular the deliberative ones) use formal models (hidden but present), which can be used for V&V. Providing the algorithm they use are sound, and that the models are valid w.r.t. the specifications, the results will be consistent w.r.t. the models. On the other end of the spectrum, code without any models, our point of vire is that there should not be any. All the running code should be developed with some level of specification and structure to enable some verification. As for learned models, we need to consider a change of paradigm, and not so much prove that a model is validated and verified, but that it will be used and deployed in such a way that the trust we put in the system is not jeopardized.

Last, for components developed using a robotics framework for which a DSL can be derived towards a formal language, then one can also expect encouraging results. In the following sections, we shall examine in detail a particular instance in this category.

We have seen that most robotic tools and frameworks do not provide any formal models per se, and if they do, it is usually up to the programmers to write both the program to run on the robot and the formal model.

Bjørner and Havelund (2014) 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."*

Following this advice, we advocate that the best way to introduce formal V&V in robotic components is to rely on existing robotic specification languages and frameworks, and to offer some automatic translation to formal

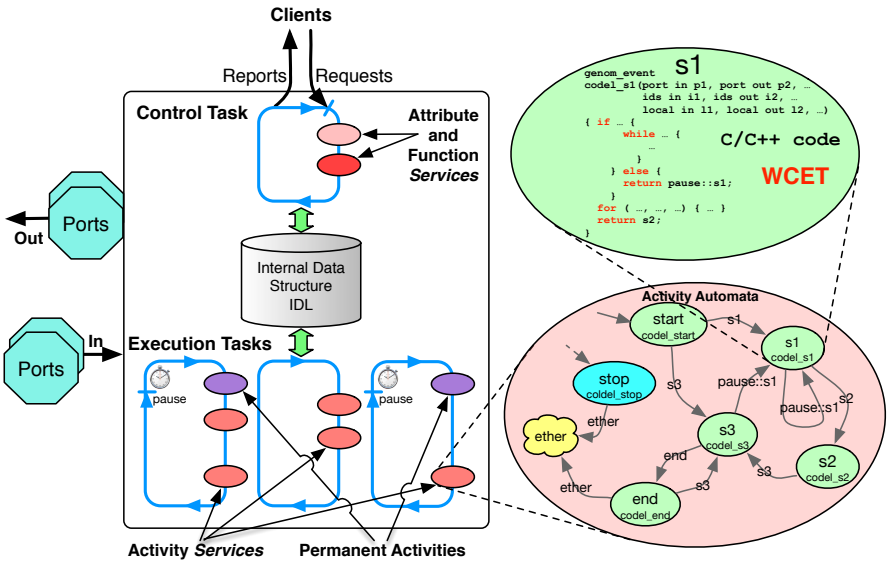

Fig. 2 A GenoM generic functional component (module).

models. For this, we need to ensure that the semantics of the specification is correct, and is properly modelled in the targeted formalism.

We shall now present an existing robotic specification language and its versatile template mechanism.

4 The GenoM tool

GenoM (GENerator Of Modules) (Mallet et al, 2010) is a tool to specify and implement robotic functional components: modules (see the nine modules "boxes" of the functional level on Fig. 4). These modules provide services in charge of functionalities that may range from simple low-level driver control (e.g. the VELODYNE or IMU modules to respectively control the Velodyne HLV32 or the XSens IMU) to more integrated computations (e.g. POM for localization with an Unscented Kalman Filter, or POTENTIALFIELD for navigation). GenoM proposes a language to completely specify the functional component down to (but not including) the C or C++ functions (codels), that implement the different stages and steps of the implemented services. This language fully specifies the shared ports (the green octagons in Fig. 4) between components (in and out), as well as the shared variables in a component, and the periodic tasks (i.e. threads) in which the services run. For each service, one defines the arguments (in and out), and the automata specifying the steps to follow to execute the codels, as well as their arguments. From a

specification point of view, there is a clear semantics of what should be done and how it should be properly implemented.

We now briefly present GenoM specification and its template mechanism.

4.1 GenoM Specification

Fig. 2 presents a generic GenoM component, composed of:

Control Task: A component always has an implicit cyclic *control task* that manages the control flow by processing *requests* and sending *reports* (from and to external clients); it also runs control (attribute and function) *services*, and activates or interrupts activity *services* in *Execution Tasks*.

Execution Task(s): Aside from the *control task*, whose reactivity must remain high, one may need one or more cyclic *execution tasks*, aperiodic or periodic, in charge of longer computations needed by activity *services*

Services: The core algorithms needed by the component are encapsulated within *services*. *Services* are associated to *requests* (with the same name). The algorithm executed by these services may require a *short* computation time or a *long* one. *Short* services are known as *control services* and are directly executed by the control task. Control services are in charge of quick computations and may be *attributes* (setters or getters) or *functions*. *Longer* services are known as *activities* and they are executed by *execution tasks*. Activities ensure longer computations and are modelled with an automaton that breaks down the computation into different *states* (see an example in the lower right part of Fig. 2). Each state is associated with a *codel*, which specifies a C or C++ function (top right part of Fig. 2). The execution of that codel leads to the next state in the automaton, to execute immediately, or in the next period if this next state name is prefixed with `pause`.

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

Ports: They specify the shared data, in and out, the component needs and produces from and for other components. Access to ports is mutually exclusive and is also specified in the *codels* arguments list.

This generic component is then instantiated for each specific component specification with a template mechanism.

4.2 GenoM Templates

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

Listing 1 A simple template code snippet.

A GenoM template is a set of text files that include Tcl code, whose evaluation in the context of a GenoM call on a specification file will produce the target of this particular template. The target can be as simple as one file with the list of the name of the services specified in the module (in which case the template file will just include a loop over all services and print their name), or it can be the C code which control the execution of an activity automaton, or it can be all the code which implement the module itself in ROS. Templates are the building blocks of any output of GenoM.

The template mechanism was initially introduced to deal with the *middleware independency* problem (Mallet et al, 2010). Indeed, the specifications presented above do not subsume any specific middleware. In short, the components are specified in a generic way using 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 module.

A template, when called by GenoM on a given module specification, has access to all the information contained in the specification file such as services names and types, ports and IDS fields needed by each codel, execution tasks periods, etc. Through the template interpreter (using Tcl syntax), one specifies what they need the template to synthesize. For instance, Listing 1 shows an excerpt of a template code and Listing 2 the C code it produces when called together with the NAVIGATION component specification file. The interpreter evaluates anything enclosed in *markers* <' '> without output, while on the code between <" ">, variables and commands substitution is performed and the result is output in the destination file, together with the text outside of the markers. For example,

`<'foreach s [$component services] {'> ... <"[$s name]"> ... <'}'>` iterates over the list of services of the component, contained in the `$component` variable; while `<"[$s name]">` is replaced by the name of the service contained in the `$s` variable bound by the foreach statement.

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

Listing 2 Excerpt of the synthesized C code for the PocoLibs NAVIGATION component corresponding to the template in Listing 1 (note how the C code is synthesized for all the services of the component).

There are already templates to synthesize: the component implementation for various middleware (*e.g.* PocoLibs[5], ROS-Com (Quigley et al, 2009), Orocos (Bruyninckx, 2001)); client libraries to control the component (*e.g.* JSON, C, OpenPRS), etc. Among the available middleware, we rather focus on Pocolibs as it is the most suitable for real-time applications (notably UAVs). Yet, its implementation, as efficient as it can be, cannot guarantee crucial properties such as schedulability of periodic tasks, for this we need formal V&V.

Fig. 3 shows the workflow to synthesize regular PocoLibs or ROS GenoM components.

[5] `https://git.openrobots.org/projects/pocolibs`

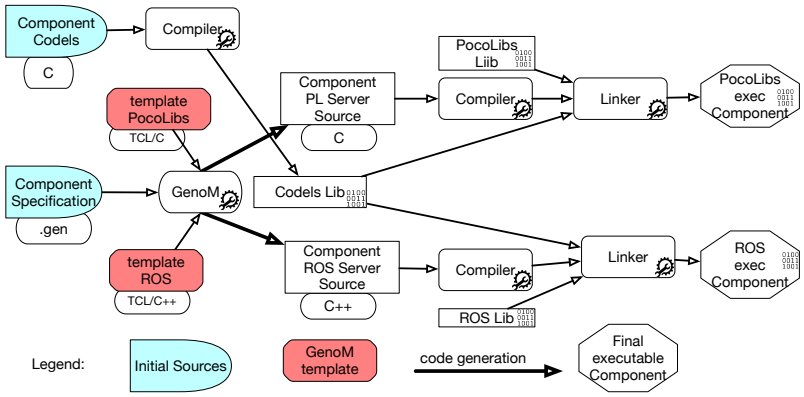

Fig. 3 Workflow to synthesize regular PocoLibs or ROS GenoM components.

We now present a non trivial realworld experiment implemented using GenoM. As we shall see, it is not an autonomous car, but it shares a lot of common sensors and effectors with one.

5 Not A Toy Example

To illustrate our approach, we present a complete navigation experiment for an RMP440 LE robot (called Minnie). Minnie has the following sensors: an XSens MTi IMU, a KVH DSP-5000 fiber optic gyro, a Novatel GPS, all connected through serial or USB lines and an HDL-32E Velodyne lidar (on an ethernet UDP interface). The RMP440 platform comes with a low-level controller (accessed through an ethernet interface), which allows controlling the robot with a speed (x-linear and z-angular) command, and returns the platform wheel odometry. There is also a wireless Sony PS2 joystick connected to a USB port. Finally, the platform includes a Nuvis 5306RT i7-6700 CPU with 16 Gb RAM and a 256 Go SSD drive, running Ubuntu 16.04.

These hardware components are controlled through their respective GenoM modules (see Fig. 4)[6]. They produce their respective ports (e.g. pose estimation, point cloud, buttons and axes pushed, etc). On top of these, POM uses an UKF to merge pose estimations from GPS, IMU and RMP440 (gyro and odometry) and to provide the position of the platform in the **Pose** port. NAVIGATION offer services to navigate in a graph of positions in a topological map of the environment, and produces a port with the next **Target** to reach. This **Target** is used by POTENTIALFIELD as the current goal to reach,

[6] The gyro does not appear as a separate module has it is managed inside the RMP440 module.

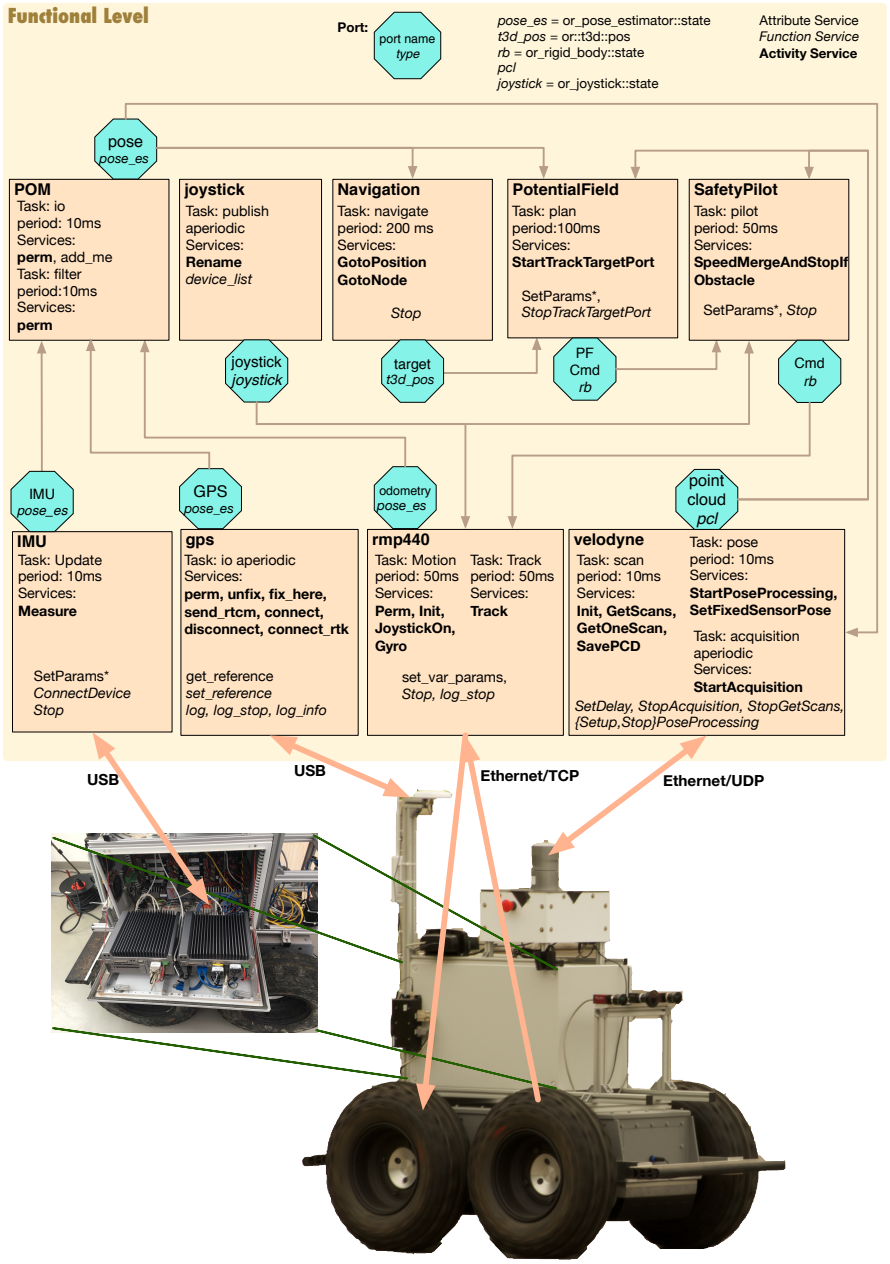

Fig. 4 Functional level of the Minnie RMP440 experiment

while avoiding obstacles in the **point cloud** using a Potential Field method inspired from (Guerra et al, 2016) (the points in the cloud are collected in an

occupancy grid which is then used to provide obstacles position in the local map). The speed reference **PF Cmd** is then read by SafetyPilot which, as last resort, checks that no obstacles is too close to the robot, and stops the robot if needed. It also considers the **joystick** port and uses it as a speed command producer if the proper joystick buttons are pushed (which is a way to gain control back of the robot platform in case something goes astray). The final speed **Cmd** produced is then read by RMP440, which pushes it to the low-level controller of the robot. The goal of this chapter is not to discuss the overall localisation and navigation implemented on Minnie, but to give a reasonable idea of the overall complexity entailed by a non-trivial robotic experiment.[7]

So for each of these modules, Fig. 4 indicates how many execution tasks each module has, the associated activity services (in bold), the function services (in italic), and the attribute services. Ports, represented by octagons, have a name and the data type they hold. For example, velodyne has three execution tasks: scan and pose running at 100 Hz, and acquisition aperiodic. scan has four services, *Init*, *GetScans*, *GetOneScans* and *SavePCD*. To further illustrate the GenoM specification, Listing 3 presents the *GetScans* activity service of the velodyne. Note the automata specification, which is also presented on Fig. 5.

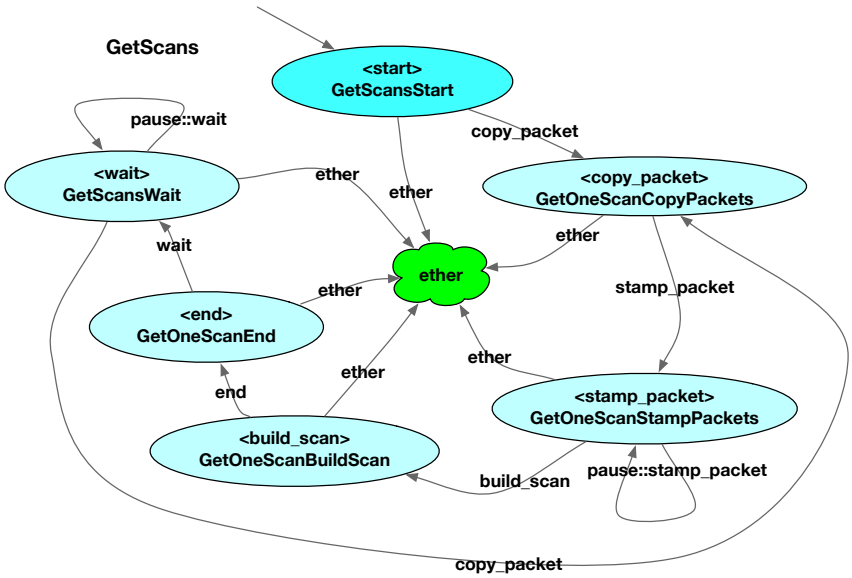

Fig. 5 Finite-state machine of the activity `GetScans` (listing 3)

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

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

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

 codel<start> GetScansStart(in acquisition_params)
                      yield copy_packets;
 codel<copy_packets> GetOneScanCopyPackets(in acquisition_params,
                             out mutex_buffer)
                      yield stamp_packets;
 codel<stamp_packets> GetOneScanStampPackets(in acquisition_params,
                             out mutex_pose_data, in timeout)
                      yield pause::stamp_packets, build_scan;
 codel<build_scan> GetOneScanBuildScan(in acquisition_params,
                             in firstAngle, in lastAngle)
                      yield end;
 codel<end> GetOneScanEnd(in acquisition_params,
                             port out point_cloud, inout usec_delay)
                      yield wait;
 codel<wait> GetScansWait(in period)
                      yield pause::wait, copy_packets;

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

Listing 3 The GenoM specification of the *GetScans* activity (VELODYNE).

Overall, the Minnie experiment includes: 9 modules, 9 ports, (13 + 9) tasks, 38 activity services (with automata), 41 function services, 43 attribute services, 170 codels over 14k loc (lines of codes) and their respective WCET. The synthesized GenoM modules amount to 200k loc for all to which one must add external libraries (middleware, PCL, Euler, etc).

We now briefly present the formal frameworks for which we have written GenoM templates.

6 Synthesized BIP, FIACRE, UPPAAL formal models

The template mechanism used to synthesize the GenoM modules from their specifications and codels, can also synthesize models for three formal frameworks. These templates also use temporal and statistical information obtained

by running the regular modules with the proper probes. In particular, for all formal models, we include the extracted Worst-Case Execution Time of each codel, as well as the distribution of state transitions in the service automata for the UPPAAL-SMC model. We now briefly present the three formal frameworks.

6.1 RT BIP

RT BIP is a framework[8] that allows modelling of embedded real-time systems, within components, including automata with guards (logical and temporal) and ports for synchronization (rendez vous and broadcast) with other components. RT BIP is not to be confused with the original BIP template used in (Bensalem et al, 2011) which did not include time information. RT BIP can be used offline with RT D-Finder (Ben Rayana et al, 2016) to prove some properties. For this, it automatically extracts invariant, from the automata, the interactions and the clock's histories and synthesizes an overapproximation of the reachable states of the system. It then tries to prove with a SAT solver that the desired property is satisfied in it. More interestingly, it can also be used online, and then use the RT BIP Engine (Socci et al, 2013) to run the model itself, linked with the codels, and enforce the properties at runtime. This runtime verification can also be augmented with more complex properties the roboticist may want to enforce.

6.2 UPPAAL and UPPAAL-SMC

UPPAAL is an integrated tool environment[9] for modeling, validation and verification of real-time systems modeled as networks of timed automata, extended with data types. Unlike BIP, it uses model checking to verify simplified TCTL (timed computation tree logic) properties in the modelled systems. The latest UPPAAL version (UPPAAL-SMC[10]) addresses the state explosion limit by offering a Statistical Model Checking extension. The timed automata can then be enriched with transition frequencies to perform sampling of the reachable states consistent with the transition frequencies. The properties are then proven with a confidence which is the ratio of state in which they are true over all the sampled states.

[8] http://www-verimag.imag.fr/RSD-Tools.html

[9] http://www.uppaal.org/

[10] http://people.cs.aau.dk/~adavid/smc/

6.3 FIACRE

FIACRE[11] is a formal language[12] for specifying concurrent and real-time systems also based on automata (behavior), ports and transitions, which can be guarded and sensitized over a time interval (similar to time Petri nets). The semantics is different from the timed automata used in BIP and UPPAAL. FIACRE provides a rich model to represent behavior and timing aspects of concurrent systems, using complex types, functions and externals. The produced model can then be analyzed with a model checking tool (see the TINA (TIme petri Nets Analyzer) toolbox[13]), but can also be deployed and executed using the Hippo model execution engine.

Table 1 sums up the various formal frameworks for which GenoM templates are available, and the corresponding tools used. The resulting formal models are automatically synthesized for the experiment, such as the one presented in Fig. 4. The offline versions abstract codels with their WCET, and are enriched with a client model (specific or generic) that specifies the sequence of requests to execute. The resulting model is then fed to the respective V&V tools (see some results in (Foughali, 2018)). The online versions (RT-BIP Engine and FIACRE-Hippo), provide an API to allow for: asynchronous external events handling to receive requests from the real clients; codels execution in separate threads, and reporting results to the clients. So the online versions run the real components (in place e.g. of the Pocolibs implementation of the component), we will show that with some added properties, these online versions perform runtime verification.

Formal Frameworks	Offline	Online PocoLibs (Herrb, 1992)	Online ROS-Comm (Quigley et al 2009)
RT BIP (Socci et al, 2013)	RT D-Finder (Ben Rayana et al 2016)	RT BIP Engine (Abdellatif et al 2010)	RT BIP Engine
FIACRE (Berthomieu et al, 2008)	Tina (Dal Zilio et al, 2015)	Hippo(Hladik, 2020)	Hippo
UPPAAL (Behrmann et al, 2006)	OK	NA	NA
UPPAAL-SMC (David et al, 2015)	OK	NA	NA

Table 1 Existing formal framework templates for GenoM.

[11] "Format Intermédiaire pour les Architectures de Composants Répartis Embarqués", french for "Intermediate Format for Embedded Distributed Component Architectures".

[12] http://projects.laas.fr/fiacre/

[13] http://projects.laas.fr/tina/

Writing these templates is tedious. It requires a very good knowledge of the GenoM specification and implementation, and of course a good knowledge of the formal frameworks used. But an interesting side effect, is that writing the formal version of a synthesized implementation (e.g. the Pocolibs implementation of the module) requires to also clarify the specification and the implementation when they are subject to ambiguities.

We shall now examine how the synthesized formal models for the Minnie experiment can be used to provide the Minnie GenoM module programmers with more confidence and proof of what the robot is doing.

7 Putting These Formal Models To Use

The three frameworks presented in the previous section have been deployed and tested online and offline with the Minnie experiment.

7.1 Online runtime verification with BIP

The BIP model of the Minnie experiment (Section 5) is 27k lines long and is linked to all the codels of the experiment. It handles requests, reports and ports exactly like the regular GenoM modules. Thus the BIP Engine can run the whole experiment, in place of the regular GenoM module implementation, but also check and enforce properties. Thus it will check that the tasks specified periods are respected, but it can also check more complex properties. For example, we may want to stop the robot when the **point cloud** port has not been properly published. For this type of temporal properties, we can add a monitor (Listing 4) such that:

- the **scan()** BIP port will be connected to the `GetOneScanEnd` codel (Listing 3, line 23) termination (which corresponds to the writing of the **point cloud** GenoM port)
- the **report()** BIP port will be connected to the BIP model of the *Track* service of the RMP440 modules, and force a transition to the *stop* state, of this automaton which makes an emergency stop of the robot (i.e. putting both linear and angular speed to 0).

To test this scenario, we introduced a *SetDelay* service in VELODYNE that artificially delays the execution of the `GetOneScanEnd` codel, and indeed, after two seconds, one can see the robot make an emergency stop.[14]

Running the BIP Engine, in the Minnie experiment, incurs a 15% increase in the CPU load, which remains acceptable considering the added safety. The

[14] BIP Engine traces and videos are available at this url `https://redmine.laas.fr/projects/minnie/gollum/index` which demonstrate the expected behavior.

```
atom type monitor_timeout()

        clock c unit millisecond
        export port Port scan()
        export port Port report()

        place idle, busy // Automata state

        initial to idle //Initial state

        on scan
        from idle to busy
        do {c = 0;} // reset clock

        on scan // scan interact, we stay in
        from busy to busy // busy state
        provided (c <= 2000) // providing it took less than 2 sec
        do {printf("monitor_timeout <= 2000.\n"); c = 0;} // reset clock

        on report // report interact
        from busy to idle // when in busy (transit to idle)
        provided (c > 2000) // if it took more than 2 sec
        do { printf("monitor_timeout > 2000.\n");}

end
```

Listing 4 A BIP monitor atom type

offline version of this model has also been tested and run with RT-DFinder, but did not bring any noticeable results.

7.2 Offline verification with UPPAAL

The synthesized UPPAAL model for the whole Minnie experiment is 9k lines. This model can be used for offline verification, but we need to add a "client" model, which represents the overall procedure, including the initialization sequence, of request to start and then run the experiment. Indeed, in robotics applications, initialization sequence are rather critical: many subsystems need to start at once, race conditions and deadlock are to be avoided. Such a "client" model is also written in UPPAAL, translated from the regular TCL script used to run the experiment. We can then check offline than no **port** is ever read before it has been written at least once. The UPPAAL property for the **Pose** port is:

```
A□ (not port_reading[Pose] or port_inited[Pose])
```

Then it is just a matter of making a conjunction for all the ports. If the UPPAAL model checker finds this formula false, it will report the sequence which led to this state.

In fact, this very property was initially false because of a race condition in the initialization sequence of the robot where the NAVIGATION module may have tried to navigate before the first **pose** position of the robot had been produced by the POM module.

7.3 Offline and online verification with FIACRE

The FIACRE model of the Minnie experiment is around 34k lines and includes the nine modules, 13 execution tasks, and 38 activity services calling 170 codels. It is the most detailed formal model we have and provides an execution fully equivalent to the regular GenoM version[15]. The big advantage of this model, is that it is exactly the same used offline with the TINA tools and online with the Hippo runtime execution engine. The differences between the two versions are minimal:

- The codel returned values, which define the next state in the activity automata, in the model executed online are replaced with nondeterministic "select" choices in the model used offline.
- Real code execution time in hippo (dispatched in a separate thread) is handled with transitions over the time interval of $[0, wcet]$ in the offline version
- We use external ports in the online version to receive requests from the client, while the offline version deploys a generic or specific FIACRE process to produce them and get reports.

In fact, the GenoM template[16] that produces the formal model is the same, there is just a flag to pass at synthesis time to select which version to produce (online or offline).

As a result, we are able to show properties similar to the ones we have shown in UPPAAL, even if the two approaches use two different time presentations (timed automata and global clock, versus local clocked guard and sensibilisation). Listing 5 shows the code to check for Uninitialized Port Read in Fiacre with Hippo; at runtime an error can be reported when the UPR state is reached. A similar version (without `start/sync`) can be used offline searching if there is a path leading to UPR.

[15] Without going into the detail, the BIP and UPPAAL models are too abstracted compared to the FIACRE one, they oversimplify the control task part of requests execution (interrupts, before, after), and do not properly model the interleaving of activities within one task (activities run completely to ether or pause before the next activity get called).

[16] `https://redmine.laas.fr/projects/genom3-fiacre-template`

```
from Navigation_start
    on (navigate_turn = my_index);
    if (navigate_activities[my_index].state = Navigation_stop) then
        to Navigation_stop
    end;
    on ((mutex_ports[Pose_port] = no_codel) and /* guard to check both */
        (mutex_ports[Target_port] = no_codel)); /* ports are available */
    mutex_ports[Pose_port] := SetTargetToPoseStart_port_codel; /* we lock */
    mutex_ports[Target_port] := SetTargetToPoseStart_port_codel; /* them */
    navigate_running_codel := SetTargetToPoseStart;
    if (not write_ports[Pose_port]) then /* Pose has not been written yet */
        to UPR /* the UPR state will report the error, */
    end; /* and take actions */
    /* The codel is called (start) in its own thread */
    start Navigation_SetTargetToPose_start(navigate_activities[my_index]);
    to Navigation_start_sync

from Navigation_start_sync
    /* waiting (sync) for the codel to return */
    sync Navigation_SetTargetToPose_start state;
    mutex_ports[Pose_port] := no_codel; /* ports are released */
    mutex_ports[Target_port] := no_codel;
    write_ports[Target_port] := true; /* Target port is marked as written */
    navigate_running_codel := 0; /* navigate task has no running codel now
         */
    to Navigation_start_dispatch
```

Listing 5 A FIACRE code snippet handling Uninitialized Port Read of the **Pose** port by the *SetTargetToPose* service from the NAVIGATION module.

Similarly, we can compute the maximum time it takes between a *stop* request sent to NAVIGATION and the writing of the zero speed on the robot HW controller by RMP440 from the **Cmd** port of SAFETYPILOT. Given a proper model of the scheduler, which can also be written in FIACRE, we can check wether the number of cores on the CPU is sufficient or not (Foughali et al, 2018). The model includes an `overshoot` state for each periodic task and if this state is reached, it means that the task has taken more time than its specified period. The number of cores available in an experiment can be considered in the model and one can adjust the value or the period or the activity services automata, or the codel to change the wcet, to ensure the schedulability of all the tasks. Note that this property is checked offline (with WCET and number of core) but also at runtime during the real execution.

So not only are these properties checked offline, but as a result, they are also true of the same model which executes them in a monitor online with the Hippo engine. Like BIP, we can write monitors in FIACRE that enforces more complex properties at runtime.

Overall, even if model checking techniques suffer from state-space explosion, the results obtained here on fairly complex robotic experiments are still encouraging.

All these results are obtained with models automatically synthesized and with automatic verification tools. One still needs to understand how to express properties in the corresponding query language (e.g. LTL and patterns for FIACRE, TCTL for UPPAAL) and how to interpret the results. Still, it is a big step forward in providing V&V tools to roboticists. To formally validate the obtained model, the semantics of GenoM has been first specified in Timed Transition Systems and then transformed in Timed Automata with Urgency and Data (Foughali et al, 2018). Still this is subject to the correctness of the semantics of GenoM in TTS. If there is a flaw in this semantics, the flaw will also ends up in the verified model. From our roboticist point of view, having exactly the same model (produced from the same template file) for execution and offline verification is a much stronger argument for validation of the model, as it exhibits a behavior at runtime perfectly similar to the regular Pocolibs implementation. Finally, one should note that all implemented modules in GenoM, get all these equivalent formal models for free, and their programmer can run the various V&V tools associated to them.

8 Conclusion and Future Work

In the proposed architecture, we distinguish between functional and decisional components. We have seen that some of these components already provide formal models (§3.3), so for these, future research may focus on verifying the correctness of the search algorithms deployed. Some rely on DSL and specific frameworks (§3.6) which could be extended to automatically provides formal models on which one can perform V&V as we have shown in Section 4. The GenoM formal frameworks templates should not be seen as the only possible path to infusing V&V in robotics. It is an example, of such a possible path, and there are numerous robotics frameworks that could do a similar automatic transformation of their specification towards formal models. We invite other robotic framework programmers to reach out and look at the possible formal frameworks they can connect to. Learned models (§3.4) have already been identified as outliers, so they probably need a "special" architectural setup for now and until we are satisfied with the confidence we can put in them. As for components with no model at all (§3.5), there are tedious solutions, if one cannot deploy them, one should at least consider reorganizing the code in such a way that it can rely on existing DSL or robotic frameworks.

On a different topic, the same way AI and robotics have to take into account human presence and interaction, V&V must also integrate it in the process. So we need to consider models of human behavior to introduce them

in the V&V process. This can be models of the users of the AS itself (e.g. passenger of an autonomous car), but also models of people around the AS (e.g. pedestrian, or drivers of a regular car)(Vicentini et al, 2020). Of course this adds another layer of variability and expands again the size of the models to explore, but we should also consider this as an opportunity to "close" the model and keep the reachable states at a reasonable size.

Another topic which needs to be considered is how these different models coexist and complete each other when it comes to proving a property of all the ASS. For example, the link between the different layers and components must also be verified. If one can rely on "compatible" formalisms between these layers, the better, as it will ease checking properties which are defined over more than one layer. The communication and middleware should also be properly modelled to be part of the V&V process. Shared memory is rather straightforward to model with locks, but publish and subscribe over XML-RPC (e.g. in ROS) is a completely different story. One needs to take into account network latencies, size of the queue, etc. We also need to guarantee consistency over the various models deployed.

Last, we should keep in mind that specifications, even if they produce formally equivalent models that can be fed to V&V tools, need to correctly capture the intents of the system designer. As pointed by (Rozier, 2016) "there is no escaping the 'garbage in, garbage out' reality". For this problem, we think that for now, we should rely on good old testing (see in this book (Gotlieb, 2021, chap. 4), (Eder, 2021, chap. 5)) of the system as to check that specifications are correct and synthesize the proper formal model.

References

Abdeddaim Y, Asarin E, Gallien M, Ingrand F, Lesire C, Sighireanu M (2007) Planning Robust Temporal Plans: A Comparison Between CBTP and TGA Approaches. In: Proceedings of the International Conference on Automated Planning and Scheduling, URL https://hal.archives-ouvertes.fr/hal-00157935

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

Amodei D, Olah C, Steinhardt J, Christiano P, Schulman J, Mané D (2016) Concrete Problems in AI Safety. arXivorg URL http://arxiv.org/abs/1606.06565v2, 1606.06565v2

Argall BD, Chernova S, Veloso MM, Browning B (2009) A survey of robot learning from demonstration. Robotics and Autonomous Systems 57(5):469–483

Bardaro G, Semprebon A, Matteucci M (2018) A use case in model-based robot development using AADL and ROS. In: ACM/IEEE Workshop on

Robotics Software Engineering, ACM Press, New York, New York, USA, pp 9–16, DOI rm10.1007/978-3-319-10783-7_13, URL http://dl.acm.org/citation.cfm?doid=3196558.3196560

Behrmann G, David A, Larsen KG (2006) A Tutorial on Uppaal 4.0. Tech. rep., Department of Computer Science, Aalborg University, Denmark, URL message:%3CC726A2F8-69CE-4CD4-A5B9-50F9B00C2C74@laas.fr%3E

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

Bensalem S, de Silva L, Ingrand F, Yan R (2011) A Verifiable and Correct-by-Construction Controller for Robot Functional Levels. Journal of Software Engineering for Robotics 1(2):1–19, URL http://arxiv.org/abs/0908.0221v1

Bensalem S, Havelund K, Orlandini A (2014) Verification and validation meet planning and scheduling. International Journal on Software Tools for Technology Transfer 16(1):1–12, DOI rm10.1007/s10009-013-0294-x, URL http://link.springer.com/10.1007/s10009-013-0294-x

Benveniste A, Berry G (1991) The synchronous approach to reactive and real-time systems. Proceedings of the IEEE 79(9):1270–1282

Benveniste A, Caspi P, Edwards S, Halbwachs N, Le Guernic P, de Simone R (2003) The synchronous languages 12 years later. Proceedings of the IEEE URL https://dblp.org/rec/journals/pieee/Benveniste CEHGS03

Berthomieu B, Diaz M (1991) Modeling and Verification of Time-Dependent Systems Using Time Petri Nets. Ieee Transactions on Software Engineering 17(3):259–273, URL http://gateway.webofknowledge.com/gateway/Gateway.cgi?GWVersion=2&SrcAuth=mekentosj&SrcApp=Papers&DestLinkType=FullRecord&DestApp=WOS&KeyUT=A1991FE66100005

Berthomieu B, Bodeveix JP, Farail P, Filali M, Garavel H, Gaufillet P, Lang F, Vernadat F (2008) Fiacre: an Intermediate Language for Model Verification in the Topcased Environment. In: Embedded Real-Time Software and Systems, HAL - CCSD, Toulouse, URL http://hal.inria.fr/docs/00/26/24/42/PDF/Berthomieu-Bodeveix-Farail-et-al-08.pdf

Bjørner D, Havelund K (2014) 40 Years of Formal Methods - Some Obstacles and Some Possibilities? FM URL https://dblp.org/rec/conf/fm/BjornerH14

Bohren J, Cousins S (2010) The SMACH High-Level Executive. IEEE Robotics and Automation Magazine 17(4):18–20, DOI rm10.1109/MRA.2010.938836, URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5663871

Boussinot F, de Simone R (1991) The ESTEREL Language. In: Proceeding of the IEEE, pp 1293–1304

Brugali D (2015) Model-Driven Software Engineering in Robotics. IEEE Robotics and Automation Magazine 22(3):155–166, DOI rm10.1109/

MRA.2015.2452201, URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=7254324

Brugali D (2021) Managing software variability for dynamic reconfiguration of robot control systems. In: Software Engineering for Robotics, Springer, URL https://doi.org/10.1007/978-3-030-66494-7

Brunner SG, Steinmetz F, Belder R, Domel A (2016) RAFCON: A graphical tool for engineering complex, robotic tasks. In: IEEE/RSJ International Conference on Intelligent Robots and Systems, IEEE, pp 3283–3290, DOI rm10.1109/IROS.2016.7759506, URL http://ieeexplore.ieee.org/document/7759506/

Bruyninckx H (2001) Open Robot Control Software: The OROCOS Project. In: IEEE International Conference on Robotics and Automation

Cassez F, Pagetti C, Roux OH (2004) A Timed Extension for ALTARICA. Fundam Inform URL https://dblp.org/rec/journals/fuin/CassezPR04

Cavalcanti A (2017) Formal Methods for Robotics: RoboChart, RoboSim, and More. In: Formal Methods: Foundations and Applications, Springer International Publishing, Cham, pp 3–6, DOI rm10.1145/1592434.1592436, URL http://link.springer.com/10.1007/978-3-319-70848-5_1

Cavalcanti A (2021) RoboStar technology - a roboticist's toolbox for combined proof and sound simulation. In: Software Engineering for Robotics, Springer, URL https://doi.org/10.1007/978-3-030-66494-7

Cicala G, Khalili A, Metta G, Natale L, Pathak S, Pulina L, Tacchella A (2016) Engineering approaches and methods to verify software in autonomous systems. In: International Conference on Intelligent Autonomous Systems, URL http://link.springer.com/chapter/10.1007/978-3-319-08338-4_121

Claßen J, Röger G, Lakemeyer G, Nebel B (2012) Platas—Integrating Planning and the Action Language Golog. KI-Künstliche Intelligenz 26(1):61–67, URL http://link.springer.com/article/10.1007/s13218-011-0155-2

Come D, Brunel J, Doose D (2018) Improving Code Quality in ROS Packages Using a Temporal Extension of First-Order Logic. In: IEEE International Conference on Robotic Computing, IEEE, pp 1–8, DOI rm10.1109/IRC.2018.00010, URL http://ieeexplore.ieee.org/document/8329874/

Costelha H, Lima PU (2012) Robot task plan representation by Petri nets: modelling, identification, analysis and execution. Autonomous Robots 33(4):337–360, DOI rm10.1142/3376, URL http://link.springer.com/10.1007/s10514-012-9288-x

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

David A, Larsen KG, Legay A, Mikučionis M, Poulsen DB (2015) UPPAAL SMC tutorial. International Journal on Software Tools for Technol-

ogy Transfer pp 1–19, DOI rm10.1007/s10009-014-0361-y, URL http://dx.doi.org/10.1007/s10009-014-0361-y

Desai A, Dreossi T, Seshia SA (2017) Combining Model Checking and Runtime Verification for Safe Robotics. RV URL https://dblp.org/rec/conf/rv/DesaiDS17

Dhouib S, Kchir S, Stinckwich S, Ziadi T, Ziane M (2012) RobotML, a Domain-Specific Language to Design, Simulate and Deploy Robotic Applications. In: IEEE International Conference on Simulation, Modeling, and Programming for Autonomous Robots, URL http://link.springer.com/chapter/10.1007/978-3-642-34327-8_16

Doose D, Grand C, Lesire C (2017) MAUVE Runtime: A Component-Based Middleware to Reconfigure Software Architectures in Real-Time. In: IEEE International Conference on Robotic Computing, IEEE, pp 208–211, DOI rm10.1109/IRC.2017.47, URL http://ieeexplore.ieee.org/document/7926540/

D'Silva V, Kroening D, Weissenbacher G (2008) A Survey of Automated Techniques for Formal Software Verification. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 27(7):1165–1178, DOI rm10.1109/TCAD.2008.923410, URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4544862

Eder K (2021) Gaining confidence in the correctness of robotic and autonomous systems. In: Software Engineering for Robotics, Springer, URL https://doi.org/10.1007/978-3-030-66494-7

Espiau B, Kapellos K, Jourdan M (1996) Formal verification in robotics: Why and how? In: International Symposium on Robotics Research, URL http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.54.3091&rep=rep1&type=pdf

Feth P, Akram MN, Schuster R, Wasenmüller O (2018) Dynamic Risk Assessment for Vehicles of Higher Automation Levels by Deep Learning. arXivorg URL http://arxiv.org/abs/1806.07635v1, 1806.07635v1

Fisher M (2021) Verifiable autonomy and responsible robotics. In: Software Engineering for Robotics, Springer, URL https://doi.org/10.1007/978-3-030-66494-7

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

Foughali M, Berthomieu B, Dal Zilio S, Hladik PE, Ingrand F, Mallet A (2018) Formal verification of complex robotic systems on resource-constrained platforms. In: FormaliSE @ The International Conference on Software Engineering ICSE, ACM Press, New York, New York, USA, pp 2–9, DOI rm10.1016/S1571-0661(05)80435-9, URL https://hal.laas.fr/hal-01778960

Gobillot N, Guet F, Doose D, Grand C, Lesire C, Santinelli L (2016) Measurement-based real-time analysis of robotic software architectures. In: IEEE/RSJ International Conference on Intelligent Robots and Systems, IEEE, pp 3306–3311, DOI rm10.1109/IROS.2016.

7759509, URL http://ieeexplore.ieee.org/xpl/articleDetails.jsp?tp=&arnumber=7759509&contentType=Conference+Publications

Gotlieb A (2021) Testing robotic systems: A new battlefield! In: Software Engineering for Robotics, Springer, URL https://doi.org/10.1007/978-3-030-66494-7

Guerra M, Efimov D, Zheng G, Perruquetti W (2016) Avoiding local minima in the potential field method using input-to-state stability. Control Engineering Practice 55(C):174–184, DOI rm10.1016/j.conengprac.2016.07.008, URL http://dx.doi.org/10.1016/j.conengprac.2016.07.008

Guiochet J, Machin M, Waeselynck H (2017) Safety-critical advanced robots: A survey. Robotics and Autonomous Systems URL http://www.sciencedirect.com/science/article/pii/S0921889016300768

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

Halder R, Proença J, Macedo N, Santos A (2017) Formal Verification of ROS-Based Robotic Applications Using Timed-Automata. In: IEEE/ACM International FME Workshop on Formal Methods in Software Engineering (FormaliSE), URL https://dblp.org/rec/conf/icse/HalderPMS17

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

Hierons R (2021) Systematic automated testing of robotic systems based on formal models. In: Software Engineering for Robotics, Springer, URL https://doi.org/10.1007/978-3-030-66494-7

Hladik PE (2020) Hippo. Tech. rep., LAAS-CNRS, URL https://redmine.laas.fr/projects/genom3-fiacre-template/gollum/hippo

Huang J, Erdogan C, Zhang Y, Moore B, Luo Q, Sundaresan A, Rosu G (2014) ROSRV: Runtime verification for robots. In: Runtime Verification, URL http://link.springer.com/chapter/10.1007/978-3-319-11164-3_20

Infantes G, Ghallab M, Ingrand F (2010) Learning the behavior model of a robot. Autonomous Robots pp 1–21, URL https://homepages.laas.fr/felix/publis-pdf/arj10.pdf

Ingrand F, Ghallab M (2017) Deliberation for autonomous robots: A survey. Artificial Intelligence 247:10–44, DOI rm10.1016/j.artint.2014.11.003, URL http://dx.doi.org/10.1016/j.artint.2014.11.003

Kai A, Hölldobler K, Rumpe B, Wortmann A (2017) Modeling Robotics Software Architectures with Modular Model Transformations. Journal of Software Engineering for Robotics 8(1):3–16, DOI rm10.6092/JOSER, URL https://www.google.com/

Kober J, Bagnell JA, Peters J (2013) Reinforcement Learning in Robotics: A Survey. International Journal of Robotics Research DOI rm10.1177/0278364913495721, URL http://ijr.sagepub.com/content/early/2013/08/22/0278364913495721.abstract

Koopman P, Wagner M (2016) Challenges in Autonomous Vehicle Testing and Validation. SAE International Journal of Transportation Safety 4(1):15–24, DOI rm10.4271/2016-01-0128, URL http://papers.sae.org/2016-01-0128/

Kortenkamp D, Simmons RG (2008) Robotic Systems Architectures and Programming. In: Siciliano B, Khatib O (eds) Handbook of Robotics, Springer, pp 187–206

Kress-Gazit H, Wongpiromsarn T, Topcu U (2011) Correct, Reactive, High-Level Robot Control. IEEE Robotics and Automation Magazine 18(3):65–74, DOI rm10.1109/MRA.2011.942116, URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6016593

Lesire C, Pommereau F (2018) ASPiC: an Acting system based on Skill Petri net Composition. In: IEEE/RSJ International Conference on Intelligent Robots and Systems, pp 1–7

Levesque HJ, Reiter R, Lesperance Y, Lin F, Scherl RB (1997) GOLOG: A logic programming language for dynamic domains. The Journal of Logic Programming 31(1):59–83, URL http://www.sciencedirect.com/science/article/pii/S0743106696001215

Li W, Miyazawa A, Ribeiro P, Cavalcanti A, Woodcock J, Timmis J (2018) From Formalised State Machines to Implementations of Robotic Controllers . In: Distributed Autonomous Robotic Systems, pp 1–14, URL https://scholar.google.com/

Lotz A, Hamann A, Lütkebohle I, Stampfer D (2016) Modeling Non-Functional Application Domain Constraints for Component-Based Robotics Software Systems. arXivorg URL http://arxiv.org/abs/1601.02379, related:kK103Nsy7GYJ

Luckcuck M, Farrell M, Dennis L, Dixon C, Fisher M (2018) Formal Specification and Verification of Autonomous Robotic Systems: A Survey. arXivorg URL http://arxiv.org/abs/1807.00048v1, 1807.00048v1

Mallet A, Pasteur C, Herrb M, Lemaignan S, Ingrand F (2010) GenoM3: Building middleware-independent robotic components. In: IEEE International Conference on Robotics and Automation, pp 4627–4632, DOI rm10.1109/ROBOT.2010.5509539, URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5509539

Meng W, Park J, Sokolsky O, Weirich S, Lee I (2015) Verified ROS-Based Deployment of Platform-Independent Control Systems. In: NASA formal methods, Springer International Publishing, Cham, pp 248–262, DOI rm10.1007/978-3-319-17524-9_18, URL http://link.springer.com/10.1007/978-3-319-17524-9_18

Miyazawa A, Ribeiro P, Li W, Cavalcanti A, Timmis J (2017) Automatic property checking of robotic applications. In: IEEE/RSJ International Conference on Intelligent Robots and Systems, URL http://dblp.org/rec/conf/iros/Miyazawa0LCT17

Mühlbacher C, Gspandl S, Reip M, Steinbauer G (2016) Improving Dependability of Industrial Transport Robots Using Model-Based Tech-

niques. In: IEEE International Conference on Robotics and Automation, pp 3133–3140, DOI rm10.1109/ICRA.2016.7487480, URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=7487480

Nordmann A, Hochgeschwender N, Wigand D, Wrede S (2016) A Survey on Domain-Specific Modeling and Languages in Robotics. Journal of Software Engineering for Robotics 7(1):1–25, URL https://scholar.google.com/

Pelliccione P (2021) Making robots usable in everyday life. In: Software Engineering for Robotics, Springer, URL https://doi.org/10.1007/978-3-030-66494-7

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

Ribeiro P, Miyazawa A, Li W, Cavalcanti A, Timmis J (2017) Modelling and Verification of Timed Robotic Controllers. In: International Conference on Integrated Formal Methods, URL http://dblp.org/rec/conf/ifm/0002MLCT17

Rozier KY (2016) Specification - The Biggest Bottleneck in Formal Methods and Autonomy. In: Verified Software: Theories, Tools, and Experiments, DOI rm10.1007/978-3-319-48869-1, URL http://link.springer.com/chapter/10.1007/978-3-319-48869-1_2

Schlegel C (2021) Composition, separation of roles and model-driven approaches as enabler of a robotics software ecosystem. In: Software Engineering for Robotics, Springer, URL https://doi.org/10.1007/978-3-030-66494-7

Schlegel C, Hassler T, Lotz A, Steck A (2009) Robotic software systems: From code-driven to model-driven designs. In: International Conference on Advanced Robotics, IEEE, pp 1–8, URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5174736

Seshia SA, Sadigh D, Sastry SS (2016) Towards Verified Artificial Intelligence. arXivorg URL http://arxiv.org/abs/1606.08514v3, 1606.08514v3

Simmons RG, Pecheur C (2000) Automating Model Checking for Autonomous Systems. In: AAAI Spring Symposium on Real-Time Autonomous Systems

Simon D, Pissard-Gibollet R, Arias S (2006) ORCCAD, a framework for safe robot control design and implementation. In: Control Architecture for Robots, URL https://hal.inria.fr/inria-00385258

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

Sorin A, Morten L, Kjeld J, Schultz UP (2016) Rule-based Dynamic Safety Monitoring for Mobile Robots. Journal Of Software Engineering In Robotics 7(1):120–141, URL https://scholar.google.fr/

Sotiropoulos T, Waeselynck H, Guiochet J, Ingrand F (2017) Can Robot Navigation Bugs Be Found in Simulation? An Exploratory Study. In: IEEE In-

ternational Conference on Software Quality, Reliability and Security, URL https://dblp.org/rec/conf/qrs/SotiropoulosWGI17

Täubig H, Frese U, Hertzberg C, Lüth C, Mohr S, Vorobev E, Walter D (2011) Guaranteeing functional safety: design for provability and computer-aided verification. Autonomous Robots 32(3):303–331, DOI rm10.1007/s10514-011-9271-y, URL http://www.springerlink.com/index/10.1007/s10514-011-9271-y

Tomlin CJ, Mitchell I, Bayen AM, Oishi M (2003) Computational techniques for the verification of hybrid systems. Proceedings of the IEEE 91(7):986–1001, DOI rm10.1109/JPROC.2003.814621, URL http://ieeexplore.ieee.org/document/1215682/

Verma V, Jónsson AK, Pasareanu C, Iatauro M (2006) Universal executive and PLEXIL: engine and language for robust spacecraft control and operations. In: American Institute of Aeronautics and Astronautics Space, AIAA Space Conference, URL http://scholar.google.com/scholar?q=related:IpQ407u5_qsJ:scholar.google.com/&hl=en&num=20&as_sdt=0,5

Vicentini F, Askarpour M, Rossi MG, Mandrioli D (2020) Safety Assessment of Collaborative Robotics Through Automated Formal Verification. IEEE Transactions on Robotics 36(1):42–61, DOI rm10.1109/TRO.2019.2937471, URL https://ieeexplore.ieee.org/document/8844289/

Williams BC, Ingham MD (2003) Model-based Programming of Intelligent Embedded Systems and Robotic Space Explorers. Proc of the IEEE: Special Issue on Modeling and Design of Embedded Software 91(1):212–237

Wong KW, Kress-Gazit H (2017) Robot Operating System (ROS) Introspective Implementation of High-Level Task Controllers. Journal of Software Engineering for Robotics 8(1):1–13, DOI rm10.6092/JOSER, URL http://joser.unibg.it/index.php/joser/issue/view/9

Woodcock J (2021) Modelling uncertainty in RoboChart using probability. In: Software Engineering for Robotics, Springer, URL https://doi.org/10.1007/978-3-030-66494-7

Woodcock J, Larsen PG, Bicarregui J, Fitzgerald JS (2009) Formal methods - Practice and experience. ACM computing surveys 41(4), URL https://dblp.org/rec/journals/csur/WoodcockLBF09

Yakymets N, Dhouib S, Jaber H, Lanusse A (2013) Model-driven safety assessment of robotic systems. In: IEEE/RSJ International Conference on Intelligent Robots and Systems, IEEE, pp 1137–1142, DOI rm10.1109/IROS.2013.6696493, URL http://ieeexplore.ieee.org/xpl/articleDetails.jsp?tp=&arnumber=6696493&contentType=Conference+Publications