



**HAL**  
open science

# GenoM3 Templates: from Middleware Independence to Formal Models Synthesis

Mohammed Foughali, Félix Ingrand, Anthony Mallet

► **To cite this version:**

Mohammed Foughali, Félix Ingrand, Anthony Mallet. GenoM3 Templates: from Middleware Independence to Formal Models Synthesis . 2017. hal-01457881v2

**HAL Id: hal-01457881**

**<https://laas.hal.science/hal-01457881v2>**

Preprint submitted on 22 Mar 2017

**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.

# **G<sup>en</sup>M3 Templates: from Middleware Independence to Formal Models Synthesis \***

Mohammed Foughali, Félix Ingrand and Anthony Mallet  
LAAS-CNRS, Université de Toulouse, CNRS, Toulouse, France  
{firstname.lastname}@laas.fr

## **Abstract**

G<sup>en</sup>M is an approach to develop robotic software components, which can be controlled, and assembled to build complex applications. Its latest version G<sup>en</sup>M3, provides a template mechanism which is versatile enough to deploy components for different middleware without any change in the specification and user code. But this same template mechanism also enables us to automatically synthesize formal models (for two Validation and Verification frameworks) of the final components. We illustrate our approach on a real deployed example of a drone flight controller for which we prove offline real-time properties, and an outdoor robot for which we synthesize a controller to perform runtime verification.

## **1 Introduction**

There is a rising concern in advanced robotic and autonomous systems software development. Can we improve the dependability of such systems by deploying formal validation and verification (V&V) techniques applied to their software? Such techniques are widespread in areas such as aeronautic, railway, etc, but are still seldom used in robotics. Nowadays, robotic software developments use model-based or model-driven software engineering approaches (e.g. SMARTSOFT [Schlegel et al., 2009], RobotML [Dhouib et al., 2012], MontiArc [Ringert et al., 2015]). These approaches and their associated middleware are numerous and surveyed in a number of papers [Brugali, 2015, Elkady and Sobh, 2012, Mohamed et al., 2008]. Still, most of these approaches remain disconnected to formal model analysis and the use of V&V techniques. Several works proposed to use the formal synchronous language ESTEREL [Boussinot and de Simone, 1991] to model functional components [Espiau et al., 1996, Sowmya et al., 2002, Kim and Kang, 2005]. The formal models were then exploited to verify behavioral and timed properties using model checking tools. These experiments were nevertheless led on simple examples and specifications were either hard-coded in ESTEREL or manually translated from robotic com-

---

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

ponents. More recently a special issue [Kress-Gazit, 2011] on this subject presented a number of interesting works along hybrid automata [Muradore et al., 2011] and controller synthesis [Kress-Gazit et al., 2011] and more recently [Jing et al., 2016]. The formal frameworks proposed are similar to the ones we use, but they are mostly deployed at decisional level or on rather simple robotic systems. Our approach is somewhat complementary in choosing to model all the software components which need to be integrated together at the functional level and then checking and enforcing properties on the integrated code. The closest work to our approach is MAUVE [Gobillot et al., 2014, Gobillot et al., 2016], where the code is instrumented to collect Worst Case Execution Time (WCET), and then temporal formal property can then be checked on the components. Our approach distinguishes itself by, simultaneously, being fully automatic, considering all timing constraints of the model and tackling rather complex integrated robotic applications.

In this paper we present the  $G^{\text{en}}\text{M3}$  framework to specify and deploy robotic functional components.  $G^{\text{en}}\text{M3}$  relies on a specification language which allows the programmer to completely define how the component will work when associated to the user provided code, and how this code will be internally organized along services, tasks, ports, etc. This specification is done independently of the middleware targeted and, thanks to its template mechanism, can also be used to synthesize more than just the final code of the component, but also client libraries, and models for various V&V frameworks.

The paper first describes, with an example, the specification language used by  $G^{\text{en}}\text{M3}$ . Section 3 presents the template mechanism used to synthesize middleware specific components, but also to produce the components formal models for two V&V frameworks. Section 4 briefly presents two examples (a drone flight controller and an outdoor robot navigation) which have been completely specified in  $G^{\text{en}}\text{M3}$ , followed by a section where we illustrate the type of properties we can formally prove on the resulting models, as well as how we synthesize a runtime controller to run the component. The paper concludes on the ongoing work along these lines and the future extensions we intend to tackle.

## 2 $G^{\text{en}}\text{M3}$ components

$G^{\text{en}}\text{M3}$  [Mallet et al., 2010] is a tool to specify and implement robotic functional components. In the overall LAAS architecture [Ingrand et al., 2007], functional components act as “servers” in charge of functionalities which may range from simple low-level driver control (e.g. the velocity control of the propellers of a drone, camera, etc) to more integrated computations (e.g. Simultaneous Localization And Mapping (SLAM), navigation, PRM or RRT motion planning, etc).

### 2.1 Requirements

We consider that a typical component is a *program* which needs to handle and manage the following aspects:

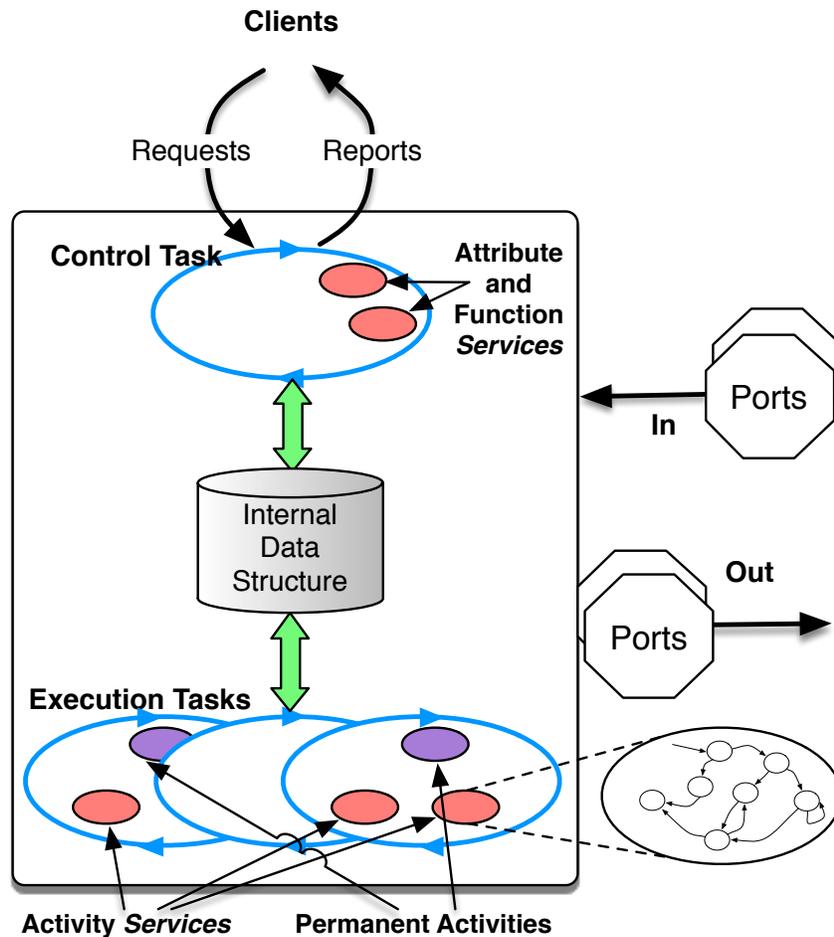


Figure 1: A generic  $G^{en}M3$  component.

**Inputs and Outputs** : a component interacts with external clients and other components. For the former, the control flow, it must handle *requests* from client(s) and asynchronously send back *reports* to the client which issued the request, to act on the result. For the latter, the data flow, it must provide a mechanism to share data with other components and read data from other components. Data flow and control flow are semantically different and correspond to two different ways components can interact.

**Algorithms** : the core algorithms needed to implement the functionality the component is in charge of must be appropriately organized within threads as to preserve the reactivity of the component and the schedulability of the various possibly concurrent algorithms. A component may have just one service to provide, but most of the time, there are a number of such services associated to the consid-

ered robotic functionality. The way algorithms are specified and organized in a component is a tradeoff. One can let the programmer organize its code the way it pleases him. But without any particular structure, chances are that little can be validated or verified. If one provides guidelines and rules as how the code must be organized, than we stand a much better chance as to prove some properties on the code.

**Internally shared data** : the various algorithms, possibly concurrent, running in the component, may have to share state variables, parameters, etc. which represent the internal state of the component.

## 2.2 Implementation

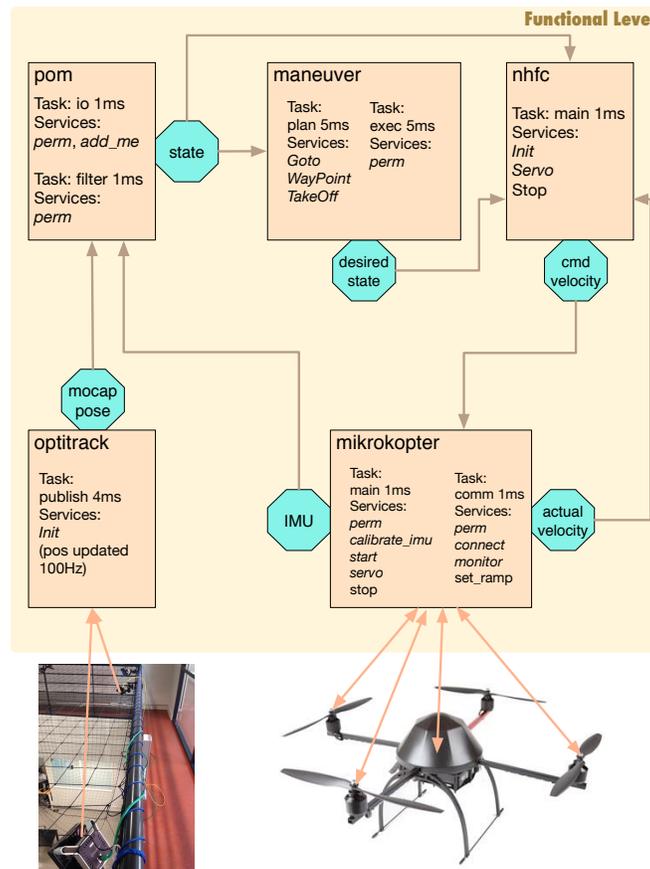


Figure 2: The quadcopter functional level. For the sake of simplicity, only a partial list of services is presented.

To achieve such requirements of a functional component, we propose to organize each one along the structure shown on Figure 1. Of course, the pros and cons of these implementation choices can be discussed, and we will attempt to justify them as we present them. We illustrate the detailed specifications on Listing 1, with the MANEUVER component of the complete functional layer of a Mikrokopter drone flying in our lab (Figure 2). This component, given a position to hover or waypoints to pass by, is in charge of producing intermediate positions to navigate to.

Specifying a component in  $G^{en}M3$  is the programmer design choice. Thus, there are a number of considerations she/he has to take into account, which may depend on the hardware constraints, the complexity of the algorithm, the needed external data, etc. Let us describe these different elements in more details and how they interact, and how they are specified. Note that at this point, we are not assuming any particular middleware for interprocess communication or data sharing (control and data flow).

Apart from the control task, each element of the specification results from choices the developer has to carefully make (how many tasks? periodic or not? if periodic, which period? which services associated to which task? how to break down long services in processing steps? etc):

**Control Task** : A component always has a *control task* that manages the control flow by processing *requests* and sending *reports* (from/to external clients); activate and stop services, etc. The control task is implicitly comprised within a component and the user does not need to specify it.

**Execution Task(s)** : Aside from the *control task*, whose reactivity must remain short, one may need one or more *execution tasks*, aperiodic or periodic, in charge of longer computations.

**Services** : The core algorithms needed to implement the functionality the component is in charge of are encapsulated within *services*. *Services* are associated to a *request* (with the same name), but one may also define *permanent activities* which are attached to an *execution task*.

**IDS** : A local *internal data structure* is provided for all the *services* to share parameters, computed values or state variables of the component. It is appropriately accessed (i.e. with proper locking) by the *services* when they need to read or write a field of the *IDS* (line 10).

**Ports** : They specify the shared data, in and out, the component needs or produces from/for other components (line 3).

**Exceptions** : One may specify *exceptions*, which can be returned by *services* to report non nominal execution (line 7).

We go in more details and see how these different elements interact and how the component internally runs.

**Services** Services hold the specifications of the algorithms handled by the component. Services can take arguments (line 35), and return values (line 29). Services are

```

1  component maneuver {
2      /* ports declaration: direction type name */
3      port in or_pose_estimator::state state;
4      port out or_pose_estimator::state desired;
5
6      /* exception declaration */
7      exception e_nostate; /*Service can throw this exception*/
8
9      /* ids declaration */
10     ids {
11         planner_s planner;
12         struct trajectory_s { ... } trajectory; ... };
13
14     /* tasks declaration */
15     task plan { /* an aperiodic task */
16         codel<start> mv_plan_start(out ::ids) yield ether wcet 26us; /* used to
17             initialize one */
18         codel<stop> mv_plan_stop(inout ::ids) yield ether wcet 12us; }; /* used to
19             cleanup once */
20
21     task exec { /* A task with only a permanent activity */
22         period 5 ms;
23         codel<start> mv_exec_start(out desired) yield wait wcet 17us;
24         codel<wait> mv_exec_wait(in state, in trajectory) yield pause::wait, main
25             1165us;
26         codel<main> mv_exec_main(in state, inout trajectory, out desired) yield wait,
27             pause::main, start wcet 1313us;
28         codel<stop> mv_exec_stop() yield ether wcet 4us; };
29
30     /* attribute services declaration */
31     attribute get_planner(out planner);
32     /* function services declaration */
33     function stop(out double x, out double y, out double z) {
34         doc "Stop a goto and return the current position";
35         codel get_current_position(out x, out y, out z) wcet 10us;
36         interrupt goto; };
37
38     /* activity services declaration */
39     activity goto(in double x, in double y, in double z, in double yaw) {
40         doc "Reach a given position from current state";
41         task plan; /* goto will execute in the plan task */
42         validate validate_goto(in z) wcet 3us; /* check z>0 */
43         codel<start> mv_current_state_start(in state, out start) yield plan wcet 43us;
44         async codel<plan> mv_goto_plan(in planner, in start, in x, in y, in z, in yaw,
45             out path) yield exec wcet 2487us;
46         codel<exec> mv_plan_exec(in planner, in path, inout trajectory) yield
47             pause::exec, wait wcet 304us;
48         codel<wait> mv_plan_exec_wait(in trajectory) yield pause::wait, ether wcet 13us;
49         codel<stop> mv_plan_exec_stop(out trajectory) yield ether wcet 10us;
50         throw e_nostate; /* No valid state position found */
51         interrupt goto; }; /* will interrupt an active goto */
52
53     activity waypoint(in double x, in double y, in double z, in double yaw) {
54         doc "Push a given position to reach after last one";
55         task plan; /* waypoint will execute in this task */
56         codel<start> mv_waypoint_start(in state, in trajectory, out start) yield plan
57             wcet 6us;
58         async codel<plan> mv_goto_plan(in planner, in start, in x, in y, in z, in yaw,
59             out path) yield exec 2487us;
60         codel<exec> mv_waypoint_add(in planner, in path, inout trajectory) yield ether
61             wcet 369us; }; };

```

Listing 1: Excerpt from the GenM3 specification of the MANEUVER component.

activated upon receiving the corresponding request. A service may have a `validate` codel (line 38). This codel is executed by the control task and checks that the arguments of the request are correct. If they are, the service is then *runnable*, otherwise, it is reported with an illegal arguments report. A service may also specify other services it interrupts (line 45) when it becomes runnable. The interrupted services execute a stop codel (line 43) if any and report to their client that they have been interrupted.

*Control Services*, are only for short execution as to not delay the control task which executes them. A control service may be an *attribute* (setter or getter of fields of the IDS, line 27), or a *function* (line 29) for quick and simple computations. A  $G^{en}M3$  component offers four predefined function services, namely: *Kill* (stop the component), *Abort* (stop an activity service), *Connect Port* to connect a local *in* port to a distant *out* port and *Connect Service* to connect a service of another component.

*Activity services* (activities for short), see line 35 and 47, are executed in the execution task specified in their declaration (line 37). They all have a `start` codel which is the entry point of their codels finite-state machine (FSM) and as many states/codels, as the programmer wants, to specify the decomposition of the long computation they are performing (e.g., the FSM defined from line 39 to 43, also drawn on Figure 4). The execution of a codel always returns the next state to which the execution must transition to in the service FSM. If the returned state is prefixed with *pause* (line 41), the control of the execution task is passed to the next service to execute in this task, if any, or back to the scheduler as to wait until the next task period. *ether* is a special state to which a terminating service can transition. An activity may be *permanent* (from line 21 to 24). It is not requested by a client and is run by its execution task when the component starts.

*Codels* specify the C or C++ function they will call, with the arguments (taken from the service arguments, the IDS and/or the ports of the component) they need (in and out). Codels are restricted to use these arguments only. Codels are also restricted to return a state/codel specified in the FSM definition of their service. Each codel may specify a WCET, which measures the worst case execution time of the codel alone (i.e., executing independently of any other execution). The organisation of activity services along FSM and codels may be seen as an unnecessary burden on the robot programmer, but nothing prevent the programmer to have one `start` codel which does it all. Yet, breaking code along a FSM brings a number of advantages when it comes to better code integration and V&V. It improves schedulability and code execution interleaving. It provides a finer model of data sharing and code interlocking.

**Control Task** As seen before, the control task manages the requests and reports of the component, as well as starting, terminating services. It runs the `validate` codels for services which specify one. If there exist activities that are incompatible with the requested service, the control task instructs the execution tasks in charge of such activities to interrupt them. If the request concerns a control service (attribute, line 27 or function, line 29), the control task executes it directly. Otherwise, the requested activity service is then put on hold until all the incompatible instances are correctly interrupted and terminated. Then the control task advises the execution task declared by the service to run it, and sends an intermediate reply to the client to inform it that

processing has started. Upon completion of services, the control task sends reports to the corresponding client (service ended nominally, service interrupted, etc.).

**Execution Tasks** Execution tasks are periodic (with a specified period, line 20) or aperiodic (line 15). With each period signal (if periodic) or event occurrence (if sporadic), the execution task runs its permanent activity (if any) and then all the *active instances* of its associated activities. An active instance of a given activity is an instance that has been requested by a client and whose execution has not yet ended.

**Internal Data Structure** Access to the IDS is mutually exclusive. One can see that the proper specifications (enforced by  $G^{\text{en}}M3$ ) of the codel arguments allows for a very fine grain locking of the IDS field. In other words, we know at any time which codels access what. Only the needed field(s) by a codel are locked in order to ensure maximal parallelism.

**Ports** Information exchange with other components is made through ports (line 3). As seen above, ports usage (in, out or in/out) is also declared in codels arguments. As such, over a large set of components composing a robotic functional layer, we have a clear model of which codels use a particular port and at what time.

### 3 Templates Approach

As seen above,  $G^{\text{en}}M3$  provides a rich language to specify functional components and how they should be organized. Still, producing the real component code to run on the robot, out of this specification, requires additional steps. This is where  $G^{\text{en}}M3$  *template* mechanism is critical.  $G^{\text{en}}M3$  without template just analyzes the specification file (extension *.gen*) and checks it for inconsistencies. The real power of  $G^{\text{en}}M3$  is to call it on a specification file, along with a template, as to automatically synthesize the target of the template.

A template when called by  $G^{\text{en}}M3$  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. Based on all these specifications,  $G^{\text{en}}M3$  can also compute information such as which codels can execute at the same time (considering their respective arguments), or which port must be locked by which codel. etc. Through the template interpreter (Tcl), one specifies what they need the template to synthesize. Since the interpreter relies on a complete scripting language, there is virtually no restriction on what a template can generate. For instance, Listing 2 shows an excerpt of a template function and Listing 3 the C code it produces when called together with the MANEUVER specification. 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 [$component services] {'>... <'>` iterates over the list of services of the component, contained in the `$component` variable; while

<"[\$s name]"> is replaced by the name of the service contained in the \$s variable bound by the foreach statement.

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

Listing 2: A simple template code snippet.

```
void
genom_maneuver_activity_report(
    struct genom_component_data *self,
    struct genom_activity *a)
{
    switch(a->sid) {
        case -1: return; /* permanent activity reports nothing */
        case MANEUVER_connect_port_RQSTID:
            genom_maneuver_connect_port_activity_report(
                self,
                (struct genom_maneuver_connect_port_activity *)a);
            return;
        ...
        case MANEUVER_goto_RQSTID:
            genom_maneuver_goto_activity_report(
                self,
                (struct genom_maneuver_goto_activity *)a);
            return;
        case MANEUVER_waypoint_RQSTID:
            genom_maneuver_waypoint_activity_report(
                self,
                (struct genom_maneuver_waypoint_activity *)a);
            return;
    }
    ...
}
```

Listing 3: Excerpt of the synthesized C code for the PocoLibs MANEUVER component corresponding to the template in Listing 2 (note how the C code is synthesized for all the services of the component).

There are a number of templates already defined, to synthesize the component code

for a given middleware (e.g., PocoLibs<sup>1</sup>, ROS [Quigley et al., 2009]), C client libraries, OpenPRS client procedures and code, a JSON client, etc. For example, the template *skeleton* generates the files containing the codel stubs with their proper function prototypes. The user can then specify the algorithmic core of their codels without worrying about the middleware.

### 3.1 Middleware Independence

The middleware “server” templates are used to synthesize the component itself to be run on the robot. They output the *glue code* in charge of making calls to the targeted middleware. The synthesized glue code manages message passing (requests/reports) as well as ports connection, and handles all the internal algorithms to manage the different tasks, services FSMs, proper locking of shared resources, etc. This is a viable solution to the problem of middleware dependency as neither the specification nor the codels rely on a specific middleware. Indeed, codel execution can only rely on the objects declared in their arg list (i.e. IDS fields, their service arguments and ports) and do not make calls to the middleware. Figure 3 shows an overview of  $G^{en}M3$  workflow for middleware independence.

So far,  $G^{en}M3$  offers several middleware templates, notably ROS-Com and PocoLibs. The former heavily uses ROS topics, ROS services and ROS actions (actionLib) in the synthesized code, while the latter uses PocoLibs primitives, such as MBox, CSM-Box, posters, etc. But from an external behavior point of view, the two resulting components (ROS and PocoLibs) behave exactly the same (apart from performance issues specific to the middleware implementation).

### 3.2 Formal Verification

$G^{en}M$  started as a robotic software development tool and methodology in the mid 90’s [Fleury et al., 1997]. Quickly, it appeared that the component specification could be used for more than components code synthesis. An earlier study, using  $G^{en}M2$ , did an ad hoc job at synthesizing a BIP [Basu et al., 2006] model and went as far as running the synthesized model of 14 components along with the BIP Engine on a real robot [Bensalem et al., 2011].  $G^{en}M3$  has a semantically cleaner specification model (arbitrarily complex user-defined FSMs, codels WCETs, etc) and thanks to its versatile template mechanism can now be used to synthesize formal models for different frameworks (Fiacre [Berthomieu et al., 2008] and Real-Time BIP (RT-BIP) [Abdellatif et al., 2010]). But one should keep in mind that we want to synthesize the model which is semantically equivalent to the resulting component. So the synthesized model goes beyond what is in the component specification file. As a result, the Fiacre or RT-BIP templates synthesize models relative to a targeted specific middleware. Moreover, unlike the previous work presented in [Bensalem et al., 2011], the current  $G^{en}M3$  component specifications include some temporal information: WCET on codels, as well as the period of execution tasks. WCET can be obtained empirically, or with more advanced techniques [Wilhelm et al., 2008], for now, we get them

---

<sup>1</sup><https://git.openrobots.org/projects/pocolibs>

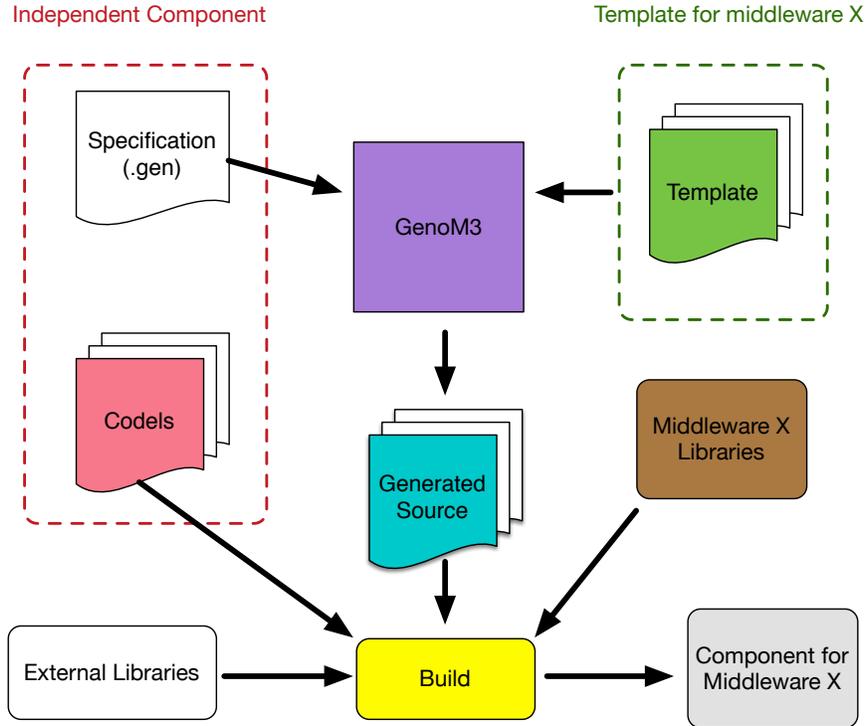


Figure 3: Generating a  $G^{\text{en}}\text{M3}$  component for a middleware X.

by running the components and collecting data. Yet, these temporal information were not taken into account in [Bensalem et al., 2011] but are now part of the formal models we synthesize.

In [Foughali et al., 2016], a template is presented which automatically synthesizes models in Fiacre [Berthomieu et al., 2008], a formal language for specifying concurrent and real-time systems based on automata (behavior) and time Petri nets (timing aspects). The synthesized models are exploited in order to verify important real-time properties using TINA [Berthomieu et al., 2004] model checkers.

$G^{\text{en}}\text{M3}$  also provides a template to automatically generate RT-BIP [Basu et al., 2006] models. RT-BIP is a formal framework based on interacting components encapsulating timed automata with urgencies. It is associated to an execution *engine* and an offline deductive verification tool: RTD-finder [Ben Rayana et al., 2016]. The latest release of the RT-BIP engine (RT-BIPE) implements *external* transitions guarded with external, non controllable events. This allowed us to run the generated models with the RT-BIPE checking for client requests while properly handling the execution of sporadic tasks, particularly the control task, and monitor our components online.

## 4 Deployed Examples

There are already a number of experiments deployed with G<sup>en</sup>M3. We illustrate our approach with the functional level of (i) a MikroKopter quadcopter flying in our lab and (ii) our outdoor robot Mana.

### 4.1 Quadcopter Flying Example

Figure 2 presents the 5 components involved in our quadcopter functional layer. Each box corresponds to a component, and each octagon is a port. Ports are written (out) by the components they are attached to, and read (in) through the arrow pointing to the reading component. Inside each box, we list the execution tasks (their period or “ap” if they are aperiodic), and a partial list of the services provided by this component. Note that this figure does not present the “supervisor” in charge of sending requests and analyzing reports, which is out of scope of this paper.

- MIKROKOPTER is the component in charge of the quadcopter low-level hardware. The quadcopter is controlled by applying a velocity to each propeller, and produces the current velocities, as well as its current IMU (Inertia Measurement Unit) values. It has two tasks i) **comm**, aperiodic, which keeps polling and parsing data from the hardware (to get the current propellers velocity and IMU) and storing them in the IDS. ii) **main**, periodic at 1ms, which reads the **cmd velocity** port, manages the servo control and writes the two ports **IMU** and the propellers **actual velocity**.
- OPTITRACK is the component handling the current position of the quadcopter as perceived by our “OptiTrack” motion capture system. It has one task **publish**, periodic at 250Hz. It provides the current position of the quadcopter in the **mocap pose** port.
- POM merges the **mocap pose** position produced by OPTITRACK and the **IMU** from MIKROKOPTER and produces an Unscented Kalman filtered position in port **state**. It has two tasks **io** and **filter** both periodic at 1KHz.
- MANEUVER is the navigation component, it has two tasks **exec** with a period of 5ms and **plan** aperiodic. Given a position or waypoints to navigate to, it reads the **state**, and computes a trajectory to reach it, producing intermediate positions to fly to in **desired state**.
- NHFC (Near Hovering Flight Controller) is the core of the flight controller. Running one task **main** at 1KHz, it reads the **actual velocity** port of the propellers, the current position in the **state** port of POM, and the desired position (port **desired state**) of MANEUVER and produces the proper **cmd velocity** port containing the desired velocity of the propellers (which is then read by MIKROKOPTER) to reach and hover near this position.

The complexity of our quadcopter functional level is such that with 5 components, 13 tasks running potentially in parallel, over 40 services and more than 65 codels,

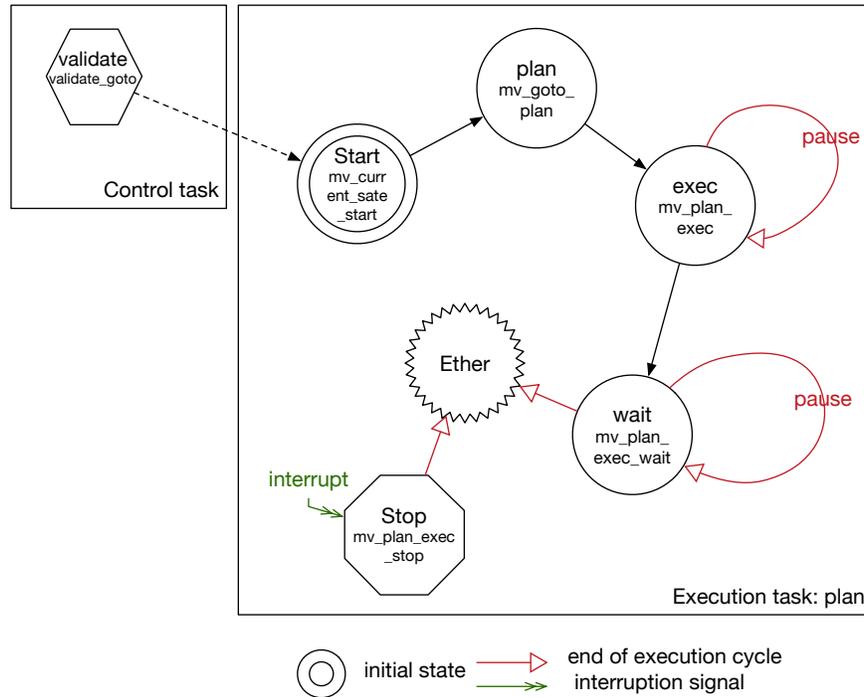


Figure 4: FSM of the goto activity of the MANEUVER component (See Listing 1, line 39-43)

checking by hand any temporal property is impossible. Note that for the sake of simplicity, only a partial list of services is presented in this paper, we refer the reader to the source repository for the complete specifications <https://git.openrobots.org/projects/telekyb3> (in the corresponding sub-projects).

## 4.2 Outdoor Navigation Example

This navigation stack is inspired from the real navigation running on our Segway RMP 400 robot, Mana.

Figure 5 presents the four components in charge of the navigation:

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

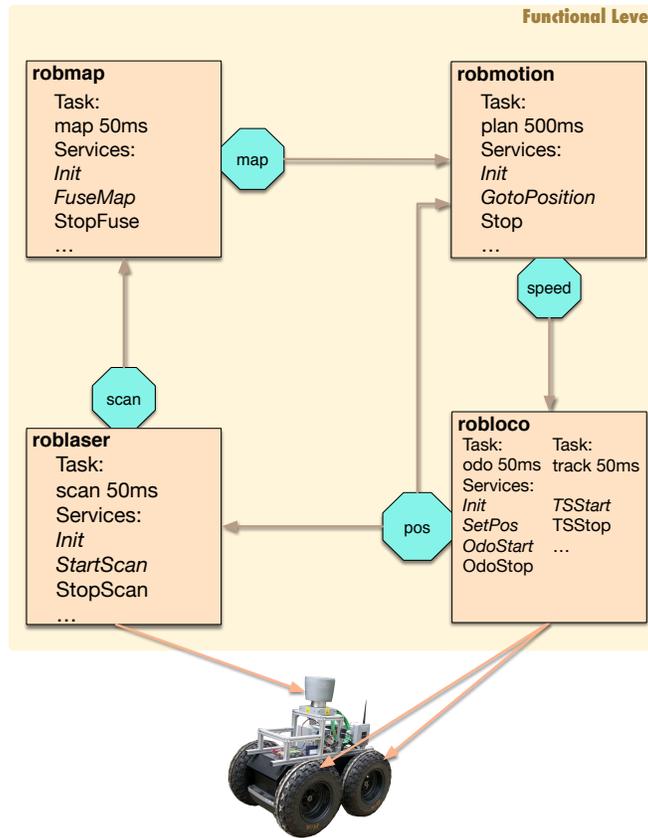


Figure 5: An outdoor robot functional level

produces, on the port **scan**, the free space in the laser's range tagged with the position where the scan has been made (read on **pos**).

- ROBMOTION aggregates the successive **scan** data in the **map** port. A fuse task (period 50 ms) and *FuseMap*, one of its activities, perform the computation. The function *FuseStop* interrupts the activity *FuseMap*.
- ROBMOTION has one task *plan* (period 500 ms) which, given a goal position (via the activity *GotoPosition*), computes the appropriate speed to reach it and writes it on **speed**, using the current position (from **pos**), and avoiding obstacles (from **map**). *GotoPosition* interrupts itself, so a new request will cancel the currently running one (if it exists) and force the execution of its stop state. Similarly, The *Stop* service (function) interrupts *GotoPosition*.

## 5 Offline and Runtime Verification

Templates are used to generate our quadcopter and outdoor robot components for various middleware. We use these components today routinely on PocoLibs and ROS-Com middleware. In this section, we focus on verification of the PocoLibs implementation as it is more adapted for real-time applications such as flight control running at 1KHz.

### 5.1 Offline Real-time Properties with Fiacre

When it comes to prove real-time properties on concurrent real-time systems, one has to take into account the hardware on which it runs, and the scheduling policy used. The latest release of the Fiacre template allows the user to provide the number of cores provided by the platform. The template consequently generates a Fiacre model of the components including a cooperative (non-preemptive) FIFO scheduler in charge of allocating the cores to the different tasks present in the components model<sup>2</sup>. We give hereafter two examples of real-time properties that we successfully prove on our, respectively, quadcopter and outdoor components. These properties are of a capital importance to robotic programmers. Note that we automatically synthesize the Fiacre models of the components and the external clients considering their behavior as they produce the requests as needed for both applications.

**Schedulability (Quadcopter)** We refer to a periodic execution task as *schedulable* if it always executes the requested services before its deadline (next period signal). Schedulability is very often a hard real-time requirement in quadcopter applications. The problem of verifying the schedulability of tasks is inherently complex in  $G^{\text{en}}\text{M3}$  because of the mutual exclusion between codels. For instance, in `MIKROKOPTER`, while the task `main` is executing a given service, it may have to wait when it reaches one of its codels since such a codel needs a resource (a field of the IDS or a port) already in use by a codel being executed by the control task or the task `comm` or even a task of another component since ports are shared among components. Hence, verifying the schedulability is more complicated than just summing the WCETs of the codels and comparing the result to the task period. Furthermore, the lack of cores may cause a delay between the period signal and the actual start of task execution, raising the risk to miss the deadline. To verify schedulability properties, we generate an external client to ensure a stationary flight (hovering).

The modeling choices in [Foughali et al., 2016] permit an easy expression of schedulability properties for all tasks e.g. for `main`:

```
property schedulability_main is
always (mikrokopter/main/state executing =>not (main_period_signal))
```

Which translates to: for all execution paths, when `main` is at its state `executing`, the boolean `main_period_signal` (which is set to true in the model when `main` period is reached) evaluates to false. The same modeling choices ease also the verification of schedulability properties. Indeed, they are expressed as invariants which allows the

<sup>2</sup>In other words, the tasks are not preempted, and release their core/CPU at the end of each cycle.

use of the TINA coarser reachability graph construction that does not preserve firing sequences (smaller state spaces). The results, obtained in about seven minutes, show that all tasks are schedulable, considering the quad-core hardware constraints. We note that in case of using a less powerful hardware, e.g. a core-duo platform, the model checker produces a counterexample as the schedulability property holds no more for filter (POM). Considering the aforementioned scheduler characteristics, we prove that in order to satisfy schedulability properties for the hovering application, a hardware with three cores minimum is indispensable.

**Bounded Stop (Outdoor Robot)** As soon as a *Stop* request is sent to ROBMOTION, thestop codel of ROBMOTION’s *GotoPosition* is executed (because the specification of the *Stop* service is to interrupt the *GotoPosition* service). This execution includes writing a null speed to **speed** port that will be sent to the motor controller via executing the codel update of *TSSstart* (ROBLOCO). For the robot programmer, it is naturally important to determine the maximum amount of time  $\tau$  between sending a *Stop* request and applying a null speed to the wheels. Due to mutual exclusion among cores and ports, manually calculating  $\tau$  would be as tedious as error-prone.

We make use of the Fiacre patterns [Abid et al., 2012] `leadsto` `within` and `leave` to compute the worst-case value of  $\tau$  considering the actual quad-core platform on Mana.

```
property bounded_stop_1 is (robmotion/control_task/state Stop_req) leadsto
  (robmotion/GotoPosition/state stop) within [0,0.5]
property bounded_stop_2 is (robmotion/GotoPosition/state stop) leads to leave
  (robloco/TSSstart/state update) within [0,0.06]
```

The pattern `leadsto` encodes the Linear Temporal Logic (LTL) combined operator  $\square \diamond$  (always eventually). The *scope modifier* `within` extends the pattern `leadsto` in order to express a timed property. The pattern `leave` expresses leaving a given state. Indeed, it is important in this context to compute the bound up to the end of update execution to make sure the null speed is sent to the wheels controller. We sum the bounds separating (1) sending the *Stop* request and reaching the codel stop of *GotoPosition* (ROBMOTION) and (2) reaching the codel stop of *GotoPosition* and leaving the codel update of *TSSstart* (ROBLOCO) to prove the sought bound  $\tau$  to be 560 ms.

## 5.2 Online Real-time Controller with RT-BIP

RT-BIP models are automatically generated out of  $G^{\text{en}}\text{M3}$  specifications. The  $G^{\text{en}}\text{M3}$ -to-RT-BIP template synthesizes readily executable models (linked with the codels) for the multi-threaded RT-BIPE. This latest release of RT-BIPE, allows the definition of asynchronous external transitions, which permits us to run the component on the RT-BIPE while communicating directly with external clients. We use these models to conduct runtime monitoring on the outdoor robot navigation components ROBMOTION and ROBLOCO while evolving with the other components implemented in PocoLibs. For instance, the RT-BIPE enforces the timed constraints of the model (respect of task period and WCET) and is able to catch when those constraints are violated, and

take appropriate measures. The multithreaded RT-BIP engine is still currently under development toward a version that will allow us to run all the components together on the engine and perform runtime verification and online enforcement of properties. As for offline verification of the model with RTD-Finder, it is still being investigated as the tool does not yet readily support urgencies and variables. RTD-Finder, being a deductive verification tool on invariants automatically extracted from the model (behavior invariants, interaction invariants and history clock invariants), may be a legitimate resort when models do not scale with model checking. The outcome of the ongoing work on integrating urgencies and/or variables in the verification process will certainly open a new horizon in verifying more complex systems in the future.

## 6 Conclusion

G<sup>en</sup>M3 defines a functional component specification language and offers a template mechanism which allows the programmer to synthesize the component code ready to be built, linked to the user written algorithms, and to deploy it on the targeted platform. There exist a number of templates, for example to synthesize the components for various middleware. Thus the middleware choice can be done later with respect to performance issues, development platform or other independently deployed components and legacy software. Similarly, one can now automatically synthesize the formal models, for two frameworks widely used by the V&V community for embedded and concurrent real-time systems (Fiacre and RT-BIP). These formal models can then be used with their respective toolboxes, to formally check offline properties, using complementary approaches: model checking for Fiacre; and invariants extraction and satisfiability for RT-BIP. RT-BIP, together with its Engine can also run the formal models linked to the user code, in place of the regular component, and enforce real-time properties at runtime. From the roboticist point of view, being able to formally check functional components alone, but also integrated with other components, with two different V&V paradigms is a win-win situation, as one can choose the most adapted technique to the particular behavioral and timed properties they want to prove. Moreover, with the rising complexity of autonomous systems (autonomous cars, drones, cobots, etc), formal analysis and proof of correct behavior is becoming a rather critical issue and can lead to new ways of certifying the software of these systems. For example, we are now starting a new project to apply this methodology to an autonomous driverless bus, and expect this approach to ease the certification of such a vehicle, with respect to regulation agency.

As for future work, we plan to take advantage of the upcoming versions of the RT-BIP Engine to be able to run all the components of a functional layer. We also plan to improve how to have roboticist (not V&V expert) to express the properties they want to prove and how to easily interpret the results.

## References

- [Abdellatif et al., 2010] Abdellatif, T., Combaz, J., and Sifakis, J. (2010). Model-Based Implementation of Real-Time Applications. In *EMSOFT*.
- [Abid et al., 2012] Abid, N., Dal Zilio, S., and Le Botlan, D. (2012). Real-Time Specification Patterns and Tools. In *FMICS*.
- [Basu et al., 2006] Basu, A., Bozga, M., and Sifakis, J. (2006). Modeling Heterogeneous Real-Time Components in BIP. In *SEFM*.
- [Ben Rayana et al., 2016] Ben Rayana, S., Bozga, M., Bensalem, S., and Combaz, J. (2016). RTD-Finder - A Tool for Compositional Verification of Real-Time Component-Based Systems. In *TACAS*.
- [Bensalem et al., 2011] Bensalem, S., de Silva, L., Ingrand, F., and Yan, R. (2011). A Verifiable and Correct-by-Construction Controller for Robot Functional Levels. *JOSER*.
- [Berthomieu et al., 2008] Berthomieu, B., Bodeveix, J.-P., Farail, P., Filali, M., Gavel, H., Gauffillet, P., Lang, F., and Vernadat, F. (2008). Fiacre: an Intermediate Language for Model Verification in the Topcased Environment. In *ERTSS*.
- [Berthomieu et al., 2004] Berthomieu, B., Ribet, P. O., and Vernadat, F. (2004). The tool TINA - Construction of abstract state spaces for Petri nets and Time Petri. *International Journal of Production Research*.
- [Boussinot and de Simone, 1991] Boussinot, F. and de Simone, R. (1991). The ESTEREL Language. In *Proceeding of the IEEE*.
- [Brugali, 2015] Brugali, D. (2015). Model-Driven Software Engineering in Robotics. *IEEE RAM*.
- [Dhouib et al., 2012] Dhouib, S., Kchir, S., Stinckwich, S., Ziadi, T., and Ziane, M. (2012). RobotML, a Domain-Specific Language to Design, Simulate and Deploy Robotic Applications. In *SIMPAR*.
- [Elkady and Sobh, 2012] Elkady, A. and Sobh, T. (2012). Robotics Middleware: A Comprehensive Literature Survey and Attribute-Based Bibliography. *Journal of Robotics*.
- [Espiau et al., 1996] Espiau, B., Kapellos, K., and Jourdan, M. (1996). Formal verification in robotics: Why and how? In *ISRR*.
- [Fleury et al., 1997] Fleury, S., Herrb, M., and Chatilla, R. (1997). GenoM: A Tool for the Specification and the Implementation of Operating Modules in a Distributed Robot Architecture. In *IROS*.
- [Foughali et al., 2016] Foughali, M., Berthomieu, B., Dal Zilio, S., Ingrand, F., and Mallet, A. (2016). Model Checking Real-Time Properties on the Functional Layer of Autonomous Robots. In *ICFEM*.

- [Gobillot et al., 2016] Gobillot, N., Guet, F., Doose, D., Grand, C., Lesire, C., and Santinelli, L. (2016). Measurement-based real-time analysis of robotic software architectures. In *IROS*.
- [Gobillot et al., 2014] Gobillot, N., Lesire, C., and Doose, D. (2014). A Modeling Framework for Software Architecture Specification and Validation. In *SIMPAR*.
- [Ingrand et al., 2007] Ingrand, F., Lacroix, S., Lemai-Chenevier, S., and Py, F. (2007). Decisional autonomy of planetary rovers. *JFR*.
- [Jing et al., 2016] Jing, G., Tosun, T., Yim, M., and Kress-Gazit, H. (2016). An End-To-End System for Accomplishing Tasks with Modular Robots. In *RSS*.
- [Kim and Kang, 2005] Kim, M. and Kang, K. C. (2005). Formal Construction and Verification of Home Service Robots: A Case Study. In *Automated Technology for Verification and Analysis*.
- [Kress-Gazit, 2011] Kress-Gazit, H. (2011). Robot challenges: Toward development of verification and synthesis techniques [from the Guest Editors]. *IEEE RAM*.
- [Kress-Gazit et al., 2011] Kress-Gazit, H., Wongpiromsarn, T., and Topcu, U. (2011). Correct, Reactive, High-Level Robot Control. *IEEE RAM*.
- [Mallet et al., 2010] Mallet, A., Pasteur, C., Herrb, M., Lemaignan, S., and Ingrand, F. (2010). GenoM3: Building middleware-independent robotic components. In *ICRA*.
- [Mohamed et al., 2008] Mohamed, N., Al-Jaroodi, J., and Jawhar, I. (2008). Middleware for Robotics: A Survey. In *RAMECH*.
- [Muradore et al., 2011] Muradore, R., Bresolin, D., Geretti, L., Fiorini, P., and Villa, T. (2011). Robotic Surgery. *IEEE RAM*.
- [Quigley et al., 2009] Quigley, M., Gerkey, B., Conley, K., Faust, J., Foote, T., Leibs, J., Berger, E., Wheeler, R., and Ng, A. Y. (2009). ROS: an open-source Robot Operating System. In *ICRA*.
- [Ringert et al., 2015] Ringert, J. O., Roth, A., and Rumpe, B. (2015). Language and code generator composition for model-driven engineering of robotics component & connector systems. *JOSE*.
- [Schlegel et al., 2009] Schlegel, C., Hassler, T., Lotz, A., and Steck, A. (2009). Robotic software systems: From code-driven to model-driven designs. In *ICAR*.
- [Sowmya et al., 2002] Sowmya, A., Tsz-Wang So, D., and Hung Tang, W. (2002). Design of a Mobile Robot Controller using Esterel Tools. *Electronic Notes in Theoretical Computer Science*.
- [Wilhelm et al., 2008] Wilhelm, R., Mitra, T., Mueller, F., Puaud, I., Puschner, P., Staschulat, J., Stenström, P., Engblom, J., Ermedahl, A., Holsti, N., Thesing, S., Whalley, D., Bernat, G., Ferdinand, C., and Heckmann, R. (2008). The worst-case execution-time problem—overview of methods and survey of tools. *TECS*.