Statistical Model Checking of Complex Robotic Systems

Mohammed Foughali, Félix Ingrand, Cristina Seceleanu

▶ **To cite this version:**

Mohammed Foughali, Félix Ingrand, Cristina Seceleanu. Statistical Model Checking of Complex Robotic Systems. 26th International SPIN Symposium on Model Checking of Software, Jul 2019, Beijing, China. ⟨hal-02152286⟩

HAL Id: hal-02152286

https://laas.hal.science/hal-02152286v1

Submitted on 11 Jun 2019

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.

HAL Authorization

Statistical Model Checking of Complex Robotic Systems

Mohammed Foughali[1], Félix Ingrand[1], and Cristina Seceleanu[2]

[1] LAAS-CNRS, Université de Toulouse, CNRS, Toulouse, France
mfoughal@laas.fr,felix@laas.fr
[2] Mälardalen University, Västerås, Sweden
cristina.seceleanu@mdh.se

Abstract. Failure of robotic software may cause catastrophic damages. In order to establish a higher level of trust in robotic systems, formal methods are often proposed. However, their applicability to the *functional layer* of robots remains limited because of the informal nature of specifications, their complexity and size. In this paper, we formalize the robotic framework GenoM3 and automatically translate its components to UPPAAL-SMC, a real-time statistical model checker. We apply our approach to verify properties of interest on a real-world autonomous drone navigation that does not scale with regular UPPAAL.

1 Introduction

Although robotic software is tested, both in the field and using simulators, its lack of safety hinders the deployment of robots in costly and human-interaction missions (*e.g.* home assistants, deep space). As an example, the NASA Remote Agent Experiment had to be stopped due to a deadlock, never detected during the one-year testing phase [25]. Other examples include the autonomous vehicle *Alice* [19] and the museum guide RoboX9 [32]. Such failures are mainly due to the nature of classical, scenario-based testing, unable to provide guarantees on important properties. *Formal methods* are a promising alternative, but their use in robotics is still marginal, and varies according to the software *layers* [33]. Indeed, at the *decisional layer*, in charge of high-level decision making functions (*e.g.* planning [17]), models are often formal with complete semantics, which facilitates their formal modeling and verification [7, 12]. In contrast, *functional layer* components, in charge of low-level actions involving sensors and actuators (*e.g.* localization and navigation), are developed within non formal frameworks (*e.g.* ROS [26]), which makes their *formalization* particularly challenging and costly. Furthermore, the formal modeling is non reusable (it needs to be redone whenever a component evolves) and models are not guaranteed to scale. Consequently, many previous works either focus on simple case studies (usually not deployed on real robots), resort to non realistic abstractions (*e.g.* ignoring timing constraints), or propose no alternatives to deal with scalability issues (Sect. 7).

We propose in this paper the use of formal methods to verify the functional layer of robotic systems. We focus on verification by means of model checking, and use statistical model checking [22] to tackle scalability issues. A particular interest is given to real-time properties, *e.g. schedulability* and *bounded response*, crucial in robotics (examples in Sect. 6). To tackle the abovementioned problems, we (1) formalize (Sect. 3) the robotic framework GenoM3 (Sect. 2), (2) develop automatic, sound transformation

from any GenoM3 specification into UPPAAL and UPPAAL-SMC models (Sect. 4, 5) and (3) verify crucial real-time properties, while avoiding non-realistic abstractions (*e.g.* all timing constraints are considered), on a real drone application (Sect. 6). We conclude with related work (Sect. 7) and lessons learned (Sect. 8).

2 Preliminaries

2.1 GenoM3

GenoM3 [23] is a tool to specify and implement robotic functional components. Each component, in charge of a functionality, ranging from sensor control (*e.g.* laser) to more integrated computations (*e.g.* navigation), is organized as shown in Fig. 1a. For space and readability, we omit in this paper *control services* and *interruption of activities*, but the interested reader may refer to [11] for details.

A component implements the core algorithms of its functionality within *activities*, which it executes following *requests* from external *clients*. Thus, the component has a (i) *control Task* to *process* the clients requests and *report* to them accordingly and (ii) one or more *execution task(s)* to execute activities. These tasks share parameters and computed values of the component through the *Internal Data Structure* (IDS). Finally, a component provides *ports* to share data with other components.

```
1  activity MoveDistance(in double distRef :"Distance in m") {
2    doc  "Move of the given distance";
3    codel <start> mdStartEngine(in distRef, in state.position, out posRef)
         yield exec, ether;
4    codel <exec> mdGotoPosition(in speedRef, in posRef, out state, port out
         Mobile) yield pause::exec, end;
5    codel <end> mdStopEngine() yield ether wcet 1 ms;
6    task motion;};
```

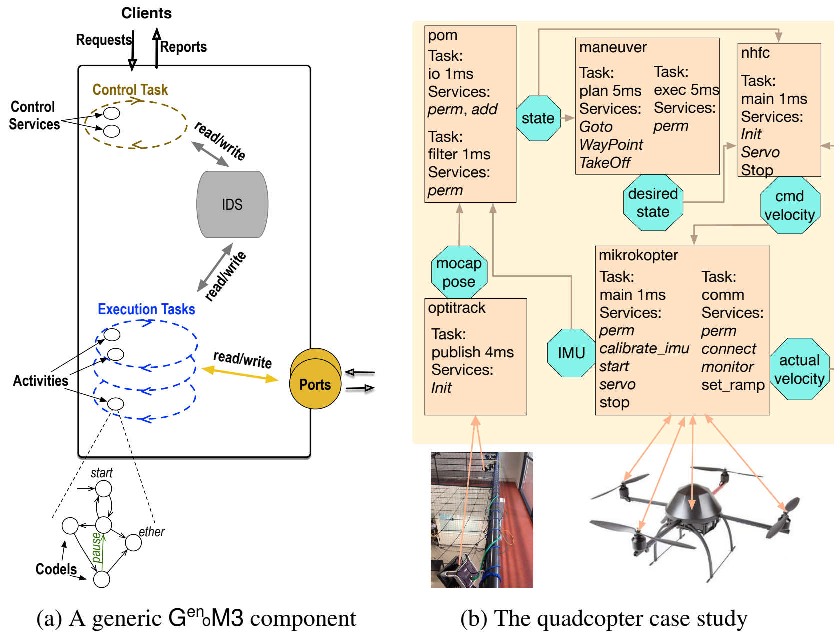
Listing 1: Activity *MoveDistance*

2.1.1 Behavior

We briefly explain how a component behaves. We use the support example of activity *MoveDistance* that belongs to the component DEMO, developed for illustration purposes (listing. 1).

Activities: activities are *finite-state machines* FSM, each state called a *codel*. An activity is executed by the execution task it specifies (line 6 specifies that activity *MoveDistance* is executed by the motion task).

FSM: define the activity behavior through *codels* and *transitions*. A codel is a state at which a chunk of C or C++ code is executed. It specifies its arguments (*e.g.* exec uses the IDS fields *speedRef*, *posRef* and *state* and the port **Mobile**, line 4) and the possible *transitions* subsequent to its execution (*e.g.* start returns exec or ether, line 3). Taking a *pause* transition *pauses* the execution of the activity until the next *cycle* (see below) of its execution task (*e.g.* taking transition pause::exec, line 4, pauses the activity at codel exec from which it will be resumed at the next cycle of task motion). A codel may (optionally) specify a WCET (worst case execution time) on a given platform (*e.g.* end has a WCET of 1 ms, line 5). An FSM has always the codels start (entry point) and ether (end point with no code attached). When the latter is reached, the activity is *terminated* and reported to the client.



(a) A generic GenoM3 component

(b) The quadcopter case study

Fig. 1: Generic GenoM3 component & case study.

Control task: manages requests and reports (from/to clients). When a request for an activity is received, the control task validates it and *activates* such activity (which informs the execution task in charge to execute it). Upon completion of any activity, the control task sends a report to the corresponding client.

Execution tasks: periodic or aperiodic. With each *cycle* (triggered by period or event), an execution task runs, sequentially, all the activities it is in charge of, previously activated by the control task. The execution of an activity ends when it is paused or terminated. In the former case, the activity is resumed at the next cycle.

IDS & concurrency: Tasks are run as parallel threads, with fine-grain concurrent access to the IDS: only the required field(s) by a codel (in its activity, run in a task) are locked when it executes and simultaneous readings are allowed.

2.1.2 Templates GenoM3 features an automatic generation mechanism based on *templates*. A template may access all the component information (*e.g.* tasks periods, activities and their codels) and generate text files with no restrictions (examples in Sect. 5.2). There are templates that, for instance, generate component implementations for PocoLibs [1] and ROS-Comm [26] middleware. These implementation templates also collect codels execution time, which are reported (average and WCET) upon completion, and the number of occurrences of transitions in all activities (Sect. 5.2).

2.1.3 Case study In this paper, we consider the quadcopter in Fig. 1b. In Sect. 6, we explain how we use the components for a navigation mission. For technical details on each component (out of the scope of this paper), we refer the interested reader to [8].

```
1     process example () {
2     clock c;
3     state l0 {;10}, l1{x<=2}, l2{x<=1}; branchpoint l1_b; init l0;
4     trans  l0 -> l1 {assign x:=0; },
5            l1 -> l1_b {guard x>0; }
6            l1_b -> l0 { probability 1; },
7            l1_b -> l2 {assign x:=0; probability 2; },
8            l2 -> l0 { guard x>0; };
9    }
```

Listing 2: STA example in .xta format

2.2 UPPAAL

UPPAAL [2] is a real-time model checker. Models are based on *timed automata* (*TA*) and supported properties are mainly safety, liveness and bounded response.

Timed Automata: A TA [16] is a tuple $\langle L, l_0, X, \Sigma, E, I \rangle$ where L is a finite set of locations, $l_0 \in L$ is the initial location, X is a finite set of clocks, Σ is a finite set of actions including synchronization and internal actions, E is a finite set of edges of the form (l, g, a, φ, l'), with $l, l' \in L$, g a predicate on $\mathbb{R}^X$, $a \in \Sigma$, and φ a binary relation on $\mathbb{R}^X$, and I assigns an invariant predicate $I(l)$ to any location l.

Extending TA: In a TA, urgencies are expressed locally through invariants. For global urgencies, *e.g.* involving different TA, UTA [4] are introduced. In a UTA, when an *eager* edge (denoted $\wr$) is enabled, time cannot progress and the edge must be taken (or disabled by taking another edge) immediately. TA can also be extended with data variables. We refer to UTA extended with data as DUTA. Fig. 2 shows a DUTA example with two locations, $l0$ (initial, denoted with an inner circle), and $l1$, and one $\wr$ edge. Guards are in green, invariants in purple and operations in blue. ex (resp. O) is a Boolean expression (resp. some operations) over some variables. In this example, if the guard remains false for more than 3 time units, the DUTA *timelocks*.

UPPAAL supports a subclass of DUTA that allows (i) *urgent channels* (over which only time-constraint-free edges may synchronize), but not eager edges (example in Sect. 5) and (ii) Boolean and integer data types and functions without pointers.

2.3 UPPAAL-SMC

UPPAAL-SMC is an extension of UPPAAL based on *stochastic timed automata STA*.

Stochastic Timed Automata: An STA is a tuple $\langle TA, \mu, \gamma \rangle$ where
$TA = \langle L, l_0, X, \Sigma, E, I \rangle$ is a timed automaton (Sect. 2.2), μ is the set of density delay functions $\mu_s \in L \times \mathbb{R}^X$, which can be either uniform or exponential distribution, and γ is the set of probability functions γ_s over Σ in TA.

In brief, STA extend TA with (i) density functions (on locations) and (ii) probabilities (on edges). Since we target STA as supported by UPPAAL-SMC, we show an STA example in the *.xta* format (listing 2). If the location has an associated invariant (*e.g.* $l1$, line 3), the density function is a uniform distribution (exponential distribution with a user-supplied rate otherwise, *e.g.* *10* on $l0$, line 3). Probabilities, uniform by default, can be added using (i) a *branchpoint* (lines 5 to 7) and (ii) the keyword “probability” followed by the number of occurrences, used to compute probabilities (the probability to take the edge from $l1$ to $l0$ (resp. to $l2$) is $1/3$ (resp. $2/3$)).

Verification in UPPAAL-SMC: In this paper, we are interested in *probability evaluation*, that is estimating the probability $Pr[<= b](Op_{x \leq d}\phi)$ where b is a time bound on runs, Op is either $\Diamond$ or $\Box$ and ϕ lies within the *Weighted Metric Temporal Logic* $WMTL_{\leq}$ [5] grammar (atomic propositions endowed with U, the *until* operator and O, the *next* operator).

3 Formalizing G$^{en}_{o}$M3

We semanticize G$^{en}_{o}$M3 components using *timed transition systems TTS* (Sect. 3.1). For readability and space, control task and aperiodic behaviors are excluded. This version preserves important mechanisms, *e.g.* concurrency, and the more complex version can be found in [11]. Since the control task is excluded, we will often refer to an execution task as simply *task*.

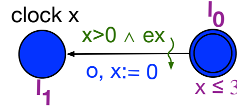


Fig. 2: A generic DUTA example

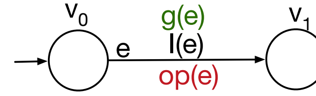


Fig. 3: A generic TTD example

3.1 Timed transition systems

We propose a variation of TTS in [15] where (i) a dense-time model (time intervals have durations in $\mathbb{R}_{\geq 0}$ with bounds in $\mathbb{Q}_{\geq 0} \cup \infty$) is considered instead of a discrete one and (ii) more general time intervals (left- and right-open) are accepted. TTS are suitable to semanticize G$^{en}_{o}$M3. For instance, they are convenient to formalize the *global urgency constraints* (*e.g.* a codel executes *as soon as* it has the required (shared) resources, Sect. 2.1.1), as opposed to clock-based transition systems such as TA where urgencies are expressed only locally (see examples in [8]). Semantics in TTS also allowed automatic mapping to *Fiacre* in [10].

Let $\mathbb{I}$ be the set of well-formed (time) intervals. An element i of $\mathbb{I}$ can have the form: (f1) $[a, b]$ (f2) $]a, b]$ (f3) $[a, b[$ or (f4) $]a, b[$, where $a \in \mathbb{Q}_{\geq 0}$, $b \in \mathbb{Q}_{\geq 0} \cup \infty$, and with $a \leqslant b$ for f1 ($a < b$ otherwise). Interval i is thus the set of reals $x \in \mathbb{R}_{\geq 0}$ such that $a \leq x \leq b$ (f1), $a < x \leq b$ (f2), $a \leq x < b$ (f3), $a < x < b$ (f4). In any form, we say that $\downarrow i = a$ (resp. $\uparrow i = b$) is the lower (resp. upper) bound of i.

A TTS is a tuple $\langle U, S, s_0, \tau, I \rangle$ where:
- U is a finite set variables,
- S is a set of states. Each state of S is an interpretation of variables in U,
- s_0 is the initial state ($s_0 \in S$) that maps each variable in U to its initial value,
- τ is a set of transitions. Each transition $t \in \tau$ defines for every state $s \in S$ a (possibly empty) set of successors $t(s) \subseteq S$,
- $I : \tau \mapsto \mathbb{I}$ maps each transition $t \in \tau$ to a *static (time) interval* $I(t) \in \mathbb{I}$.

The semantic "meaning" of time intervals depends on the *enabledness* of transitions: if transition t is enabled at s (s is the current state of the TTS and $t(s) \neq \varnothing$) since date Δ then we can *take* t starting at date d s.t. $\Delta + \downarrow I(t) < d$ if $I(t)$ is of form (f2) or (f4) ($\Delta + \downarrow I(t) \leq d$ otherwise) and *must* take it no later than date $d' < \Delta + \uparrow I(t)$ if $I(t)$ is of form (f3) or (f4) ($d' \leq \Delta + \uparrow I(t)$ otherwise), unless it is disabled in between

by taking another transition. If t is disabled, then *I(t)* has no semantic effect (detailed semantics in [11]).

3.1.1 TTDs A timed transition diagram TTD (inspired from [15]) is a finite directed graph with a set of vertices V and a set of edges E. The unique initial vertex is $v_0 \in V$. Each *edge* $e \in E$ is labeled with: an interval $I(e)$ (omitted if equal to $[0, \infty[$); a guard g_e (omitted if tautology); and an atomic sequence of operations op_e (omitted if has no side effects). An edge e connecting vertex v to vertex v' is denoted, interchangeably, $e \in E$ or $v \xrightarrow{e} v' \in E$. Fig. 3 shows a simple generic TTD with two vertices, v_0 (initial, denoted with an incoming edge without source) and v_1, and one edge e.

3.1.2 Composition of TTDs The parallel composition of n TTDs, $P_1, \ldots, P_n$, over a set of shared variables, U_s, results in a TTS $\{\Theta\}[\,\|_{i \in 1..n}\, P_i\,]$, where Θ gives the initial valuations of each variable in U_s and each component P_i accesses U_s and a set of local variables U_i. For detailed semantics of such TTS, we refer the interested reader to [11].

For simplicity, we stop referring to the names of edges in TTDs: $v \xrightarrow{e} v'$ (Sect. 3.1.1) will be referred to, from now on, as simply $v \to v'$, or $v \to$ (resp. $\to v'$) when the identity of v' (resp. v) is irrelevant. This is because in our G^{en}oM3 semantics (Sect. 3.3), edges are uniquely defined through their source and target vertices.

3.2 Syntax and syntactical restrictions of a GenoM3 component

3.2.1 Activity An activity A is a tuple $\langle C_A, W_A, T_A, T_A^P \rangle$ where:
- C_A is a set of codels with at least two codels (for starting and termination, Sect. 2.1.1): $\{start_A, ether_A\} \subseteq C_A$,
- $W_A : C_A \backslash \{ether_A\} \mapsto \mathbb{Q}_{>0}$ associates to every codel its WCET (Sect. 2). The codel $ether_A$ (reserved for termination) is excluded (no code attached to it, Sect. 2.1.1),
- T_A is a set of transitions of the form $c \to c'$ (each transition is uniquely defined through its source codel c and target codel c'). We denote this relation by simply $c \to$ (or $\to c'$) when the identity of codel c (or c') is unimportant,
- $T_A^P \subseteq T_A$ is the set of *pause* transitions.

3.2.2 Task A task T is a triple $\langle Per, \mathcal{A}, V \rangle$ where:
- $Per \in \mathbb{Q}_{>0}$ is the period,
- $\mathcal{A}$ is the non-empty set of activities T is in charge of,
- V is a set of variables.

3.2.3 Component A component $Comp$ is a triple $\langle E, V, \mu \rangle$ where:
- E is a set of tasks,
- V is a set of variables,
- $\mu : C \mapsto \mathcal{P}(C)$ is the *conflict* function, where C is the union of all codels in all activities of all tasks in E and $\mathcal{P}(C)$ its powerset. $\mu(c)$ is the set of codels that are *in conflict* (cannot execute simultaneously) with c. If $\mu(c) = \varnothing$ then c is *thread safe* (*thread unsafe* otherwise).

3.2.4 Well-formed components *Well-formed* components are defined by the following syntactic restrictions. For any activity A, we require that (i) each codel in $C_A \backslash \{ether_A\}$ has at least one successor in the relation defined by T_A, (ii) T_A must not include any transition whose source codel is $ether_A$ (reserved for termination), and (iii) $ether_A$ cannot be the target of a *pause* transition because the latter is for pausing

while the former is for termination. These requirements can be expressed succinctly as follows:

$$\forall c \in C_A \backslash \{ether_A\}\, \exists c' \in C_A : (c \to c' \in T_A)$$
$$\forall c, c' \in C_A : (c \to c' \in T_A) \Rightarrow (c \neq ether_A)$$
$$\forall c, c' \in C_A : (c \to c' \in T_A^P) \Rightarrow (c' \neq ether_A)$$

Finally, $ether$ codels are thread safe. Also, there is no conflict within the same task: any two activities A and B in the same task are executed sequentially "by construction" (one task = one thread). Therefore, we require that $\mu(c) \cap C_B = \mu(c') \cap C_A = \varnothing$ for all c in C_A and c' in C_B.

3.3 Operational semantics of a GenoM3 component

Before we go further, we need to distinguish between what the programmer specifies (reflected at the syntactical level, *e.g.* in transitions T_A, Sect. 3.2.1), and what is enforced to produce the expected behavior (*e.g. starting* and *mutual exclusion* edges, Definition 3). We present operational semantics "top-down", from component to activities.

3.3.1 Component semantics A component $Comp$ semantics is given by the TTS $Comp = \{\Theta\}[\|_{i \in 1..n}\ T_i]$ where $n = \mid E \mid$ is the number of tasks in E (Sect. 3.2.3) and T_i are tasks. For each codel $c \in C$ s.t. $\mu(c) \neq \varnothing$ (Sect. 3.2.3), there is a Boolean r_c in the set of shared variables U_s (V in Sect. 3.2.3), initially false ($\Theta(r_c) = False$ for all $r_c \in U_s$). These variables help semanticize concurrency (Definition 3).

3.3.2 Task semantics. The semantics of a task is given by the TTS $T = \{\Theta\}[Tim||M||(\|_{A \in \mathcal{A}}\ A)]$ where Tim is the *timer* (Definition 1), M is the task *manager* (Definition 2), and $\|_{A \in \mathcal{A}}\ A$ is the composition of all activities A (Definition 3) in $\mathcal{A}$ (Sect. 3.2.2). The set of shared variables U_s (V in Sect. 3.2.2) contains: N, the set of "names" of activities to execute, sig, the period signal, and Π, the *control passing* variable. Π ranges over TTDs "names" (by abuse of notation, M is the name of the manager TTD and the name of activity A is A), N has the same type as Π excluding M, and sig is a Boolean. The initial values are $\Theta(N) = \varnothing$, $\Theta(sig) = False$, and $\Theta(\Pi) = M$ (the manager has the control when the system starts).

Definition 1 ***Timer semantics.*** *The timer semantics is given in Fig. 4.*

Changing the value of sig to *true* corresponds to transmitting a signal asynchronously to the *manager* (Definition 2). The time interval $[Per, Per]$ ensures that this signal is transmitted at exactly each period (each Per time units).

Definition 2 ***Manager semantics.*** *The manager semantics is given in Fig. 5.*

Vertex *wait* denotes waiting for the next period and *manage* is to execute activities, if any. The operation $\Pi := rand(N)$ gives the control to one of the activities in N (by assigning randomly an element from N to Π). The manager transits back to *wait* as soon as it has the control and N is empty.

Since $\Theta(N) = \varnothing$, no activity would ever be executed. This is because fulfilling activities requests is the role of the control task that we do not represent here. Therefore, the manager performs the operation $rrand(N)$ to initialize N randomly, over the set of activities T is in charge of; while respecting the condition $(A \in N \wedge B \in N) \Rightarrow (A \neq$

B). The operation $rrand(N)$ covers all the possible evolutions of tasks, as the resulting set of configurations of N is a superset of that obtained when a control task is present (details in [11]). Note how the guard on the edge from *wait* to *manage* does not contain

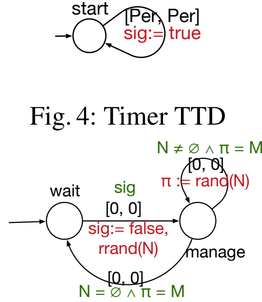


Fig. 4: Timer TTD

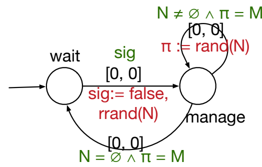


Fig. 5: Manager TTD

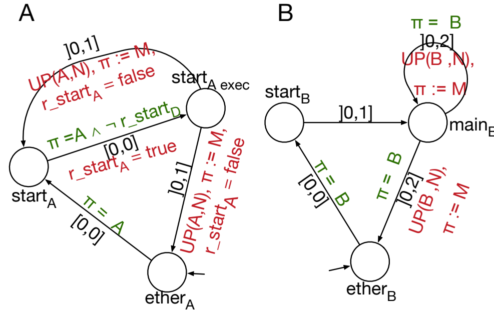


Fig. 6: Activities A and B in task T

the clause $\Pi = M$ because this is always true at vertex $wait$ ($\Theta(\Pi) = M$ and the manager cannot lose the control at vertex *wait*).

Definition 3 ***Activities semantics.*** *The operational semantics of an activity* $\langle C_A, W_A, T_A, T_A^P \rangle$ *(Sect. 3.2.1) is given by a TTD such that:*
- Vertices V: each $c \in C_A$ is mapped to a vertex $c \in V$. A vertex $c_{exec} \in V$ is added for each thread-unsafe codel c ($\mu(c) \neq \varnothing$, Sect. 3.2.3). The initial vertex v_0 is $ether_A$,
- Edges $E = E^N \cup E^A$ are nominal (in E^N) or additional (in E^A):

- *E^N: each transition $c \to c'$ in T_A is mapped to an edge $c \to c'$ (resp. $c_{exec} \to c'$) in E^N if $\mu(c) = \varnothing$ (resp. otherwise). We distinguish three disjoint sets of nominal edges: $E^N = E^P \cup E^T \cup E^X$. E^P is the set of pause edges that maps the set of pause transitions T^P; E^T is the set of termination edges of the form $\to ether$ and E^X the set of the remaining (execution) edges.*
- *$E^A = E^S \cup E^M$ where E^S contains the starting edge $ether \to start$ and E^M the mutual exclusion edges of the form $c \to c_{exec}$ (for each thread-unsafe codel c).*

- Time intervals I: $I(e) =]0, W_A(c)]$ iff $e \in E^N$ ($I(e) = [0, 0]$ otherwise).
Now we define the guards and operations:
- Each edge in $E^T \cup E^P$ is augmented with the operation $\Pi := M$ and the operation $UP(A, N)$ that removes A (the activity "name") from N,
- The edge in E^S, and each edge $c \to$ in $E^N \cup E^M$ such that exists an edge $\to c$ in E^P, are guarded with $\Pi = A$,
- Each edge $c \to$ in E^M is augmented with the operation $r_c := true$ (see shared variables in Sect. 3.3.1).
Finally, (i) the guard of each edge $c \to$ in E^M is conjuncted with the expression $\forall c' \in \mu(c) : \neg r_c'$ and (ii) $r_c := false$ is added to the operations of each edge $c_{exec} \to$ in E^N.

Nominal edges map transitions that the programmer specifies, while additional edges reflect actions enforced by G^{en}$_{o}$M3 to handle starting and concurrency. Edges are uniquely defined through their source and target vertices. For activities, this can be con-

cluded from syntax, restrictions and semantics (Sect. 3.2.1, Sect. 3.2.4 and Definition 3). For the *manager* and the *timer*, it is shown in Fig. 4 and Fig. 5.

Let us illustrate through an example how activities evolve following these semantics, and how this coincides with the behavior in Sect. 2.1.1. We consider a component with two tasks T and T'. T is in charge of two activities A and B (on which we focus) while T' is in charge of one activity D. We give the syntactical definitions of A and B:

Activity A
- $C_A = \{start_A, ether_A\}$,
- $W_A(start_A) = 1$,
- $T_A = \{start_A \rightarrow start_A,$ $start_A \rightarrow ether_A\}$,
- $T_A^P = \{start_A \rightarrow start_A\}$.

Activity B
- $C_B = \{start_B, main_B, ether_B\}$,
- $W_B(start_B) = 1$, $W_B(main_B) = 2$,
- $T_B = \{start_B \rightarrow main_B,$ $main_B \rightarrow main_B, main_B \rightarrow ether_B\}$,
- $T_B^P = \{main_B \rightarrow main_B\}$.

Now, because of the mutual exclusion between T and T', the $start$ codels of A (in T) and D (in T') are in conflict: $\mu(start_A) = \{start_D\}$ (and symmetrically $\mu(start_D) = \{start_A\}$). The remaining codels are thread safe.

We apply Definition 3 to get the TTDs of A and B in Fig. 6 evolving within T (the *manager* and *timer* (generic) TTDs are given in Fig. 5 and Fig. 4, respectively). Starting an activity, from $ether$ or wherever it was paused last, is subject to having the control through Π (*e.g.* edge $ether_B \rightarrow start_B$). At the end of execution, either by pausing (*e.g.* edge $main_B \rightarrow main_B$) or terminating (*e.g.* edge $start_{A\ exec} \rightarrow ether_A$), the control is given back to the manager ($\Pi := M$), and the activity removes its "name" from N ($UP()$, no further execution for this activity in this cycle). Π ensures thus a *sequential* behavior within the same task, that is between the manager and each A in $\mathcal{A}$ (no two edges in two different TTDs can be enabled simultaneously).

At the codels level, outgoing edges of vertices c (the underlying codel is thread safe, *e.g.* $start_B$) and c_{exec} (otherwise, *e.g.* $start_A$) are associated with the interval $]0, W(c)]$ to reflect that the execution of a codel takes between a non-null time and its WCET. Boolean expressions involving r_c' variables, which take part in the guards on edges $c \rightarrow c_{exec}$, prevent the thread-unsafe codel c to execute if there is at least a codel in $\mu(c)$ that is already running, and the time interval $[0, 0]$ allows it to execute *as soon as* this is no longer the case. For instance, the guard on $start_A \rightarrow start_{A\ exec}$ disables this very edge (even when A has the control) as long as the activity D (in the concurrent task T', not shown here) is at vertex $start_{D\ exec}$ (denoting the execution of $start_D$), captured through the truth of the Boolean r_start_D. Similarly, operations $r_c := true$ on edges $c \rightarrow c_{exec}$ prevent thread-unsafe codels in $\mu(c)$ to run in parallel with c (*e.g.* $r_start_A := true$ on $start_A \rightarrow start_{A\ exec}$). Finally, operation $r_c := false$ on edges of the form $c_{exec} \rightarrow$ (*e.g.* $r_start_A := false$) allow activities with codels in conflict with c to capture the end of execution of c through the falseness of r_c.

4 Translation

TTS semantics are translated to DUTA in order to automatically map G^{en}oM3 to UPPAAL and UPPAAL-SMC. We show the translation for activities, since it is rather straightforward for the manager and the timer (Fig. 7).

Mapping intervals into clock constraints and $\wr$ edges may lead to incorrect translations, as shown in Fig. 8 (activity B). Indeed, if B_{ta} is paused (taking $main_B \rightarrow main_B$),

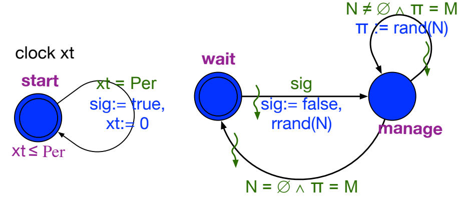


Fig. 7: DUTA translation of manager and timer

it will timelock after 2 time units unless it resumes the control before then (all outgoing edges from location $main_B$ are disabled). This is encountered when there is a vertex in the TTD that (i) maps a thread-safe codel and (ii) is the target of a pause edge. This problem is due to clocks evolving independently from edges enabledness in DUTA (in contrast to intervals in TTDs). We propose a generic translation for all activities.

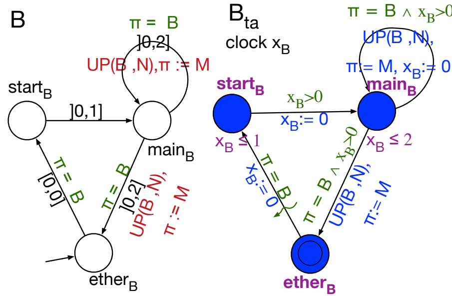


Fig. 8: Incorrect translation (activity B)

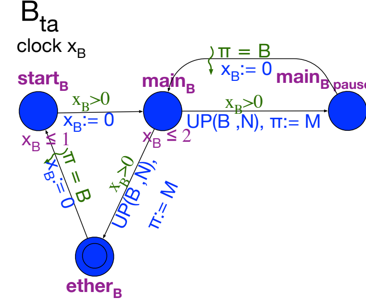


Fig. 9: Correct translation (activity B)

Definition 4 ***Activities*** A_{ta} ***(DUTA).*** *The DUTA translation A_{ta} of the TTD A (Definition 3), is given by the following rules:*
- Clocks: A_{ta} has a clock x_A, whose initial valuation is zero,
- Locations: Each vertex c in A of a thread-safe codel c s.t. there exists $\rightarrow c$ in T^P is mapped to two locations c and c_{pause}. Each remaining vertex in A is mapped to a location with the same name. Each location c that maps a vertex $c \neq ether$ of a thread-safe codel is associated with an invariant $x_A \leq \uparrow I(c \rightarrow)$ with $c \rightarrow$ any outgoing edge of c. The same invariant rule is applied to each location c_{exec},
- Edges: - Each pause edge $c \xrightarrow{g,op} c'$ in A s.t. c' is thread safe is mapped to an edge $c \xrightarrow{x_A>0,op} c'_{pause}$, and an eager edge $c'_{pause} \xrightarrow{g,x:=0} c'$ is added.
- Each remaining edge in A is mapped to an edge in A_{ta} with the same source and target, where: (1) intervals $[0,0]$ are mapped into $\wr$ edges, (2) each outgoing (resp. incoming) edge of a location associated with an invariant is guarded (resp. augmented) with $x_A > 0$ (resp. with $x_A := 0$), then (3) guards (resp. operations) associated with each edge result from the conjunction (resp. sequencing) of guards (resp. operations) of its TTD counterpart and the guards (resp. resets) over clocks.

These rules allow clocks to evolve unboundedly at locations c_{pause} (when the activity is paused). Resuming the activity is then equivalent to taking the edge $c_{pause} \rightarrow c$ with a clock reset to count the WCET of c starting from 0, which we may see when applying Definition 4 to activity B (Fig. 9).

Translation soundness: DUTA models must be faithful to the G^{en}oM3 semantics. We use weak timed bisimulation to prove that the translation is sound. Details on the proof may be found in [11].

5 Automatic mapping

We see how the DUTA models are automatically mapped into UPPAAL and UPPAAL-SMC. In order to do so, we first present the current implementation.

Implementation: In the actual implementation (either in ROS-Comm or PocoLibs middleware), the set of activities to execute (N) is substituted with an array run of size $n = |\mathcal{A}|$ (the number of activities in the task) of *records*, starting at index 0. Each record is composed of two fields: an activity "name" m and its "status" s, that may be $requested$ (r) or $idle$ (d), equivalent, respectively, to $A \in N$ and $A \notin N$ in the semantics. The operation $arand(t)$ initializes the status s fields of array t randomly. The variable i, initially equal to 0, ranges from 0 to n. The function $next(t, b)$ browses the array t, starting from index b, and returns the index of the first element with $s = r$ ($|t|$ if such an element is not found or $b = |t|$).

The implementation of a task is then derived from its semantics as follows. For any activity A, each operation $UP(A, N)$ is replaced by $i := i + 1, i := next(run, i)$. In the manager, the guard $N \neq \varnothing$ (resp. $N = \varnothing$) is replaced by $i \neq n$ (resp. $i = n$), the operation $\Pi := rand(N)$ by $\Pi := run[i].m$, and the operation $rrand(N)$ by $arand(run), i := next(run, i)$ (in reality, the *run* array is updated by the control task, not considered in our presentation). Finally, the edge $manage \rightarrow wait$ in the manager is augmented with the operation $i := 0$. Accordingly, the implementation model of task T (Fig. 5, 4, 6) is given in Fig. 10. Trivially, the semantics (allowing random "scheduling" of activities) is a superset of the implementation (where the order of execution of activities is predefined when initializing names fields (m) in run). The random scheduling at the semantics level allows to derive different implementations if needed. For DUTA, it is sufficient to apply the TTD-DUTA translation rules. Fig. 11 gives the DUTA implementation of activity A (Fig. 10).

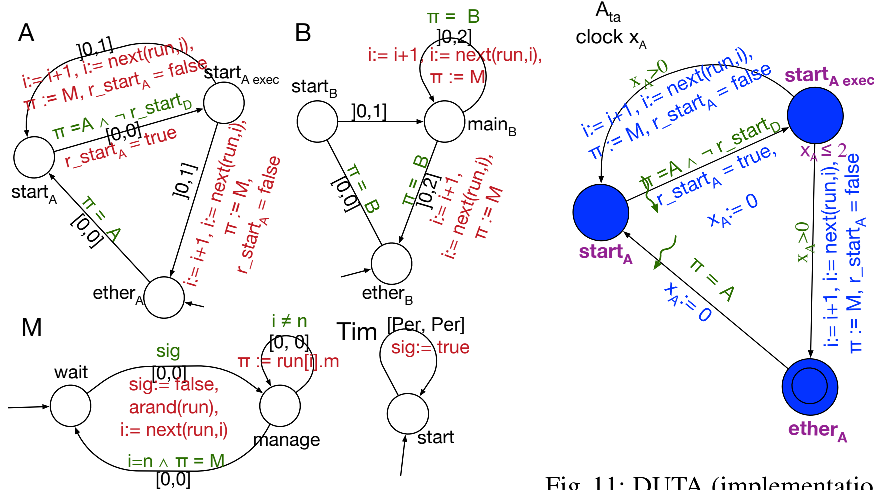


Fig. 10: TTDs in task T (implementation).

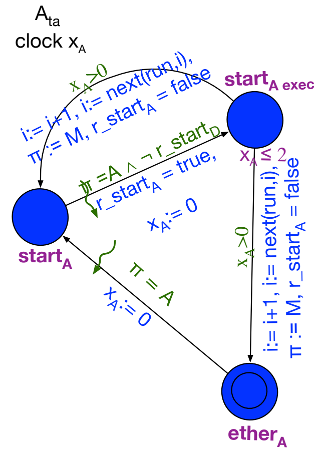


Fig. 11: DUTA (implementation) of activity A.

```
1    process A (urgent chan &exe, int[M, size_run] &pi, int[0, size_run] &i,
         CELL &run[size_run], bool &mut[size_mut]) {
2    clock x;
3    state ether, start, start_exec {x<=1}; init ether;
4    trans // behavior
5    // additional edges
6    ether -> start { guard pi = A ; sync exe!; };
7    start -> start_exec { guard pi = A && !mut[r_start_D]; sync exe!;
         assign x:= 0, mut[r_start_A]:= true; };
8    // nominal edges
9   ...
```

Listing 3: Process A (UPPAAL)

5.1 Mapping to UPPAAL

We see how to model an activity. First, we deal with urgent edges (UPPAAL only allows urgent *channels*, Sect. 2.2). We add a process $urgency$ and synchronize its unique edge, over an urgent channel exe, with each $eager$ edge in the activity (and with all eager edges in the system):

```
process Urgency(urgent chan &exe) {
state idle; init idle;
trans
       idle -> idle { sync exe?; };
}
```

Now we can model *e.g.* activity A. Listing. 3 is a partial UPPAAL model of A (only additional edges are shown). Constant $M = \mathit{0}$ denotes the manager, so $\mathit{\Pi}$ ranges over $[M, \mathit{size_run}]$ (line 1) where activity names are encoded in turn as constant integers in this range. CELL is the record type for run (line 1) and mut is an array that facilitates implementing mutual exclusion variables (r_c becomes $mut[r_c]$, line 1, 7).

5.2 Automatic synthesis

We generalize the approach for automatic synthesis using the template mechanism (Sect. 2.1.2). We develop a template that generates automatically the UPPAAL model for any G^{en}oM3 specification (made of any number of components). We show an example on how additional edges are generated for a given activity a (listing 4). The interpreter outputs everything as is, except what is enclosed in <' '> that it evaluates in Tcl, and in <" "> that it evaluates and outputs the result.

In lines 3 to 7, we check each outgoing transitions of each codel (keyword yields), and append the successor to the list p if such transition is a pause. We also append the codel c to the list tu if its field $mutex$, which contains the codels c is in conflict with, is not empty. Therefore, p contains all the codels targeted by a pause and tu all thread-unsafe codels in a. At line 11, we generate the starting edge, then the mutual exclusion edges from line 12 to 20, where, for each thread-unsafe codel c, we add the guard on having the control through $\mathit{\Pi}$ if c is also in p (applying Definition 4 and inductively Definition 3). The task name is added to distinguish variables in different tasks.

Extending to UPPAAL-SMC: Implementation templates (Sect. 2.1.2) generate, for each transition in each activity, a line with the number of its occurrences:

```
task_name/activity_name/source_codel_name/target_codel_name/<#occurrence>
```

```
1  <'set p [list]'>
2  <'set tu [list]'>
3  <'foreach c [$a codels] {'>
4  <'  foreach y [$c yields] {'>
5  <'    if {[$y kind] == "pause" && !($y in $p)} {lappend p $y}}'>
6  <'  if {[llength [$c mutex]]} {lappend tu $c}'>
7  <'} '>
8  ...
9  trans //behavior
10 // additional edges
11 ether-> start {guard pi_<"[$t name]"> = <"[$a name]">; sync exe!; };
12 <'foreach c in $tu {'>
13 <"[$c name]"> -> <"[$c name]">_exec {guard
14 <'  if {$c in $p} {'>
15   pi_<"[$t name]"> = <"[$a name]"> &&
16 <'  }'>!(
17 <'    foreach m [$c mutex] {'>mut[r_<"[$m name]">]
18 <'      if {$m != [lindex [$c mutex] last]} {'> || <' }'>
19 <'    }'>); sync exe!; assign x:=0, mut[r_<"[$c name]">]:= true;};
20 <'}'>
21 // nominal edges
22 ...
```

Listing 4: Generating additional edges (for an activity a in task t)

A *.proba* file is thus constructed, then passed as an argument to the UPPAAL-SMC template, together with the G^en_oM3 specification. Listing 5 shows an excerpt of the UPPAAL-SMC template . For simplicity, we only show the case where the source codel is thread safe and none of its outgoing transitions is pause or termination. Line 3 conditions adding probabilities by the existence of more than one successor. Line 5 connects the edge to a branchpoint (as shown in Sect. 5.1). Lines 6-8 generate the outgoing edges of the branchpoint and extract occurrences from the *.proba* file.

```
1 <' foreach c [$a codels] {'>
2        ...
3 <' if {[llength [$c yields]] > 1} {'>
4 <' set pr [join [list [$t name] [$a name] [$c cname] [$y cname]] /]'>
5 <"[$c name]"> -> <"[$c name]">_b {guard x>0; },
6 <' foreach y [$c yields] {'>
7 <"[$c name]">_b -> <"[$y name]"> {;probability <"[dict get $argv $pr]">;},
8 <' }}'>
9        ...
```

Listing 5: Generating probabilistic transitions (for an activity a in task t)

6 Verification results

We use the automatically generated models (Sect. 5) to specify and verify important real-time properties on the quadcopter case study (Sect. 2.1.3). Experiments are carried out on a laptop (Intel Core i7; 16 GB of RAM). Tasks are assigned to independent cores on the hardware. Experiments, with instructions on how to reproduce them, are freely clonable from https://github.com/Mo-F/uppaal-smc-exp.

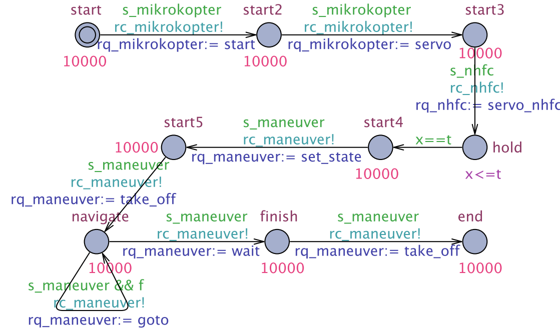


Fig. 12: UPPAAL-SMC client (quadcopter navigation).

6.1 Model checking

With UPPAAL, we get the same results as with the Fiacre template in [8]: the *stationary flight* application (excluding the component MANEUVER) scales, while the navigation application (involving all components) does not. We use UPPAAL-SMC for the latter.

6.2 Statistical model checking

As seen in Sect. 2, components need to receive requests from clients to run. For that, we add a client to ensure a navigation application (see below). The automatically generated UPPAAL-SMC model of the quadcopter plus the added client make 36 complex processes overall, on which we carry out the statistical verification.

6.2.1 Client The client (Fig. 12) uses urgent channels rc_X (X is a component) to send activities requests to components, through rq_X variables. Since UPPAAL-SMC supports only broadcast channels, we guard each channel rc_X with the Boolean s_X, true only when X is ready to receive a request (which forces a rendezvous behavior). Location $hold$ is for waiting an amount t between sending servoing requests (NHFC and MIKROKOPTER) and taking off (MANEUVER), as servoing must have already started before taking off (which is an important property to verify). Exponential rates are required on invariant-free locations (high rates imply a high probability to leave the location at smaller time values, but values are unimportant here because of the urgencies enforced by rc_X channels). The self-loop at location $navigate$ enables, using the Boolean f, issuing a new *goto* request each time the last *goto* activity (to navigate) has ended (goal invalid, reached, or unreachable). From the same location, a request *wait* then *take_off* can be sent (to land). The client covers thus all the possible scenarios of navigation.

6.2.2 Properties of interest The following properties are crucial such that accidents may occur if they are not satisfied.
Readiness: When requests are sent to MANEUVER, the previously requested activities from MIKROKOPTER and NHFC must have already started executing. Find the minimum value of t to satisfy this property with the highest possible probability.
Schedulability: Estimate the probability of schedulability of periodic tasks in the critical components POM, MIKROKOPTER and NHFC.

6.2.3 Verification with UPPAAL-SMC Statistical parameters are set to a high confidence (0.98) and precision (0.005), and the runs are bounded to $b = 10\,\mathrm{s}$.

Readiness: Readiness is typically a bounded response property, not supported by UPPAAL-SMC. We propose an alternative using the *Until* operator. An activity starts once its codel `start` begins executing, which is equivalent to reaching the location *start_exec* (since none of the codels `start` in this context is thread safe). Therefore, the *client* "cl" must not reach location *start4* (from which it sends requests to MANEUVER) before locations *start_exec* of each previously requested activity (*start* and *servo* (MIKROKOPTER) and *servo* in NHFC) is reached. Readiness boils down then to the conjunction of the three *Until* properties in listing 6.

```
1 ap: cl.start or cl.start2 or client.start3 or cl.hold
2 p1: ap U start_mikrokopter.start_exec
3 p2: ap U servo_mikrokopter.start_exec
4 p3: ap U servo_nhfc.start_exec
5 rp: p1 and p2 and p3
```

Listing 6: Readiness property rp

Note that attempting to reduce these properties to only one using the conjunction of their right terms would result in a stricter property (*e.g.* *start_exec* of *servo* may be left before *start_exec* of *servo_nhfc* is reached). We tune t starting from 1 ms. The highest possible probability is returned by the verifier ($\geq 99\%$ considering the precision, 0.005 ± 0.005) for all of the three properties as soon as t is equal to 8 ms. Results for $p3$ for different values of t are given in table 1. Therefore, in order to ensure a high probability of satisfying *Readiness*, t may have any value larger than 8 ms. We fix it to 1 s.

t (ms)	Results	Runs	Time
7	$Pr \in [0.98, 0.99]$	*3279*	*12*
8	$Pr \in [0.99, 1]$	*1595*	*6*
100	$Pr \in [0.99, 1]$	*390*	*3*

Table 1: Analysis results for p3 (listing 6) with the query $Pr[<= b]p3$.

Task	Query	Results	Runs	Time
io	$Pr[<= b]vs_{io}$	$Pr \in [0, 0.01]$	*390*	*966*
filter	$Pr[<= b]vs_{filter}$	$Pr \in [0, 0.01]$	*390*	*962*

Table 2: Analysis results for schedulability (POM)

Schedulability: It is reduced to a reachability property. Indeed, it is sufficient to verify that whenever the *manager* is executing activities (at location *manage*), no new period signal is received (sig is false), see Fig. 7. The probability of violating this property is the lowest possible for all tasks of the critical components POM, MIKROKOPTER and NHFC ($\leq 1\%$). Examples of results on POM tasks are given in table 2 with vs_T being the violation of schedulability of task T: $<> manager_T.manage \ and \ sig_T$.

6.2.4 Discussion While we cannot verify some properties in a precise way (due to scalability issues with model checking), the results we get with UPPAAL-SMC are encouraging. We verify important properties up to a high probability, which is better than classical scenario-based testing. The verification is cost effective: around 15 minutes in the worst case, and a remarkably low memory consumption (less than 15 mb). Nevertheless, two main issues are encountered, besides non exhaustivity. First, though 99% is fair for this application, we generally lack precise requirements expressed probabilistically in the robotics domain. Second, the expressiveness of UPPAAL-SMC query language is limited (*e.g.* bounded response properties are not supported). While we often manage, with some artefacts, to verify closer alternatives, such artefacts need a proficiency with formal languages that robotic practitioners do not possess.

7 Related work

Model checking: The synchronous language ESTEREL [3] is used in some model-checking-based verification works such as [18, 30, 31], where the robotic specifications are either translated by hand to, or hard-coded in ESTEREL. Efforts such as [24] rely on automatic translation of *RoboChart* models into CSP [27] in order to verify real-time properties. However, RoboChart is not a robotic framework (its models are not executable on robotic platforms). That is, robotic applications, initially specified in a robotic framework, need to be modeled first in RoboChart, then translated into CSP. An attempt to formalize ROS components is developed in [13] where UPPAAL is used to verify buffer-related properties (no overflow). Only the message passing part (publisher/subscriber) is modeled, manually, and crucial bounded response properties (*e.g.* messages are delivered within a bounded amount of time), are not verified. Our work distinguishes itself across three main aspects: (i) this is the first work that fully formalizes a robotic framework for functional-layer specifications, (ii) modeling is fully automatized and (iii) only real-world applications are analyzed.

Statistical & probabilistic model checking: Real-time statistical/probabilistic model checking has been used to verify systems in various domains such as communication protocols [21], railway systems [6] and decisional robotics [29]. At the functional layer of robotic systems, statistical and probabilistic model checkers are seldom used. The work presented in [14] is a notable exception. ROS graphs are formalized in an ad-hoc fashion (no operational semantics given), then, on an autonomous vehicle case study, PRISM [20] estimates the probability of finding an object in a bounded amount of time. To the best of our knowledge, our work presented here is the first that applies real-time statistical model checking to complex, concurrent functional layer, where formal models are sound and automatic. The choice of UPPAAL-SMC is motivated by the fact that the automatic translation gives us the opportunity to use regular UPPAAL and resort to UPPAAL-SMC when models do not scale.

Comparison to our previous work: In our previous efforts to verify the quadcopter, model checking scaled only for the stationary flight, excluding the MANEUVER component [8, 9]. This is the first work that verifies the navigation application, involving all the components, through sound and automatic bridging with UPPAAL-SMC.

8 Conclusion

We propose in this paper automatic and sound generation of formal models from robotic specifications, and obtain encouraging results on a real application. Our contributions advance the state of the art toward a correct and practical verification of robotic systems.

However, it is difficult to set the probabilities for properties because we lack this kind of requirements in robotics. We need to investigate further this problem. Moreover, the restricted query language of UPPAAL-SMC forced us to reason on alternatives using the supported operators only. For a robotic programmer, this could be discouraging since it requires a good knowledge of the tool, the query language and the underlying logic. A possible future work consists therefore in developing query-to-query transformations that are transparent to the practitioner. Finally, we are interested in verifying some hardware-related properties using SMC such as energy consumption (as in [28]).

References

1. The PocoLibs middleware https://git.openrobots.org/projects/pocolibs.
2. G. Behrmann, A. David, and K.G. Larsen. A tutorial on UPPAAL. In *Formal Methods for the Design of Real-Time Systems*, pages 200–236. 2004.
3. G. Berry. *The Esterel v5 language primer: version v5_91*. Centre de mathématiques appliquées, Ecole des mines and INRIA, 2000.
4. S. Bornot, J. Sifakis, and S. Tripakis. Modeling urgency in timed systems. In *International Symposium on Compositionality (ISC): the significant difference*, pages 103–129. 1998.
5. P. Bulychev, A. David, K-G. Larsen, A. Legay, G. Li, D. B. Poulsen, and A. Stainer. Monitor-based statistical model checking for weighted metric temporal logic. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning (LPAR)*, pages 168–182. Springer, 2012.
6. Q. Cappart, C. Limbrée, P. Schaus, J. Quilbeuf, L-M. Traonouez, and Axel Legay. Verification of interlocking systems using statistical model checking. In *2017 IEEE 18th International Symposium on High Assurance Systems Engineering (HASE)*, pages 61–68. IEEE, 2017.
7. A. Cimatti, M. Roveri, and P. Bertoli. Conformant planning via symbolic model checking and heuristic search. *Artificial Intelligence*, 159(1-2):127–206, 2004.
8. M. Foughali. Toward a Correct-and-Scalable Verification of Concurrent Robotic Systems: Insights on Formalisms and Tools. In *International Conference on Application of Concurrency to System Design (ACSD)*, pages 29–38, 2017.
9. M. Foughali, B. Berthomieu, S. Dal Zilio, P.-E. Hladik, F. Ingrand, and A. Mallet. Formal verification of complex robotic systems on resource-constrained platforms. In *International Conference on Formal Methods in Software Engineering (FormaliSE)*, pages 2–9, 2018.
10. M. Foughali, B. Berthomieu, S. Dal Zilio, F. Ingrand, and A. Mallet. Model checking real-time properties on the functional layer of autonomous robots. In *International Conference on Formal Engineering Methods (ICFEM)*, pages 383–399. Springer, 2016.
11. M. Foughali, S. Dal Zilio, and F. Ingrand. On the Semantics of the GenoM3 Framework. Technical report, LAAS-CNRS, 2019.
12. D. Hähnel, W. Burgard, and G. Lakemeyer. GOLEX —- bridging the gap between logic (GOLOG) and a real robot. In *Annual Conference on Artificial Intelligence*, pages 165–176. Springer, 1998.
13. R. Halder, J. Proença, N. Macedo, and A. Santos. Formal verification of ROS-based robotic applications using timed-automata. In *International Conference on Formal Methods in Software Engineering (FormaliSE)*, pages 44–50. IEEE/ACM, 2017.
14. M. Hazim, H. Qu, and S. Veres. Testing, verification and improvements of timeliness in ROS processes. In *Annual Conference Towards Autonomous Robotic Systems (TAROS)*, pages 146–157, 2016.
15. T. Henzinger, Z. Manna, and A. Pnueli. Timed transition systems. In *Research and Education in Concurrent Systems*, pages 226–251, 1991.
16. T. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic model checking for real-time systems. *Information and computation*, 111(2):193–244, 1994.
17. F. Ingrand and M. Ghallab. Deliberation for autonomous robots: A survey. *Artificial Intelligence*, 247:10–44, 2017.
18. M. Kim and K. Kang. Formal construction and verification of home service robots: A case study. In *International Symposium on Automated Technology for Verification and Analysis (ATVA)*, pages 429–443. Springer, 2005.
19. H. Kress-Gazit, T. Wongpiromsarn, and U. Topcu. Correct, reactive, high-level robot control. *IEEE Robotics & Automation Magazine*, 18(3):65–74, 2011.

20. M. Kwiatkowska, G. Norman, and D. Parker. PRISM 4.0: Verification of probabilistic real-time systems. In *International Conference on Computer-Aided Verification (CAV)*, pages 585–591. Springer, 2011.
21. M. Kwiatkowska, G. Norman, and J. Sproston. Probabilistic model checking of deadline properties in the IEEE 1394 FireWire root contention protocol. *Formal Aspects of Computing*, 14:295–318, 2003.
22. A. Legay, B. Delahaye, and S. Bensalem. Statistical model checking: An overview. In *International Conference on Runtime Verification (RV)*, pages 122–135. Springer, 2010.
23. A. Mallet, C. Pasteur, M. Herrb, S. Lemaignan, and F. Ingrand. GenoM3: Building middleware-independent robotic components. In *International Conference on Robotics and Automation (ICRA)*, pages 4627–4632. IEEE, 2010.
24. A. Miyazawa, P. Ribeiro, W. Li, A. Cavalcanti, and J. Timmis. Automatic property checking of robotic applications. In *International Conference on Intelligent Robots and Systems (IROS)*, pages 3869–3876. IEEE, 2017.
25. C. Pecheur. Verification and validation of autonomy software at NASA. Technical report, NASA Ames Research Center, 2000.
26. M. Quigley, B. Gerkey, K. Conley, J. Faust, T. Foote, J. Leibs, E. Berger, R. Wheeler, and A. Ng. ROS: an open-source Robot Operating System. In *ICRA workshop on open source software*, page 5, 2009.
27. A. Roscoe. *Understanding concurrent systems*. Springer Science & Business Media, 2010.
28. C. Seceleanu, A. Vulgarakis, and P. Pettersson. REMES: A resource model for embedded systems. In *International Conference on Engineering of Complex Computer Systems (ICECCS)*, pages 84–94, 2009.
29. T. Sekizawa, F. Otsuki, K. Ito, and K. Okano. Behavior verification of autonomous robot vehicle in consideration of errors and sisturbances. In *International Computer Software and Applications Conference (COMPSAC)*, pages 550–555, 2015.
30. D. Simon, R. Pissard-Gibollet, and S. Arias. Orccad, a framework for safe robot control design and implementation. In *National workshop on control architectures of robots: software approaches and issues (CAR)*, 2006.
31. A. Sowmya, D. Tsz-Wang So, and W. Hung Tang. Design of a mobile robot controller using Esterel tools. *Electronic Notes in Theoretical Computer Science*, 65(5):3–10, 2002.
32. N. Tomatis, G. Terrien, R. Piguet, D. Burnier, S. Bouabdallah, K. Arras, and R. Siegwart. Designing a secure and robust mobile interacting robot for the long term. In *International Conference on Robotics and Automation (ICRA)*, pages 4246–4251. IEEE, 2003.
33. R. Volpe, I. Nesnas, T. Estlin, D. Mutz, R. Petras, and H. Das. The CLARAty architecture for robotic autonomy. In *Aerospace Conference*, pages 1–121, 2001.