https://doi.org/10.1007/s00165-020-00509-0
The Author(s) © 2020
Formal Aspects of Computing (2020) 32: 187–227

Formal Aspects of Computing

PuRSUE -from specification of robotic environments to synthesis of controllers

Marcello M. Bersani[1] and Matteo Soldo[1] and Claudio Menghi[3] and Patrizio Pelliccione[4,5] and Matteo Rossi[2]

[1] Dipartimento di Elettronica, Informazione e Bioingegneria, Politecnico di Milano, Milan, Italy
[2] Dipartimento di Meccanica, Politecnico di Milano, Milan, Italy
[3] SnT - University of Luxembourg, Luxembourg, Luxembourg
[4] Chalmers | University of Gothenburg, Gothenburg, Sweden
[5] University of L'Aquila, L'Aquila, Italy

Abstract. Developing robotic applications is a complex task, which requires skills that are usually only possessed by highly-qualified robotic developers. While formal methods that help developers in the creation and design of robotic applications exist, they must be explicitly customized to be impactful in the robotics domain and to support effectively the growth of the robotic market. Specifically, the robotic market is asking for techniques that: (i) enable a systematic and rigorous design of robotic applications though high-level languages; and (ii) enable the automatic synthesis of low-level controllers, which allow robots to achieve their missions. To address these problems we present the PuRSUE (Planner for RobotS in Uncontrollable Environments) approach, which aims to support developers in the rigorous and systematic design of high-level run-time control strategies for robotic applications. The approach includes PuRSUE-ML a high-level language that allows for modeling the environment, the agents deployed therein, and their missions. PuRSUE is able to check automatically whether a controller that allows robots to achieve their missions might exist and, then, it synthesizes a controller. We evaluated how PuRSUE helps designers in modeling robotic applications, the effectiveness of its automatic computation of controllers, and how the approach supports the deployment of controllers on actual robots. The evaluation is based on 13 scenarios derived from 3 different robotic applications presented in the literature. The results show that: (i) PuRSUE-ML is effective in supporting designers in the formal modeling of robotic applications compared to a direct encoding of robotic applications in low-level modeling formalisms; (ii) PuRSUE enables the automatic generation of controllers that are difficult to create manually; and (iii) the plans generated with PuRSUE are indeed effective when deployed on actual robots.

Keywords: Robotics; software engineering; controller synthesis; formal methods

Correspondence to: P. Pelliccione, E-mail: patrizio.pelliccione@univaq.it

1. Introduction

Market forecasts estimate an increasing trend of development for the industrial robotics sector. The World Robotics Survey [WRs] evidenced an increment in the use of service robots for professional use and indoor logistics. In particular, sales of professional service robots registered a 28% growth in 2014, which resulted in an increase of USD 2.2 billion in sales. This growth emphasizes the need for techniques to support the effective development of robotic applications.

In contrast, the H2020 Multi-Annual Robotics Roadmap ICT-2016 [WRs] evidenced various hindrances that are still limiting a more thriving growth of the market, such as the lack of consolidated development processes, the frequent use of craftsmanship production practices, and the constant need for highly specialized designers. Indeed, in the robotic domain, solutions are generally ad-hoc and conceived on a per-problem basis, rather than being designed for enabling their reuse and for making them accessible to non-experts. As a consequence, robotic development remains confined to a small set of highly-qualified specialists, thus preventing it from scaling up to current industry needs.

To effectively support a rapid and constant growth of the robotic market, we believe that it is necessary to make the development of robotic applications more accessible to engineers with a limited knowledge of the physics, the theory of control and the technology of robots. Specifically, the boost in the robotic market can only be sustained by effectively handling two problems: (**P1**) supporting a *systematic and rigorous design* of the robotic applications through high-level languages, and (**P2**) enabling automatic *synthesis of low-level controllers* that allow robots to achieve their missions.

P1: Supporting a systematic and rigorous design Current practices in robotic development require designers to build their applications using low-level primitives and do not help reasoning on a high-level logical design. For example, ROS (Robot Operating System [QCG+09]), the de facto standard platform for developing robotic applications, requires developers to deal with low-level messages on the ROS master node to control the robot's behaviour. These messages must be managed and defined carefully by the designer as they represent low-level primitives that instruct the robot on several aspects defining its dynamics. For example, to regulate the autonomous navigation of a robot, a message should include at least the time-stamp and the target position in terms of coordinates in a previously defined map of the environment stored in the robot. Thus, rather than conceiving and reasoning about the high-level robot behavior, designers are committed to tackling low-level problems such as converting the coordinates in the selected system, formatting the ROS messages, etc. The lack of high-level constructs introduces an "error-prone" process even for experienced designers and demands a deep knowledge of the robots' dynamics and kinematics.

We advocate a more systematic development process, where a rigorous high-level design of the problem domain and of the components of the robotic application is performed first. During the modeling activity designers may want to consider different scenarios, possibly accounting for different environments and actors, to automatically program the high-level robot tasks and to verify the application feasibility. Hence, a preliminary modeling activity is pivotal for enabling automatic reasoning from a high-level. Low-level details of the robotic application, such as the technology adopted for the implementation of the robots and sensors, the algorithms employed for managing the motion and perception of robots, the communication framework enabling the information exchange among the agents in the environment and so on, should be considered in a second phase of the design activity.

We envision our work as part of a modeling framework where it is possible to include and reuse the models of the robots. This would enable the reuse of information about the dynamics and kinematics of the robot (i.e., its speed, the actions that can be executed and the time needed to execute these actions) every time it is required.

P2: Enabling automatic synthesis of low-level controllers Robots are essentially agents that are deployed within a given environment to fulfill some mission. A mission is a high-level goal that a robotic application (i.e., a single robot or a set of robots) must accomplish [LRF+15, MTP+19, MTB+18, MTBP19]. The mission achievement is reached through the execution of movements as well as the execution of a set of actions that specify how the robots change the environment state and react to environmental changes. Controllers are software components that are designed to compute from a high-level mission a set of actions that, if executed, ensure the mission achievement. The computation of controllers is far from trivial, as it must take into account not only the robots' behavior, but also the evolution of the environment in which they are deployed.

The controller synthesis problem has been deeply studied in the formal methods domain. For example, in [JRLD07, CLRR11], and [LKZ16] a control problem is encoded into a finite state machine. While these techniques effectively handle the controller synthesis problem, they are not designed to be reusable and applicable in the robotic domain as-is. This makes their usage difficult, as designers must have a background in formal methods and control theory as well as a clear understanding of low-level formalisms, such as Timed Automata [AD94], Timed Computation Tree Logic [AD93, Bou09] and others, rather than a high-level design perspective that is supported by a generic high-level language equipped with domain-specific elements. In essence, there is the need to make controller synthesis techniques accessible not only to experts and, at the same time, to turn formal methods-based algorithms for controller synthesis, such as those in [JRLD07, CLRR11, LKZ16], into widely-used robotic solutions.

We promote a systematic and rigorous design methodology of robotic applications through a language with a precise and well defined semantics integrated with of off-the-shelf tools, that enable controller synthesis, and make the usage of formal techniques accessible in the robotic domain. Despite some works that address these two goals (e.g., [LRJ06, FJKG10, RGC94, GLR+11]), our research is tailored to robotic applications that work under two assumptions: (**A1**) *uncontrollable agents* can move and interact with the robot and their environment, and (**A2**) requirements possess an *explicit representation of time*. As further discussed through our motivating example (Sect. 2), these are two central aspects in the development of a relevant class of robotic applications, that, as detailed in Sect. 6, state-of-the-art approaches are not able to deal with effectively.

A1: Handling uncontrollable agents Robots constantly interact with their environments. In many applications, the collaboration of robots alongside with human workers is essential, as they are deployed in many industrial fields (e.g., mechanical, chemical and manufacturing industries) to help human activities. However, the human behavior is sometimes not predictable. For this reason, one of the most prominent challenges in planning and verifying robotic systems involves their interactions with their environment.

Formal models are prone to the problem of the reality gap, where models produced are never close enough to the real world to ensure successful transfer of their results except for specific problems (such as collision avoidance). A first step in this direction has been taken via static models of the environment, in which the robots are the sole actors [RGC94]. However, these models fail to capture the uncertainty in the environment's behavior. For example, even in a fully known environment, there might be uncontrollable actors, such as humans, that can interact with the robot and the environment itself. In some works, uncontrollable events are modeled as an input. In [LTL+16], uncontrollable events are modeled by means of automata that describe all of the system's capabilities. The designers must know the exact behavior of the uncontrolled agents to specify their behavior in the model, and the nondeterminism of the uncontrolled agents should be known in advance. Some works include an explicit representation of uncontrollable agents [MAIL+16, GMM+18], but they focus on the verification of system properties rather than on the generation of a control strategy.

A2: Handling missions with an explicit representation of time The specification of temporal aspects has a prominent role in the definition of robotic missions. Forcing a robot to achieve a certain goal within a bounded time frame, or being able to specify that a reaction has to occur within a specific time frame are examples of timed mission requirements that may need to be specified in robotic applications. Allowing designers to consider these requirements is extremely important in novel robotic applications. Unfortunately, while controller synthesis techniques that are able to consider these requirements do exist, their usage is mainly confined to robotic or formal methods experts and to ad-hoc applications (e.g., [VVB+19, BDJT19]).

Overview of the work We address the problems **P1** and **P2** by considering the assumptions **A1** and **A2**. We present PuRSUE (Planner for RobotS in Uncontrollable Environments), a framework that aims to support developers in the design of a high-level runtime control strategy for robotic applications. The PuRSUE framework helps designers in a systematic and rigorous design of the robotic application. Specifically, PuRSUE provides a set of features that allow addressing **P1** and **P2**:

- **F1:** PuRSUE supports designers in *modeling the robotic application*. It provides a Domain Specific Language (DSL), called PuRSUE-ML (*PuRSUE Modelling Language*), that allows non-expert users to easily and intuitively describe (i) the environment where the robotic application is executed, (ii) the controllable and uncontrollable agents acting and moving within it, (iii) a set of constraints between the defined events, and (iv) a mission that the robotic application should achieve.
- **F2:** PuRSUE allows designers to *specify missions that contain explicit temporal constraints* such as “the medicine needs to be delivered within 60 seconds once it has been requested”.

- **F3:** PuRSUE allows designers to automatically synthesize a control strategy, when one exists, in complex situations where multiple agents, both controllable and uncontrollable, interact within an environment.

We evaluated the support provided by PuRSUE from three different perspectives: for modeling robotic applications, for the automatic computation of controllers, and for the deployment of the controllers on real robots. We considered three different robotic applications inspired by examples tjat we collected from the literature [QLFAI18, AMRV19, TSF+16]: a robotic application in which a robot has to catch a thief, a work cell and an eco-robot that is in charge of collecting the trash. For each of these robotic applications we considered different scenarios with varying complexity, leading to 13 distinct scenarios.

To check how PuRSUE helps in modeling robotic applications, we used PuRSUE-ML to model our scenarios, and we evaluated the size of the proposed model, in terms of PuRSUE-ML constructs. We compare the number of constructs used to model the robotic application in PuRSUE-ML with respect to the number of states and transitions that are necessary to model our problem using Timed Game Automata (TGA). The TGA models are obtained using our automatic translation. The results show that the number of constructs used for modeling the robotic application in PuRSUE-ML is less than 19% of the number of constructs of the TGA that is automatically generated from the PuRSUE-ML specification using our translation, showing that the specification written by using our PuRSUE-ML is more concise than the corresponding TGA.

To examine how PuRSUE allows developers to automatically compute controllers, we evaluated how many times PuRSUE could generate a run-time controller, its size, and the time required for the generation. The results show that, in 10 out of 13 cases, PuRSUE was able to generate a run-time controller within a reasonable amount of time. The order of magnitude of the time required for computing the controller is limited to tens of seconds (or seconds, in the majority of the cases). Two scenarios do not admit a strategy. However, in the context of this work, this result allows us to point out a peculiarity of our approach to robotic development. By using a tool such as PuRSUE designers can rapidly prototype and fix their design, without digging into the details of a low-level formal model. The higher perspective on the design with PuRSUE-ML can simplify the analysis of the issues, and leads the designers to refine their model by means of a comprehensive view (see the discussion in Sect. 5.3). In addition, the experiments were conducted with the version of UPPAAL-TIGA compiled for 32-bit architectures.[1] For this reason, one controller synthesis failed because the tool ran out of memory. Finally, the size of the generated controllers shows that a manual design is complex, if not outright impossible, and allows us to prove the effectiveness of our approach using PuRSUE.

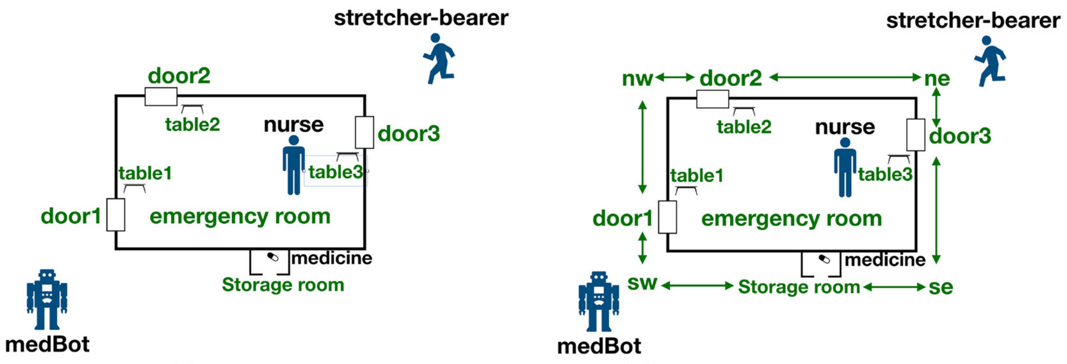
To evaluate how the controllers generated by PuRSUE behave in real scenarios, we deployed two of them on an actual robot, and checked whether the robot behaved as expected. Since this work mainly focuses on the PuRSUE modeling language, and on the procedure to translate PuRSUE models into TGA, the run-time execution of the plan is performed in an ad-hoc manner, with the only purpose of enabling the evaluation of the work in real scenarios. To this end, we developed a solution that enables PuRSUE to deploy executable controllers on real robots using the Robotic Operating System (ROS) [QCG+09]. The run-time controller is generated from the strategy computed by UPPAAL- TIGA, which is used as an off-the-shelf component to synthesize plans from TGA models. The validation of our approach has been carried out via two use cases, and our experiments showed that the robot behaved properly in both scenarios. For this reason, we claim that the design process realized by using PuRSUE, from the high-level description of the scenario to the controller generation, is indeed feasible (videos are available online [vid]).

Organization This article is structured as follows. Section 2 presents our motivating example. Section 3 presents the background knowledge needed to understand the rest of the work. Section 4 presents our contribution. Section 5 reports the evaluation we performed. Section 6 presents related work. Section 7 summarizes our findings and conclusions.

2. Motivating example

We consider a realistic example from the medical domain, indicated in the following as the Drug Delivery (DD) example. In the DD example, a robot (`medBot`) has to retrieve some medicine (`medicine`) from a storage room, and deliver it to an emergency room, while avoiding any interference with the transportation of patients on stretchers. A high-level graphical representation of the example is presented in Fig. 1a.

[1]The version 4.0.13 of UPPAAL-TIGA was used for the experiments that were performed on a machine equipped with an Intel(R) Core(TM) i7-4770, CPU (3.40GHz) with 8 cores, 16GB of RAM and Debian Linux (version 8.8).



(a) Drug Delivery **(b)** Drug Delivery with relevant points.

Fig. 1. A graphical representation of the Drug Delivery example

The emergency room is graphically indicated through a solid line that describes its boundaries. It has three entrances (`door1, door2, door3`), that can be either opened or closed. The robot, for security reasons, is not allowed to traverse the emergency room. The delivered medicines can be positioned on one of the tables set inside the emergency room, next to each of the entrances (`table1, table2, table3`). The medicines are located in the storage room (`storage room`), and the robot should bring the selected medicine, if requested, to one of the three entrances, when the corresponding door is open. At the same time, additional agents, such as the nurse, and the stretcher-bearer, can move freely in the area. The nurse is located in the emergency room, and can open closed doors to allow the `medBot` to deliver the medicines. The stretcher-bearer can move in the environment, around the emergency room, and he/she can close doors that are open when they hinder his/her movements in the corridors. The developer has to define a controller for the robot that (i) prevents the robot from bumping into the stretcher-bearer in the corridors, and (ii) always guarantees the delivery of the medicine within a specific time frame, regardless of the behavior of the nurse/stretcher-bearer, and given specific geometric information about the environment and the speed of the agents. In a game-theoretic representation, the robot and the nurse are the "players", which we assume to cooperate with one another (their goal is the effective delivery of the medicines). However, the stretcher-bearer is the "opponent" whose unpredictable behavior might hamper the realization of the goal that the players have set.

Developing a controller without any automatic support is not easy. Manually designing a controller for a player that always guarantees the delivery of the medicine requires the evaluation of all the possible evolutions of the system. This is an error-prone and time-consuming activity. In the DD example, one may for example program the robot to drop the medicine when reaching the table at `door1`. However, this may be a failing strategy, since the stretcher-bearer may reach `door1`, thus the robot fails to avoid interfering with the transportation of patients on stretchers. Intuitively, as mentioned, the correct strategy should take into account the position of the stretcher-bearer before choosing which door to use to deliver the medicine; this needs to consider the specific distances between locations, the speeds of the different agents, and the duration of actions such as picking up the medicine and setting it on the table—all of which are nontrivial, especially in topologies that are not as simple as the one presented (consider, for instance, industrial warehouses where automatic tracks and humans collaborate for goods logistics). Moreover, agents may behave in many different ways, and the number of agents might be higher in realistic scenarios (e.g., patients and doctors also moving as agents in the same area).

Referring to the features that were presented in the Introduction, we implemented them in PuRSUE; the resulting tool helps in solving the problems above in the following ways:

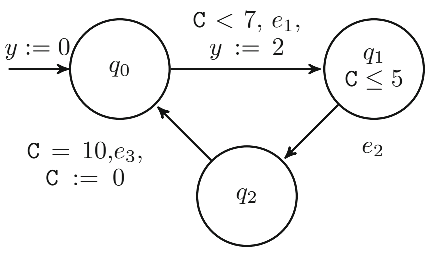
- **F1:** *Providing a language that allows designers to easily model robotic applications where uncontrollable agents can move and interact with the robot and the environment.* As argued in the introduction, providing a language that allows easily identification of controllable and uncontrollable agents, the actions that they can perform, and how they can move within their environment, is a necessary pre-condition for enabling the use of Model-driven Engineering (MDE) techniques for robotic applications. In our example, the designer is interested in modeling the presence of the storage room and the doors where the robot should deliver the medicine. The uncontrollable agents are represented by the nurse and stretcher-bearer.

- **F2:** *Providing a framework that supports reasoning about properties that contain an explicit representation of time.* In our example, the designer is interested in producing a controller for the robot that ensures the delivery of the medicine within a specific time frame. In order to achieve this objective, it is necessary to provide a framework that accommodates the specification of missions with explicit temporal constraints.
- **F3:** *Automatically synthesizing a runtime controller capable of achieving the mission defined in the scenario.* As mentioned, manually designing a run-time controller for the DD scenario is impractical. To automate the synthesis of a runtime controller, we employed well-known and established formal methods.

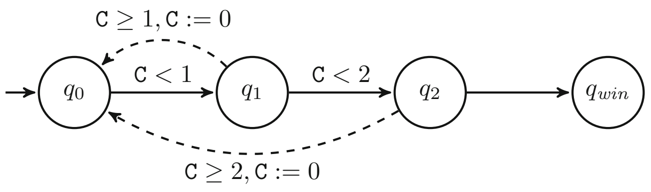
In the rest of this paper, we discuss how PuRSUE provides these features.

3. Timed games

This section introduces the formalisms used in this work. In particular, it provides the basic definitions of Timed Automata (TA, [AD94]) and of Timed Game Automata (TGA) (which are an extension of the former), and it introduces Timed Computation Tree Logic (TCTL) [AD93, Bou09], a well-known logic for expressing properties of TA and TGA.



(a) A Timed Automaton.


$\mathtt{C} \geq 1, \mathtt{C} := 0$
q_0
$\mathtt{C} < 1$
q_1
$\mathtt{C} < 2$
q_2
q_{win}
$\mathtt{C} \geq 2, \mathtt{C} := 0$

(b) A Timed Game Automaton.

We introduce TGA because they are the target formalism into which the PuRSUE-ML models are compiled (see Sect. 4.2). Indeed, the agents acting in the environment, the geometry of the space where the agents move, the ordering of events executed by the agents, the collaboration among the actors, etc., which are initially described in PuRSUE-ML, are automatically translated into TGA. We first introduce TA, as they allow us to introduce the notions of discrete transitions, time transitions and runs. We then extend the definition of those concepts to TGA. Finally, we recall TCTL, since it is used in the encoding of the mission of the robotic application (see Sect. 4.3).

In the rest of this work, we will indicate automata in calligraphic font, e.g., $\mathcal{A}$, and sets with standard roman font, e.g., X.

Timed automata Let X be a finite set of clock variables and $\Gamma(X)$ be the set of *clock constraints* defined as $\eta := x \sim c \mid \neg\eta \mid \eta \wedge \eta$, where $\sim \in \{<, =\}$, $x \in X$ and c is a natural number. Let $\Gamma(Y)$ be the set of *variable constraints* ζ defined as $\zeta := y \sim d \mid y \sim y' \mid \neg\zeta \mid \zeta \wedge \zeta$, where y and y' are variables in Y and d is an integer number. Let $assign(Y)$ be the set of assignments of the form $y := d$, where $y \in Y$.

Given a set of event symbols Σ and a null event τ, a Timed Automaton (TA) $\mathcal{A}$ is a tuple $\mathcal{A} = \langle Q, q_0, v_0, I, \Sigma, T\rangle$, where: Q is a finite set of locations, $q_0 \in Q$ is the initial location, $v_0 : Y \rightarrow \mathbb{Z}$ is a function assigning an integer value to each variable in Y, $I : Q \rightarrow \Gamma(X)$ is an invariant assignment function, and $T \subseteq Q \times Q \times \Sigma \cup \{\tau\} \times \Gamma(X) \times \Gamma(Y) \times \wp(X) \times \wp(assign(Y))$ is a finite set of transitions. Intuitively, a transition $(q, q', \sigma, \eta, \zeta, V, U) \in T$ consists of a source state q, a destination state q', a synchronization event $\sigma \in \Sigma \cup \{\tau\}$, a clock guard η that is a constraint on the values of the clock variables, a variable guard ζ that is a constraint on the values of the variables, a set V of clocks to be reset, and a set U of variable assignments. For example, Fig. 2a contains a TA with locations q_0, q_1 and q_2 defined over the set of clock variables $X = \{\mathtt{C}\}$ and variables $Y = \{y\}$. Location q_0 is the initial location, e_1, e_2 and e_3 are synchronization events, $\mathtt{C} < 5$ and $x = 10$ are clock guards, $\mathtt{C} := 0$ resets the clock $\mathtt{C}$ and $y := 2$ is a variable assignment.

The standard semantics of a TA is given in terms of *configurations*. A configuration is a pair (q, v) defining the current location q of the automaton and the value v of all clocks and variables, where $q \in Q$ and v is a function over $X \cup Y$ assigning a non-negative real value to every clock of X and an integer to every variable of Y. A configuration change from (q, v) to (q', v') can happen because either a transition in T is taken or because time elapses. The adopted semantics is standard, and it are outlined below. The configuration change is a relation

$$\rightarrow \subseteq Q \times \mathbb{R}^X_{\geq 0} \times \mathbb{Z}^Y \times Q \times \mathbb{R}^X_{\geq 0} \times \mathbb{Z}^Y \tag{1}$$

where $\rightarrow = \xrightarrow{\sigma} \cup \xrightarrow{\delta}$, and $\xrightarrow{\delta}$ and $\xrightarrow{\sigma}$ are defined as follows.

A *discrete transition* $(q, v) \xrightarrow{\sigma} (q', v')$ is determined by the tuple $(q, q', \sigma, \eta, \zeta, V, U) \in T$, and it is such that: (i) the clock and the variable values defined by the valuation v satisfy, respectively, guards η and ζ, and v' satisfies the invariant $I(q')$; (ii) for each clock x, if x is in X, then $v'(x) = 0$, otherwise $v'(x) = v(x)$; (iii) for each variable $y \in Y$, $v'(y) = d$ if $y := d$ is an assignment in U. For example, a discrete transition for the TA in Fig. 2a, associated with event e_1, can change the configuration (q_0, v), where $v(\mathtt{C}) = 1$ and $v(y) = 0$, to the configuration (q_1, v'), where $v'(\mathtt{C}) = 1$ and $v'(y) = 2$.

A *time transition* $(q, v) \xrightarrow{\delta} (q', v')$, for any $\delta \in \mathbb{R}_{\geq 0}$, is such that $q = q'$, each variable $y \in Y$ retains its value, and $v'(x) = v(x) + \delta$ for all $x \in X$. In addition, the invariant $I(q)$ is satisfied by all assignments of the clocks given by v to v'. For example, a time transition for the TA in Fig. 2a can change the configuration (q_0, v), where $v(\mathtt{C}) = 0.3$ and $v(y) = 0$, to the configuration (q_0, v'), where $v'(\mathtt{C}) = 1$ and $v'(y) = 0$.

A *run*, or *execution*, of a TA $\mathcal{A}$ is a (possibly infinite) sequence of configurations $(q_0, v_0)(q_1, v_1)(q_2, v_2)\cdots$ such that, for any $i \geq 0$, $(q_i, v_i) \rightarrow (q_{i+1}, v_{i+1})$ is a discrete transition or a time transition. It is customary to define $v_0(x) = 0$, for every $x \in X$, whereas variables can be initialized with a specific value. For example, the initial configuration of the TA described in Fig. 2a is (q_0, v), where $v(\mathtt{C}) = 0$ and $v(y) = 0$. The set of all the executions of a TA $\mathcal{A}$ is denoted by $R(\mathcal{A})$.

When several TA are considered, the configuration contains the locations of all of the automata and the values of all their clocks and variables. The symbols in Σ are used to constrain the executions of the TA, i.e., the ways in which a network of TA synchronize while changing the configuration of the system. Each symbol $\sigma \in \Sigma$ that labels a transition has the form $\sigma?$ or $\sigma!$. Informally, two TA synchronize when one performs a discrete transition labeled with $\sigma?$ and the other with $\sigma!$ at the same time. In this work, we consider broadcast synchronization: when a TA fires a transition labeled with $\sigma!$, every TA that has an outgoing transition that is enabled (its guards are satisfied) and is labeled with $\sigma?$ is taken at the same time (the existence of $\sigma?$ is not necessary for the execution of $\sigma!$). In this paper, we indicate the composition of TA $\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_n$ through broadcast channels as $\mathcal{A}_1 \mid\mid \mathcal{A}_2 \mid\mid \ldots \mid\mid \mathcal{A}_n$ [BDL04].

Timed game automata A Timed Game Automaton (TGA) is a TA $\mathcal{A} = \langle Q, q_0, v_0, I, \Sigma, T\rangle$ where Σ is split into two disjoint sets: Σ_c, which is the set of controllable events, and Σ_u, which is the set of uncontrollable events. An example of a TGA is shown in Fig. 2b, where controllable events are drawn with a continuous line and uncontrollable ones are dashed (for the sake of simplicity, we do not report the name of events). Some edges in the TGA are labeled with clock constraints and variable assignments, as is the case for standard TA.

Two kinds of games can be defined and effectively solved with TGA: reachability games and safety games. Let *Goal* be a set of configurations in $Q \times \mathbb{R}^X_{\geq 0} \times \mathbb{Z}^Y$. A game (be it a reachability game or a safety game) for $\mathcal{A}$ is a pair $(\mathcal{A}, \mathit{Goal})$. For a reachability game, a *winning execution* is a finite (or infinite) run $(q_0, v_0)(q_1, v_1)\ldots \in R(\mathcal{A})$ such that $(q_h, v_h) \in \mathit{Goal}$, for some $h \geq 0$. Intuitively, given the set *Goal*, an execution is a winning execution if one of the configurations in the set *Goal* is in the run associated with the execution. For example, consider a scenario where a robot is engaged to carry an item from a conveyor belt to a shelf, that are located in two positions inside a warehouse. In such a case, the goal is the actual shipping of the item, that is achieved by defining a suitable path through the warehouse. For a safety game, instead, a *winning execution* is a finite (or infinite) run such that $(q_h, v_h) \in \mathit{Goal}$, for all $h \geq 0$. Intuitively, given the set *Goal*, an execution is a winning execution if all the configurations in the run associated with the execution are within the set *Goal*. For example, consider a robot and a human that load packets, with different weight, on two conveyor belts and that cooperate to guarantee that the belts never remain empty for more than 10 seconds. In a realistic scenario, where the human only moves light packets and cannot be compelled to work constantly, an execution is winning if both the belts are always reloaded within 10 seconds. We define the set $W(\mathcal{A}, \mathit{Goal}) \subseteq R(\mathcal{A})$ as the set of winning executions $(q_0, v_0)(q_1, v_1)\ldots$ for a game $(\mathcal{A}, \mathit{Goal})$.

Given a reachability or safety game $(\mathcal{A}, Goal)$, a *strategy* indicates which controllable transitions of the TGA can be fired, and the amount of time the automaton can spend in each configuration of the run such that all the executions of the TGA are winning executions.

Definition 1 A *strategy* ν for a TGA $\mathcal{A} = \langle Q, q_0, v_0, I, \Sigma_c \cup \Sigma_u, T\rangle$ is a *partial* mapping $\nu : R(\mathcal{A}) \to \Sigma_c \cup \{d\}$, from the set of *finite* executions $R(\mathcal{A})$ to the union of the set of controllable events Σ_c and the symbol d (with $d \notin \Sigma_c$) that indicates that the time progresses, and no event in $\Sigma_c \cup \Sigma_u$ occurs, such that, for any execution ρ of the form $(q_0, v_0)\ldots(q_k, v_k)$,

- if $\nu(\rho) = d$, then $(q_k, v_k) \xrightarrow{\delta} (q_k, v_k + \delta)$, for some $\delta > 0$,
- if $\nu(\rho) = \sigma$, then $(q_k, v_k) \xrightarrow{\sigma} (q, v)$, for some configuration (q, v).

Observe that, if $\nu(\rho)$ is not defined, then the only events that are allowed in (q_k, v_k) are the uncontrollable ones in Σ_u. When a TGA $\mathcal{A}$ is controlled using a strategy ν, all configuration changes in the resulting executions, called outcomes, are compliant with ν.

Definition 2 An *outcome* of a strategy ν for a TGA $\mathcal{A} = \langle Q, q_0, v_0, I, \Sigma_c \cup \Sigma_u, T\rangle$ is an execution in $R(\mathcal{A})$ of the form $(q_0, v_0)\ldots(q_k, v_k)$ such that:

- for every $(q_i, v_i) \xrightarrow{\delta} (q_i, v_i + \delta)$, with $\delta > 0$, and for all $\delta' < \delta$, $\nu((q_0, v_0)\ldots(q_i, v_i + \delta')) = d$.
- for every $(q_i, v_i) \xrightarrow{\sigma} (q_{i+1}, v_{i+1})$, with $\sigma \in \Sigma_c$, it holds that $\nu((q_0, v_0)\ldots(q_i, v_i')) = \sigma$, for any v_i'.

An infinite execution ρ is an outcome if all finite prefixes of ρ are outcomes. Given a game $(\mathcal{A}, Goal)$, we indicate by $O(\mathcal{A}, Goal, \nu)$ the set of outcomes in $R(\mathcal{A})$ obtained by means of ν.

The baseline assumption we consider when we deal with games is that the uncontrollable transitions of $\mathcal{A}$ in a game $(\mathcal{A}, Goal)$ are played by an "opponent" that wants to prevent the execution of $\mathcal{A}$ from being winning. The effectiveness of a strategy only depends on the player executing the controllable transitions, as the strategy cannot enforce a specific event through the opponent. For this reason, when we calculate a strategy ν for a game $(\mathcal{A}, Goal)$, we want $O(\mathcal{A}, Goal, \nu)$ to include only certain outcomes, called maximal. A *maximal outcome* is (i) an infinite execution or (ii) a finite execution whose final configuration allows either the opponent to perform an event, or time to elapse, or it belongs to $Goal$. Formally, ρ is *maximal* if: (i) ρ is an infinite execution; (ii) ρ is of the form $(q_0, v_0)\ldots(q_k, v_k)$ and q_k is in $Goal$ or, if there exists (q, v) such that $(q_k, v_k) \xrightarrow{\sigma} (q, v)$, then $\sigma \in \Sigma_u$. Hence, a finite execution is winning only because of the events played before the last configuration (and not because of the opponent's event or the elapsing of time at the end).

A strategy ν is *winning* for a game $(\mathcal{A}, Goal)$ if all maximal outcomes in $O(\mathcal{A}, Goal, \nu)$ are in $W(\mathcal{A}, Goal)$. For example, a winning strategy for the TGA in Fig. 2b is the one that allows the system to reach location q_{win} by taking advantage of the controllable events associated with the two transitions between q_0 and q_1 and between q_1 and q_2. The strategy should force the player "to be quick" in every configuration (q, v) such that q is either q_1 or q_2. In fact, if the player waits too much time in q_1, then the opponent can prevent the player from taking the transition leading to q_2, and can set the system location to q_0 by taking the uncontrollable action between q_1 and q_0. In fact, when the system is in q_1 and clock `C` becomes greater than 1, then the opponent can perform the uncontrollable action leading the system to q_0. A similar argument also holds for q_2. Therefore, in order to win the reachability game $\mathtt{AF}q_{win}$, i.e., every execution eventually reaches location q_{win}, the player must perform the two controllable events in less than one time unit since the beginning of the game in location q_0. The reachability/safety problem for a game $(\mathcal{A}, Goal)$ consists in finding a winning strategy for the game. The decidability [AMP95, CDF+05] of the two classes of games considered in this work follows from the use of memory-less strategies. Let ρ and ρ' be two executions of the form $(q_0, v_0)\ldots(q_k, v_k)$ and $(q_0', v_0')\ldots(q_h', v_h')$, respectively. We say that ρ and ρ' terminate with the same configuration when $(q_k, v_k) = (q_h', v_h')$. A strategy ν is *memoryless* when, for all pairs of distinct executions ρ and ρ' terminating with the same configuration (q, v), then $\nu(\rho) = \nu(\rho')$.

Similar to TA, two or more TGA can define a network of TGA with the same synchronization mechanism of a network of TA, realized through labels $\sigma?$ or $\sigma!$, for any $\sigma \in \Sigma$.

The objectives that collaborating agents realize can be specified by means of a logical language that formally captures a required goal that agents must achieve. Timed Computation Tree Logic has been adopted to specify reachability or safety games in [BCD+07].

Timed computation tree logic A widely adopted language for the specification of temporal properties of timed systems is Timed Computation Tree Logic (TCTL).

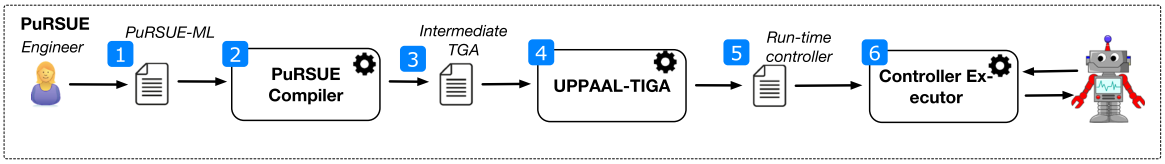


Fig. 2. Graphical representation of the relation among the artifacts and the automatic procedures of the PuRSUE Framework

Its semantics is standard (the interested reader can refer to [AD93, Bou09]). Given a TA (or TGA) $\mathcal{A} = \langle Q, q_0, v_0, I, \Sigma, T\rangle$, we assume that every location $q \in Q$ of the TA (or TGA) is associated with an atomic proposition that is evaluated as true if the configuration of $\mathcal{A}$ is (q, v), for any v. These atomic propositions, together with arithmetical formulae $\Gamma(Y)$, can be used in TCTL formulae to define a set of configurations. Let ϕ_{Goal} be a propositional formula that is evaluated as true when the TA is in one of the configurations of the *Goal* set, $\texttt{AF}(\phi_{\text{Goal}})$ and $\texttt{AG}(\phi_{\text{Goal}})$ are TCTL formulae, where $\texttt{A}$ means "for every system execution", $\texttt{G}$ means "globally" and $\texttt{F}$ means "eventually". We write $\mathcal{A} \models_\nu \texttt{AF}(\phi_{\text{Goal}})$ (resp. $\mathcal{A} \models_\nu \texttt{AG}(\phi_{\text{Goal}})$) to indicate that ν is a winning strategy for the game such that all the maximal outcomes are in $W(\mathcal{A}, Goal)$, i.e., every maximal outcome in $O(\mathcal{A}, Goal, \nu)$ satisfies $\texttt{F}(\phi_{\text{Goal}})$ (resp. $\texttt{G}(\phi_{\text{Goal}})$). The reachability (resp. safety) problem for a game $(\mathcal{A}, \phi_{\text{Goal}})$ amounts to finding a strategy ν such that $\mathcal{A} \models_\nu \texttt{AF}(\phi_{\text{Goal}})$ (resp. $\mathcal{A} \models_\nu \texttt{AG}(\phi_{\text{Goal}})$).

An extension of the previous two problems considered in this paper is defined on a game $(\mathcal{A}, Goal, Safe)$, where *Goal* and *Safe* are two sets of configurations representing goal and safe configurations, respectively. In such a case, the set $W(\mathcal{A}, Goal)$ of winning executions is limited to all the runs of $\mathcal{A}$ which include only configurations of *Safe* and eventually reach a configuration in *Goal*. The problem can be formulated in terms of TCTL formula $\texttt{A}(\phi_{\text{Safe}}\ \texttt{U}\ \phi_{\text{Goal}})$, and consists of finding a strategy ν such that $\mathcal{A} \models_\nu \texttt{A}(\phi_{\text{Safe}}\ \texttt{U}\ \phi_{\text{Goal}})$. When ϕ_{Safe} evaluates to $\top$ (i.e., it is vacuously true), then $\mathcal{A} \models_\nu \texttt{A}(\phi_{\text{Safe}}\ \texttt{U}\ \phi_{\text{Goal}})$ evaluates to $\mathcal{A} \models_\nu \texttt{AF}(\phi_{\text{Goal}})$.

We consider UPPAAL- TIGA [BCD+07] as the tool to solve reachability/safety games. Indeed, this tool is able to effectively process TGAs and TCTL formulae, which form the basis of this work.

4. The PuRSUE framework

PuRSUE is a framework[2] that allows developers to model robotic applications and synthesize controllers that can be implemented on the controlled agents to achieve a specific mission. To this end, PuRSUE relies on the artifacts and the automatic procedures presented in Fig. 2 as follows.

- 1 PuRSUE-ML (Sect. 4.1). It is a high-level language for modeling robotic applications with a human-readable format.
- 2 *PuRSUE compiler* (Sect. 4.2). It translates the PuRSUE-ML model of the robotic application into an intermediate TGA.
- 3 *Intermediate TGA* (Sect. 4.2). It is an intermediate encoding of the robotic application defined by the TGA that is tailored to the creation of a run-time controller. The controller is automatically obtained from the TGA model by using any tool solving the controller synthesis for TGA problems.
- 4 UPPAAL- TIGA. We adopt UPPAAL- TIGA to generate the controller from the intermediate TGA.
- 5 *Run-time controller*. It is a runnable implementation of the synthesized controller that can be executed by all controllable agents. A run-time controller enforces the control logic in the environment by governing the events that autonomous agents must take to finally achieve the mission goal.
- 6 *Controller executor*. It executes the controller on the controllable agents by acting as interface between the events as modeled in PuRSUE-ML and the ROS API. For instance, if the runtime controller instructs an agent to move to a certain location, the controller executor takes care of signaling the command to the agent through the ROS interface.

[2]The implementation is available at https://github.com/deib-polimi/PuRSUE.

Remark This work focuses on PuRSUE-ML and the translation of PuRSUE-ML constructs into an intermediate TGA. It will not discuss how the strategy synthesized by UPPAAL- TIGA is transformed into an executable controller, and how it is managed at run-time by the controller executor, even if these aspects have been considered to realize the experimental evaluation of Sect. 5. However, for completeness, we have made the source code available on our online website [pur19].

4.1. The PuRSUE modeling language

PuRSUE-ML is a high-level modeling language that supports designers in the development of robotic applications. By using PuRSUE-ML, the designer of a robotic application constructs a high-level model that describes: (i) the environment in which the application is deployed—e.g., to explicitly say that agent `medBot` can move from location `nw` to `door2`; (ii) the events that can occur in the environment or those that can be generated by controllable and uncontrollable agents—for instance, to model that `medBot` can collect a medicine in `storage room`; (iii) the precedence relations among events (e.g., event consequences), for instance, to state that before crossing `room` a `nurse` must first open a door; (iv) the agents, including whether they are controllable or not, and how they can trigger events—e.g., that `stretcherBearer` can freely move in the environment, whereas `medBot` is controlled by the system, and it can be instructed to collect a medicine in `storage room`; (v) the initial configuration of the system; and (vi) the objectives, which represent the final goal that the robotic application has to achieve—for example, to bring the medicine to the nurse in the room.

We describe how PuRSUE-ML helps designers in modeling each of these elements in the following sections, considering the Drug Delivery scenario as an example. Specifically, a graphical representation of the scenario with some additional information that is used in the description of the PuRSUE-ML model is presented in Fig.1b.

4.1.1. Environment

The physical environment within which the robotic application is deployed is modeled by the designer through a set of Points Of Interest (POIs) and connections between POIs, which describe the point-to-point spatial relationships among them.

POIs are an abstraction of the points of the physical environment used to represent locations that can be occupied by the agents. For instance, they can represent buildings, rooms, or logical areas according to the considered scenarios. A POI x is defined in PuRSUE-ML with the keyword "`poi`" as follows:

`poi` x

In the DD scenario, 12 POIs have been identified: `medicine`, representing the storage room where the medicines are located; `room`, representing the emergency room where the nurse is located, and where the medicine should be delivered; `door1`, `door2`, and `door3`, modeling the doors that can be crossed by the `medBot` to deliver the medicine; south west (`sw`), north west (`nw`), south east (`se`), and north east (`ne`), and `table1`, `table2`, and `table3` modeling the tables. These POIs are encoded in PuRSUE-ML as follows:

```
poi medicine
poi room
poi door1, door2, door3
poi sw, nw, ne, se
poi table1, table2, table3
```

Connections represent the physical links that the agents traverse, while moving from one POI to another. A connection is an abstraction of a path in the environment (i.e., the distance of POIs). Connections can represent complex path in the environment such as corridors, elevators or any other sort of physical or logical connection among POIs. For any two POIs x, y, and a positive integer $dist$, a point-to-point connection between x and y is defined in PuRSUE-ML with the keyword "`connect`", as follows:

`connect` x `and` y `distance` $dist$ (`unidirectional`)?

where symbol ? marks optional parts of statements. For example, connections are bidirectional by default, but unidirectional connections can also be modeled by adding the keyword `unidirectional` at the end of the definition.

In the DD example, connections are all bidirectional, and they are modeled as follows:

```
connect sw and door1 distance 6
connect door1 and nw distance 13
connect nw and door2 distance 8
...
```

As connections represent paths that agents can follow to move among the different POIs, if the path from A to B traverses C, the PuRSUE-ML specification includes two connections: one from A to C, and another from C to B. For example, in the DD case, a connection links POIs `sw` and `door1`, and another links POIs `door1` and `nw`.

4.1.2. Events

Events represent atomic actions that agents can execute by means of actuators, whose effects modify the configuration of the environment. Every event is associated with an agent that performs it. In fact, if an event was not associated with any concrete agents in the environment, a dummy agent, that performs that event, can always be introduced without loss of generality. For example, in case a user wants to represent a blackout, an additional agent, named as "system" or "environment" can be introduced in the model, and equipped with an event "blackout". Events are atomic entities, whose occurrence cannot be interrupted, and that are held instantaneously or with a given duration. Being PuRSUE-ML a language to model real scenarios, we assume that only one event can occur at a time, as the real time is dense.

For any event e, POI x, and positive integer n, an event is defined in PuRSUE-ML by means of the keyword "`event`" as follows:

`event` e: (`collaborative`)? (`location` x)? (`duration` n)?

where the keywords have the following meanings:

- `collaborative`: the event necessitates *collaboration* between at least two agents to occur. Intuitively, in the DD scenario, if the `medBot` could fetch a medicine from a delivery agent, then the event representing the "medicine retrieval" would be modeled as collaborative. Collaborative events are further elaborated in Sect. 4.1.4.
- `location` x: the event can only occur when the agent performing it is in POI x. If no location is specified, then the event can occur in every POI.
- `duration` n: the event requires n time units to be completed. The event duration is introduced for tasks whose execution is non negligible. An event is *durable* or *instantaneous*, depending on its duration: in the former case $n > 0$, while in the latter $n = 0$ (or simply the keyword and the value are omitted). For example, the time that the nurse takes to move in the room from one door to another is 20 time units.

All of the events of DD are described in Table 1. They are specified in PuRSUE-ML as follows (we only show some of them since the others are very similar):

```
event giveMedicine1 location door1 duration 3
event takeMedicine location medicine duration 2
event confirmDelivery
...
```

We remark that the specification of the events representing the act of moving is not envisaged in the language. In fact, these events are automatically instantiated all of the agents that the designer defines as "capable of moving" (further details can be found in Sect. 4.1.4). To prevent an agent from moving between two POIs x and y, PuRSUE-ML includes the keyword `prevent`:

`prevent` a `from moving between` x `and` y (`unidirectional`)?

If unidirectional connections are considered, then the order between x and y determines the direction that is not allowed. Furthermore, to prevent an agent from doing an event e while it is located in a POI x or while moving between two POIs x and y, PuRSUE-ML includes the following `prevent` statements:

`prevent` a `from doing` e `between` x `and` y (`unidirectional`)?
`prevent` a `from doing` e `in` x

Table 1. Description of the events of the Drug Delivery example

Name	collaborative	location	duration	Description
`giveMedicine1`	No	`table1`	3	The delivery of the medicine on, respectively, table1, table2, and table3
`giveMedicine2`		`table2`		
`giveMedicine3`		`table3`		
`takeMedicine`	No	`medicine`	3	Picking up the medicine from the storage room
`confirmDelivery`	No	–	1	Signal that the medicine has been delivered
`openDoor1`	No	`room`	2	The opening of respectively door1 / door2 / door3 (from the emergency room)
`openDoor2`				
`openDoor3`				
`crossRoom`	No	`room`	20	A general representation of the nurse being busy between opening of doors
`closeDoor1`	No	`door1`	2	The closing of respectively door1 / door2 / door3 (from the outside)
`closeDoor2`		`door2`		
`closeDoor3`		`door3`		

4.1.3. *Rules*

Rules model the evolution of the environment by constraining the behavior of the agents. In particular, rules specify the allowed sequences of events in the system by means of a set of regular expressions defined over the set of events of the environment. We adopt regular expressions because they are expressive enough to model realistic scenarios (all those considered in this work), and they are commonly known by designers and practitioners. The set of regular expressions allowed in the rules is defined using concatenation (before) and disjunction (or). Moreover, all rules are cyclical, that is, the Kleene star is applied implicitly to the whole expression. For any rule identifier r, and regular expression *rule*, a rule is defined by means of the keyword "`rule`" as follows:

$$\texttt{rule}\ r\text{: } \mathit{rule}$$

In DD, the following two rules are considered. Rule `nurseBehaviour` models realistic behavior of the nurse moving in the room, who interleaves movements with the opening of a door. Rule `robotTask` constrains the occurrence of event `confirmDelivery`, which always follows the delivery of the medicine, and the event `takeMedicine`. The rules of Drug Delivery are specified in PuRSUE-ML as follows:

```
rule nurseBehaviour: (openDoor1 or openDoor2 or openDoor3) before crossRoom
rule robotTask: takeMedicine before (giveMedicine1 or giveMedicine2 or giveMedicine3) before confirmDelivery
```

Rules are blocking, as they compel a precise ordering of events. If an event appears in a rule, then no agent in the system can trigger it if the occurrence of the event does not respect the sequence enforced by the rule. However, rules do not enforce the triggering of any of the events included in them, as they represent necessary conditions for their occurrence. For instance, rule $(A\ before\ B)\ before\ C$ does not impose the occurrence of event C, but it only prescribes that, if C is triggered, then event B must have been triggered earlier than C.

4.1.4. *Agents*

Agents describe the entities in the environment that affect the evolution of the environment because of their behavior, which is defined by the sequence of events that they perform over time. Agents are either *controllable* or *uncontrollable*. An agent is controllable when its behavior is controlled by an external system (i.e., the controller) that is not part of the environment, and that determines the events it has to perform based on the current system configuration. An agent is uncontrollable when it spontaneously moves or performs events. For any agent identifier a, positive integer sp, and disjoint sets of events $\{e^d_1, \dots, e^d_n\}$ and $\{e^r_1, \dots, e^r_m\}$, an agent is defined using the keyword "`agent`", as follows:

$$\texttt{agent}\ a\text{: }(\texttt{controllable})?\ (\texttt{mobile}\ sp)?\ \texttt{location}\ x\ (\texttt{can\_do}\ e^d_1, \dots, e^d_n)?\ (\texttt{reacts\_to}\ e^r_1, \dots, e^r_m)?$$

where the keywords have the following meanings:

Table 2. Agents of the Drug Delivery case

Name	Controllable	Mobile	Location	can_do	reacts_to
`medBot`	Yes	Yes ($sp = 1$)	`sw`	`giveMedicine1`, `giveMedicine2`, `giveMedicine3`, `takeMedicine`, `confirmDelivery`	–
`nurse`	Yes	Yes ($sp = 1$)	`room`	`openDoor1`, `openDoor2`, `openDoor3`, `crossRoom`	`giveMedicine1`, `giveMedicine2`, `giveMedicine3`
`Stretcher-bearer`	No	Yes ($sp = 1$)	`ne`	`closeDoor1`, `closeDoor2`, `closeDoor3`	–

- `controllable`: agent a is controllable.
- `mobile` sp : agent a can move in the environment and it covers a unit of distance in exactly sp time units.
- `location` x : agent a is in POI x at the beginning of the execution of the system.
- `can_do` $e^d_1, \ldots, e^d_n$: agent a can perform the events associated with events $e^d_1, \ldots, e^d_n$.
- `reacts_to` $e^r_1, \ldots, e^r_m$: agent a collaborates with other agents in the environment by means of events $e^r_1, \ldots, e^r_m$.[3] A collaborative event (Sect. 4.1.2) always requires the presence of at least two agents: one performing the event (`can_do`), and another reacting to it (`reacts_to`). Moreover, a collaborative event can occur only when at least one acting and one reacting agent, both associated with that event, are in the same POI.[4] Notice that, even if two or more agents act and react to a collaborative event, only one occurrence of that event occurs at a time. If an event is defined as collaborative, but no reacting agents are defined, the event will always be allowed.

Table 2 lists all agents involved in DD.

Agent `medBot`, for instance, is specified as follows:

```
agent medBot controllable mobile 1 location sw can_do giveMedicine1, giveMedicine2, giveMedicine3,
    takeMedicine, confirmDelivery
```

4.1.5. *States and state dependencies*

States represent the configuration of entities of the environment that are not modeled by agents, as they cannot perform events, but whose configuration affects the evolution of the system. PuRSUE-ML allows designers to specify state dependencies, such as the relation between an event, and the state change that the event enforces. The current version of the language includes states with boolean values. Multi-valued states, however, can be expressed as a combination of boolean states. For any state identifier s, and *disjoint* sets of events $\{e^t_1, \ldots, e^t_n\}$ and $\{e^f_1, \ldots, e^f_m\}$, state s (or state variable s) is defined using the keyword "`state`", as follows:

$$\texttt{state}\ s\ \texttt{initially}\ (\mathit{true}/\mathit{false}),\ \texttt{true\_if}\ e^t_1, \ldots, e^t_n\ \texttt{false\_if}\ e^f_1, \ldots, e^f_m$$

where `initially` is used to set the initial condition of the state, either *true* or *false*. The occurrence of an event in $\{e^t_1, \ldots, e^t_n\}$ changes the value of the state of s to *true*, while the occurrence of an event in $\{e^f_1, \ldots, e^f_m\}$ changes the value of the state of s to *false*.

The states of the Drug Delivery are related to the configuration of the doors, that can be either open or closed. For i in $\{1, 2, 3\}$, the event `openDoor`i (resp. `closeDoor`i) changes the value of the state `door`i`open` of `door`i to true (resp. to false). For example, the state `door1open` is specified in PuRSUE-ML as follows:

```
state door1open: initially false, true_if openDoor1 false_if closeDoor1
```

The definition of events and states is a prerequisite for the definition of state dependencies. More precisely, state dependencies specify necessary conditions that allow the occurrence of an event, i.e., a guard on the occurrence of an event. For any event e, and any propositional formula ϕ defined on state variables, a state dependency is defined through keyword `stateDependency` as follows:

$$\texttt{stateDependency}\ e\ \texttt{only\_if}\ \phi$$

[3]Note that multiple agents can react to the occurrence of the same event e.

[4]For clarity and conciseness, cooperation in this work is restricted to basic interactions performed by two agents at the same physical place. Extending the PuRSUE language—and the corresponding translation to TGA—to allow cooperation between agents that are not located at the same POI is fairly straightforward: one should add suitable syntactic constructs to PuRSUE and modify the definition of guard g_{coll} in the agent automata presented in Sect. 4.2.6.

Table 3. Objectives that can be used to specify the mission that the robotic application has two achieve. Each objective (except for Execution objectives) is specified in two different versions, one for infinite executions, and the other for finite executions satisfying $\mathtt{F}\phi_{\text{Goal}}$. e, e_1, e_2 are events occurring in the environment, ϕ is a propositional formula on state variables.

Name	Type	Syntax	TCTL
Execution (e,n)	Reachability	`do e (after n)?` / `do e within n`	$\dfrac{\mathtt{AF}_{\geq n}\, e, \text{ with } n \geq 0}{\mathtt{AF}_{\leq n}\, e, \text{ with } n > 0}$
Reaction (e_1,e_2,n)	Liveness	`if e_1 then e_2 within n`	$\dfrac{\mathtt{AG}(e_1 \Rightarrow \mathtt{AF}_{\leq n}\, e_2)\ /}{\mathtt{A}((e_1 \Rightarrow \mathtt{AF}_{\leq n}\, e_2)\mathtt{U}\phi_{\text{Goal}})}$, with $n > 0$
EventAvoidance (e)	Safety	`avoid e`	$\mathtt{AG}(\neg e)$ / $\mathtt{A}(\neg e \mathtt{U} \phi_{\text{Goal}})$
PositionalAvoidance (a_1,a_2)	Safety	`a_1 never_with a_2`	$\dfrac{\mathtt{AG}(\neg(\ell(a_1) \approx \ell(a_2)))\ /}{\mathtt{A}(\neg(\ell(a_1) \approx \ell(a_2))\mathtt{U}\phi_{\text{Goal}})}$
StateAvoidance(ϕ)	Safety	`avoid` ϕ	$\mathtt{AG}(\neg\phi)$ / $\mathtt{A}(\neg\phi\mathtt{U}\phi_{\text{Goal}})$

In DD, the state dependency defining the necessary condition for the occurrence of event `giveMedicine1` is specified as follows (the others are similar):

```
stateDependency giveMedicine1 only_if door1open
```

The condition prescribes that event `giveMedicine1`, which represents the act of giving the medicine to the nurse at `door1`, can only occur if the door is open. Note that the nurse reacts to the event `giveMedicine`i, with $i \in \{1, 2, 3\}$, that can be performed by the `medBot`.

4.1.6. *Objectives*

An objective describes a desired goal, or a behavior, that the controlled agents operating in the environment have to achieve. Objectives are prescriptions that are expressed in PuRSUE-ML in terms of events or states. The objectives are the instruments developers can use to specify what is the final goal, i.e., the mission, the robotic application should achieve. PuRSUE also enables the specification of multi-objective scenarios, where two or more objectives have to be achieved. In such cases, all the defined objectives are considered in the computation of the strategy: if a strategy exists for the scenario, all the objectives are achieved together. Table 3 lists the objectives supported by PuRSUE-ML. In particular, the table shows the syntax that is used to declare each type of objective. In addition, for each type, a TCTL formalization is provided to better explain the exact meaning of the objective. Notice, however, that the corresponding formal model that is used to generate plans for the mission, though equivalent, is formalized in a different way to make it implementable in the Uppaal- TIGA tool, as described in Sect. 4.2.7. We have three different classes of objectives, namely, Execution, Reaction and Avoidance objectives (where Avoidance objectives are further classified as Event, Positional, and State Avoidance).

Execution(e, n**)** constrains the occurrence of the event e, and it specifies the following reachability properties: in all the executions, event e must occur after/within n time units (if the keyword "after" is omitted $n = \infty$) from the initial instant. The corresponding TCTL formula that expresses the objective is $\mathtt{AF}_{\geq n}\, e/\mathtt{AF}_{\leq n}\, e$, with $n \geq 0$, depending on kind of bound with respect to n, i.e., a lower bound ("after") in the former case, and an upper bound ("within") in the latter case.

The semantics of (Event, Positional, and State) Avoidance and Reaction objectives is influenced by the presence (or absence) of one or more Execution objectives. If Avoidance (resp. Reaction) and Execution objectives are paired in a scenario, then the safety (resp. liveness) property entailed by the Avoidance (resp. Reaction) objective is applied only over finite executions. Otherwise, Avoidance (resp. Reaction) is defined over infinite executions. In the following, we indicate by ϕ_{Goal} a TCTL state formula that encodes an Execution objective in the environment (see the formal definition in Sect. 4.3).

Reaction(e_1**,**e_2**,**n**)** specifies a relation between the occurrence of two events e_1 and e_2, and it specifies the following liveness property: for every execution, it always holds that, if event e_1 happens, then e_2 occurs within n time units. If no execution objectives are specified, the corresponding TCTL formula, that expresses the reaction objective, is $\mathtt{AG}(e_1 \Rightarrow \mathtt{AF}_{\leq n}\, e_2)$, with $n \geq 0$. If, instead, an execution objective—captured by TCTL formula ϕ_{Goal}—is present, then the TCTL formula is $\mathtt{A}((e_1 \Rightarrow \mathtt{AF}_{\leq n}\, e_2)\mathtt{U}\phi_{\text{Goal}})$.

EventAvoidance(e) constrains the occurrence of the event e, and specifies the following safety property: in all executions it always holds that event e does not occur. If no execution objective is specified, the corresponding TCTL formula, that expresses the avoidance of event e, is $\mathtt{AG}(\neg e)$. Otherwise, the TCTL formula is $\mathtt{A}(\neg e \mathtt{U} \phi_{\text{Goal}})$.

PositionalAvoidance(a_1,a_2) constrains the location of agents a_1 and a_2, and specifies the following safety property: for every execution, it always holds that agents a_1 and a_2 are not located in the same POI, at the same time, and they are not traveling simultaneously on the same connection. In Table 3, $\ell(a)$ is either the POI where agent a is currently located, e.g., $\ell(a) = x$ if agent a is in x; or, it represents the connection between two POIs x and y if agent a is moving from x to y. Moreover, symbol $\approx$ has the following meaning: if $\ell(a_1)$ and $\ell(a_2)$ are two POIs (rather than a connection), then $\ell(a_1) \approx \ell(a_2)$ holds if a_1 and a_2 are in the same POI; in addition, if $\ell(a_1)$ and $\ell(a_2)$ represent that a_1 and a_2 are moving on the same connection—but in opposite direction—between some POIs x and y, then $\ell(a_1) \approx \ell(a_2)$ also holds. In all the other cases $\ell(a_1) \approx \ell(a_2)$ does not hold. The formal definitions of ℓ and $\approx$ are provided at the end of Sect. 4.2.6, where they are linked to the automata-based model described in Sect. 4.2. If no execution objective is specified, the corresponding TCTL formula that expresses the avoidance of configurations where agents might collide with one another is $\mathtt{AG}(\neg(\ell(a_1) \approx \ell(a_2)))$. Otherwise, the TCTL formula is $\mathtt{A}(\neg(\ell(a_1) \approx \ell(a_2))\mathtt{U}\phi_{\text{Goal}})$.

StateAvoidance(ϕ) forces the system to avoid reaching states where ϕ holds, and specifies the following safety property: for every execution, it always holds that the system does not reach any configuration that satisfies ϕ, where ϕ is a propositional formula on state variables. If no execution objective is specified, the corresponding TCTL formula, that expresses the avoidance of configurations where ϕ holds, is $\mathtt{AG}(\neg\phi)$. Otherwise, the TCTL formula is $\mathtt{AG}(\neg\phi\mathtt{U}\phi_{\text{Goal}})$.

As an example of compound objective, mixing execution and avoidance goals, in DD the mission is accomplished when the medicines are delivered to the emergency room by avoiding collisions between the robot and the stretcher. This occurs when event `confirmDelivery` is triggerd and agents `stretcher` and `medBot` are never in the same location or travelling in opposite directions between the same two locations. The objective in PuRSUE-ML is written as follows:

```
objective: do confirmDelivery, stretcher-bearer never_with medBot
```

4.2. Intermediate TGA

This section describes the structure of the TGAs that are automatically generated by PuRSUE. In particular, a PuRSUE-ML model is translated into a network of TGAs that properly captures all of the aspects characterizing the evolution of the modeled scenario over time. All aspects discussed in Sect. 4.1 are taken into account by means of specific constructs or resources, such as automata, integer values, or synchronization channels. The final network of TGAs representing a robotic application is composed of the following automata:

- Rule automata (Sect. 4.2.3), accepting only the sequences of events allowed by the rules;
- State automata (Sect. 4.2.5), performing the update of the variables related to the state constraints;
- Agent automata (Sect. 4.2.6), representing the behavior of the agents of the system; and
- Objective automata (Sect. 4.2.7), encoding the goal to be achieved by controllable agents.

Remark in the final network of automata, all events occurring in the environment are associated with broadcast channels, and only agent automata include transitions labeled with synchronization labels of the form $e!$, for some event e. Conversely, all other automata can synchronize only by means of transitions labeled with $e?$. For this reason, agents are only capable of triggering events by executing transitions synchronized with $e!$ whereas rule, state and objective automata only progress through the transitions synchronizing with $e?$. This way, the evolution of the entire system modeling the environment is coordinated, and the behavior of the agents (the sequence of events they execute) is compliant with the other automata in the network.

A PuRSUE-ML model $\mathcal{M}$ can be represented in an abstract form as a tuple $(P, C, E, R, S, D, A, F, O)$ where:

- P is a finite set of POIs defined through the keyword "`poi`".
- C is a multiset of elements in $P \times P \times \mathbb{N}$, defined though the keywords "`connection`" and "`distance`". A triple $(x, y, dist) \in C$ is a connection from x to y with distance $dist$. If a connection is bidirectional, then both $(x, y, dist) \in C$ and $(y, x, dist) \in C$ hold, whereas only one of the two is in C if the connection is unidirectional.
- E is a multiset of elements in $\{0, 1\} \times \mathbb{N} \times P \cup \{\#\}$ representing events defined through the keyword "`event`". An event e is a tuple $(b, n, x) \in E$ such that the flag b indicates whether the event is collaborative $(b = 1)$ or not $(b = 0)$, n indicates its duration, x indicates which POI is associated with e (if $x \in P$) or, if $x = \#$, that e is not bound to any POI. The event (b, n, x) is called *instantaneous* if $n = 0$.
- R is the set of regular expressions defined by the `rule` keyword.
- S represents elements defined by the `state` keyword. It is a set of elements in $\{0, 1\} \times 2^E \times 2^E$. For every $(b, E_t, E_f) \in S$ it holds that $E_t \cap E_f = \emptyset$. Every triple $(b, E_t, E_f) \in S$ is a state, which is initially set to b and that becomes true if some event in E_t occurs (keyword "`true_if`"), and false if some event in E_f occurs (keyword `false_if`).
- D represents elements defined by the `stateDependency` keyword. It is composed of elements in $E \times \Phi(S)$, where $\Phi(S)$ is the set of propositional formulae that are built through conjunction and negation of formulae that predicate on state variables corresponding to elements of S, which are the atomic formulae of $\Phi(S)$. A pair $(e, \phi) \in D$ is a state dependency constraining the occurrence of e with ϕ.
- A represents elements defined by the `agent` keyword. It is a multiset of elements in $\{0, 1\} \times \mathbb{N} \times P \times 2^E \times 2^E$. A tuple (b, v, x, E_1, E_2) is an agent that is controllable if $b = 1$ (keyword `controllable`), it is mobile if the velocity $v > 0$ (`mobile`), it is initially located in x (keyword `location`), it can perform the events associated with events E_1 (keyword `can_do`), and it can react to events in E_2 (keyword `reacts_to`). Events in E_1 are called *action events* and events in E_2 are called *reaction events*.
- F is a set of elements in $\{P \times P \times A\} \cup \{P \times A \times E\} \cup \{P \times P \times A \times E\}$ defined through keyword `prevent`. A tuple $(x, y, a) \in \{P \times P \times A\}$ is a prohibition for agent a of moving from POI x to POI y; a tuple $(x, a, e) \in \{P \times A \times E\}$ is a prohibition for agent a in performing event e in POI x; finally, a tuple $(x, y, a, e) \in \{P \times P \times A \times E\}$ is a prohibition for agent a in performing event e while agent a is traversing the path from x to y (if the prohibition is not unidirectional, both (x, y, a, e) and (y, x, a, e) are in F).
- O is a set of objectives, which are elements of the form:
  - $(e, n, \geq)$, with $e \in E$ and $n \geq 0$, for Execution objectives of the kind $\mathtt{AF}_{\geq n} e$; and $(e, n, \leq)$, with $e \in E$ and $n > 0$, for Execution objectives of the kind $\mathtt{AF}_{\leq n} e$,
  - (e_1, e_2, n), with $e_1, e_2 \in E$ and $n > 0$, for Reaction objectives,
  - e, with $e \in E$, for Event avoidance objectives,
  - (a_1, a_2), with $a_1, a_2 \in A$, for Positional avoidance objectives and
  - propositional formulae $\phi \in \Phi(S)$, where $\Phi(S)$ is the set defined above.

  In the rest of the paper, we indicate by O_A the set of objectives of types Execution, Reaction, and Event avoidance.

The rest of this section describes how, given a PuRSUE-ML model $\mathcal{M} = (P, C, E, R, S, D, A, F, O)$, the corresponding network of TA $\mathcal{M}_\mathcal{A}$ is defined. More precisely: (i) each rule $r \in R$ corresponds to a rule automaton $\mathcal{A}_r$; (ii) the set of states S corresponds to a single state automaton $\mathcal{A}_S$; (iii) each agent $a \in A$ corresponds to an agent automaton $\mathcal{A}_a$; and, (iv) each objective $o \in O_A$ corresponds to an objective automaton $\mathcal{A}_o$ (Positional and State avoidance objectives are formalized differently). Then, the network of automata $\mathcal{M}_\mathcal{A}$ is defined as the composition

$$\mathcal{M}_\mathcal{A} = \{\mathcal{A}_r\}_{r \in R} \ || \ \mathcal{A}_S \ || \ \{\mathcal{A}_a\}_{a \in A} \ || \ \{\mathcal{A}_o\}_{o \in O_A}, \tag{2}$$

where $\{\mathcal{A}_k\}_{k \in K}$ stands for $||_{k \in K} \mathcal{A}_k$, for some generic set K of automata. Notice that the network of automata $\mathcal{M}_\mathcal{A}$ essentially defines the semantics of the PuRSUE-ML model $\mathcal{M}$.

4.2.1. Modeling assumptions

The translation that produces a TGA from PuRSUE-ML has been designed to work under the following assumptions:

- Once an agent is committed to an event then no other events can be performed by this agent until the current event is completed. Therefore, simultaneous events are not allowed: while an agent that is executing an event e with a duration n or is traversing a connection, has to wait until e terminates or the destination of the connection is reached before executing a new event or performing another movement.
  The assumption allows us to model realistic behaviors of autonomous agents that can perform one event at a time such as robotic agents with one service arm, which are commonly used in industry [SK16]. Moreover, the assumption is not too restrictive. Simultaneous events can still be modeled, for instance, by means of a durable event and a number of instantaneous ones that occur immediately after it, by following a certain user-defined order that can be enforced by a specific rule. Furthermore, the simultaneous occurrence of an event and a movement can be modeled, as it is always possible to add an extra POI between the start and the end POIs defining a path and associate the new POI with the event to be performed.
- At least one time unit elapses when an agent traverses a POI, or starts a durable event in a POI. Specifically, one time unit passes from the time instant the agent gets to the POI and the instant when it leaves the POI or starts a durable event. The assumption is motivated by the following two facts. Every agent delimits a physical space with a certain geometric shape. Therefore, the act of moving over a POI, which includes entering and leaving the POI, always requires at least some strictly positive amount of time. We assume that 1 time unit is the least time delay measurable in the environment. Similarly, a durable event cannot be considered as "operating" if a positive amount of time has not elapsed since the end of the last movement or event. It is reasonable to consider that the agent always performs a preliminary setup of a durable event that lasts (at least) 1 time unit.
- The collaboration that is realized through a collaborative event always involves all of the agents that are enabled to collaborate.
- A collaborative event with a duration is considered to be available for triggering if the following condition is verified: acting and reacting agents are in the same location, and none of the two is prevented from executing the event. In the case of a collaborative event without duration, the event is considered available for triggering if the acting and reacting agents are either in the same location, or they are traveling between two locations in opposite directions, and none of the two is prevented from executing the event. In such a case, the two agents eventually meet along the path, and collaborate with each other.
- All events in the environment are observable. As we are interested in scenarios where the knowledge of the entire environment is realized in favor of the achievement of a certain goal, we assume that the controller has full observability of the system. Such an assumption does not entail the absence of uncertainty in the sensing function implemented by the autonomous agents and the controller. Observations of the environment are, however, always enough to determine the state of all the agents and all the objects (e.g., doors) interacting with them, yet they might, in general, be affected by some skew (consider, for instance, the Monte Carlo localization algorithm for estimating the agent position within a map that uses the perceived observation of the environment). Applications that meet this requirement can be found in realistic scenarios. According to the taxonomy elaborated in [FIN04], they are commonly classified as follows: i) *cooperative*, i.e., applications where agents realize a given global task by means of coordinated event; ii) realized by agents that are *aware*, i.e., where every agent has some degree of knowledge of other agents in the environment; and iii) *strongly coordinated*, i.e., applications that are based on an information exchange system allowing agents to implement a predefined coordination protocol, governed by a central actor called leader. Several works fall into the latter class, and a survey can be found in [FIN04]. For instance, [BREC07] considers an application where a Mars lander manages a group of rovers by using observations that they collect on the field.

4.2.2. Support variables

To correctly model the evolution of the environment, i.e., the coordinated evolution of all the automata modeling the interaction of the agents, the network of TGAs is endowed with integer variables, clocks, and channels, which are used to model (i) the progress of the agents performing actions in the environment, (ii) the time elapsing with respect to relevant events occurring in the environment, and (iii) the coordinated evolution of two, or more, automata on the occurrence of certain events.

Channels In a TGA network, channels allow for the synchronization of several automata, therefore enabling a coordinated and simultaneous evolution of various automata modeling the environment. For this reason, channels model the occurrence of all the events occurring in it.

For every event $e \in E$, a broadcast channel $\mathtt{e}$ is introduced. The semantics of a synchronization occurring through the channel is that "the execution of event e has started". Furthermore, if $e = (b, n, x) \in E$ is such that $n > 0$, then a channel $\mathtt{e}_{\text{done}}$ is also introduced. The semantics of a synchronization occurring through the channel is that "the execution of event e has ended".

Clocks Clocks keep track of the time elapsed from the occurrence of an event, that can be either a movement, or an event performed by some agent in the environment. For every agent $a \in A$, a clock $\mathtt{C}_a$ is introduced, and it is reset every time an event is performed by a (details in Sect. 4.2.6). Moreover, a clock $\mathtt{C}_o$ is introduced for each objective $o \in O$. Clock $\mathtt{C}_o$ is reset according to the mission as presented in Table 3 (details in Sect. 4.1.6).

Integer variables Variables are introduced to model the collaboration among agents and the concurrent and coordinated evolution of their configuration, driven by the occurrence of events.

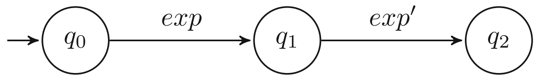
- For every rule $r \in R$, a variable $\mathtt{P}_r$ ranging over $\{1, \ldots, M_r\}$ is introduced, where M_r is the number of locations of the finite state automaton (adopted for) accepting the language associated with r. The variables $\mathtt{P}_r$ keep track of the evolution of every rule automaton, i.e., its current location, which is stored in $\mathtt{P}_r$. Agent automata exploit these variables on their transition guards to guarantee that they only evolve through the sequences of events that are allowed by the rules.
- For every agent $a \in A$, a bounded integer $\mathtt{P}_a$, whose domain is $[- \mid P^2 \mid, \mid P \mid]$, is introduced to manage the collaboration among the agents. Variable $\mathtt{P}_a$ is, in fact, used in the guards of the agent automata to rule the simultaneous progress of two collaborative agents. Specifically, every $\mathtt{P}_a$ keeps track of the current location occupied by a or of the current action that a is performing. Let $l : P \to \{1, \ldots, |P|\}$ be a function associating a unique integer value with every POI in P and the operator $\circ$ be a bijection between the sets $\{1, \ldots, |P|\} \times \{1, \ldots, |P|\}\{1, \ldots, |P|\}$ and $\{1, \ldots, |P|^2\}$. The value of $\mathtt{P}_a$ is:
  - $\mathtt{P}_a = l(x)$, if agent a is in POI x and not performing any event;
  - $\mathtt{P}_a = -l(x) \circ l(x)$, if agent a is in POI x and performing some event;
  - $\mathtt{P}_a = -l(x) \circ l(y)$, if agent a is moving from POI x to y.
- For every state $s \in S$, an integer $\mathtt{S}_s$ is introduced to implement state dependencies. The value of $\mathtt{S}_s$ can be either 0 or 1, and is updated by the state automaton according to the logic defined by the user though PuRSUE-ML as explained in Sect. 4.1.5. Agent automata exploit these variables on their transition guards to guarantee that the events they perform satisfy the necessary conditions determined by the current value of the state variables.

4.2.3. Rule automata

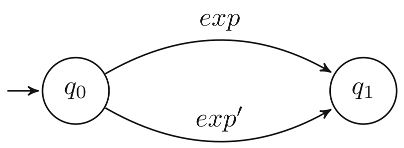
For each rule $r \in R$, a rule automaton $\mathcal{A}_r$ is built that accepts the same regular language captured by the regular expression corresponding to r. This can be done using standard techniques from the formal languages literature, such as the Berry-Sethi method [BS86]; however, it is even simpler for PuRSUE-ML rules, which are a subset of regular expressions. First of all, rule automata do not use clocks and location invariants, because rules do not capture real-time constraints. Then, since PuRSUE-ML allows only for the concatenation and disjunction operators of regular expressions, there are two automata templates that capture these operators, and that can be composed as dictated by the regular expression in which the concatenation and disjunction operators appears.

Figure 3a and b show the two templates for the operators exp `before` exp' and for exp `or` exp', respectively, where exp and exp' are two generic PuRSUE-ML rule expressions that are recursively expanded. Every rule is cyclic, i.e., if exp_r is the regular expression corresponding to rule r, then the sequences of events constrained by the rule are those in $(exp_r)^*$.

All transitions of $\mathcal{A}_r$ are synchronized with events and perform updates. First, if $e \in E$ is the event associated with a transition, then the latter is synchronized with $e?$.



(a) Automaton accepting exp `before` exp'.



(b) Automaton accepting exp `or` exp'.

Fig. 3. Rule automata

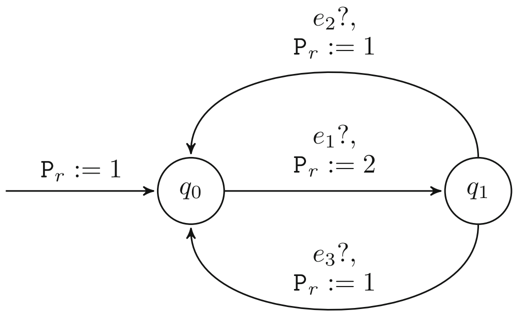

$e_2?$, $\mathtt{P}_r := 1$
$\mathtt{P}_r := 1$
q_0
$e_1?$, $\mathtt{P}_r := 2$
q_1
$e_3?$, $\mathtt{P}_r := 1$

Fig. 4. Automaton accepting e_1 `before` (e_2 `or` e_3), where $h_r(q_0) = 1$ and $h_r(q_1) = 2$

Moreover, as explained in Sect. 4.2.2, every rule $r \in R$ is associated with an integer $\mathtt{P}_r$, whose domain is the set $\{1, \ldots, M_r\}$ and M_r is the number of locations of the automaton accepting words of r. Let Q be the set of states of $\mathcal{A}_r$ and h_r be a bijection from Q to $\{1, \ldots, M_r\}$. All incoming transitions in a location q update $\mathtt{P}_r$ through the assignment $\mathtt{P}_r := h_r(q)$.

The automaton corresponding to the rule e_1 `before` (e_2 `or` e_3) is shown in Fig. 4.

4.2.4. Event-related automata

PuRSUE-ML includes events with a duration, which are defined through the keyword `duration` (see Sect. 4.1.2). As mentioned in Sect. 4.2.2, if an event $e \in E$ has a duration, PuRSUE automatically introduces event e_{done} (and its corresponding channel) to model the conclusion of the event. In addition, the following rule r_e is added to set R (hence, it is translated into the corresponding rule automaton $\mathcal{A}_{r_e}$, as explained in Sect. 4.2.3)

e `before` e_{done}

to constrain the order between the beginning and the end of the event. The timing constraints enforcing the event duration are included in the agent automata (see Sect. 4.2.6), and they are expressed using the (unique) clock used by the agents (clock $\mathtt{C}_a$ of agent a, for any $a \in A$). This way, the elapsing of time that has to be evaluated to model the event with duration can be measured without introducing a clock in the rule automaton $\mathcal{A}_{r_e}$ (recall that an agent can perform only one event at a time). Furthermore, every occurrence of a non-instantaneous event e in a rule $r \in R$ is automatically replaced with the corresponding expression e `before` e_{done}; for instance, if e-dur is an event with a duration, e-inst is an instantaneous event, and rule e-dur `before` e-inst belongs to R, then the latter is replaced in R by the new rule e-dur `before` e-dur$_{\text{done}}$ `before` e-inst.

4.2.5. State automata

A state automaton manages the updates of the values of the state variables throughout the evolution of the environment over time. To this end, for every robotic application only one state automaton $\mathcal{A}_S$ is introduced. The state automaton has only one location. For every state $s \in S$ of the form (b, E_t, E_f) (where $E_t \subseteq E$, $E_f \subseteq E$, and $E_t \cap E_f$ hold, as described in Sect. 4.2.2), $\mathcal{A}_S$ is endowed with $| E_t | + | E_f |$ distinct transitions, one for each event $e = (b, n, x) \in E_t \cup E_f$, with $b \in \{0, 1\}$, $n \in \mathbb{N}$ and $x \in P$.

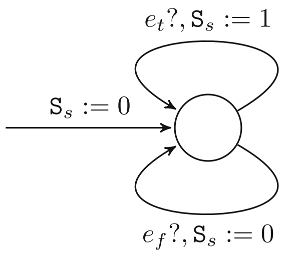


Fig. 5. State automaton with three transitions updating the value of a state s

In particular, if $n = 0$ (i.e., e is instantaneous) then every transition is labeled with a synchronization guard of the form $e?$ and with the assignment $\mathtt{S}_s := 1$ (if $e \in E_t$) or $\mathtt{S}_s := 0$ (if $e \in E_f$). If $n > 0$ (i.e., e is not instantaneous) the synchronization labels are specified by using $e_{\text{done}}?$ instead of $e?$, and assignments are specified as above. This allows the update of the state only when event e is concluded. Figure 5 exemplifies the template automaton by showing only the two transitions associated with a state $s \in S$ of the form $(0, \{e_t\}, \{e_f\})$.

4.2.6. *Agent automata*

For each agent $a = (b, v, x_0, E_1, E_2) \in A$, the corresponding agent automaton $\mathcal{A}_a$ is introduced, where $\mathcal{A}_a$ models the following aspects:

- the position of a, if a is idle;
- the act of motion of a, or the act of performing of an event; and
- whether a is capable of triggering a certain event, according to the constraints defined by the user.

We model the previous aspects in $\mathcal{A}_a$ through the following means, and by using a single clock $\mathtt{C}_a$ and variable $\mathtt{P}_a$. We recall that $\mathtt{P}_a$ is a global variable (all automata can read its value) that contains the "current status" of agent a, i.e., whether it is performing an event in a certain location, or it is traversing a path between two POIs (see Sect. 4.2.2; also, we recall that E_1 is the set of action events (modeling events that the agent "can do") and E_2 is the set of reaction events (modeling events that the agent "reacts to"). Finally, if the agent is controllable (i.e., $b = 1$), then all transitions in the agent automaton $\mathcal{A}_a$ are controllable. Otherwise, when $b = 0$ holds, all transitions are uncontrollable.

The rest of this section describes how the locations and transitions of $\mathcal{A}_a$ are defined from a.

First of all, the set Q^a of locations of automaton $\mathcal{A}_a$ is defined as $Q^a = Q^a_{\text{POI}} \cup Q^a_{\text{mov}} \cup Q^a_{\text{ev}}$, where the various subsets of Q^a capture the following situations: the agent standing in a POI of the environment (Q^a_{POI}); the agent moving in the environment (Q^a_{mov}); and, the agent performing an event of non-null duration that is not a movement (Q^a_{ev}). We describe the various sets of locations in more detail, and their incoming and outgoing transitions.

POI locations Q^a_{POI}. POI locations model the presence of agent a in a certain POI of the environment, and that a resides there for a strictly positive amount of time. If agent a is mobile (i.e., $v > 0$), then we define $Q^a_{\text{POI}} = \{q_x \mid x \in P\}$, because the agent can generally reside in every POI; otherwise, we define $Q^a_{\text{POI}} = \{q_{x_0}\}$. The initial state of $\mathcal{A}_a$ is q_{x_0} (notice that an agent is always bound to its initial POI if it is not mobile).

As already discussed in Sect. 4.1.1, every POI is an abstraction of a physical space in a realistic world, that has no geometric characterization and dimension. However, modeling the realistic behaviors of agents requires the following assumption on the physical space occupied by the POI: an agent passing through a POI, while moving towards another position, takes at least a certain positive amount of time (Sect. 4.2.1). To this end, suitable constraints on clock $\mathtt{C}_a$ are introduced in the guards of the transitions of automaton $\mathcal{A}_a$ that enter locations of set Q^a_{POI}. Moreover, these transitions include suitable updates of variable $\mathtt{P}_a$ to represent the fact that agent a reaches a POI.

More precisely, for each POI location $q_x \in Q^a_{\text{POI}}$:

- All incoming transitions reset $\mathtt{C}_a$ and update the value of $\mathtt{P}_a$ with $l(x)$ (i.e., agent a is in POI x), as explained in Sect. 4.2.2.

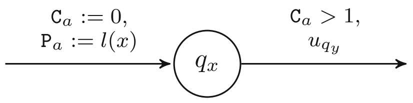


Fig. 6. POI location of an agent automaton $\mathcal{A}_a$ with one incoming and one outgoing transition

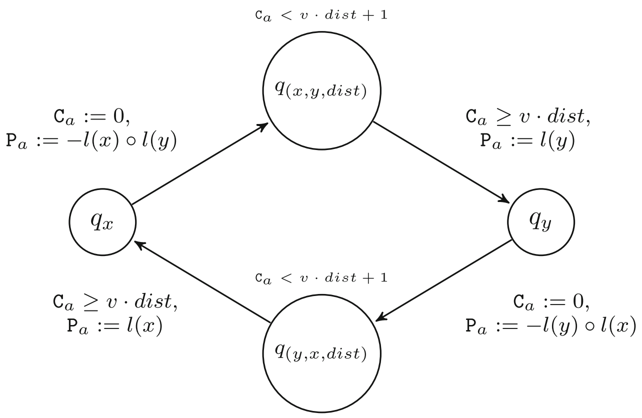

$\mathtt{C}_a < v \cdot dist + 1$
$q_{(x,y,dist)}$
$\mathtt{C}_a := 0,$
$\mathtt{P}_a := -l(x) \circ l(y)$
$\mathtt{C}_a \geq v \cdot dist,$
$\mathtt{P}_a := l(y)$
q_x
q_y
$\mathtt{C}_a < v \cdot dist + 1$
$\mathtt{C}_a \geq v \cdot dist,$
$\mathtt{P}_a := l(x)$
$\mathtt{C}_a := 0,$
$\mathtt{P}_a := -l(y) \circ l(x)$
$q_{(y,x,dist)}$

Fig. 7. Bidirectional connection between POIs x and y, where $dist$ is the distance between the POIs

- All outgoing transitions are labeled with clock constraint $\mathtt{C}_a > 1$, which ensures that the system maintains a physically feasible behavior, i.e., agents are prevented from reaching and leaving a POI in an infinitesimal[5] amount of time (Sect. 4.2.1). Each transition leaving q_x also includes an update of variable $\mathtt{P}_a$ that depends on the location connected to q_x through the transition (as explained later).

Figure 6 shows an example of a POI location q_x with one incoming and one outgoing transition. In this example, $y \in P$ is the POI reached by the agent after leaving POI x, and it holds that POIs x and y are connected through connection $(x, y, dist) \in C$, for some distance $dist$. Abbreviation u_{q_y} stands for the update performed when the outgoing transition is taken, which is not detailed here for the sake of clarity, but which is presented below.

Movement locations Q^a_{mov} Movement locations model the motion, from one POI to another, that is performed by a mobile agent a. We define $Q^a_{\text{mov}} = \{q_{(x,y,dist)} \mid (x, y, dist) \in C \wedge (x, y, a) \notin F\}$; that is, for every connection in C linking POIs x and y with distance $dist$, a movement location $q_{(x,y,dist)}$ is introduced, if the direction is not prohibited for agent a.

Every location $q_{(x,y,dist)} \in Q^a_{\text{mov}}$ is connected to POI location q_x (resp. q_y) by means of a transition from q_x to $q_{(x,y,dist)}$ (resp. from $q_{(x,y,dist)}$ to q_y) such that:

- The transition entering $q_{(x,y,dist)}$ resets clock $\mathtt{C}_a$ to model the duration of the movement. The value of $\mathtt{P}_a$ is updated according to Sect. 4.2.2 with the value $-l(x) \circ l(y)$, to represent the motion from x to y.
- The guard of the transition exiting $q_{(x,y,dist)}$ includes clock constraint $\mathtt{C}_a \geq v \cdot dist$, which guarantees that the time required by agent a to traverse the path from x to y is higher than, or equal to, the product of the speed v of the agent and the distance $dist$ between x and y. The update of $\mathtt{P}_a$ sets its value to $l(y)$ to represent that agent a reaches POI y when the outgoing transition is taken.

Location $q_{(x,y,dist)}$ also has an invariant, $\mathtt{C}_a < (v \cdot dist) + 1$, which guarantees that agent a cannot take more than $(v \cdot dist) + 1$ time units to traverse the path from x to y. Figure 7 shows an example of locations and transitions formalizing the movement of agent a along the bidirectional connection between x and y (of distance $dist$).

Durable event locations Q^a_{ev} Each durable event location models the act of agent a performing an event $e = (b, n, x)$ that is not a movement, and whose duration is non-null (i.e., $n > 0$ holds). More precisely, set

[5]A weaker model where $\mathtt{C}_a > 1$ is not considered would allow executions where the time expended by an agent to pass through a POI can be any $\epsilon > 0$. This affects negatively the modeling. For instance, in the Catch-the-thief scenario presented in Sect. 5, the thief is always able to escape a POI earlier than a cop that is gaining that location.

Q^a_{ev} is defined as

$$\{q_{(e,x)} \mid e = (b, n, \#) \in E_1 \cup E_2 \land x \in P \land n > 0 \ \land (x, a, e) \notin F\} \cup$$
$$\{q_{(e,x)} \mid e = (b, n, x) \in E_1 \cup E_2 \land n > 0 \land (x, a, e) \notin F\} \quad .$$

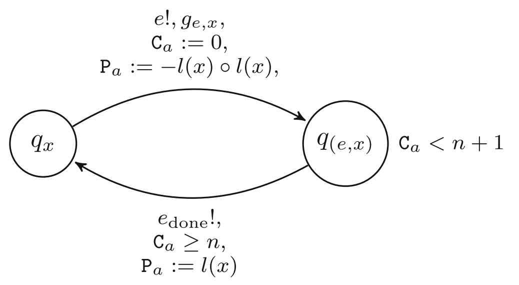
In other words, for every event $e \in E$ with non-null duration:

- if $e = (b, n, \#)$, for each POI $x \in P$, a location $q_{(e,x)}$ is introduced in Q^a_{ev} (since e is not bound to a specific POI and agent a can perform it in every POI), if agent a is not prevented from executing e in x (i.e., if $(x, a, e) \notin F$ holds);
- otherwise, if $e = (b, n, x)$, a location $q_{(e,x)}$ is introduced in Q^a_{ev} only for POI $x \in P$ (since e is bound to POI x and agent a can perform the event only there), unless $(x, a, e) \in F$ holds.

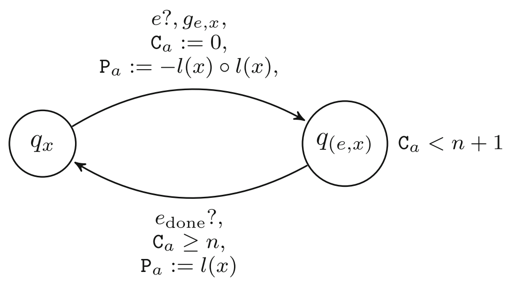
Figure 8 shows the locations and transitions describing the execution of event e, whose duration is n, in POI x, in the two cases where e is an action event of set E_1 (Fig. 8a), or it is a reaction event of E_2 (Fig. 8b). Every location $q_{(e,x)} \in Q^a_{\text{ev}}$ has one incoming transition from the POI location q_x and one outgoing transition towards the POI location q_x such that:

- The incoming transition to $q_{(e,x)}$ is synchronized with $e!$, if $e \in E_1$, or with $e?$, if $e \in E_2$, to model the beginning of event e. Clock $\mathtt{C}_a$ is reset to measure the duration of e. The value of $\mathtt{P}_a$ is updated according to Sect. 4.2.2 with value $-l(x) \circ l(x)$, to represent that agent a is performing event e in POI x. Guard $g_{e,x}$ is defined as $g_\phi \land g_{coll} \land g_r \land g_{\mathtt{C}_a}$, where the conjuncts are defined as follows.
  - g_ϕ is the "state dependency guard": if event e is associated with a State Dependency, i.e., $(e, \phi) \in D$, then g_ϕ is derived from ϕ by replacing every positive occurrence of a state $s \in S$ in ϕ with the formula $\mathtt{S}_s = 1$ (where variable $\mathtt{S}_s$ is defined in Sect. 4.2.2), and every negated occurrence with $\mathtt{S}_s = 0$. Otherwise, g_ϕ is set to true.
  - g_{coll} is the "collaboration guard": if event e is collaborative, i.e., $e = (1, n, x)$ for some n and x, guard g_{coll} is defined differently according to whether $e \in E_1$ or $e \in E_2$ holds. More precisely, if $e \in E_1$, then g_{coll} is $\bigvee_{a' \in \alpha(e)} (\mathtt{P}_{a'} = l(x))$, where $\alpha(e)$ is the set of agents of A that can react to e, $\alpha(e) = \{a' \in A \mid a' = (b', v', p', E'_1, E'_2) \land e \in E'_2\}$. Intuitively, if e is an action event, then the guard forces at least one of the agents a' that can react to the execution of e to be in location x. Otherwise, if $e \in E_2$ holds, then g_{coll} is $\bigvee_{a' \in \beta(e)} (\mathtt{P}_{a'} = l(x))$, where $\beta(e)$ is the set of agents of A that can perform e, $\beta(e) = \{a' \in A \mid a' = (b', v', p', E'_1, E'_2) \land e \in E_1\}$. In other words, g_{coll} guarantees that at least one acting agent a' is in POI x when a reacts to collaborative event e in x.
  - g_r is the "rule guard": in case e appears in some rule $r \in R$, then g_r enables the transition only if the occurrence of e is compliant with the sequence of events allowed by the languages of the rules where e occurs. Let $\rho(e)$ be the set of rules $r \in R$ such that e appears in r and $\eta(\mathcal{A}_r, e)$ be the set $\{q \mid q \xrightarrow{e} q'$ is a transition of $\mathcal{A}_r\}$, i.e., the set of locations of $\mathcal{A}_r$ with an outgoing transition labeled with e. The formula g_r is defined as $\bigwedge_{r \in \rho(e)} \bigvee_{q \in \eta(\mathcal{A}_r, e)} (\mathtt{P}_r = h_r(q))$ (where $\mathtt{P}_r$ and $h_r(q)$ are defined in Sect. 4.2.2). Informally, g_r imposes that, for each rule r in which e appears, there exists at least one state q in automaton $\mathcal{A}_r$ that has an outgoing transition labeled with e, and that the automaton can take, if e occurs (condition $\mathtt{P}_r = h_r(q)$).
  - $g_{\mathtt{C}_a}$ is the "agent clock guard" $\mathtt{C}_a > 1$, because the transition is outgoing from a POI location.
- The outgoing transition from $q_{(e,x)}$ is labeled with a synchronization $e_{\text{done}}!$, if $e \in E_1$, or with $e_{\text{done}}?$, if $e \in E_2$, notifying the termination of the action, and with the clock constraint $\mathtt{C}_a \geq n$ to guarantee that the duration of the event is at least n time units. The update of $\mathtt{P}_a$ is carried out with value $l(x)$, to indicate that agent a is no longer performing the event e in x, yet it still resides in x.

Instantaneous event transitions Instantaneous event transitions model agent actions that have a null duration, i.e., that are associated with an event $e = (b, 0, x)$ for some $b \in \{0, 1\}$ and $x \in P$. Recall that b indicates if the event is collaborative ($b = 1$), and x is the POI in which it can be executed (x is # if the event can be executed in any POI). Figure 9 depicts examples of fragments of automata handling instantaneous events: the one in Fig. 9a deals with POI locations, and the one in Fig. 9b deals with movement locations.
For every action event $e \in E_1$:



(a) A durable event location, modeling an agent a performing an action event e in POI x.



(b) A durable event location, modeling an agent a performing a reaction event e in POI x.

Fig. 8. Template automaton handling durable event locations in two cases: **a** e is an action event or **b** e is a reaction event

- If $e = (b, 0, x)$, then a transition from q_x to q_x is introduced (e is bound to POI x and agent a can only perform it there), unless agent a is prevented from executing e in x (i.e., $(x, a, e) \in F$). The synchronization label is $e!$ and the guard $g_{e,x}$ on the transition is determined in the same manner as the one on the transitions from q_x to $q_{(e,x)}$ in the case of durable events (see Fig. 8).
- Otherwise, if $e = (b, 0, \#)$, a transition from the POI location q_x to q_x is introduced for each POI $x \in P$, labeled with the same guard and synchronization as in the previous case, unless agent a is prevented from executing e in x (i.e., unless $(x, a, e) \in F$). Moreover, a transition from the movement location $q_{(x,y,dist)}$ to $q_{(x,y,dist)}$, for all $(x, y, dist) \in Q^a_{\text{mov}}$, is introduced. The synchronization label is $e!$ and the guard $g_{e,x,y}$ is $g_\phi \wedge g_{mov\text{-}coll} \wedge g_r$, where g_ϕ and g_r are the same as the ones previously defined for durable events, while $g_{mov\text{-}coll}$ is $\bigvee_{a' \in \alpha(e)} (\mathtt{P}_{a'} = -l(y) \circ l(x))$, where $\alpha(e)$ is the set of agents of A that can react to e, $\alpha(e) = \{a' \in A \mid a' = (b', v', p', E'_1, E'_2) \wedge e \in E_2\}$. Constraint $g_{mov\text{-}coll}$ guarantees that an instantaneous and collaborative event e, that takes place while a is moving from x to y, is actually performed only if another agent that can react to e is moving from y to x.

Finally, in both cases, the transitions modeling an instantaneous event do not update $\mathtt{P}_a$ because they neither modify the location of the automaton, nor do they enable a movement of the agent.

Remark In the case of an instantaneous and collaborative event e, the presence of two collaborating agents a and a' in a certain POI, or the simultaneous movement of the agents traversing the same path between two POIs in opposite directions, is enough to guarantee that the collaboration between a and a' by means of e is realized as soon as the action event e is performed. For this reason, no transitions labeled with $e?$ are introduced in the agent automaton $\mathcal{A}_a$. Observe, however, that some rules in the environment might include e and, hence, their automata synchronize by means of $e?$. Furthermore, without loss of generality, we can always assume that for every event e in the whole network of TGAs there is at least one synchronizing label $e!$ and one $e?$. This can be, in fact, realized by means of a syntactical check of the tuple defining the environment.

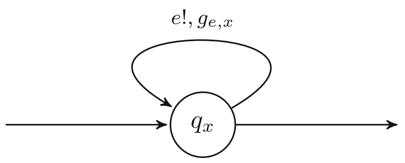
Remark In Table 3 we use function $\ell(a)$, where $a \in A$, to indicate the position or the movement of an agent $a \in A$ at any instant of a possible execution of automaton $\mathcal{A}_a$. In particular, if $(q_0, v_0)(q_1, v_1)\ldots$ is an execution of $\mathcal{A}_a$ (see Sect. 3), then:

- $\ell(a) = x$ holds at position i if q_i is either q_x or $q_{(e,x)}$, for some $x \in P$ and $e \in E$,
- $\ell(a) = (x, y, dist)$ holds at position i if q_i is $q_{(x,y,dist)}$, for some $q_{(x,y,dist)} \in Q^a_{\text{mov}}$.

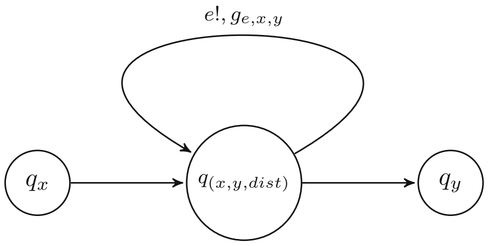
We define relation $\approx \subseteq \{P \cup \bigcup_{a\in A} Q^a_{\text{mov}}\} \times \{P \cup \bigcup_{a\in A} Q^a_{\text{mov}}\}$ as follows: $(x \approx x)$ holds for every $x \in P$; in addition, $(x, y, dist) \approx (y, x, dist)$ holds for every $x, y \in P$ and $dist \in \mathbb{N}$ such that $(x, y, dist)$ is in $\bigcup_{a\in A} Q^a_{\text{mov}}$; for all other pairs the relation does not hold.

4.2.7. *Objective automata and formulae*

An objective automaton $\mathcal{A}_o$ implements an objective $o \in O$ that is of the kind Execution, Reaction or Event avoidance (Sect. 4.1.6). Every automaton is mission-specific, as it depends on the objective o that it represents.



(a) Self-loop on location x dealing with the occurrence of an instantaneous event e in location x.



(b) Self-loop dealing with the occurrence of instantaneous event e in a movement location between POIs x and y.

Fig. 9. Instantaneous event transitions handling event e

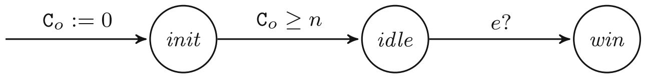


Fig. 10. Automaton representing an Execution objective $\mathtt{AF}_{\geq n} e$, for $n > 0$

Objectives of the kind Position and State avoidance, instead, are encoded through a propositional formula rather than an automaton. The complete formal model of the objectives is a combination of objective automata and of formulae, that are included in the TGA model to capture the reachability/safety game (see Sect. 4.3).

Some objectives are defined in terms of tuples, whose elements can be events in E. Since events might be instantaneous or durable, and they are used to synchronize some transitions of the objective automata, in the following constructions, if an event e is instantaneous, then the considered label is $e?$, otherwise it is $e_{\text{done}}?$ (the effect of a durable event on the environment is determined at its conclusion). For every objective $o \in O$, either an automaton, or a formula is introduced, according to the following rules:

Execution. Objective o has form $(e, n, \geq)$ or $(e, n, \leq)$, with $e \in E$ and $n \geq 0$. The corresponding automaton $\mathcal{A}_o$ is shown in Fig. 10 for the case $(e, n, \geq)$, and in Fig. 11 for the case $(e, n, \leq)$. If o is $(e, n, \geq)$ and $n > 0$, then the automaton has three locations, *init*, *idle* and *win*, and two transitions. The clock constraint $C_o \geq n$ on the transition between *init* to *idle* ensures that at least n time units elapse before event e occurs. Clock C_o is reset entering the initial state at the beginning of the system execution. In the case of $n = 0$ (which means that there is no time constraint on the event), the automaton only has two locations, *idle* and *win*, that are connected with a transition labeled with $e?$, and clock C_o is not introduced. If o is $(e, n, \leq)$, $\mathcal{A}_o$ has two locations, *init* and *win*, and one transition. Clock C_o is reset entering the initial state and is used in the guard of the transition to guarantee that the occurrence of e allows $\mathcal{A}_o$ to reach the *win* location before n time units from the origin.
As explained in Table 3, the presence of Execution objectives influences the semantics of the other types of objectives (which are meaningful only as long as the Execution objectives have not been achieved). As mentioned in Sect. 4.3, this is captured in the TGA model by linking the goal of the reachability game (captured by formula ϕ_{Goal}) to the *win* locations of Execution objective automata.

Reaction Objective o is a tuple of the form (e_1, e_2, n), with $e_1, e_2 \in E$ and $n > 0$. The corresponding automaton $\mathcal{A}_o$, which is shown in Fig. 12, has three locations (*idle*, *atRisk*, *lose*) and three transitions. The automaton has a clock, C_o, which measures the time elapsing between the occurrence of e_1 and e_2. Clock C_o is reset when $\mathcal{A}_o$ moves from *idle* to *atRisk*. The invariant $C_o \leq n$ in location *atRisk* ensures that no more than n time units elapse between the occurrence of e_1 and e_2. In fact, if $C_o \leq n$ holds, and e_2 occurs when the automaton is in *atRisk*, then the automaton moves to *idle*; otherwise, it moves to *lose*.

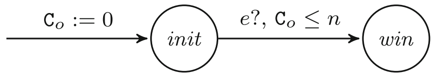


Fig. 11. Automaton representing an Execution objective $\mathtt{AF}_{\leq n} e$, for $n > 0$

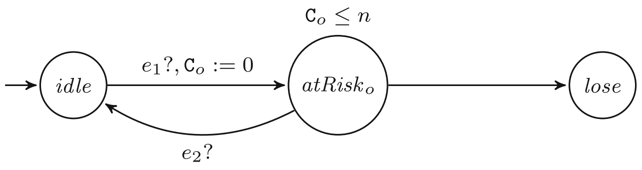


Fig. 12. Automaton representing a Reaction objective

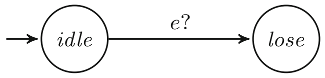


Fig. 13. Automaton representing an Event avoidance objective

Event avoidance Objective o is simply an event $e \in E$. The corresponding automaton $\mathcal{A}_o$, which is shown in Fig. 13, has two locations, *idle* and *lose*. The transition going from *idle* to *lose* ensures that the game is lost when event e occurs.

Positional avoidance Objective o is a tuple of the form (a, a'), where $a, a' \in A$. Positional avoidance for two agents $a, a' \in A$ is translated into a TCTL formula that is included as part of the mission (see Sect. 4.3), rather than an automaton. First of all, we introduce the following two TCTL formulae, where $q_x \in Q^a_{\text{POI}}$ and $q'_x \in Q^{a'}_{\text{POI}}$ are POI locations of agents a and a', respectively, and $q_{(x,y,dist)} \in Q^a_{\text{mov}}$ and $q'_{(y,x,dist)} \in Q^{a'}_{\text{mov}}$ are movement locations of agents a and a', respectively, for any $x, y \in P$:

$$\phi_{\text{same-pos}}(a, a') = \bigvee_{\substack{x \in P \mid \\ q_x \in Q^a_{\text{POI}},\, q'_x \in Q^{a'}_{\text{POI}}}} (q_x \wedge q'_x)$$

$$\phi_{\text{same-move}}(a, a') = \bigvee_{\substack{x, y \in P \mid \\ q_{(x,y,dist)} \in Q^a_{\text{mov}},\, q'_{(y,x,dist)} \in Q^{a'}_{\text{mov}}}} (q_{(x,y,dist)} \wedge q'_{(y,x,dist)})$$

Formula $\phi_{\text{same-pos}}(a, a')$ encodes the conditions that hold when the agents a and a' reside in the same POI, i.e., when the current configurations of the automata (q, v) and (q', v') are such that locations q and q' refer to the same POI $x \in P$. Similarly, formula $\phi_{\text{same-move}}(a, a')$ encodes the conditions that hold when agents a and a' are traversing the same path between POIs $x, y \in P$ in opposite directions. According to the definitions of ℓ and $\approx$ introduced in Sect. 4.2.6, given an execution of (a network including) $\mathcal{A}_a$ and $\mathcal{A}_{a'}$, for every position of the execution it holds that $\ell(a) \approx \ell(a')$ if, and only if, $\phi_{\text{same-pos}}(a, a') \vee \phi_{\text{same-move}}(a, a')$ is true. Then, positional avoidance is simply formula $\neg(\phi_{\text{same-pos}}(a, a') \vee \phi_{\text{same-move}}(a, a'))$.

State avoidance Objective o is a propositional formula over states of set S. As in the case of a Positional Avoidance objective, State Avoidance is formalized through a TCTL formula ϕ_o included in the definition of the mission, rather than an automaton. More precisely, formula ϕ_o is obtained from o, where all the positive occurrences of $s \in S$ are rewritten as $\mathtt{S}_s = 1$ and the negated ones are rewritten as $\mathtt{S}_s = 0$.

4.3. Encoding the mission of the robotic application

We now define the game that encodes the robotic application, and whose strategy (if it exists) allows the robots to achieve their objectives. As mentioned in Sect. 3, a game is formally defined as $(\mathcal{A}, Goal, Safe)$, where $\mathcal{A}$ is the TGA for which the game is solved, *Goal* is a set of configurations of $\mathcal{A}$ that need to be reached, and *Safe* is a set including the only configurations that can be traversed while reaching the goal (i.e., all configurations that are not in *Safe* must be avoided).

The TGA for which the game must be solved is $\mathcal{M}_\mathcal{A}$, which is the composition of TGA defined in Formula (2). In the rest of this section, we first define TCTL formulae ϕ_{Goal} and ϕ_{Safe}, which formalize sets *Goal* and *Safe*, and which are input—together with automaton $\mathcal{M}_\mathcal{A}$—to UPPAAL-TIGA to solve the timed game; then (Sect. 4.3.1), we briefly show that objective automata $\mathcal{A}_o$ (for $o \in O_A$) and formulae $\phi_{\text{Goal}}, \phi_{\text{Safe}}$ correctly capture the semantics of objectives described in Sect. 4.2.7.

For every objective $o \in O_A$ (i.e., of type Execution, Reaction, or Event avoidance), we use the notation $q \in \mathcal{A}_o$ to indicate a location q of automaton $\mathcal{A}_o$ and q_{win}, q_{lose} to indicate locations that are called $win, lose$ in Sect. 4.2.7. In addition, recall that $\Phi(S)$ is the set of propositional formulae expressed using variables $\mathtt{S}_s$ associated with states $s \in S$. The following formula ϕ_{Safe} encodes Reaction, Event avoidance, Positional avoidance, and State avoidance objectives.

$$\phi_{\text{Safe}} := \underbrace{\bigwedge_{\substack{q_{lose} \in \mathcal{A}_o \text{ s.t. } o \in O_A \text{ and} \\ o = (e_1, e_2, n) \text{ or } o = e}} \neg q_{lose}}_{part(a)} \wedge \underbrace{\bigwedge_{(a,a') \in O \setminus O_A} \neg(\phi_{\text{same-pos}}(a, a') \vee \phi_{\text{same-move}}(a, a'))}_{part(a)} \wedge \underbrace{\bigwedge_{\phi_o \in (O \setminus O_A) \cap \Phi(S)} \neg\phi_o}_{part(c)}$$

Notice that ϕ_{Safe}, which is based on formulae $\phi_{\text{same-pos}}(a, a')$ and $\phi_{\text{same-move}}(a, a')$ defined in Sect. 4.2.7, is a TCTL state formula, as no temporal operators occur in it. Hence, its value is determined only by the current configuration (q, v) of $\mathcal{M}_\mathcal{A}$. More precisely, the three subformulae in ϕ_{Safe} have the following meaning: formula $part(a)$ holds in a configuration (q, v) if the current location of each automaton $\mathcal{A}_o$, where o is an objective of the form (e_1, e_2, n) (Reaction objective) or $o = e$ (Event avoidance objective), is not q_{lose}; formula $part(b)$ encodes Positional avoidance objectives (a, a') (which by definition belong to set $O \setminus O_A$), and it holds in (q, v) if no pair of agents $(a, a') \in O \setminus O_A$ reside in the same POI, or traverse the same path between two POIs in opposite directions; formula $part(c)$ encodes State avoidance objectives (where each State avoidance objective, which by definition belongs to set $O \setminus O_A$, is a formula $\phi_o \in \Phi(S)$), and it holds if the assignments to variables $\mathtt{S}_s$ in (q, v) are such that each ϕ_o does not hold.

The following formula ϕ_{Goal} (which is also a state formula that does include any temporal operators) encodes Execution objectives.

$$\phi_{\text{Goal}} := \bigwedge_{\substack{q_{win} \in \mathcal{A}_o \text{ s.t.} \\ o \in O_A \text{ and } o = (e, n, \sim)}} q_{win}$$

The formula holds when the current location of each automaton $\mathcal{A}_o$ for an objective of the form $(e, n, \sim)$, with $\sim \in \{\leq, \geq\}$, is q_{win}.

Recall that empty conjunctions are vacuously true. For instance, we consider $\phi_{\text{Goal}} = \top$ when O does not include any Execution objectives.

Finally, the reachability/safety problem for a (network of) TGA $\mathcal{M}_\mathcal{A}$ amounts to finding a strategy ν such that, if O does not contain any Execution objectives, then the following condition holds:

$$\mathcal{M}_\mathcal{A} \models_\nu \mathtt{AG}(\phi_{\text{Safe}}), \tag{3}$$

otherwise the following condition holds:

$$\mathcal{M}_\mathcal{A} \models_\nu \mathtt{A}(\phi_{\text{Safe}}\ \mathtt{U}\ \phi_{\text{Goal}}). \tag{4}$$

4.3.1. Correctness of the mission encoding

As stated in Sect. 4.2, the network of automata $\mathcal{M}_\mathcal{A}$ provides the semantics of PuRSUE-ML model $\mathcal{M}$. In particular, the components $\{A_r\}_{r \in R}$, $\mathcal{A}_S$ and $\{A_a\}_{a \in A}$ formalize the behavior of agents operating in the environment. Automata $\{\mathcal{A}_o\}_{o \in O_A}$ and formulae ϕ_{Safe} and ϕ_{Goal}, instead, formalize the objectives to be pursued by the robotic application. In the rest of the section, we discuss the correctness of the encoding of the mission as a timed game.

In particular, we aim to show the following: let ν be a winning strategy of the game captured by Formula (3) (resp., Formula (4)) if set O includes (resp., does not include) an Execution objective; then, automaton $\mathcal{M}_{\mathcal{A}}$ controlled with strategy ν satisfies all TCTL formulae of Table 3 corresponding to objectives of set O.

We study the two cases (3) and (4) separately.

$\mathtt{AG}(\phi_{\text{Safe}})$**:** in this case, in each configuration of every run of $\mathcal{M}_{\mathcal{A}}$, formula ϕ_{Safe} holds. Then, subformulae (a), (b) and (c) of ϕ_{Safe} hold. Hence, formula $\mathtt{AG}\ \neg(\ell(a) \approx \ell(a'))$ holds for every Positional avoidance objective $(a, a') \in O \setminus O_A$; in addition, formula $\mathtt{AG}(\neg\phi_o)$ holds for every State avoidance objective $\phi_o \in (O \setminus O_A) \cap \Phi(S)$. Formula $\mathtt{AG}(e_1 \Rightarrow \mathtt{AF}_{\leq n} e_2)$ holds for every Reaction objective $(e_1, e_2, n) \in O_A$. Indeed, if objective $o \in O_A$ is (e_1, e_2, n), subformula (a) imposes that location q_{lose} is never reached in any run of the corresponding automaton $\mathcal{A}_o$ (see Fig. 12). It is easy to see that, in this case, the sequence of events e_1 and e_2 is such that, in each run, $\mathcal{A}_o$ visits alternatively locations q_{idle} and q_{atRisk} and, therefore, every occurrence of e_2 is no more than n time units later than the previous occurrence of e_1.
Similar considerations show that formula $\mathtt{AG}(\neg e)$ holds for every automaton $\mathcal{A}_o$ corresponding to an Event avoidance objective $o \in O_A$ (see Fig. 13).

$\mathtt{A}(\phi_{\text{Safe}}\ \mathtt{U}\ \phi_{\text{Goal}})$**:** in this case, along every run of $\mathcal{M}_{\mathcal{A}}$ formula ϕ_{Goal} eventually holds, and formula ϕ_{Safe} holds in all configurations until then. Formula ϕ_{Goal} holds in all configurations in which every automaton A_o corresponding to an Execution objective $o \in O_A$ of the form $(e, n, \sim)$ is in location q_{win}. As a consequence, the sequence of events in every execution is such that there is an occurrence of e that lets $\mathcal{A}_o$ move from q_{init} to q_{win} and:

- If $\sim$ is $\geq$ (see Fig. 10), then more than n time units passed since the beginning of the execution. Indeed, the clock constraint on the transition between q_{init} and q_{idle} forces e to occur more than n time units after the initial position.
- If $\sim$ is $\leq$ (see Fig. 11), then no more than n time units have passed since the beginning of the run. In this case, the clock constraint on the transition between q_{init} and q_{win} forces e to occur no more than n time units after the initial configuration.

Hence, formula $\mathtt{AF}_{\sim n}(e)$ holds for any Execution objective $(e, n, \sim) \in O_A$.
In addition, since, as remarked above, formula ϕ_{Safe} holds in all configurations of each run of $\mathcal{M}_{\mathcal{A}}$ until ϕ_{Goal} holds, then in a similar manner as for case $\mathtt{AG}(\phi_{\text{Safe}})$ it is easy to see that all Reaction and Event, Positional, and State avoidance objectives are enforced until the configuration where ϕ_{Goal} holds. That is, for every $(e_1, e_2, n) \in O_A$, $o \in O_A \cap E$, $(a, a') \in O \setminus O_A$, $o \in (O \setminus O_A) \cap \Phi(S)$, formulae $\mathtt{A}((e_1 \Rightarrow \mathtt{AF}_{\leq n} e_2)\ \mathtt{U}\ \phi_{\text{Goal}})$, $\mathtt{A}(\neg e\ \mathtt{U}\ \phi_{\text{Goal}})$, $\mathtt{A}(\neg(\ell(a_1) \approx \ell(a_2))\ \mathtt{U}\ \phi_{\text{Goal}})$ and $\mathtt{A}(\neg\phi\ \mathtt{U}\ \phi_{\text{Goal}})$ hold, respectively.

5. Evaluation

We evaluate PuRSUE with respect to the following aspects.

- To understand how PuRSUE helps designers model and analyze robotic applications, we check whether PuRSUE allows for modeling complex robotic applications, and we compare the size of the PuRSUE-ML specification with the size of the generated TGA (Sect. 5.2).
- To evaluate how PuRSUE supports designers in generating controllers for robotic applications, we apply the approach to some applications, and we estimate the saved development time (Sect. 5.3).
- To provide a qualitative estimation of the scalability approach, we analyzed how the number of agents that are present in the robotic application affects the performance of PuRSUE(Sect. 5.4).
- To evaluate whether the control strategy generated by PuRSUE is effectively implementable on actual robots, we deployed the controller generated by UPPAAL- TIGA on a Turtlebot [FWom] (Sect. 5.5).

The analysis was carried out by considering three robotic applications, Catch the Thief (CT), Work Cell (WC) and EcoBot (EB), further described in Sect. 5.1, that are inspired by case studies in the literature, respectively in [QLFAI18, AMRV19], and [TSF$^+$16]. Thirteen variants (scenarios) were obtained by changing the specifications of the application (6 for CT, 3 for WC, and 4 for EB). The Drug Delivery scenario has also been considered. All of the results can be found in [Onl19].

Table 4. Variations of the Catch the Thief robotic application.

Sc.	Env.	Description
CT1	E1	The environment includes four POIs (`a`, `b`, `c` and `d`), and two agents, called `policeBot` and `thief`. The act of catching is a collaborative event `catch` between `policeBot` (acting) and `thief` (reacting), and the mission is expressed with an Execution objective of the form (`catch`, 0). Agent `thief` move two times faster than `policeBot`
CT2	E2	The environment of *CT1* is enriched with a `WeaponCloset`, connected to `a`. The agents are the same of *CT1*. An additional constraint, imposing that the robot needs first to pick up its weapon in a closet before catching the thief, is implemented either through a rule (*CT2a*), or through states (*CT2b*). In *CT2a*, `catch` can only be triggered after event `pickUpBaton` performed by `policeBot` in `WeaponCloset`. In *CT2b*, state `hasBaton` is set to true when `pickUpBaton` is triggered, and a state dependency ensures that `catch` can only be triggered when the state `hasBaton` is true.
CT3	E1	There are two (*CT3a*) or three (*CT3b*) `policeBot` robots that are active in catching `thief` in the same environment of the base scenario. The `policeBot` will move at the same speed as the `thief`
CT4	E3	The base scenario, but in a more complex environment (see the textual description of E3).
CT5	E4	The environment of *CT4* is enriched with three additional POIs: `window1`, `window2` and `stairs`. `thief` can steal objects, located in `a`, `b` and `d`, and can escape through a `window` or the `stairs`. The goal of `policeBot` is to catch `thief` before `thief` leaves the area after a successful theft. To model the capability of the thief to enter or exit the building, a state `away` is added; when `away` is $true$, `thief` is outside the building. Three location specific events, namely, `leave1`, `leave2` and `leave3`, are defined to change `away` to $true$, and the location specific event `enter` is defined to turn `away` to $false$. To model the capability of the thief to steal objects in the three locations, three location specific events, namely, `steal1`, `steal2` and `steal3`, are defined. These events are constrained by state dependecies, that enable their occurrence only if `away` is $false$. The event `stolen` represents a successful theft and the evasion. A rule prescribes that either `stolen` or `catch` can occur only after one of the three stealing events has been performed. A state dependency constrains the occurrence of `stolen`, that can only be triggered when `away` is $true$; and, a different dependency imposes that `catch` can occur only if `away` is $false$. The mission is expressed with an Event Avoidance objective for event `stolen`
CT6	E4	The same scenario as *CT5* where controllable and non-controllable agents swap roles. The controllable robot has to steal an object from the office, while an uncontrollable security agent has to stop it. The speeds of the two agents are set equal. The mission is expressed with an Execution objective of the form (`stolen`, 0, $\geq$).

5.1. Case studies

Catch the Thief (CT) This case study has been inspired by the example presented by Quattrini et al. [QLFAI18], and has been considered as it represents a scenario in which two agents (the robot and the thief) have opposite goals. A robot-cop (`policeBot`) and a human (`thief`) are both located in a complex environment. The `policeBot` has to catch the thief by means of an immobilizer mounted on a baton. The `thief` is free to move in the environment to avoid being captured. The environments taken into consideration are the following:

- E1 is a room of the Jupiter building of the University of Gothenburg and Chalmers, that is surrounded by a corridor (4 POIs and 4 connections).
- E2 is the same room of the Jupiter building as in E1, with an additional weapon closet next to it.
- E3 is the entire third floor of the Jupiter building of the University of Gothenburg and Chalmers is considered as the environment in which the robotic application is deployed (14 POIs and 17 connections).
- E4 is the same floor of the Jupiter building as in E3, with the addition of windows and stairs (17 POIs and 23 connections).

Modeling details and some variations to the scenario are reported in Table 4, where column **Sc.** is the acronym identifying the scenario, **Env.** is the environment in which it is set, and **Description** is a short summary of what characterizes this variation.

Work cell (WC) A work cell is composed of a work unit, operating specific tasks on boxes, and a conveyor belt that carries boxes to the work unit. The following sequence of tasks is performed by a human (`human`) on a box upon its arrival to the work unit: the box is first lifted, and then jigs are screwed into it; finally, the box is put back on the conveyor belt. The whole sequence of tasks must be concluded within a time limit set by the factory.

Table 5. Variations of the Work Cell robotic application.

Sc.	Description
WC1	The whole work cell is modeled as a single POI (`station`) and by means of two agents, namely, `human` and `assistBot`. The events `pickUpBox`, `screw` and `putDownBox` model the collection of a box, the screwing of jigs and the dispatching of the box on the conveyor belt, respectively; the arrival of a new box is modeled through event `newBox`. Both agents can perform all the tasks, while `newBox` is performed only by `human`. A rule describes the entire expected workflow that is determined by the following ordered sequence of events: the arrival and the lifting of a new box, the screwing and, finally, the dispatching. The mission is expressed with a Reaction objective of the form (`newBox`, `putDownBox`, T), ensuring that the box is put on the conveyor belt within T seconds since its arrival.
WC2	The work cell is similar to the one in *WC1*, except for the number of operating robots. In this scenario, since the jigs cannot be screwed while a robot is retaining the box, additional robots are in charge of the screwing (one robot R_1 in *WC2a* and two robots R_1, R_2 in *WC2b*). The robotic agents are multi-purpose, as they can perform every task. For this reason, an event is defined for every task and for every agent (e.g. for `human`, `pickUpBoxH`, `screwH` and `putDownBoxH`). To model that an agent is holding a box, a state *busy* is added for every agent (e.g. `busyH`). This state is initially *false*, it becomes *true* if the agent picks up the box and *false* if the agents puts the box on the belt. For every agent, one state dependency ensures that the screwing can only be done if the agent is not busy; and a rule ensures that the dispatching of a box can happen after its collection. The workflow is modeled similarly to *WC1* by using robotic agents that can perform every task. For instance, in *WC2a* the screwing of the jigs is done either by `screwH` or `screwR1`. The workflow is completed when event `done` occurs, that always follows the dispatching. The mission is expressed with a Reaction objective of the form (`newBox`, `done`, T), for some $T > 0$.

Table 6. Variations of the EcoBot robotic application.

Sc.	Description
EB1	The environment is modeled with three POIs (`room`, `base`, `trash`), all connected to the POI `hallway`. The agents are `human`, `ecoBot` and `bin`, the latter being the unique non-mobile agent in the environment, located in `trash`. Event `callBot` models the `human` calling for a cleaning task in the `office`, `getTrash` models the act of collecting the trash, and `throwTrash` is a collaborative event involving the robot and the bin, that models the disposal of the trash. Event `officeClean` models the robot notifying the conclusion of the task. A rule specifies the behavior of the system by constraining that `callBot` is followed by `getTrash` and by `officeClean`. To model that the robot can only hold one piece of trash at the time, a rule forces `trowTrash` to occur after every `getTrash`. To delay consecutive calls of the robot, a rule imposes that an event with duration (`wait`) must occur between every pair of events `callBot`. The mission is expressed with a Reaction objective of the form (`callBot`, `officeClean`, T), for some $T > 0$
EB2	Same scenario as *EB1*, except for `bin`, that is mobile and non-controllable
EB3	The environment is the same as the one of *EB1*, but the non-mobile agents `paperBin` and `plasticBin` model two different bins. State `isPaperOrPlastic` is introduced to distinguish the material of the item to be disposed: `isPaperOrPlastic` is *true* if event `isPaper` occurs, and *false* if event `isPlastic` occurs. Events `isPaper` and `isPlastic` are performed by `human`, that selects the material, by performing either `isPaper` or `isPlastic`, before calling the robot for the garbage collection. This constraint is enforced by a rule that determines the sequence of events `isPaper` or `isPlastic` before `callBot`, and finally `wait`. All the rules and events in *EB1* are extended to cover two different materials (e.g. instead of only having `getTrash` two events are defined, `getPaperTrash` and `getPlasticTrash`). A state dependency allows the robot to trigger `getPaperTrash` only if `ispaperOrPlastic` is *true* and `getPlasticTrash` only if `ispaperOrPlastic` is *false*
EB4	A combination of *EB2* and *EB3*. There are two distinct bins for paper and plastic and both can be moved in the environment.

A robotic assistant (`assistBot`) is provided to the `human` to help him perform the tasks. Both the robot and the human are capable of performing any of the three tasks required at the work unit. Moreover, all combinations of robot/manual tasks are admitted and, as such, the robot will take over any task that the human delays. Modeling details and variants are reported in Table 5.

EcoBot (EB) A robot (`ecoBot`) collects the trash from an office (`office`), and throws it into the trash bin (`bin`), located in a separate area (`trash room`). The robot is initially located in another room, called `base`. The three locations are connected though the hallways of the building (`hallway`). A human (`human`), located in the `office`, can activate the robot to start a cleaning task in the `office`. Upon calling, the robot is expected to collect the trash from the office within a specified amount of time. The robot can only carry one piece of trash at a time. The scenario is set in a portion of a floor of the Jupiter building of the University of Gothenburg and Chalmers. The variations of this scenario are reported in Table 6.

Table 7. Size of the PuRSUE-ML and TGA specification.

	PuRSUE-ML	UPPAAL- TIGA				
Scenario	Constructs	| variables	Locations	Transitions	Total |	Objective
CT1	12	4	26	45	75	Execution
CT2a	16	4	36	61	101	Execution
CT2b	18	5	35	62	102	Execution
CT3a	13	6	38	73	117	Execution
CT3b	14	8	50	101	159	Execution
CT4	36	4	98	185	287	Execution
CT5	56	7	121	291	419	Avoidance
CT6	56	6	121	291	418	Avoidance
WC1	9	6	19	23	48	Reaction
WC2a	19	10	31	44	85	Reaction
WC2b	26	14	42	63	119	Reaction
EB1	19	10	36	40	86	Reaction
EB2	19	10	45	52	107	Reaction
EB3	28	14	49	62	125	Reaction
EB4	28	14	63	86	163	Reaction
DD	42	11	78	135	226	Execution and positional avoidance

The considered examples are significantly different from each other, and ensure that all the features presented in Sect. 4 are considered. Moreover, EcoBot and Work Cell contain missions defining specific timing constraints, i.e., the mission should be achieved within a specified time bound. All the scenarios require an explicit representation of the time to model the duration of the actions, the speed of the robots, and time required to cover a certain path connecting the locations.

5.2. Evaluation of the modeling support

The modeling features offered by PuRSUE-ML are evaluated with respect to the following aspects.

Expressiveness of PuRSUE-ML. PuRSUE-ML successfully allowed the modeling of all the considered scenarios. A detailed analysis of how each of these scenarios was encoded in PuRSUE-ML can be found in Tables 4, 5 and 6, and the PuRSUE-ML specifications can be found in the online repository [Onl19].

Assessing the design effort saved by the usage of PuRSUE-ML The evaluation of the effort in using PuRSUE-ML to model real applications has been carried out by comparing the number of PuRSUE-ML constructs needed to model the robotic application and the size of the TGA obtained from the PuRSUE-ML specification by applying automatically the transformation in Sect. 4.2 (Table 7). Indeed, the latter provides a rough estimation of the magnitude of the number of different "modeling objects", that a human should manage if a manual modeling had been undertaken. Specifically, the size of the PuRSUE-ML specification is measured by computing the total number of constructs (POIs, Connections, Events, Rules, States, State Dependencies, Agents and Objectives) used in it. The size of the TGA is the total number of locations, transitions and variables (Clocks and Integers) of the TGA specification (Column **total** in Table 7).

The size of the TGA model ranges from a minimum of 48, to a maximum of 419 elements. PuRSUE, instead, allows the designer to define complex scenarios with a manageable number of constructs, which in none of the considered cases exceeds 20% of the number of modeling objects included in the TGA model.

It is worth mentioning that most of the lines of the two most complicated scenarios (*CT5* and *CT6*) are needed for the environment definition. This definition can be easily automatized by relying on a graphical interface that further simplifies the design of robotic applications. Furthermore, PuRSUE allows handling different scenarios with simple changes. Consider, for example, the scenarios *EB1* and *EB2*; a simple change in one of the keywords of *EB1* allows the designer to model *EB2*, which is a conceptually very different scenario from *EB1*. The same task would require the addition of 9 states and 12 transitions in the TGA. Indeed, we might experimentally compare manual modeling against synthesizing the models by following our approach in order to have empirical evidence to support the evaluation of this aspect. We will consider this as an interesting future work. However, thanks to the observations and discussion above, we can conclude that PuRSUE is effective in supporting designers in the formal modeling of robotic applications compared to a manual definition of the TGA.

The results clearly show that modeling a robotic application in PuRSUE-ML is significantly easier than developing the same robotic application by directly modeling it using TGA. Indeed, the number of constructs used in PuRSUE-ML is dramatically lower than the total number of constructs required to model the same scenario in UPPAAL- TIGA.

5.3. Automatic controller generation

To evaluate how PuRSUE helps designers in generating controllers for robotic applications (i) we evaluated whether PuRSUE can compute the controllers for the considered robotic applications, and how it helps designers in the design activity; and (ii) we estimated the development time that can be saved by taking advantage of PuRSUE.

In the EB scenarios, the value assigned to the delay between two consecutive requests from the human (i.e., the duration of event `wait`) determines whether the robot is capable of fulfilling the task (assuming a given assignment to the other parameter values of the scenario). Indeed, if it is too low, the robot will not have enough time to throw the trash in the bin. The analysis of the scenario with PuRSUE allowed us to refine the parameter values, and obtain a strategy for the controlled agent. To quantify the duration of `wait`, making the scenario feasible, PuRSUE was run several times with different parameter values. For instance, in *EB1*, the minimum duration is 37 time units (tu), whereas, in *EB2*, the duration varies according to the speed of the bin. If the speed is 2 tu, the duration of `wait` is at least 60 tu in order to obtain a strategy for the robot, while if we set it to 5 tu, the minimum wait turns out to be 65 tu. In *EB3*, the minimum duration is 36 tu. The difference of one time unit between *EB1* and *EB3* is due to the fact that the human needs to select the type of trash before calling the robot. Unfortunately, UPPAAL- TIGA for 32-bit architectures did not complete the analysis of *EB4*, because it ran out of memory. The version supporting 64-bit architectures can likely solve the scenario, but it is not publicly available. For this reason, we were not able to further investigate *EB4*, and determine the existence of the controller. In all other cases, PuRSUE was able to generate a run-time controller.

In the CT use case, not all scenarios allow for the generation of a controller. First of all, by analyzing some instances of CT with PuRSUE, it turns out that the robot must be faster than the human, otherwise all scenarios are trivially unfeasible, and no strategy exists. Even when the robot is faster than the human, no plan was actually generated for *CT3a*, because when the robot reaches the thief in a POI, the thief can always leave the POI while the robot initiates the act of catching (by assumption, as described in Sect. 4.2.1, every agent requires at least one time unit to start the execution of any action upon the arrival in a location). Furthermore, since all the locations in the environment are connected with two others, the thief can always move to a POI that is not guarded. *CT3b* has been devised to improve *CT3a*, and design a safe scenario. Since three robots are employed to surveil the environment, the thief can always be surrounded by the robots that, in any case, protect all POIs around him. Hence, a strategy for the controlled agents exists, and the controller can be defined. No plan can be generated for *WC2a* either, for reasons similar to those of *CT3a*.

To estimate the time saving in the development of controllers with PuRSUE, we evaluated the time to compute a controller, and its size in every scenario. The measures are relevant, as both provide an estimation of the complexity of the generated controllers that a human designer would face without using an automatic approach such as PuRSUE. The time required to generate the strategy includes the time to translate the PuRSUE-ML specification into a TGA, based on the translation in Sect. 4.2, and the time to compute it from the TGA by means of UPPAAL- TIGA. UPPAAL- TIGA is executed with two input parameters, namely, the network of automata $\mathcal{M}_{\mathcal{A}}$ of the form (2), derived from the PuRSUE-ML model, and a TCTL formula encoding the mission of the application, defined according to the definitions in Sect. 4.3. In particular, based on the formulae ϕ_{Safe} and ϕ_{Goal}, derived from the PuRSUE-ML model, the TCTL formula is either $\mathtt{AG}(\phi_{\mathrm{Safe}})$ or $\mathtt{A}(\phi_{\mathrm{Safe}}\ \mathtt{U}\ \phi_{\mathrm{Goal}})$. Table 8 contains the results of our experiments. Since a strategy can be represented in terms of a TA, the output produced by UPPAAL- TIGA is a textual description that includes several cases of the form:

`State` : s

`While you are in:` c_0, wait.

`When:` c_1 `take transition` t_1.

...

`When:` c_n `take transition` t_n.

Table 8. Time (ms) required for generating the TGA from the PuRSUE-ML models (PuRSUE-ML2TGA), and to synthesize the controllers (Uppaal- TIGA), and size of the generated controllers (states and when).

Scenario	PuRSUE-ML2TGA	States	When	Uppaal-TIGA	Total time
CT1	1231	164	194	83	1314
CT2a	1639	588	751	207	1846
CT2b	1238	617	800	240	1478
CT3a	1238	✗	✗	Unfeasible	✗
CT3b	1319	2301	3022	815	2134
CT5	1307	7989	9005	915	2222
CT6	1018	9547	12896	2399	3417
WC1	963	10	6	58	1021
WC2a	985	✗	✗	Unfeasible	✗
WC2b	1015	16	14	78	1093
EB1	1020	77	82	86	1106
EB2	1002	887	987	6850	7852
EB3	1021	332	348	182	1203
EB4	1287	–	–	Out of memory	–
DD	1311	5376	9768	32680	33991

where s specifies the current location for every automaton in the TGA, and the *while* case specifies a condition c_0 on all the clocks and variables in the system that enables a time transition; in other words, if c_0 is satisfied and s holds, then no action is performed by the agents, and a positive amount of time must elapse before the next query to the strategy that will determine the next action. The ith *when* case triggers the specific discrete transition t_i, corresponding to an edge between two locations of a TA in the model when the associated condition c_i holds. For instance, the following excerpt is extracted from the strategy calculated for the DD scenario:

```
 State: ( states.base medBot.going_c_to_door3 nurse.doing_crossRoom_in_room
         stretcher.going_door2_to_c reachObj.initial_location
         nurseBehaviour._nurseBehaviour0s_doing_crossRoom robotTask._robotTask_initial_location )
         PnurseBehaviour=5 ProbotTask=0 PmedBot=-85 Pnurse=-22
         Pstretcher=-48 Sdoor1open=1 Sdoor2open=0 Sdoor3open=0
...
When you are in (1<CmedBot && Cnurse-Cstretcher<=1 && Cstretcher<6 && Cstretcher<=Cnurse) ||
                (1<CmedBot && Cnurse<140 && Cstretcher<7 && Cstretcher-Cnurse<-1),
take transition medBot.a->medBot.going_a_to_door1
                { CmedBot > 1, medBot_a2door1!, PmedBot := -63, CmedBot := 0 }
```

The size of a strategy is obtained by computing the sum of the number of states (column **states**) and the number of transitions (column **when**) in the corresponding TA. The average size of the generated strategy is around 5000 objects (states and transitions), while the average time required for computing it is in the order of magnitude of seconds for CT, WC and EB and of tens of seconds for DD. Moreover, it is worth noticing, that states and transitions are associated with conditions, each one representing a specific subset of the state space of the system for which a suitable action must be performed. In fact, every condition defining a state is a boolean formula on the locations of the TA and on the value of the integer values that are used in the model (e.g., `medBot.going_c_to_door3` and `PnurseBehaviour=5` in the "State" part of the previous excerpt); and every condition defining a transition is a boolean formula that includes several equalities, or inequalities, between clocks and constants or between pairs of clocks. (e.g., $1 <$ `CmedBot` and `Cnurse`$-$`Cstretcher` ≤ 1 in the "When" part of the previous excerpt). For these reasons, the magnitude of the estimations clearly evidence that a manual design of controllers for the considered class of scenarios is neither achievable nor convenient.

5.4. Scalability analysis

To provide a qualitative evaluation of the scalability of our approach, we analyzed how the performance of PuRSUE is affected by the number of agents in the application. To perform the analysis, we considered the baseline scenario (CT1) of the Catch the Thief case study. We considered an increasing number of thieves (from one to ten) and a single `policeBot`. The `policeBot` has to catch all the thieves by catching the thief n only after the thief $n-1$ has been caught. To analyze the performance of PuRSUE both when a controller can be generated, and when it can not, we considered two different cases: one in which the speed of the `policeBot` allows the robot to catch all the thieves (`Case A`), and one in which the speed of the `policeBot` is not enough to catch all the thieves (`Case B`).

The results show that the influence of the number of agents on the procedure that translates the PuRSUE-ML language into TGA is negligible. The average time required to translate the PuRSUE-ML language into TGA is always lower than one second (0.11 ms on average). Differently, the number of agents has a significant influence on the computation of the controller. Uppaal- TIGA was able to produce the controller for the instances of the Catch the Thief case study that contained up to two thieves, and it runs out of memory when more than three agents were considered. In particular, when two thieves are considered, the maximum time required to compute the controller for `Case A` is 51 seconds, and 1.8 seconds for `Case B`. As mentioned in the introduction, the experiments were conducted with the version 4.0.13 of Uppaal- TIGA compiled for 32-bit architectures, since a version compiled for 64-bit architectures is not available. For this reason, more complex scenarios can still be analyzed, but cannot be currently solved.

To analyze if the main cause of the low performance in the computation of the controller is a non-optimal intermediate TGA encoding, or the well-known state-explosion problem, we compared our results with the one reported by Cassez et al. [CDF+05]. The results reported by Cassez show that Uppaal- TIGA was able to process a production cell with a maximum of three plates (i.e., that represent the agents in the case study they considered). Even with specific configurations, enabling internal optimizations, the authors managed not more than six plates.

From our scalability analysis, and the qualitative comparison, we conclude that only the synthesis of controllers highly impacts on the performance of PuRSUE, whereas the influence of the translation of PuRSUE-ML models into TGA, and of the translation of the strategies into executable runtime controllers, is rather limited. Moreover, the translation that generates TGA does not restrict the class of applications that can be analyzed by means of PuRSUE. Indeed, all the considered scenarios includes two or three agents, and a number of logical entities in the environment, that are characterized with specific features that affect the overall evolution of the system (e.g., doors and windows that can be open or closed, or the kind of waste). Every scenario also satisfies specific constraints on the temporal ordering of events. For this reason, the scenarios that can be modeled with PuRSUE-ML are at least as complex as those captured by means of an ad-hoc modeling with TGA. In addition, PuRSUE allows designers to rapidly develop and prototype their robotic applications, as the same class of scenarios, that can be captured by a direct encoding with TGA, can be equivalently specified in PuRSUE-ML with a higher level of abstraction and in less time.

5.5. Experimental evaluation

The implementability and the effectiveness of the strategies generated by PuRSUE were tested by deploying them on a real robot and by checking the expected behavior in the real world. The two scenarios considered for the experiments are *EB2* and *EB3* (Sect. 5.1). The strategy has been implemented in a controller that was automatically generated by PuRSUE by transforming the strategy emitted by Uppaal- TIGA into executable code. The controller was executed on a Turtlebot robot [FWom].[6] The two scenarios were executed in the real world with humans and a Turtlebot acting together. Both were solved with a Reactive objective of the form "*if* `callBot` *then* `officeClean` *within* 40". In the real environment, a time unit was equal to 1 second.

EB2 Scenario We observed that the Turtlebot performed the expected actions according to the uncontrollable events triggered by the humans. In the considered execution, a human in the `office` triggers the event `callBot`, thus making the Turtlebot move towards the `office` through the `hallway`. Meanwhile, another human starts moving the bin from `base` to the `hallway`. Once the Turtlebot reaches the `office`, the robot first informs the human in the room to put the trash in it (event `takeTrash`), then notifies the system of the occurred cleaning (event `officeClean`). Simultaneously, the bin is moved towards `base` by another uncontrollable action. Hence, the controller informs the robot to move to `base`. After the arrival in `base`, the robot and the bin collaborate with event `throwTrash`.

[6]Available video at [vid]

EB3 scenario This scenario is similar to *EB2*, but differs from it in the possible kinds of garbage that can be either paper or plastic, and in the presence of two bins, `plasticBin` and `paperBin`, located in `base` and in `trashRoom`, respectively. Moreover, the bins are fixed and cannot be moved from their locations. As in *EB2*, the robot correctly accomplished the mission. In the experimented execution, the robot was called twice by the human to clean the office from two different waste kinds, first a paper one and then a plastic one.

6. Related work

Several surveys related to the verification and specification of robotic systems have been recently conducted by the research community. Luckcuck et al. [LFD+19] presented a comprehensive overview of the usage of formal methods for the specification and verification of autonomous robotic systems, while Nordmann et al. [NHW14] presented a survey on the domain-specific languages used in robotics. Farrell et al. [FLF18] summarized a set of challenges that should be addressed by current robotic applications, discussed how existing formal methods approaches can be used to address those challenges, and advocated the integration of several formal methods techniques to tackle the increasingly complex nature of robotic systems. Finally, Truszkowski et al. [TRHR] presented a survey of formal methods techniques that can be used in the verification of intelligent swarms of robots. We list several works that propose tools or theoretical frameworks aimed at the design of robotic applications without considering the robotic application as a multi-player game (Sect. 6.1), and works in which scenarios and their missions are explicitly modeled as multi-player games (Sect. 6.2).

6.1. Supporting the design of robotic applications

We discuss a number of formal methods tools and domain specific languages (DSL) that have been used to model and reason about robotic applications.

Tools to model robotic applications. Different formalisms have been proposed to model problems of interest through formal languages, including LTL (Linear Temporal Logic) and FSA (Finite State automata) (e.g., [MGPT18a, MGPT18b, TD16, TD14]). For example, Konur et al. [KDF10] use probabilistic finite state machines to model a swarm of foraging robots in order to verify their behavior. The possible behaviors of a single robot are described as probabilistic finite state automata, which are given as input to the model checker PRISM [KNP11] together with a probabilistic temporal property expressed as PCTL. Then, a counting abstraction approach is taken in order to model the behavior of the entire swarm.

Lopes et al. [LTL+16] use FSM as a baseline formalism to represent Supervisory Control Theory (SCT) for robotic applications. The controller is generated by considering synchronous composition between system descriptions and specification generators. This work is implemented using the open source software Nadzoru [PLLJ15].

Gainer et al. [GDD+17] proposed an automatic translation of the control rules of a Care-O-Bot system into the formal input of the model checker NuSMV [CCGR00]. These control rules are a set of preconditions and a sequence of actions. Such specifications are translated into LTL, which is in turn translated into SMV, the modeling language accepted by NuSMV. The extent to which the uncontrollable agents are modeled is a series of preconditions which can either be true of false.

Morse et al. [MAIL+16] model a non-deterministic environment using MDPs, where probabilistic transitions model the behavior of the uncontrollable agents in the environment. The tool enables the computation of the probability of the system meeting requirements in an unknown environment.

Vicentini et al. [VARM20] proposed a methodology to assess the physical safety of Human-Robot Collaboration (HRC) via a rich formal model of collaborative systems. The methodology features a model of both normative and erroneous human behavior [AMRV17, AMRV19] in order to detect the hazardous situations caused by human error, which are typically overlooked during the assessment. It allows engineers to incrementally refine the model of their system until all the safety violations are removed.

RoboChart [MRL+19, MRL+17] provides a graphical user-interface, i.e., a de facto restriction of UML, that allows designers to describe low-level components of a robotic application, the controller, and the expected behavior of the whole system. This description is used to automatically generate mathematical definitions that are employed to demonstrate some key properties of the robotic controllers by means of theorem provers. In contrast to our work, RobotChart is focused on software components and their interactions rather than on the creation of the domain model of the robotic application, where uncontrollable agents are explicitly modeled, and on the automatic computation of the controller that is presented in this work.

All of the aforementioned works differ from the one presented in this paper, as PuRSUE contains a DSL that allows end users to describe the robotic application with a human-readable high-level language, to automatically compute the TGA model of the controllable and uncontrollable agents in the environment, and, finally, to generate a run-time controller for the application.

DSL for robotics Providing DSLs that support developers in the creation of robotic applications is a problem that has been considered in several works. A survey on the topic has been conducted by Nordmann et al. [NHW14]. The authors identified a number of state-of-the-art DSLs for robotics, that were selected considering the Precision Placement Test (PPT) from the RoboCup@Work competition. The PPT exemplifies the complexity and capabilities that are common in robot applications, and it turned out to be a relevant metrics for the survey. In this work, the authors consider nine different sub-domains to classify the collected works: Robot Structure, Coordinate Representations and Transformations, Perception, Reasoning and Planning, Manipulation and Grasping, Coordination, Motion Control, Components and Architecture. Only two contributions can be found in Reasoning and Planning (discussed later in this section), whereas the majority of works focuses on Coordination.

Loetzsch et al. [LRJ06] presented a DSL for programming the behavior of autonomous agents. This work translates a system defined at a high level by the user into a finite state machine. The proposed DSL allows the designer to model the control strategy and the reaction of the robots as a response to certain state changes. Conversely, our work allows the designer to describe the robotic application, its objective and the uncontrollable agents, while the controller is automatically computed.

Götz et al. [GLR$^+$11], proposed a DSL to program Nao robots [nao] called NaoText. This work does not include an explicit representation of the opponents in order to generate a controller.

Kunze et al. [KRB11], presented SRDL, a DSL that allows designers to provide a semantics for describing robot components, capabilities and the actions. The aim of this work is to provide the autonomous agents with knowledge of their capabilities of performing different tasks. This is done by describing actions required by the user as a set of sub-actions, each of which is tied to the capability of each robot to perform such action, which ultimately depends on the components that the agent is provided with. While this work is similar to our work in the ambition of modeling the high-level actions of a robot, it does not provide an explicit representation of the environment, nor does it support the generation of controllers.

Finucane et al. [FJKG10] presented a DSL for writing the specifications of the robot using structured English. The desired behavior of the robot is specified through a set of assumptions on the environment, and the desired behavior of the agent is expressed as conditions between events. Explicit descriptions of other agent's behavior, while conceptually achievable through environmental assumptions, is not an explicit concept as it is in this work. Finally, they do not handle teams of robots and explicit time.

Rugg-Gunn and Cameron [RGC94] proposed a formalism that allows operators to explicitly describe the topology of an environment and the tasks that need to be performed in different locations, expressed as services to be delivered in certain locations and time frame. Multi-robot scheduling is also included, allowing the designer to schedule the movements of a set of robots having to deliver a set of services without interfering with each other. However, this work does not allow for the representation of complex tasks, and uncontrollable agents are not considered.

García et al. [GPM$^+$19] presented PROMISE, a DSL that enables domain experts to specify missions at a high level of abstraction for teams of autonomous robots in a user-friendly way. As PROMISE makes use of the mission specification patterns presented in [MTP$^+$19], it turned out to be very useful for specifying missions that robots should accomplish. However, PROMISE does not include the specification of the environment, while PuRSUE-ML does.

Bozhinoski et al. [BRM$^+$15] and Ruscio et al. [RMPT16] proposed FLYAQ, a platform and DSLs to enable non-technical operators to define and execute missions for a team of multicopters, that also supports generic robots [CRMP16]. These works present a user-friendly language, that allows the operator to specify precise waypoints that the robots need to cover. In this way, it might be easier for the mission specifier to declaratively model the mission, i.e. to state the goal of the mission instead of the steps needed to be performed in order to achieve the goal. Finally, the modeling of the environment only concerns no-fly-zones, obstacles, and places where to land in the case of problems during the mission execution.

Campusano and Fabry [CF17] introduced Live Robot Programming (LRP), a language for nested state machines, which enables the design and programming of robot behaviors. LRP enables the user to reduce the cognitive distance between the code and the resulting behavior. LRP does not support the specification of the environment in which the robots will operate.

Finally, SPECTRA [MR19], a specification language for reactive systems, allows considering GR(1) discrete LTL specifications, but does not support reasoning with explicit and continuous time specifications, such as the one considered in this paper.

6.2. Multi-player support for modeling and reasoning on robotic applications

The idea of generating control strategies by modeling the robotic application control problem as a multi-player game has been considered in many works. Various solutions have been proposed by different communities to tackle the problem of controller synthesis. The specification of objectives is commonly realized through logical languages, such as TCTL adopted in this work, whereas the synthesis algorithms are designed depending on the formalisms adopted to model the systems. There are three families of models for which synthesis algorithms are used to synthesize controllers. Automata-based synthesis algorithms are defined for systems that are abstracted as a Kripke structure, whose objectives are commonly specified with LTL formulae. Markov Decision Processes, and the probabilistic version of CTL, are adopted when the designer aims to maximize the probability of satisfying the goal. Game-based approaches are used when the goal of the synthesis problem amounts to define a finite state controller that encodes the winning strategy. Furthermore, specific approaches to the synthesis of controllers, that are realized in a compositional manner, have been proposed in [ZA18] and [ZPMT11]. A survey on controller synthesis for robotics can be found in [KGLR18].

In our work, we use UPPAAL- TIGA to synthesize controllers, as it is the de-facto standard tool for the controller synthesis when explicit time concerns are present, and when the model is realized by using an automata-based formalism. However, other tools, such as Synthia [PEM11] and PRISM-games [CFK$^+$13], can be used as alternatives to UPPAAL- TIGA.

In this section, we consider a number of works that adopted automata-based synthesis algorithms, and specifically those that used TGA to model the system. We discuss papers that consider a specific version of the robotic control problem, and manually encode those problems using TGAs, and works that proposed specific translations from already existing formalisms to TGAs.

TGAs to encode specific scenarios Damas and Lima [DL04] used TGAs to model a multi-agent game. This work shows the potential of modeling a strategic problem in an uncontrollable environment as a timed game. The controller for the specific mission is obtained via dynamic programming rather than using the UPPAAL- TIGA support.

Jessen et al. [JRLD07] and Cassez et al. [CLRR11] show how to translate control problems into TGAs, which are then given as input to UPPAAL- TIGA to automatically synthesize a controller. The first presents the solution of a temperature control problem, while the second of an embedded system composed of an oil pump and an accumulator. Both focus on low-level models of actuators and sensors, rather than providing a high-level language for modeling robotic applications.

Largouët et al. [LKZ16] modeled a control problem using Priced Timed Game Automata (PTGA) with the purpose of automatically synthesizing a control strategy using model-checking. The model is extended to include many agents and uncontrollable agents, including explicit temporal constraints as well, but does not provide a DSL for modeling robotic applications.

The aforementioned works focus on how TGA allows for a convenient and automatic synthesis of controllers in different scenarios and scopes. However, no effort has been presented in the direction of automatic generation of the models themselves; all of the models, instead, were designed ad-hoc by the authors.

Translation of existing formalisms into TGA Cimatti et al. [CHMR14] used TGA to encode Simple Temporal Networks with Uncertainty (STNU). STNU is a data structure for representing and reasoning about temporal knowledge in domains where some time-points are controlled by the executor (or agent) while others are controlled by the environment. The authors first expand the formulation of STNU to more explicitly define the scenario as a 2-player game between the controllable agents and the environment. Then, they offer an encoding from the presented formalism into TGAs, and prove its correctness. While this work offers an automatic generation of a

TGA, starting from a higher level description of the problem, it turns out to be useful for solving a problem that is already encoded into the STNU formalism, and it does not offer a framework to solve a problem coming from a real-life scenario.

Mayer and Orlandini [MO15] used TGAs to encode the formalism of Timeline-based Planning. Timeline-based Planning is a paradigm in which the system is expressed in terms of a set of states and transitions between the states, and the objectives are expressed as state values over temporal intervals. A limitation of Timeline-based Planning is that the paradigm does not provide an explicit definition of action, and it differs from the classical action-based planning, because it is declarative, and events corresponding to agent actions cannot be characterized with a specific duration. Moreover, other forms of uncertainty, such as non-determinism (i.e., which tasks the environment chooses to perform), are not supported. Ceballos et al. [CBC+11] used this Timeline-based Planning for solving the planning of a rover that performs an exploration task, whereas Orlandini et al. [OFC+11] exploited this framework for domain, planning and scheduling validation, by encoding the resulting plan and the state variables into TGAs. The framework has been further extended in Cesta et al. [COU13], where a general purpose library is proposed, and Gigante et al. [GMM+18] considered game-like scenarios, in which the evolution of state variables is triggered by a user, or an opponent, in a turn-based setting.

This set of works is strongly related to the one presented in this paper, as they translate existing formalisms into TGA. However, none of them propose a general purpose language for modeling robotic-applications that can be used by non-experts.

7. Conclusions

We presented PuRSUE, an approach to help developers in the rigorous and systematic design of runtime control strategies for robotic applications. The approach is composed of a high-level language, PuRSUE-ML, for modeling robotic applications, and of a module that enables the automatic synthesis of low-level controllers enabling the robots to achieve their missions. The computation of the low-level controllers makes use of a formal model based on TGA, which is automatically generated from the PuRSUE-ML.

We evaluated PuRSUE considering the following aspects:

(i) The capability of PuRSUE in helping designers in the modeling of robotic applications by describing several robotic applications using PuRSUE-ML. We were capable of modeling them, as well as reason on variations thereof through simple changes of the PuRSUE-ML model.

(ii) The capability of PuRSUE to support designers in the generation of controllers and to reason about real-time properties for robotic applications. We generated controllers using PuRSUE for the scenarios considered. This has shown that PuRSUE is capable of generating controllers for applications described with PuRSUE-ML and its capability to support dealing with temporal properties.

(iii) The scalability of PuRSUE. We analyzed how the number of agents that are deployed in the robotic application affect the performances of PuRSUE. Our results show that the synthesis of the controller has the highest impact on the scalability of PuRSUE.

(iv) The effectiveness of the generated controllers on real applications. We deployed the controllers of two robotic applications on the Turtlebot robot. The robot behaved as expected and achieved the mission in both scenarios, showing that the controllers generated by PuRSUE are effective.

As future work, we plan to identify and develop techniques that allow improving the scalability of PuRSUE. We plan to analyze how other existing tools for controller synthesis, such as Synthia [PEM11] and PRISM-games [CFK+13], and alternative encodings for translating PuRSUE-ML specifications into TA, affect the scalability of PuRSUE. We also plan to improve the expressiveness of the language in several directions; e.g., by allowing users to define the value of state variables using LTL formulae, and its metric extension called CLT-Loc [BRSP16], or different forms of collaboration. Moreover, we aim to to apply the approach to more complex scenarios and with other robots in collaboration with our industrial partners in the Co4Robots project (BOSCH (www.bosch-ai.com) and PAL robotics (www.pal-robotics.com)). We will also investigate the possibility to integrate our previous work on mission patterns for the specification of robot missions [MTBP19, MTB+18, MTP+19]. Those patterns do not natively contain information about the environment, and have been collected and designed with the "mission specification" problem in mind, i.e., allowing the user to specify what is the final goal the robotic application should achieve. For this reason, we envision the use of those patterns in the definition of the objectives of the robotic application (Sect. 4.1.6). We will also investigate the integration of PuRSUE-ML with DSLs that have been proposed for specifying missions for robots, such as PROMISE [GPM+19], which make use of the mission specification patterns.

Acknowledgements

Open access funding provided by University of Gothenburg. The authors acknowledge financial support from the Centre of EXcellence on Connected, Geo-Localized and Cybersecure Vehicle (EX-Emerge), funded by Italian Government under CIPE resolution n. 70/2017 (Aug. 7, 2017). The work is also supported by the European Research Council under the European Union's Horizon 2020 research and innovation programme GA No. 694277 and GA No. 731869 (Co4Robots).

8. Appendix A: PuRSUE-ML specification of DD.

```
//locations
poi "medicine"
poi "room"
poi "door1"
poi "door2"
poi "door3"
poi "a"
poi "b"
poi "c"
poi "d"

//connections
connect a and door1 distance 6
connect door1 and b distance 13
connect b and door2 distance 8
connect door2 and c distance 20
connect c and door3 distance 7
connect door3 and d distance 12
connect d and medicine distance 14
connect medicine and a distance 14

//events
event "giveMedicine1" location door1 duration 3
event "giveMedicine2" location door2 duration 3
event "giveMedicine3" location door3 duration 3
event "takeMedicine" location medicine duration 2
event "confirmDelivery"
event "bumpInto" collaborative
event "openDoor1" location room duration 2
event "openDoor2" location room duration 2
event "openDoor3" location room duration 2
event "crossRoom" location room duration 400
event "closeDoor1" location door1 duration 2
event "closeDoor2" location door2 duration 2
event "closeDoor3" location door3 duration 2
```

```
//rules
rule "nurseBehaviour" : ((openDoor1 or openDoor2) or openDoor3) before crossRoom
rule "robotTask" : takeMedicine before ( ( (giveMedicine1 or giveMedicine2) or giveMedicine3 ) before
     confirmDelivery)

//states and dependencies
state "door1open": initially false, true_if openDoor1 false_if closeDoor1
state "door2open": initially false, true_if openDoor2 false_if closeDoor2
state "door3open": initially false, true_if openDoor3 false_if closeDoor3
stateDependency: giveMedicine1 only_if door1open is_true
stateDependency: giveMedicine2 only_if door2open is_true
stateDependency: giveMedicine3 only_if door3open is_true
stateDependency: openDoor1 only_if door1open is_false
stateDependency: openDoor2 only_if door2open is_false
stateDependency: openDoor3 only_if door3open is_false
stateDependency: closeDoor1 only_if door1open is_true
stateDependency: closeDoor2 only_if door3open is_true
stateDependency: closeDoor3 only_if door3open is_true

//agents
agent "medBot" controllable mobile 1 location a can_do giveMedicine1, giveMedicine2, giveMedicine3,
     takeMedicine, confirmDelivery reacts_to bumpInto
agent "nurse" controllable mobile 1 location room can_do openDoor1, openDoor2, openDoor3, crossRoom reacts_to
     giveMedicine1, giveMedicine2, giveMedicine3
agent "stretcher" mobile 1 location c can_do closeDoor1, closeDoor2, closeDoor3, bumpInto

//objectives
objective: avoid bumpInto
reach_objective: do confirmDelivery after 0
```

References

[AD93] Alur R, Dill DL (1993) Model-checking in dense real-time. Inf Comput 104(1):2–34

[AD94] Alur R, Dill DL (1994) A theory of timed automata. Theor Comput Sci 126(2):183–235

[AMP95] Asarin E, Maler O, Pnueli A (1995) Symbolic controller synthesis for discrete and timed systems. In: Hybrid Systems II. Springer

[AMRV17] Askarpour M, Mandrioli D, Rossi M, Vicentini F (2017) Modeling operator behavior in the safety analysis of collaborative robotic applications. In: Computer safety, reliability, and security. Springer, pp 89–104

[AMRV19] Askarpour M, Mandrioli D, Rossi M, Vicentini F (2019) Formal model of human erroneous behavior for safety analysis in collaborative robotics. Robot Comput Integr Manuf 57:465–476

[BCD+07] Behrmann G, Cougnard A, David A, Fleury E, Larsen KG, Lime D (2007) Uppaal-tiga: time for playing games!. In: Computer aided verification. Springer, pp 121–125

[BDJT19] Barbosa FS, Duberg D, Jensfelt P, Tůmová J (2019) Guiding autonomous exploration with signal temporal logic. IEEE Robot Autom Lett 4(4):3332–3339

[BDL04] Behrmann G, David A, Larsen KG (2004) A tutorial on uppaal. In: Formal methods for the design of real-time systems. Springer, pp 200–236

[Bou09] Bouyer P (2009) Model-checking timed temporal logics. In: Electronic notes in theoretical computer science. pp 231:323–341. Workshop on Methods for Modalities (M4M5 2007)

[BREC07] Barrett A, Rabideau G, Estlin T, Chien S (2007) Coordinated continual planning methods for cooperating rovers. IEEE Aerosp Electron Syst Mag 22(2):27–33

[BRM+15] Bozhinoski D, Ruscio DD, Malavolta I, Pelliccione P, Tivoli M (2015) FLYAQ: enabling non-expert users to specify and generate missions of autonomous multicopters. In: International conference on automated software engineering. IEEE, pp 801–806

[BRSP16] Bersani MM, Rossi M, Pietro PS (2016) A tool for deciding the satisfiability of continuous-time metric temporal logic. Acta Inf 53(2):171–206

[BS86] Berry G, Sethi R (1986) From regular expressions to deterministic automata. Theor Comput Sci 48:117–126

[CBC+11] Ceballos A, Bensalem S, Cesta A, de Silva L, Fratini S, Ingrand F, Ocón J, Orlandini A, Py F, Rajan K, Rasconi R, van Winnendael M (2011) A goal oriented autonomous controller for space exploration. http://robotics.estec.esa.int/ASTRA/Astra2011/Astra2011_Proceedings.zip1

[CCGR00] Cimatti A, Clarke E, Giunchiglia F, Roveri M (2000) NuSMV: a new symbolic model checker. Int J Softw Tools Technol Transf 2(4):410–425

[CDF+05] Cassez F, David A, Fleury E, Larsen KG, Lime D (2005) Efficient on-the-fly algorithms for the analysis of timed games. In: Concurrency theory. Springer, pp 66–80

[CF17] Campusano M, Fabry J (2017) Live robot programming: The language, its implementation, and robot API independence. Sci Comput Program 133:1–19

[CFK+13] Chen T, Forejt V, Kwiatkowska M, Parker D, Simaitis A (2013) Prism-games: a model checker for stochastic multi-player games. In: International conference on TOOLS and algorithms for the construction and analysis of systems. Springer, pp 185–191

[CHMR14] Cimatti A, Hunsberger L, Micheli A, Roveri M (2014) Using timed game automata to synthesize execution strategies for simple temporal networks with uncertainty. In: AAAI. AAAI Press, pp 2242–2249

[CLRR11] Cassez F, Larsen KG, Raskin JF, Reynier PA (2011) Timed controller synthesis: an industrial case study. *Deliverable no.: D5. 12 Title of Deliverable: Industrial Handbook*, p 150

[COU13] Cesta A, Orlandini A, Umbrico A (2013) Toward a general purpose software environment for timeline-based planning. https://core.ac.uk/download/pdf/37835325.pdf

[CRMP16] Ciccozzi F, Di RD, Malavolta I, Pelliccione P (2016) Adopting MDE for specifying and executing civilian missions of mobile multi-robot systems. IEEE Access 4:6451–6466

[DL04] Damas B, Lima P (2004) Stochastic discrete event model of a multi-robot team playing an adversarial game. IFAC Proc Vol 37(8):974–979

[FIN04] Farinelli A, Iocchi L, Nardi D (2004) Multirobot systems: a classification focused on coordination. Trans Syst Man Cybern Part B (Cybern) 34(5):2015–2028

[FJKG10] Finucane C, Jing G, Kress-Gazit H (2010) LTLMoP: Experimenting with language, temporal logic and robot control. In: Intelligent Robots and Systems. IEEE, pp 1988–1993

[FLF18] Farrell M, Luckcuck M, Fisher M (2018) Robotics and integrated formal methods: necessity meets opportunity. In: Integrated formal methods. Springer, pp 161–171

[FWom] Foote T, Wise M (2019) Turtlebot home page, Last access 27 March https://www.turtlebot.com/

[GDD+17] Gainer P, Dixon C, Dautenhahn K, Fisher M, Hustadt U, Saunders J, Webster M (2017) Cruton: automatic verification of a robotic assistant's behaviours. In: Critical systems: formal methods and automated verification. Springer, pp 119–133

[GLR+11] Götz S, Leuthäuser M, Reimann J, Schroeter J, Wende C, Wilke C, Aßmann U (2011) A role-based language for collaborative robot applications. In: International symposium on leveraging applications of formal methods, verification and validation. Springer, pp 1–15

[GMM+18] Gigante N, Montanari A, Mayer MC, Orlandini A, Reynolds M (2018) A game-theoretic approach to timeline-based planning with uncertainty. arXiv preprint arXiv:1807.04837

[GPM+19] García S, Pelliccione P, Menghi C, Berger T, Bures T (2019) High-level mission specification for multiple robots. In: International conference on software language engineering. ACM, pp 127–140

[JRLD07] Jessen JJ, Rasmussen JI, Larsen KG, David A (2007) Guided controller synthesis for climate controller using uppaal tiga. In: Formal modeling and analysis of timed systems. Springer, pp 227–240

[KDF10] Konur S, Dixon C, Fisher M (2010) Formal verification of probabilistic swarm behaviours. In: International conference on swarm intelligence. Springer, pp 440–447

[KGLR18] Kress-Gazit H, Lahijanian M, Raman V (2018) Synthesis for robots: guarantees and feedback for robot behavior. Ann Rev Control Robot Auton Syst 1(1):211–236

[KNP11] Kwiatkowska M, Norman G, Parker D (2011) Prism 4.0: verification of probabilistic real-time systems. In: Computer aided verification. Springer, pp 585–591

[KRB11] Kunze L, Roehm T, Beetz M (2011) Towards semantic robot description languages. In: International conference on robotics and automation. IEEE, pp 5589–5595

[LFD+19] Luckcuck M, Farrell M, Dennis LA, Dixon C, Fisher M (2019) Formal specification and verification of autonomous robotic systems: a survey. ACM Comput Surv 52(5):100:1–100:41

[LKZ16] Largouët C, Krichen O, Zhao Y (2016) Temporal planning with extended timed automata. In: International conference on tools with artificial intelligence. IEEE, pp 522–529

[LRF+15] Lignos C, Raman V, Finucane C, Marcus M, Kress-Gazit H (2015) Provably correct reactive control from natural language. Auton Robots 38(1):89–105

[LRJ06] Loetzsch M, Risler M, Jüngel M (2006) XABSL-A pragmatic approach to behavior engineering. In: International Conference on Intelligent Robots and Systems. IEEE/RSG, pp 5124–5129

[LTL+16] Lopes YK, Trenkwalder SM, Leal AB, Dodd TJ, Groß R (2016) Supervisory control theory applied to swarm robotics. Swarm Intell 10(1):65–97

[MAIL+16] Morse J, Araiza-Illan D, Lawry J, Richards A, Eder K (2016) Formal specification and analysis of autonomous systems under partial compliance. arXiv preprint arXiv:1603.01082

[MGPT18a] Menghi C, García S, Pelliccione P, Tůmová J (2018) Multi-robot LTL planning under uncertainty. In: International symposium on formal methods. Springer, pp 399–417

[MGPT18b] Menghi C, García S, Pelliccione P, Tůmová J (2018) Towards multi-robot applications planning under uncertainty. In: International conference on software engineering: companion proceeedings. ACM, pp 438–439

[MO15] Mayer MC, Orlandini A (2015) An executable semantics of flexible plans in terms of timed game automata. In: Temporal Representation and Reasoning. IEEE, pp 160–169

[MR19] Maoz S, Ringert JO (2019) Spectra. http://smlab.cs.tau.ac.il/syntech/spectra/userguide.pdf

[MRL+17] Miyazawa A, Ribeiro P, Li W, Cavalcanti A, Timmis J (2017) Automatic property checking of robotic applications. In: International conference on intelligent robots and systems. IEEE/RJS, pp 3869–3876

[MRL+19] Miyazawa A, Ribeiro P, Li W, Cavalcanti A, Timmis J, Woodcock J (2019) RoboChart: modelling and verification of the functional behaviour of robotic applications. Softw Syst Model

[MTB+18] Menghi C, Tsigkanos C, Berger T, Pelliccione P, Ghezzi C (2018) Poster: property specification patterns for robotic missions. In: International conference on software engineering: companion proceeedings. IEEE/ACM, pp 434–435

[MTBP19] Menghi C, Tsigkanos C, Berger T, Pelliccione P (2019) Psalm: specification of dependable robotic missions. In: International conference on software engineering: companion proceedings. IEEE/ACM, pp 99–102.

[MTP+19] Menghi C, Tsigkanos C, Pelliccione P, Ghezzi C, Berger T (2019) Specification patterns for robotic missions. IEEE Trans Softw Eng, pp 1–1

[nao] https://www.ald.softbankrobotics.com/en/robots/nao/find-out-more-about-nao

[NHW14] Nordmann A, Hochgeschwender N, Wrede S (2014) A survey on domain-specific languages in robotics. In: International Conference on Simulation, Modeling, and Programming for Autonomous Robots. Springer, pp 195–206
[OFC+11] Orlandini A, Finzi A, Cesta A, Fratini S, Tronci E (2011) Enriching apsi with validation capabilities: The keen environment and its use in robotics. In: Advanced Space Technologies in Robotics and Automation. http://robotics.estec.esa.int/ASTRA/Astra2011/Astra2011_Proceedings.zip1
[Onl19] Online appendix, (2019) https://drive.google.com/drive/folders/117-MVWDYYo-6Gir6KAzlVTsvshGu2Uvf
[PEM11] Peter H, Ehlers R, Mattmüller R (2011) Synthia: Verification and synthesis for timed automata. In: International conference on computer aided verification. Springer, pp 649–655
[PLLJ15] Pinheiro LP, Lopes YK, Leal AB, Junior RS (2015) Nadzoru: a software tool for supervisory control of discrete event systems. IFAC-PapersOnLine 48(7):182 – 187. International Workshop on Dependable Control of Discrete Systems
[pur19] PuRSUE - Planner for RobotS in Uncontrollable Environments, (2019). https://github.com/deib-polimi/PuRSUE
[QCG+09] Quigley M, Conley K, Gerkey B, Faust J, Foote T, Leibs J, Wheeler R, Ng AY (2009) ROS: an open-source robot operating system. In: ICRA workshop on open source software, vol 3. IEEE, p 5
[QLFAI18] Quattrini Li A, Fioratto R, Amigoni F, Isler V (2018) A search-based approach to solve pursuit-evasion games with limited visibility in polygonal environments. In: International conference on autonomous agents and multiAgent systems. ACM, pp 1693–1701
[RGC94] Rugg-Gunn N, Cameron S (1994) A formal semantics for multiple vehicle task and motion planning. In: International conference on robotics and automation. IEEE, pp 2464–2469
[RMPT16] Ruscio DD, Malavolta I, Pelliccione P, Tivoli M (2016) Automatic generation of detailed flight plans from high-level mission descriptions. In: International conference on model driven engineering languages and systems. ACM, pp 45–55
[SK16] Siciliano B, Khatib O (2016) Springer handbook of robotics, 2nd edn. Springer, Berlin
[TD14] Tůmová J, Dimarogonas DV (2014) A receding horizon approach to multi-agent planning from local LTL specifications. In: 2014 American Control Conference. IEEE, pp 1775–1780
[TD16] Tůmová J, Dimarogonas DV (2016) Multi-agent planning under local LTL specifications and event-based synchronization. Automatica 70:239–248
[TRHR] Truszkowski W, Rash J, Hinchey M, Rouff C. A survey of formal methods for intelligent swarms. https://ntrs.nasa.gov/archive/nasa/casi.ntrs.nasa.gov/20050156631.pdf
[TSF+16] Tunggal TP, Supriyanto A, Faishal I, Pambudi I, et al. (2016) Pursuit algorithm for robot trash can based on fuzzy-cell decomposition. Int J Electr Comput Eng, 6(6):2863
[VARM20] Vicentini F, Askarpour M, Rossi M, Mandrioli D (2020) Safety assessment of collaborative robotics through automated formal verification. IEEE Trans Robot 31(1):42–61
[vid] Ecobot. https://youtu.be/w6SfCmgdCsk, https://youtu.be/XmOY-urEDD0
[VVB+19] Verginis CK, Vrohidis C, Bechlioulis CP, Kyriakopoulos KJ, Dimarogonas DV (2019) Reconfigurable motion planning and control in obstacle cluttered environments under timed temporal tasks. In: International Conference on Robotics and Automation. IEEE, pp 951–957
[WRs] *International Federation of Robotics*. https://ifr.org/worldrobotics/
[ZA18] Zamani M, Arcak M (2018) Compositional abstraction for networks of control systems: a dissipativity approach. IEEE Trans Control Netw Syst 5(3):1003–1015
[ZPMT11] Zamani M, Pola G, Mazo M, Tabuada P (2011) Symbolic models for nonlinear control systems without stability assumptions. Trans Autom Control 57(7):1804–1809

Received 1 April 2019
Accepted in revised form 28 February 2020 by Ana Cavalcanti and Pedro Ribeiro
Published online 23 March 2020