

**POLITECNICO DI MILANO**  
Scuola di Ingegneria Industriale e dell'Informazione  
Master's degree in Automation and Control Engineering  
Dipartimento di Elettronica, Informazione e Bioingegneria



**USER-FRIENDLY CONTROLLER  
SYNTHESIS FOR MULTI-AGENT  
ROBOTIC APPLICATIONS IN  
PRESENCE OF AGENTS WITH  
UNKNOWN BEHAVIOR**

**Supervisor: Prof. Matteo Rossi**  
**Assistant Supervisor: Dr. Marcello M. Bersani**  
**Dr. Claudio Menghi**  
**Prof. Patrizio Pelliccione**

**Master's thesis of:**  
**Matteo Soldo, Matr. 864397**

**Academic Year 2017-2018**

*Alla mia famiglia e i miei amici*

# Contents

<b>Sommario</b>	<b>3</b>
<b>Acknowledgements</b>	<b>5</b>
<b>1 Introduction</b>	<b>6</b>
1.1 The Drug Delivery Example (DD)	10
1.2 Research Challenges	12
1.3 Contributions of the Thesis	15
1.4 Structure of the Thesis	19
<b>2 Related Work</b>	<b>21</b>
2.1 Classification of related work	21
2.2 Support to Designers of Robotic Applications	23
2.2.1 Formal Methods Formalisms as Tool to Model Robotic Applications	23
2.2.2 DSL for Robotics	24
2.3 Game Models	26
2.3.1 TGAs to encode specific scenarios	26
2.3.2 TGAs to encode existing formalisms	27
<b>3 Background</b>	<b>29</b>
3.1 Timed Game Automata	29
3.2 Timed Computation Tree Logic	30
3.3 UPPAL-TiGA	31
3.4 Implementation Tools	34
3.4.1 Xtext	34
3.4.2 Robot Operating System (ROS)	35
3.4.3 TurtleBot	35

<b>4</b>	<b>Contribution</b>	<b>37</b>
4.1	The PuRSUE Framework . . . . .	37
4.2	The PuRSUE Modeling Language . . . . .	39
4.2.1	Environment . . . . .	41
4.2.2	Events . . . . .	42
4.2.3	Rules . . . . .	43
4.2.4	States and State Dependencies . . . . .	46
4.2.5	Agents . . . . .	46
4.2.6	Objectives . . . . .	48
4.3	Intermediate TGA . . . . .	49
4.3.1	Variables declaration . . . . .	51
4.3.2	State Automaton . . . . .	55
4.3.3	Rules Automata . . . . .	56
4.3.4	Agent Automata . . . . .	58
4.3.5	Objective Automata . . . . .	67
4.4	DSL to TGA translation . . . . .	70
4.4.1	Abstract Representation of the PuRSUE-ML Model . . . . .	70
4.4.2	Transformation between the PuRSUE-ML Model and TGA . . . . .	71
4.5	Controller Generation . . . . .	76
<b>5</b>	<b>Implementation</b>	<b>78</b>
5.1	Design-Time Support . . . . .	78
5.2	Run-Time Support . . . . .	80
<b>6</b>	<b>Evaluation</b>	<b>85</b>
6.1	Scenarios . . . . .	86
6.1.1	Catch the Thief (CT) . . . . .	86
6.1.2	Work Cell (WC) . . . . .	88
6.1.3	EcoBot (EB) . . . . .	89
6.2	Modeling support (RQ1) . . . . .	90
6.2.1	Effectiveness in Encoding the Considered Scenarios . . . . .	90
6.2.2	Assessing the Design Effort Saved by the Usage of PuRSUE-ML . . . . .	94
6.3	Automatic Controller Generation (RQ2) . . . . .	96
6.4	Experimental Evaluation (RQ3) . . . . .	98
6.4.1	EB2 Scenario . . . . .	99
6.4.2	EB3 Scenario . . . . .	99

**7 Conclusions and future work** **101**  
7.1 Scenarios from Evaluation modeled in PuRSUE-ML . . . . . 105

# List of Figures

1.1	A graphical representation of the Drug Delivery example . . .	12
3.1	Example of a TGA . . . . .	30
3.2	screenshot of the UPPAAL-TiGA graphical interface . . . . .	33
3.3	the TurtleBot . . . . .	36
4.1	The PuRSUE Framework. . . . .	38
4.2	the Drug Delivery example. . . . .	39
4.3	State automaton with two transitions updating the value of a state $s$ . . . . .	55
4.4	Example of a state automaton for the Drug Delivery case . . .	56
4.5	Automaton representing the statement “ $e_1$ before $e_2$ ” . . . . .	57
4.6	Automaton representing the statement “ $e_1$ or $e_2$ ” . . . . .	57
4.7	Automaton representing the rule <code>robotTask</code> . . . . .	58
4.8	POI location with one incoming and one outgoing transition	59
4.9	example of a POI location of agent <code>turtleBot</code> . . . . .	60
4.10	a couple of movement locations associated to a single connection POIs . . . . .	61
4.11	A durable event location, modeling agent $a$ performing $e$ in POI $x$ . . . . .	63
4.12	example of a durable event location for agent <code>turtleBot</code> . . .	64
4.15	Automaton representing the Reaction mission. . . . .	69
4.16	Automaton representing the Avoidance mission. . . . .	69
4.17	Automaton representing the Execution mission. . . . .	69
4.18	Execution Mission automaton in the Drug Delivery case . . .	70
5.1	PuRSUE: design time workflow . . . . .	79
5.2	ScreenShot of the Eclipse Plug-In . . . . .	79
5.3	Run-time architecture . . . . .	81
5.4	modules of runtime controller . . . . .	84

5.5	run-time communication diagram . . . . .	84
6.1	Graphical representation of the environments used in the different scenarios. . . . .	87
6.2	A graphical representation of the EB example . . . . .	89

# List of Tables

2.1	classification of related work . . . . .	22
4.1	POIs for the Drug Delivery case . . . . .	41
4.2	Events Relevant to the Drug Delivery case . . . . .	44
4.3	Available Objectives . . . . .	48
6.1	Variations of the Catch the Thief application. . . . .	86
6.2	Variations of the Work Cell application. . . . .	89
6.3	Variations of the EcoBot application. . . . .	90
6.4	comparison of scenarios to answer <b>RQ1</b> . . . . .	95
6.5	comparison of scenarios to answer <b>RQ2</b> , time reported in milliseconds . . . . .	97



# Abstract

Developing robotic applications is a complex task that requires skills that are usually possessed by highly qualified robotic developers. We believe that, while formal methods techniques that support developers in the creation and design of robotic applications exist, they must be explicitly customized to be impactful in the robotic domain and to effectively support the growth of the robotic market. Specifically, the robotic market is asking for techniques that: (i) support a systematic and rigorous design of robotic applications through high-level languages; and (ii) enable the automatic synthesis of low-level controllers that allow robots to achieve their missions. To address those problems we present the PuRSUE (Planner for RobotS in Uncontrollable Environments) approach, which aims at supporting developers in the rigorous and systematic design of high-level run-time control strategies for robotic applications. The approach introduces PuRSUE-ML, a high-level language that allows modeling the environment, the agents deployed therein, and their missions. PuRSUE is able to automatically check whether a controller that allows robots to achieve their missions exist and synthesize it. We evaluated if PuRSUE offers a more compact way than Timed Game Automata (TGA) for modeling robotic applications, the effectiveness of its automatic computation of controllers, as well as whether it helped designers in reasoning on real-time properties of the scenarios, and how the approach supports the deployment of controllers on actual robots. To answer those questions we considered 13 scenarios coming from 3 different robotic applications presented in the literature. The results show that: (1) PuRSUE-ML does offer designers a more compact way for formal modeling of robotic applications compared to a direct encoding of the latter in low-level modeling formalisms; (2) PuRSUE effectively supports designers in the generation of controllers, compared to their manual development and supports them in reasoning on their temporal properties; and (3) the plans generated with PuRSUE are indeed effective when deployed on actual robots.

# Sommario

Negli ultimi anni le applicazioni robotiche sono diventate parte integrante delle nostre vite. Piattaforme mobili e manipolatori vengono sempre più utilizzati nella nostra società per eseguire azioni ripetitive, alienanti o pericolose per un essere umano. I robot, fianco a fianco con i lavoratori, sono utilizzati in ogni settore industriale, dalla meccanica, chimica al settore manifatturiero, per il trasporto di materiali e persone. Grazie alla loro flessibilità, personalizzabilità e vasta gamma di scelta nel mercato globale, questo genere di prodotto è in grado di soddisfare richieste e bisogni di molte tipologie. Per questo motivo le aziende che si affidano ad applicazioni robotiche necessitano di strumenti di programmazione sempre aggiornati in modo da rendere i loro prodotti sempre più disponibili per necessità di tipologie differenti.

Gli sviluppatori di software hanno il compito di garantire strumenti flessibili e riutilizzabili in modo da soddisfare ogni necessità del consumatore. Il progetto Co4robots [44] all'interno del quale questa tesi è stata sviluppata, ha l'obiettivo di fornire strumenti che agevolano la programmazione nel contesto di applicazioni robotiche. Co4robots mira ad introdurre nel campo della robotica principi e tecniche caratteristiche dell'ingegneria del software per fornire un approccio ingegneristico alla programmazione di robot. Il progetto analizza principalmente applicazioni multi-robot nelle quali un team di robot collabora al fine di raggiungere il soddisfacimento di una data missione, che risulterebbe non eseguibile da un singolo robot. Per esempio, una missione potrebbe richiedere al team di robot di caricare e muovere oggetti o rifornimenti in un determinato ambiente p.e. uffici, alberghi od ospedali. L'obiettivo finale del team di robot è descritto e assegnato come una missione definita da un linguaggio di alto livello.

Co4robots progetta di elaborare un approccio sistematico per lo sviluppo di applicazioni robotiche. Questo approccio include tecniche che riguardano diversi aspetti tra cui (i) la possibilità di lavorare con agenti eterogenei, tra cui lavoratori e supervisori, (ii) lo sviluppo di algoritmi per la computazione delle azioni che i vari agenti devono eseguire per raggiungere la missione

assegnata, (iii) un controllo decentralizzato che permette ai robot di interagire con l'ambiente circostante attraverso l'utilizzo di sensori e la comunicazione tra i componenti del team, (iv) facile integrazione con i diversi sistemi presenti sul mercato, (v) l'utilizzo di un linguaggio di alto livello per la definizione delle missioni, in modo da non richiedere all'utente particolari nozioni di robotica.

Questa tesi tratta uno dei vari aspetti considerati all'interno del progetto Co4robot: la generazione di controllori partendo da un linguaggio di alto livello.

La nostra analisi dello stato dell'arte ha sottolineato richiesta nell'ambito robotico di strumenti che (i) diano la possibilità di descrivere ad alto livello, attraverso una grammatica strutturata, un'applicazione robotica e (ii) generino un controllore per i robot coinvolti nell'applicazione. In particolare, ci focalizziamo sul fornire tali funzioni in situazioni in cui il designer desidera descrivere ad alto livello agenti controllabili e non controllabili e di considerare una rappresentazione esplicita del tempo. Questo lavoro propone PuRSUE, un framework mirato al risolvere i problemi sopra citati. PuRSUE fornisce:

- Un Domain Specific Language (DSL) chiamato PuRSUE-ML. PuRSUE-ML offre la possibilità di descrivere applicazioni robotiche in termini di descrizione dell'ambiente e degli agenti che interagiscono in questo ambiente, includendo rappresentazione esplicita del tempo.
- Una traduzione automatica del modello definito con PuRSUE-ML in un modello nel formalismo dei Timed Game Automata (TGA). La traduzione in questo formalismo permettere di usare tecniche di model-checking esistenti per generare un controllore per l'applicazione robotica presa in considerazione.

Per valutare PuRSUE, abbiamo preso in considerazione i seguenti punti: se PuRSUE offre un modo più compatto dei TGA per modellare un applicazione robotica, se PuRSUE supporta i designer nella generazione di controller i robot negli scenari considerati e nel ragionare su proprietà temporali e se i controller generati da PuRSUE sono efficaci se utilizzati su un robot reale.

Sono state prese in considerazione alcune varianti di scenari robotici ispirati dalla letteratura, per un totale di 13 scenari totali. I risultati mostrano che (1) PuRSUE-ML offre un modo più compatto di modellare formalmente un'applicazione robotica, (2) PuRSUE è efficace nel supportare i designer nella generazione di un controllore per applicazioni robotiche e nel ragionare su proprietà temporali del sistema descritto e (3) i controller generati da PuRSUE sono efficaci quando utilizzati per comandare robot reali.

# Acknowledgements

I would like to express my gratitude to Prof. Matteo Rossi and Patrizio Pelliccione who gave me the opportunity to develop this thesis at Chalmers University in Gothenburg, Sweden.

A special thanks goes to my supervisors, Claudio Menghi and Marcello Bersani, it's been a great experience working with you to develop this thesis.

I would also like to thank BEST Milan, which immensely influenced my personal growth during my university years

Finally, a special thanks goes to my family, which always supported me, and to my friends

# Chapter 1

## Introduction

Robotic applications are becoming pervasive in human lives. Robots are increasingly used in our society to perform alienating and repetitive tasks. This occurs for example inside airports, where baggage management and delivery are mostly automatized by using appropriate robots, inside hospitals, to assist doctors during surgeries, or in many manufacturing plants where automation is nowadays unavoidable.

Autonomous robotic applications are moving from industrial to more commonplace scenarios, such as driverless cars and people houses, as domestic assistance for the elderly and autonomous vacuum cleaners, which can be easily found nowadays in many households. These systems are becoming even more and more autonomous and complex with the advance of the IoT (Internet of Things) technology and the industry 4.0.

The increased interest in these technologies is leading to a huge development of the global market in the sectors of robotic applications.

Market predictions estimate an increasing trend of development for the robotics industrial sector. The spread of robotic applications has also been confirmed by the world Robotics Survey [1], which evidenced an increment of the use of service robots for professional use and indoor logistics. For example, sales of professional service robots had registered a growth of 28% in 2014 in logistics that resulted in an increase of USD 2.2 billion in the value of sales. This growth must be sustained by techniques that allow the effective development of robotic applications.

Developing robotic applications currently requires a set of skills and a level of craftsmanship that are not possessed by average designers. This problem has also been evidenced by the H2020 Multi-Annual Robotics Roadmap ICT-2016 [1] which states that usually, there are no consolidated system

development processes for robotic applications. Specifically, in the robotic domain, solutions are generally conceived ad-hoc and on a per problem basis, rather than being designed for supporting their reuse and for making them accessible to non robotic experts. As a consequence, robotic development remains confined to a small set of highly qualified robotic experts.

We believe that, to effectively support the growth of the robotic market it is necessary to make development of robotic applications more accessible to general developers. 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; (**P2**) enabling automatic *synthesis of controllers* that allow robots to achieve their missions.

**P1: Enabling systematic and rigorous design.** Current practices in robotic development require designers to use low-level primitives, and do not support reasoning on high-level logical elements. For example, the Robot Operating System (ROS) [42], the de-facto standard platform for developing robotic applications, requires developers to publish low level messages on the ROS master node to control the robot behavior. These messages represent low level primitives, whose composition depends on the channel and the kind of information that needs to be transmitted. For example, for the autonomous navigation to take place, a message needs to be sent including the time stamp and the target position in terms of coordinates in a previously defined map of the environment stored in the robot. The lack of high-level constructs introduces an error-prone process even for experienced designers and asks the programmer to deeply know the robots dynamics and kinematics, as well as the ROS System.

Rather than being obsessed by developing ad-hoc solutions and implementing them, we believe that robotic applications development should proceed in a more systematic way, where a rigorous high-level design of the problem domain and of the components of the robotic application is performed first. This practice is in line with current Model-Driven Engineering (MDE) techniques, which ask for the creation of domain models that allow designers to reason on the problem under consideration. During the modeling activity designers may want to consider different applications and scenarios, accounting for different environments and actors, and automatically fulfill the tasks of both programming of the robots behavior and verification of the application feasibility. Furthermore, this preliminary modeling activity is pivotal for enabling automatic reasoning.

**P2: Controller Synthesis.** Robots are essentially agents that are deployed within a given environment to fulfill some mission. A mission is a

high-level goal a robotic application (i.e., a single robot or a set of robots) must accomplish [25, 30, 34, 35]. The mission achievement is reached through the execution of a set of actions that specify how robots change the state of the environment, and how robots 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 its 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 (FM) domain. For example, in [19], [7] and [23] a control problem is encoded into a finite state machine in order to generate a controller. While these techniques effectively handle the controller synthesis problem, they are not designed for being reusable and applicable in different domains. This fact makes their usage difficult in the robotic domain as designers need to know the low level modeling constructs and languages proposed by FM rather than having a generic high-level language equipped with domain specific elements. In essence there is a need to make controller synthesis techniques accessible to people working on the robotic domain in order to turn FM controller synthesis algorithms into widespread used robotic solutions.

We believe that enabling a systematic and rigorous design of robotic applications through rigorous languages and the integration of these languages with off-the-shelf tools that enable controller synthesis is a primary goal to make the usage of FM techniques accessible in the robotic domain. While works that address these two problems exist (e.g., [26], [13] [43], [18] — see Chapter 2), our research is tailored for robotic applications that work under two assumptions: **(A1)** *uncontrollable agents* can move and interact with the robots and their environment; and **(A2)** missions and system model require an *explicit representation of time*. As discusses through our motivating example, these are two central aspects in the development of novel robotic applications.

**A1: Handling uncontrollable agents.** Robots constantly interact with their environments. In many applications, the collaboration of robots side by side with human workers is essential, as they are deployed in every industrial field, from mechanic, chemical and manufacturing industries, to the support of human activities. However, human behavior is sometimes not predictable. Expressing the correct intended behavior becomes more critical in case of human-robot collaboration, where the operator needs to execute certain actions before (or after) that robot performs some other actions, for the sake of his/her physical safety.

For this reason, one of the most prominent challenges in planning and

verifying robotic systems involves its interaction with the environment surrounding it.

Formal and non-formal models of the real world 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. A first step in this direction has been taken via static models of the environment, in which the robots are the sole actors (e.g. [43]). Some other works have contributed to planning with partial-information on the environment such as [33], [32]. These models however fail to capture the uncertainty in the environment’s behavior, that is, even under the assumption of a fully known environment, there might be uncontrollable actors, e.g. humans, who can interact with the robot and the environment itself. In some works such uncontrollable events are only modeled as an input (e.g. [27]), these works fail to take advantage of further knowledge on the possible behaviors of the environment which do not directly affect the controlled system. An explicit representation of uncontrollable agents can be found in some works (e.g. [37], [17]), but these works more currently cater to the verification of some of the system’s properties rather than the generation of a control strategy.

**A2: Handling missions and system models that require explicit representation of time.** Specification of time aspects has a prominent role in the definition of robotic missions. Forcing a robot to achieve a certain mission within a bounded time, 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 able to consider these requirements exist, their usage is mainly confined to robotic or formal method experts.

**Overview of the work.** This work tries to address problems **P1** and **P2** by considering the assumptions **A1** and **A2**. Specifically, it

- (i) proposes a language that allows designers to easily model robotic applications;
- (ii) enables automatic controller synthesis.

This work has been developed within the Co4robots project. The Co4robots project [44] tries to overcome the issues outlined in the previous paragraphs. It aims at introducing software engineering principles and techniques within the robotics domain. The final goal is to shift the design of robotic applications towards well-defined engineering approaches,



which stimulate components reuse and to have a final impact on the robotic market-places. Co4robots assumes that a robotic application is composed of robots that aim at performing a (set of) missions in a collaborative way. A *mission* is a high-level goal a robotic application must achieve. For example, a mission may require a team of robots to bring medical supplies to a surgery room in the hospital environment. Missions are typically defined in terms of a high-level (formal) description of the goals that robots shall achieve.

Co4robots tries to develop systematic software engineering techniques for the development of robotic applications. These techniques include, among others: (i) new techniques that allow heterogeneous agents (including humans) to collaborate in manipulating and moving objects; (ii) techniques for decentralized real-time planning algorithms; (iii) techniques for decentralized real-time perception, allowing robots to perceive their environment either by using its own sensors or by exchanging sensory information with other nearby agents. In addition, Co4Robots promotes the development of: (iv) a software integration platform that enables an easy deployment of robots, this platform aims at integrating the software produced within the project and drives the implementation of the outcomes of research made within the project; (v) a user-friendly specification language that enables users, without expertise in robotics and information technology (such as the personnel of a hotel or a hospital), to specify missions to be accomplished by the robots.

## 1.1 The Drug Delivery Example (DD)

We consider a realistic example from the medical domain indicated in the following as Drug Delivery example (DD). 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 it has to avoid interfering with the transportation of patients on stretchers. A high-level graphical representation of the example is presented in Figure 1.1.

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 open or close. The robot, for security reasons, is not allowed to walk through the emergency room, and any delivered medicine can be positioned on one of the tables set inside the emergency room, next to all of the entrances (**table1**, **table2**, **table3**). The medicine is initially located in the storage room (**storage room**) and the robot should move it to one of the three entrances, when it is open. At the same time, additional agents are present in the area, i.e. the nurse and the stretcher-bearer. The nurse

is located in the emergency room and she has the capability to open closed doors to allow the `medBot` to deliver the medicines, this can be requested by the controller through a beeper, though this request can only be triggered between set time intervals. The stretcher-bearer can move in the environment, around the emergency room, and can possibly close open doors when they hinder his/her movements. The robot needs to always avoid being in the same location as the stretcher-bearer, as that could obstacle his/her movement.

The goal of this work is provide an automatic tool which allows designers to (i) systematically model and design robotic applications like the one presented (**P1**); and (ii) synthesize suitable sequences of actions performed by the robot (if it exists) that always guarantees, in this case, the delivery of the medicine, regardless of the behavior of the stretcher-bearer and given specific geometric information on the environment and the speed of the agents (**P2**). As clearly evidenced by this example, in real life scenarios, it is necessary to handle uncontrollable agents (**A1**). Specifically, in a game-theoretic representation, the robot is the “player”, as well as the nurse, which we assume to be cooperative and immediately open a door upon request, while the stretcher-bearer is the “opponent” whose unpredictable behavior might hamper the realization of the goal that the player has set. In such theoretical framing, the problem we are investigating amounts to determining the existence of a winning strategy for the player, that always guarantees the player to win the game. In the DD scenario, the existence of a winning strategy corresponds to the successful delivery of the medicine operated by the robot in any situation, given a model of the scenario including real-time constraints (**A2**). Hence, the designer has to determine a strategy that can then be implemented on the `medBot` robot. If this task is performed manually, it could easily result in an error prone and time consuming activity. It is indeed necessary for the developer to reason about the behaviors of the uncontrollable agents and program the robot to properly react to these behaviors. In the DD example, one may for example program the robot to pick up the medicine and reach the table at `door1`. However, this may be a failure strategy since the stretcher-bearer may reach `door1`, thus leaving the robot to fail the mission of avoiding to interfere 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 take into account 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, which is indeed nontrivial, especially in topologies not as simple as the one presented. Moreover, as agents may behave in many different ways, and

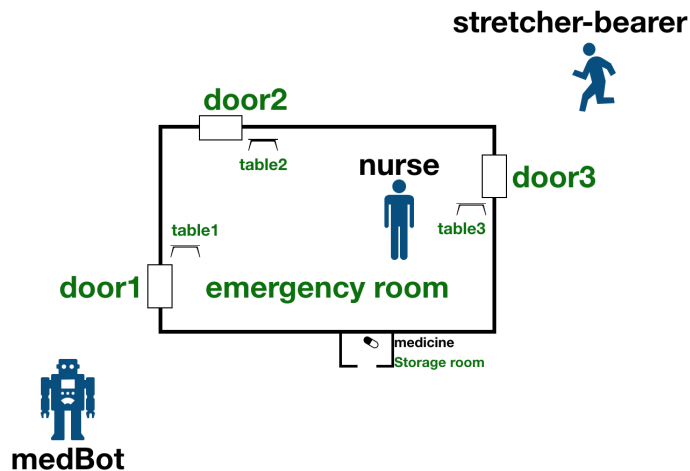


Figure 1.1: A graphical representation of the Drug Delivery example

the number of agents might be usually high in real scenarios, it is desirable to have some *automatic* support in designing the robot strategies.

Hereafter, we will refer to the actors in the system which are controllable by the designer, e.g. the medBot, as “controllable agents”, while all the others are “uncontrollable”. Any detectable action performed by controllable or uncontrollable agents will be referred to as “event”.

We will call “collaborative events” those events who require the presence of another specific agent, from now on called “reacting” agent, so that an agent, called “acting”, can trigger said event.

## 1.2 Research Challenges

To effectively tackle the high-level problems previously described and highlighted by the Drug Delivery motivating example (Sec. 1.1), it is necessary to handle the following challenges:

- **C1:** *Providing a language that allows designers to easily model robotic applications where uncontrollable agents can move and interact with the robot and the environment.* To handle problem **P1** it is necessary to define a language that easily allows describing a robotic problem,

and that possesses precise semantics that allow reasoning about the behaviors of the considered application.

DSLs catered to high-level planning in the domain of robotics have been proposed in several works, as shown in [39]. This is done both for the purpose of verification [36], [21], and for the purpose of planning [26], [13], [43]. While the different focuses of these works will be better discussed in Chapter 2, they do not take into account explicitly possible uncontrollable agents and their behavior relevant to the control problem in the environment, or missions that contain real-time constraints. This is however an important field of research. Given all the possible environment configurations, the best possible guarantee a producer can provide is that an agent will never deliberately make a choice that will lead to a condition it believes to be unsafe [14], it is therefore a relevant task to provide a high-level system description, including uncontrollable agents, that allows for either the verification of properties of the system or the computation of a control strategy capable of achieving a certain mission.

As the developer focuses on high-level control, he/she desires a high language which abstracts the description of the problem from implementation details i.e. issues such as the ability to reach a location on the map and low level motion control. We consider this to be a weak assumption, as it is indeed very common for a robot (Sec. 3.4.3) to have lower level modules to provide navigation and motion control features.

**Example.** *The developer desires a language that allow easily identifying the controllable and uncontrollable agents, the actions that they can perform and how they can move within their environment. 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 only uncontrollable agent is the stretcher-bearer, who can interact with the robot by interfering with each other and with the environment by closing the doors.*

- **C2:** *Providing a framework that supports reasoning on properties that contain an explicit representation of time.* To effectively handle both **P1** and **P2**, it is necessary to include an explicit representation of time. Indeed developers of robotic applications are usually interested in the capability of a framework to handle real-time, especially when the goal of the model is to generate a run-time controller. Indeed, for practical applications, a mission such as "if asked, deliver the item eventually"

might not be a sufficient guarantee, the designer might want to specify constraints such as "if asked, deliver the item within 60 seconds".

Explicit time representation is indeed the focus of research in papers such as [29], [8], [40] and [9], where the paradigm of Timeline-Based Planning is used to solve planning problems. Real time is easily implementable using the formalism of TA (Timed Automata) and its extension, as is done in this thesis, which explains why such formalism is very common in the context of formal verification of robotic applications [19], [7], [23]. Though the aforementioned works do use the formalism to generate a controller, they do not provide an automatic generation of the model through a DSL to provide support to designers for more general situations. A more general framework that allows for modelling of a robotic application with real-time representation can be found in [16] and [37], though these works limit themselves to the verification of the system rather than the generation of a controller.

**Example.** *In our example, explicit modeling of time is necessary in order to include in the control strategies information such as distance between locations, speed of movement and duration of actions. The timed property could be an explicit request in the mission of the control problem, e.g. the medicine having to be delivered within a certain amount of time from request.*

- **C3:** *Automatically synthesizing a run-time controller capable to achieve the mission defined in the scenario.* The automatic generation of high-level controllers is directly related to the problem **P2**. Indeed, the generation of controllers for robotic applications is an active field of research. This is why works such as [11], [29], [8], [40] and [9] make a contribution towards the automated generation of models and controllers capable of achieving the designer-specified mission. All these works however do not really support the designer looking for a solution catered to the specific challenges faced in robotic applications.

There are works more focused on robotic applications, such as [26], [13] and [43], which provide a framework for automatic generation of both a model of the scenario and the controller, but they either do not include the capability to model explicitly the environment in which the robots move and the avoidance/cooperation of/with other agents in the environment. In [19] and [7] there is a more clear description of the controlled system interaction with the environment, and in [23] there is an explicit description of the possible behavior of uncontrollable agents; all these methods generate a controller, but they focus on a single case

study and do not provide a framework capable of generating controllers from designer-defined scenarios.

We believe that a framework allowing designers to freely define a robotic application with all the aforementioned features and being capable of automatically generate a controller for it to be a very valuable contribution.

The controllers generated in this work are assumed to have full knowledge of the state of the system. While this is a strong assumption, it reasonable to expect a smart building in which such robotic application is deployed to have a network of sensors allowing the controller to have full observability of the system. In case such system wasn't available, a predictor could be implemented on top of the control system, able to guess the current state based e.g. on the available information and historical data.

**Example.** *In our example, the designer is interested in generating a controller defining, for each state of the system, an action to be performed by the `medBot` and `nurse`, depending on the state of the system as a consequence of all of the agents interacting with it.*

### 1.3 Contributions of the Thesis

We present PuRSUE (Planner for RobotS in Uncontrollable Environments), a framework aimed at supporting designers in the design of a high-level run-time controller for robotic applications. PuRSUE supports developers in synthesizing a high-level control strategy in cases in which the considered robotic application contains both controllable and uncontrollable agents. The PuRSUE framework allows the designer to focus the description of the system as the environment, the agents in it and the robotic mission, and obtain a controller for the controllable agents, as long as one exists. PuRSUE provides a set of features that allow addressing the problems introduced in Sec. 1.2:

- **F1:** PuRSUE supports the designer in modeling the robotic application. It provides a DSL called PuRSUE-ML that contains a set of keywords and DSL constructs that allow to easily describe (i) the environment where the robotic application is executed; (ii) the controllable and uncontrollable agents acting and moving in it; (iii) a set of constraints between the defined events and (iv) a mission the robotic application should achieve. The support provided by PuRSUE through **F1** allows

tackling problem **P1**. The designer only needs to describe the environment, the controllable and uncontrollable agents acting and moving in it, a set of constraints between the defined events and a mission, at high-level through PuRSUE-ML that will be further described in 4.2.

**Example.** *In our Example, the designer is supported through the use of certain keywords in PuRSUE-ML in defining the locations relevant to the control problem: doors, storage room and emergency room. Then he/she can describe how they are located in the environment with respect to each other. Similarly he/she can describe at high-level agents, such as medBot, where they are located in the environment and the actions that they can perform. In the case of agent medBot, it will be defined as an agent which can move in the environment and examples of actions it can perform are picking up the medicine and delivering it.*

- **F2:** PuRSUE allows the designer to use explicit time to specify the duration of events in time units, as well as the distance between locations and the movement speed of different agents. PuRSUE-ML also allows the designer to include *real-time* constraints to the mission. This fully addresses **P2**.

**Example.** *In our example, the designer is able to define the distance between locations and the traveling speeds of both robot and stretcher-bearer, and he/she can prescribe a mission such as “the medicine needs to be delivered within 60 seconds after it is requested”.*

- **F3:** With PuRSUE the designer is capable of reasoning and automatically synthesizing a control strategy, when one exists, on complex situations where multiple agents, both controllable and uncontrollable, interact in an environment. It is indeed very simple for the designer, thanks to PuRSUE-ML, to devise plans including any number of controllable agents and uncontrollable agents indistinctly, thus effectively handling a team of robots as well as scenarios with any number of opponents. The designer can also easily change conditions in the environment to reason on how such changes affect the control strategy. This is done by describing the scenario at a high-level, rather than the possible interactions between the agents and the environment, thus avoiding the error-prone job of hard-coding all the possible conditions that this environment might reach, while also avoiding the useless task of taking into considerations conditions that could never be reached as a result of how the agents are positioned in the environment. The

system as described through the PuRSUE-ML is thus automatically coded into a Timed Game Automaton (TGA) [2], and the controller will be generated via model checking.

**Example.** *In our Example, focusing on the task of avoiding collision between the two agents, the designer does not need to think of when and where the medBot and the stretcher might run into each other, he/she only needs to describe the distances between the several locations, the speed of movement of the agents and their initial location. Furthermore, if the designer finds that the mission results impossible to be accomplished with one robot, he/she can easily try to see if a faster robot, or two robots, are capable of accomplishing the mission. In a similar manner, the designer can add another stretcher-bearer to produce a controller able to handle a situation with two elements to avoid.*

All these features are implemented in a tool which allows for automatic generation and deployment of the controller, taking as input the description of the system in PuRSUE-ML, and deploying the run-time controller on the selected hardware. A high-level description of the PuRSUE components and how these components address the previous features is described in the following.

- PuRSUE takes as input a description of the system through the PuRSUE-ML. The language (Sec. 4.2), allows the designer to describe the environment at a high-level (F1).
- A parser then translates this description of the environment into a network of TGAs structured as shown in Sec. 4.3. The transformation used is shown in Sec. 4.4. The capability of TGAs to handle explicit time allows the system to formally model scenarios including real time, both in terms of model of the system and mission (F2), and it allows for a simple representation of controllable and uncontrollable agents.
- The finite state machine representation of the scenario is then provided as input to the model checker UPPAAL-TiGA. UPPAAL-TiGA is an integrated tool for creating models, validation and verification of real-time systems implemented as networks of TGA. UPPAAL-TiGA supports the verification of specifications written in TCTL language, which allows the designer to describe the desired behavior of the controlled system and generate a control strategy accordingly. In PuRSUE, UPPAAL-TiGA is used to analyze the model of the system and to produce a controller to fulfill the designer defined mission (F3).



We evaluated (**RQ1**) does PuRSUE offer a more compact way than TGA to model robotic applications? (**RQ2**) how does PuRSUE support designers in generating controllers for robotic applications? (**RQ3**) is the control strategy generated by PuRSUE effectively implementable on actual robots? To answer these questions we considered 3 different robotic applications inspired by case studies from literature ([24], [3], [46]): a robotic application in which a robot has to catch a thief, a work cell and a robot in charge of collecting the thrash. For each of these robotic applications we considered different scenarios with varying complexity, leading to 13 distinct scenarios.

To answer **RQ1** we used PuRSUE-ML to model our scenarios and evaluated the size of the proposed model, in terms of PuRSUE-ML constructs. We evaluated whether PuRSUE-ML allowed the designer to model the system and 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 in TGA. The TGA models are obtained by using our automatic translation. The results show that the number of constructs used to model the robotic application in PuRSUE-ML is less than 19% of the the number of construct that would have been necessary to model our problem using TGA, showing the advantage of using PuRSUE-ML.

To answer **RQ2** we evaluated how fast was PuRSUE able to generate the model of the system and a run-time controller, as well as the size of the generated controller. The results show that in 10 cases over 13 PuRSUE was able to generate a run-time controller. The average size of the generated controller is 4603, while the average time required for computing the controller is 1083 milliseconds. The large size of the generated controller shows that manually designing such controllers is complex, if not even impossible, showing the effectiveness of the automatic computation procedure provided by PuRSUE. The time required for computing those controllers is reasonable for practical usage (given the size of the controllers). In 2 of the three cases no controller was generated as there was no controller able to ensure the satisfaction of the robotic missions, i.e. the scenario provided was unfeasible. In one case PuRSUE was not able to compute the run-time controller due to exhaustion of the address space available with the 32-bit version of the model-checking software.

To answer **RQ3** we deployed the controller generated by PuRSUE to the robot TurtleBot (Sec. 3.4.3), and checked whether the hardware would behave as expected.

We implemented 2 of the 13 scenarios analyzed, simulating the exchange of information with the environment by sending the appropriate signals to the observer-controller system generated. The robot was indeed capable of

achieving both missions.

## 1.4 Structure of the Thesis

This thesis is organized as follows:

- **chapter 2 - Related Work**

Discusses the state of the art. The analysis is focused on works solving the problem of planning for robotic applications, especially in uncontrollable environments, as well as the use of TGAs for the modeling of uncontrollable events.

- **chapter 3 - Background**

Presents the background concepts and notation necessary to understand this work. We describe Timed Game Automata (TGA), the language adopted in this work to model the system and the missions of interest. We briefly introduce the UPPAAL-TiGA Model Checker, explaining the support provided by this tool to generate a TGA model, to simulate its behavior, and to verify the properties of interest. Finally we introduce the material used for the practical experimentation, as well as the robots used for the experimental evaluation.

- **chapter 4 - Contribution** Presents the PuRSUE framework. First the PuRSUE modeling language is thoroughly explained. Then the encoding of the model in TGAs is explained, using our motivating example as constant reference. Then the procedure to translate a PuRSUE-ML model in TGA is presented. Finally, the procedure to generate a run-time controller using UPPAAL-TiGA is discussed.

- **chapter 5 - Implementation**

Presents our implementation of the PuRSUE framework. These include a description of the parser, the algorithm to translate the specification into TGA, how the controller is synthesized and how the controller is executed in the ROS environment. The design-time and run-time components of PuRSUE are explained in detail.

- **chapter 6 - Evaluation**

Reports the experiments run to evaluate PuRSUE and the analysis of the results of said experiments. Different possible scenarios of applicability of the framework are presented, with focuses on different capabilities of PuRSUE. We evaluated PuRSUE's capability in supporting designers

in modeling robotic applications, its capability in supporting designers in the generation of controllers and its effectiveness when deployed on a robot.

- **chapter 7 - Conclusions and Future Work**

Presents the conclusions of this thesis and provides an overview of some possible extensions of the presented work.

## Chapter 2

# Related Work

Autonomous robots are becoming more popular and accessible every year, as such there is a growing amount of works aimed at either modeling a specific problem, building tools or formalizing techniques especially catered to model them, for either planning or property checking. Game models to solve control problems is also an area of research that has been explored. What follows is an overview of the state of the art in all these areas of research, to help the reader better contextualize the work presented and its difference with what has been proposed so far in the literature. First, in sec. 2.1, we present a table where all the analyzed works have been classified.

We identified the following two major categories of works:

- Supporting the design of Robotic Applications (Sec. 2.2). We analyze all the works that aimed at providing tools or theoretical frameworks aimed at supporting developers in the robotics field in either the development, verification or the creation of controllers or plans.
- Game Models (Sec. 2.3). In this section we analyze all the works in which scenarios and their missions are modeled as multi-player games in order to create strategies capable of achieving mission.

### 2.1 Classification of related work

Table 2.1 classifies related work considering the following dimensions:

- **Mod.** : whether the paper provided automatic model synthesis from a higher level description of the problem.

Table 2.1: classification of related work

Paper	F1				F2	F3	exp.
	Mod.	Env.	unc.	Team	Time	Con.	
[27]	X	✓	X	✓	X	✓	✓
[16]	✓	X	X	X	✓	X	✓
[37]	✓	✓	✓	X	✓	X	X
[36]	✓	X	X	X	✓	X	X
[26]	✓	X	X	X	✓	✓	✓
[13]	✓	✓	X	X	X	✓	✓
[28]	✓	✓	X	X	X	✓	✓
[43]	X	✓	X	✓	✓	✓	X
[18]	X	X	X	✓	X	✓	X
[19]	X	-	-	-	✓	✓	X
[7]	X	-	-	-	✓	✓	X
[23]	X	-	✓	✓	✓	✓	X
[12]	X	X	X	✓	✓	✓	X
[11]	✓	-	X	X	✓	✓	X
[29] [8] [40] [9]	✓	-	X	✓	✓	✓	✓
[17]	X	-	✓	X	✓	X	X

- **Env.** : whether the environment in which the agents act is explicitly described.
- **unc.:** whether uncontrollable agents interacting with the controllable ones are explicitly described.
- **Team:** whether teams of agents are considered.
- **Time:** whether explicit time is included.
- **Con.** : whether the paper provides automatic controller synthesis for the scenario.
- **exp.** : whether the work was implemented in experimental setups.

If a column appears marked with the “-” symbol, it means that the corresponding paper treated a topic that differed a lot from a robotic application, as such the discussion of more robotic specific features is meaningless.

We classified in total 19 works. none of them provides a framework catered to robotic application set in a known environment, that allows designers to describe at high level a robotic application, inclusive of any number of controllable and uncontrollable agents, including explicit time in the model and generate from it a run-time controller capable of functioning in experimental setup.

## 2.2 Support to Designers of Robotic Applications

The works presented in this chapter aim at supporting developers in the creation of robotics applications. In Sec. 2.2.1 we discuss how formal methods can be used to model and reason on robotic applications. In Sec. 2.2.2 we present Domain Specific Languages (DSL) tailored to the robotic domain.

### 2.2.1 Formal Methods Formalisms as Tool to Model Robotic Applications

Different formalisms have been proposed to model problems of interests through formal languages that enable automatic support for developers. Among these formalisms, the most commonly found in the state of the art are LTL (Linear Temporal Logic) and FSM (Finite State Machine) (e.g. [32, 33, 45, 47]). Example of this can be found in works such as the one in [20], where probabilistic finite state machines are used to model a swarm of foraging robots in order to verify their behavior. Here the possible behaviours of a single robot are described as a probabilistic finite state automata, which is given as input to the model checker PRISM [22] 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, rather than simulation the product of the models of the all FSA representing the elements of the swarm. This work shows how model checking can be used to verify a robotic application, to show how certain properties can hold for all possible runs/configurations.

In [27] FSM are used as baseline formalism to represent supervisor control theory (SCT). In this work, SCT is applied in order to obtain a system capable of automatically generating code according to a description of what the system can do and what the system should do. The controller is generated by considering synchronous composition between system description and specification generators. This work is implemented using the open source software Nadzoru [41].

In both the aforementioned contributions there is no explicit representation of the possible behaviours of the environment in which the agents exist. We believe that one of the most prominent and key challenges in verifying robotic systems involves their interaction with a possible uncontrollable environment, given all the possible environment configurations, the best possible guarantee a producer can provide is that an agent will never deliberately make a choice that will lead to a condition it believes to be unsafe (as argued in [14]).

In [16] an effort was made toward automating the generation of formal input for the model checker NuSMV [10]. The software focuses on translating the Control Rules of the Care-O-Bot system. These control rules are specified through behaviors, where a behavior is a set of atomic preconditions and a sequence of actions. Such specifications are translated into LTL, which is in turn translated into SMV, the modelling language accepted by NuSMV. The extent to which the environment is modeled in this system is as again as a series of preconditions which can either be true or false (i.e. sensor outputs).

In [37] the undeterministic environment has been modelled using MDPs, where probabilistic transitions model the uncontrollable events in the environment. The tool support allows checking for the probability of the system to meet requirements in an unknown environment. The tool is catered to support design-time decision making and does not provide the feature of controller-generation. The authors of the paper discuss how the information provided by the tool is strictly tied to the accuracy of the probabilistic model, which is what makes it especially interesting to analyze design decisions.

However all the aforementioned works differ from the one presented in this paper, as they do not provide robotic applications designers with a framework allowing them to describe the application at high level, to create in an automatized manner a model of the controllable and uncontrollable agent in the environment, and use this model to automatically generate a run-time controller.

### 2.2.2 DSL for Robotics

Automatic generation of models from a DSL in the field of robotic applications is a topic of interest that has been explored in many works. This domain is well explored in the survey presented in [39].

In [36] RoboChart is presented. A graphical user-interface, di facto restriction of UML, allows designers to describe low level components of a robotic application, as well as the controller and the expected behavior of the system. This description is used for an automatic generation of mathematical definitions that support automatic proof of key properties of

robotic controllers. This paper’s focus is indeed on the validation of the model, and the model itself is centered on low level control problems instead of high level decision making. Furthermore, they do not include explicitly opponents in the high level description of the system.

In [26], a DSL for programming the behavior of autonomous agents is presented. This paper shares with this thesis the idea of translating a system defined at a high level by the user into a FSM. This finite state machine is essentially a decision graph that connects between them options and their subsequent basic behaviors. It also focuses on having a modular system, indeed any “option” is it self composed of other “options” or basic behaviors, thus forming a hierarchy that leads to the aforementioned decision graph. However the focus is placed on allowing the designer to define the desired strategy and reaction of the autonomous agent as a response to a certain state of the system or environmental variables, while the work here presented allows the designer to simply describe the system and the objective o the controller, and subsequently output an always winning strategy. Furthermore it does not allow the designer to model the environment and adversary agents in order to check for the correctness of the generated strategy in front of their possible behavior.

A similar work was performed in [18], where collaborative robots are programmed through NaoText, a DSL based on the concepts of contexts and roles in these contexts. This work ,as well, does not include an explicit representation of the opponents in order to generate a controller accordingly.

An interesting effort towards the description of high level activties for robots is presented in [21], where SRDL is presented, a semantic for describing robot components, capabilities and 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 from the components the agent is provided with. Furthermore an explicit description of the robot past attempts to perform actions is provided, so that the robot can estimate its chance of success of fulfilling the user request. This work is tied to the one presented in this paper as it provides a semantic to model high level actions of a robot, but it differs as it does not concern with an explicit representation of the environment, nor with controller generation.

The work done in [13] presents a DSL that allows designers to write in a structured English grammar the specifications of the robot. The desired behavior of the robot is specified through a set a of assumptions on the environment and desired behavior of the agent expressed as conditions be-



tween events. They however focus on interfacing these high level decisions with low level modules with a hybrid controller. Explicit description of other agent behavior, while conceptually achievable through environmental assumptions, is not an explicit concept as it is in the work presented in this thesis. Furthermore, they do not handle teams of robots and explicit time.

SPECTRA [28], a specification language for reactive systems. It allows for modeling of the environment behavior through assumptions and guarantees. SPECTRA allows to consider discrete LTL specifications, but does not support reasoning with explicit and continuous time specifications, such as the one considered in this work.

Finally, in [43] a formalism is proposed which allows, much like the work in this thesis, to explicitly describe the topology of an environment and tasks that need to be performed in different locations, expressed as services to be delivered in certain locations in a certain time range. Multi-robot scheduling is also included, thus allowing the designer to schedule the movements of a set of robots having to deliver a set of services in a certain time frame without interfering with each other movements. This work, while strongly focused on vehicle routing strategy, does not really allow for representation of more complex tasks; furthermore, uncontrollable agents are not taken into consideration.

## 2.3 Game Models

The idea of generating control strategies by modeling the control problem as a 2-player game against the environment has been used in many works. In Sec. 2.3.1 we show papers which take a specific control problem and encode it in TGAs, with the aim of showing the potential of such technique to solve control problems. In Sec. 2.3.2 we discuss how TGAs have been used for a more systematic modeling and planning in works that translate other existing formalisms into state machines in order to solve them via model checking.

### 2.3.1 TGAs to encode specific scenarios

In [12] TGAs are used 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 a general technique such as model checking.

The works in [19] and [7] show how to translate control problems into TGAs, which are then given as input to UPPAAL-TiGA to automatically synthesize a controller. The first work presents the solution of a temperature

control problem, while the second of an embedded system composed of an oil pump and an accumulator. Especially [7] focuses on showing how this technique is indeed effective at solving the control problem, as well as convenient for the designer. Both these works focus on a low level model of actuators and sensors, rather than a more high-level strategic scenario.

In [23], a control problem related to the transport domain is modeled as 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, and it is focused, like the contribution presented in this thesis, on a high-level planning problem.

In all the aforementioned works, while it was showed how TGA allow for a convenient and automatic synthesis of controllers in different scenarios and scopes, no effort has been presented in the direction of automatic generation of the models themselves, as indeed all the models were designed ad-hoc by the authors. This is however a tedious and time-consuming task that does not lend itself well to a more systematic use of TGAs and model checking in order to generate controllers for different missions and scenarios and to reason on their temporal properties.

### 2.3.2 TGAs to encode existing formalisms

In [11] TGA are used 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-players game between the controllable agents and the environment. Then, they offer an encoding from the presented formalism into TGA and prove its correctness. While this work offers an automatic generation of a TGA starting from a higher level description of the problem, this is useful to a user looking to solve a problem already encoded into the STNU formalism rather than offering a framework to solve a problem any real-life scenario.

In [29] TGAs have been use to encode the formalism of Timeline-based Planning. This is a paradigm in which the system is expressed as a set of states and the allowed transitions between those states, the control system objectives are expressed as values those states should assume in certain temporal intervals. A limitation of this method is that it does not provide a explicit definition of action and it differs from the classical action-based

planning, as it is more declarative and it focuses on actions with uncertain duration. Other forms of uncertainty, such as non-determinism (i.e., which tasks the environment chooses to perform), are not supported. This technique has been applied for plan generation for a rover tasked with in a space exploration mission in [8]. The framework presented has been expanded in [40], where the output timeline-based plan and the state variables of the system are encoded into TGAs in order allow for planning and validation. The framework has been further extended in [9], where a general purpose library is proposed. Finally, in [17] game-like scenarios in which the evolution of state variables is triggered by either a user or an opponent in a turn-based mechanic are considered.

This set of works is strongly related to the one presented in this thesis, as it presents a general framework for automatic generation of both model and controller. However it suffers from the limitation of using TGAs as a tool to solve flexible timeline-based planning problems. While for some scenarios this might be ideal, it also does not take full advantage of the strong representative power of TGAs for timed games and it requires the designer to be knowledgeable about this particular formalism for being effectively used.

# Chapter 3

## Background

This chapter presents background concepts and formalism. Section 3.1 provides an overview over the notation of Timed Game Automata. Section 3.3 described UPPAAL-TiGA, a key component that will be exploited to synthesize controllers. Section 3.4 describes the main tools used for the implementation of the thesis.

### 3.1 Timed Game Automata

To introduce Timed Game Automata (TGA), we first present quickly the simplest model of Timed Automata (TA) [2] and then extend it to TGA.

Let  $\chi$  be a set of clock variables, which are real, positive variables with values evolving with the same rate as the time. The set  $C(\chi)$  of clock constraints over  $\chi$  contains formula  $\phi$  generated by the following grammar:

$$\phi := x \sim c \mid x - y \sim c \mid \phi \wedge \phi \quad (3.1)$$

where  $c \in \mathbb{Z}$ ,  $x, y \in \chi$ , and  $\sim \in \{<, \leq, =, \geq, >\}$ . We denote by  $B(\chi)$  the subset of  $C(\chi)$  that uses only rectangular constraints of the form  $x \sim c$ .

A timed automaton (TA) is a tuple  $A = \langle \Sigma, \chi, Q, I, \Delta \rangle$ , where  $\Sigma$  is a finite set of actions,  $\chi$  is a finite set of clocks, and  $Q$  is a finite set of states. The mapping  $I : Q \rightarrow B(\chi)$  associates with each state  $q \in Q$  an invariant  $I(q)$ , also called staying condition. The set of transitions is  $\Delta \subseteq Q \times C(\chi) \times \Sigma \times 2^\chi \times Q$ .

A transition in  $\Delta$ , also written  $q \xrightarrow{g, l, r} q'$ , specifies the source  $q$  and the target  $q'$  states, the guard  $g$  which is a clock constraint to be satisfied when the transition is executed, the label of the transition  $l$ , and the subset of clocks to be assigned to 0 ( $r$ ) as effect of the transition. This basic model

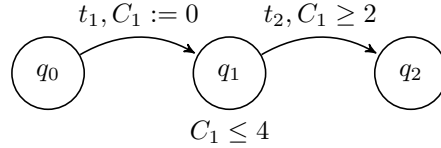


Figure 3.1: Example of a TGA

of TA can be extended to allow state variables with finite values in guards, invariants, and assignments.

For instance the automaton depicted in Fig. 3.1 is defined as  $A_{ex} = \langle \Sigma_{ex}, \chi_{ex}, Q_{ex}, I_{ex}, \Delta_{ex} \rangle$  where the actions are  $\Sigma_{ex} = \{t_1, t_2\}$ , the clocks are  $\chi_{ex} = \{C_1\}$ , the locations are  $Q_{ex} = \{q_0, q_1, q_2\}$ , the invariants are  $I_{ex} = \{q_1 \xrightarrow{C_1} 1 \leq 2\}$  and the transitions are  $\Delta_{ex} = \langle q_1 \xrightarrow{C_1 := 0} q_0, q_1 \xrightarrow{C_1 \geq 2} q_2 \rangle$ .

A timed game automaton is a timed automaton  $A = \langle \Sigma, \chi, Q, I, \Delta \rangle$  where the set of actions  $\Sigma$  is split in two disjoint sets:  $\Sigma_c$  the set of controllable actions and  $\Sigma_u$  the set of uncontrollable actions. For instance the automaton depicted in Fig. 3.1 the controllable actions are  $\Sigma_{c,ex} = \{t_1\}$  and the uncontrollable are  $\Sigma_{u,ex} = \{t_2\}$ .

The standard semantics of a TA is given in terms of *configurations*, i.e., pairs  $(q, v)$  defining the current location of the automaton and the value of all clocks and variables, where  $q \in Q$  and  $v$  is a function over  $X \cup Y$  assigning every clock of  $X$  with a real and every variable of  $Y$  with an integer.

**Definition 3.1.1.** Given a TGA  $A$  and three symbolic configurations  $Init$ ,  $Safe$ , and  $Goal$ , the reachability control problem or reachability game  $RG \langle A, Init, Safe, Goal \rangle$  consists in finding a strategy  $f$  s.t.  $A$  starting from  $Init$  and supervised by  $f$  stays in  $Safe$  and enforces  $Goal$ . More precisely, a strategy is a partial mapping  $f$  from the set of runs of  $A$  starting from  $Init$  to the set  $\Sigma \cup \lambda$ . For a finite run  $\rho$ , the strategy  $f(\rho)$  may say (1) no way to win if  $f(\rho)$  is undefined (2) do nothing, just wait in the last configuration of  $\rho$  if  $f(\rho) = \lambda$ , or (3) execute the discrete, controllable transition labeled by  $l$  in the last configuration of  $\rho$  if  $f(\rho) = l$ .

A strategy  $f$  is state-based or memory-less whenever its result depends only on the last configuration of the run.

## 3.2 Timed Computation Tree Logic

UPPAAL properties allow for specifying missions through a BNF-grammar based on the CTL logic, which also contains "time related" constraints

(TCTL). Let  $e$  be a boolean combination of formulae on variables and clocks such as, for instance,  $x \leq 10 \wedge c_1.Done$  indicating that clock  $x$  is less than or equal to 10 and that component  $c_1$  is in location  $Done$ . TCTL allows the specification of properties in the form

- $A\Box e$  ("for all path globally  $e$  holds");
- $A\Diamond e$  ("for all paths finally  $e$  holds");
- $E\Box e$  ("exists a path in which always  $e$  holds") and
- $E\Diamond e$  ("exists a path in which finally  $e$  holds").

### 3.3 UPPAL-TiGA

UPPAAL [6] is a toolbox for modeling, simulation and verification of real-time systems, based on constraint-solving and on-the-fly techniques, developed jointly by Uppsala University and Aalborg University. UPPAL-TiGA [5] is part of the UPPAAL toolbox for verification of real-time systems which provides an algorithm to synthesize controllers.

UPPAAL-TiGA allows the user to generate strategies for input models specified through a set of Timed Game Automata (3.1), together with winning and losing conditions specified through TCTL formulae. Given these inputs, UPPAAL-TiGA allows the user to:

- Test possible runs of the TGA network through the graphical interface.
- Generate a controller, provided one exists.
- Play the game against the a controller implementing the generated strategy.

A system of TGA is defined in UPPAAL-TiGA through the following elements:

- *Channels*: used to synchronize between transitions; once a channel has been defined, that channel can be used as synchronization label on transitions either with the ! (master) or the ? (slave) appendix. Transitions exhibiting the ! appendix are allowed to trigger the transition, while transitions exhibiting the ? appendix are only able to synchronize with that channel, that is, when a transition master to that channel is triggered, the slave channel will also evolve accordingly. There are two main types of channels, normal and broadcast. A normal channel

is used for binary communication; a master transition can only trigger the evolution when at least one slave transition is also available for triggering, when the transition is triggered, only one slave transition will also be triggered. A broadcast channel does not have such blocking property; a master transition can trigger at any point in time, however all slave transitions available for triggering will synchronize with it instead of only one.

- *Integer variables*: used to store information, they can be updated by transitions and used in guards.
- *clocks*: they represent the clocks of the system.
- *automata model*: they are composed of *locations* (states in the model presented in Sec. 3.1) and *transitions* between those locations. Each location can be endowed with an identifier and an invariant. An *invariant* is a conjunction of conditions of the form  $x < i$  or  $x \leq i$  where  $x$  is a clock reference and  $i$  evaluates to an integer. If the invariant isn't satisfied the system cannot evolve, as such, the invariant forces the system to leave a state endowed with it within the time limit prescribed by the condition.

Transitions are defined through one origin location and one target location, which the transition connects. They can be endowed with the following labels:

- *Synchronization labels*: they are used to synchronize between locations as explained earlier, though the use of channels and the ! and ? appendices.
- *Update labels*: they assign a new value to either a variable or a clock.
- *Guard labels*: they are logical conditions on either variables or clocks that must hold true in order for the transition to be triggered.

Furthermore, each transition can be set as *controllable* or *uncontrollable*.

The UPPAL-TiGA graphical interface, depicted in Figure 3.2, allows the user to inspect the TGA by running possible evolutions of the system. The interface allows the user to **1** trigger both controllable and uncontrollable transitions; **2** read the trace of the evolution thus far; **3** inspect the configuration of the system; It also provides **4** its Gantt chart; and a **5** communication diagram.

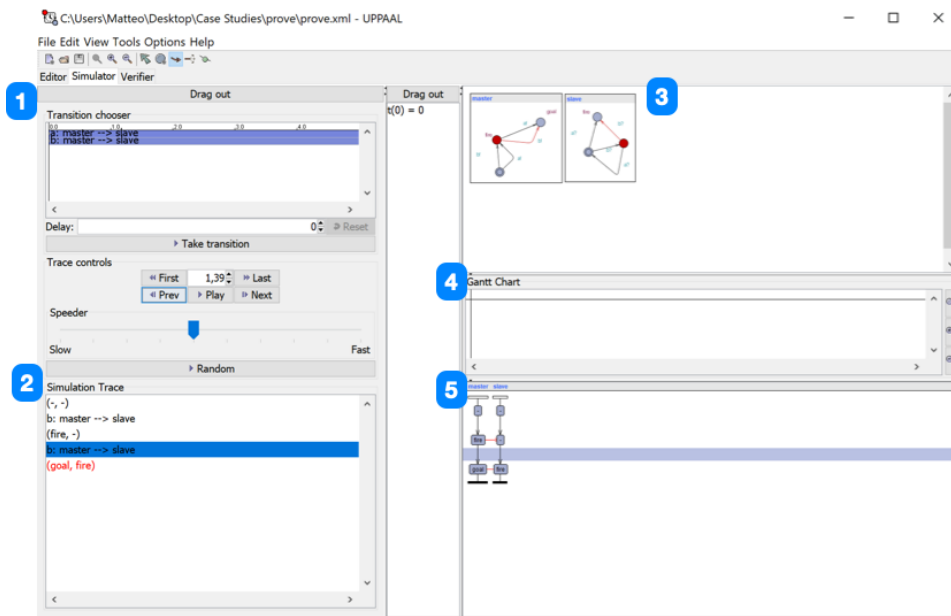


Figure 3.2: screenshot of the UPPAAL-TiGA graphical interface

### Plans Generated by UPPAAL-TiGA.

UPPAAL-TiGA supports both reachability and safety games. Given a timed game automaton  $A$  in starting condition  $init$ , a set of goal states ( $win$ ) and/or a set of bad states ( $lose$ ), four types of winning conditions can be issued, which are expressed as a more restricted version of TCTL:

- **Pure Reachability**  
 $control:A \diamond win$  (*must reach win*)
- **Strict Reachability with Avoidance**  
 $control:A[ \text{not}(lose) \ U \ win ]$  (*must reach win and must avoid lose*)
- **Weak Reachability with Avoidance**  
 $control:A[ \text{not}(lose) \ W \ win ]$  (*should reach win and must avoid lose*)
- **Purse Safety**  
 $control:A \square \text{not}(lose)$  (*must avoid lose*)

For all of them, the goal is to find a controllable strategy  $f$  such that  $A$  supervised by  $f$  satisfies the conditions imposed by the formula.



Once the model of the TGA and the formula representing the winning conditions, a strategy can be generated using the *verifytga* executable provided by UPPAAL-TiGA, though the use of the command:

```
./verifytga -t0 UPPAAL_model.xml
```

Where *UPPAAL\_model* is the specific name given to the file containing the model. A winning strategy generated by UPPAAL-TiGA is composed of a **state:** statement for every reachable configuration of the TGA network. A **state:** statement is composed as shown in Listing 3.3.

- 1 **State:** (*automa\_states*) *variables\_states*:
- 2 **While you're in** (*clock\_conditions*), **wait**.
- 3 **When you're in** (*clock\_conditions*), **take transition** *transition*

Each **state:** statement could include any number of **when you're in** statements, prescribing different transitions for different clock configurations in each state. Each *transition* is reported as which states it connects, as well as any guards, updates and synchronizations associated with it. All other transitions that would be triggered due to synchronization are reported as well. An example of of a generated controller is reported in 4.4.

Furthermore, UPPAAL-TiGA provides the possibility of playing the 2-player game. In case of successful generation of the winning strategy, the user will play as the opponent, choosing the uncontrollable transitions to be triggered, while UPPAAL-TiGA will play the controller according to the strategy generated. In case the controller generation fails due to UPPAAL-TiGA not being able to find a strategy to always win, the user is instead given the role of the controller, choosing the controllable transitions to be triggered, while UPPAAL-TiGA will trigger the controllable ones, performing a trace that shows how the controller cannot win. This function is used with the command:

```
./verifytga -p UPPAAL_model.xml
```

## 3.4 Implementation Tools

In this section we shortly present the main tools used to implement the contribution.

### 3.4.1 Xtext

Xtext is an open-source software framework for developing programming languages and domain-specific languages (DSLs). Unlike standard parser

generators, Xtext generates not only a parser, but also a class model for the abstract syntax tree, as well as providing a fully featured, customizable Eclipse-based IDE.

To specify a language, the developer has to write a grammar in Xtext's grammar language. This grammar describes how an Ecore model is derived from a textual notation. From that definition, a code generator derives an ANTLR parser and the classes for the object model. Both can be used independently of Eclipse.

### 3.4.2 Robot Operating System (ROS)

Robot Operating System (ROS) [42] is robotics middleware (i.e. collection of software frameworks for robot software development). ROS is an open-source, meta-operating system for robots. It provides the services such as hardware abstraction, low-level device control, implementation of commonly-used functionality, message-passing between processes, and package management. It also provides tools and libraries for obtaining, building, writing, and running code across multiple computers.

Running sets of ROS-based processes are represented in a graph architecture. The ROS runtime "graph" is a peer-to-peer network of processes (potentially distributed across machines) that are loosely coupled using the ROS communication infrastructure, structured as follows:

- A *node* is a process that performs computation. Nodes are combined together into a graph and communicate with one another using streaming topics.
- *Topics* are named buses over which nodes exchange messages.
- *Communication* is handled via the topics: (i) Nodes that are interested in data subscribe to the relevant topic; (ii) nodes that generate data publish to the relevant topic. There can be multiple publishers and subscribers to a topic.

### 3.4.3 TurtleBot

TurtleBot [15](Figure 3.3) is a personal robot kit with open source software. TurtleBot was created and developed at Willow Garage by Melonee Wise and Yully Foote in November 2010. TurtleBot is a robot capable of driving in the environment and has enough power and skills to preform different actions and create various types of applications. The TurtleBot kit is composed by

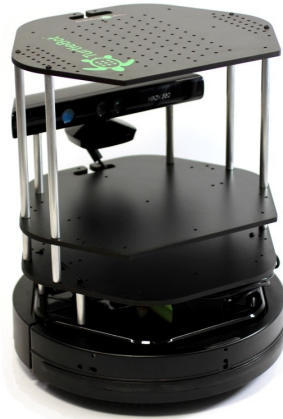


Figure 3.3: the TurtleBot

a mobile base, 2D/3D distance sensor, laptop computer and the TurtleBot mounting hardware kit. In addition to the TurtleBot kit, users can download the TurtleBot SDK from the ROS wiki. TurtleBot is designed to be easy to buy, build, and assemble, using off the shelf consumer products and parts that easily can be created from standard materials. As an entry level mobile robotics platform, TurtleBot has many of the same capabilities of the company's larger robotics platforms, like PR2.

## Chapter 4



# Contribution

PuRSUE (Planner for RobotS in Uncontrollable Environments) is a framework that allows developers to model robotic applications and to synthesize controllers for those applications that can be implemented on robots to achieve a specific mission.

While many control problems can be modeled with PuRSUE, the focus is on modeling scenarios in which one or more robots move and interact with both the environment and possibly many uncontrollable agents. The scenarios that PuRSUE focuses on will be hereafter called Robotic Applications with Uncontrollable Agents (RAUA applications).

Section 4.1 presents an overview on PuRSUE. Section 4.2 presents the PuRSUE modeling language that supports developers in the creation of a RAUA application. Section 4.3 describes the desired format for the intermediate TGA that should be obtained from the PuRSUE modeling language. Section 4.4 describes how the PuRSUE modeling language is translated into an intermediate TGA. Finally, Section 4.5 describes how the intermediate TGA can be used to compute a controller that can be used at run-time.

### 4.1 The PuRSUE Framework

An overview on the PuRSUE Framework is presented in Figure 4.1. PuRSUE allows designers to tackle the challenges described in Section 1.2. The framework allows them to model RAUA applications, with real-time time properties, and generate a controller capable of achieving the specified mission when deployed on a robot in the environment. PuRSUE is made of a set of artifacts and automatic procedures, indicated in Fig. 4.1 by means of a symbol  and a symbol , respectively. Specifically, PuRSUE relies on the

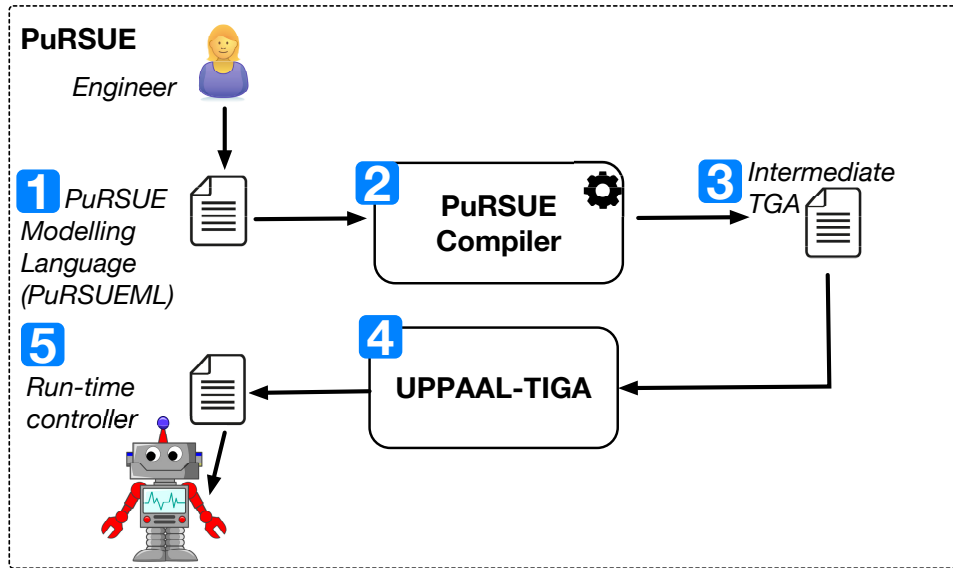


Figure 4.1: The PuRSUE Framework.

following artifacts:

- **1** *PuRSUE Modelling Language ( PuRSUE-ML )* (Section 4.2). It is a high level language that easily allows describing RAUA applications.
- **3** *Intermediate TGA* (Section 4.3). It is an intermediate encoding of the RAUA application defined through TGA that is tailored to the creation of a run-time controller. The controller is automatically obtained from the TGA model by using the UPPAAL-TiGA tool.
- **5** *Run-time controller* (Section 4.5). It encodes the run-time controller that is executed by the robot to achieve the mission of interest in the scenario modeled by means of **1**.

PuRSUE relies on the following automatic procedures:

- **2** *PuRSUE Compiler* (Section 4.4). It allows the translation of the PuRSUE-ML model of the RAUA application into a (intermediate) TGA.
- **4** *UPPAAL-TiGA* (Section 4.5). It allows the generation of the Run-time controller from the (intermediate) TGA.

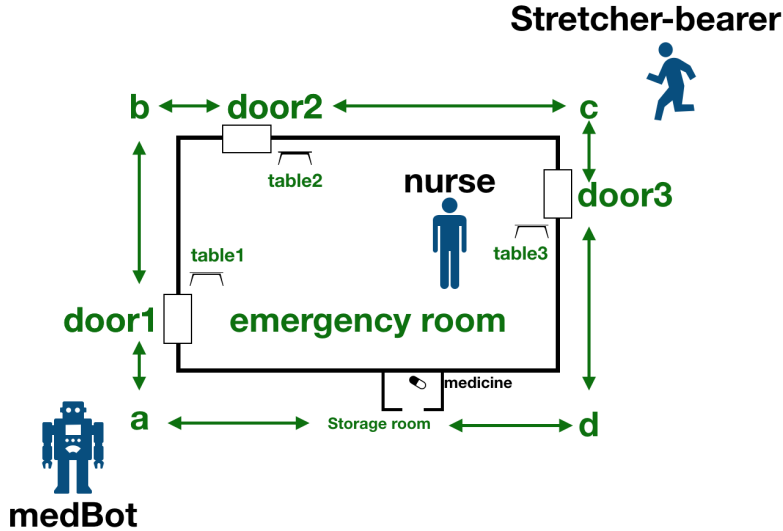


Figure 4.2: the Drug Delivery example.

## 4.2 The PuRSUE Modeling Language

The PuRSUE Modeling Language ( PuRSUE-ML ) is a high level modeling language for RAUA applications. The designer of RAUA applications can describe the control problem through (i) a series of locations relevant to the control problem and how they are connected, (ii) the agents interacting in these locations, (iii) the actions they can perform, (iv) constraints between those actions and (v) the mission the controllable agents need to achieve. To present the features of the language, the Drug Delivery scenario (Sec. 1.1) is used to facilitate the analysis of the motivations and the needs that led to the design of the language. The physical environment is shown in Figure 4.2 with some additional information that is useful to understand the encoding in PuRSUE-ML. The PuRSUE-ML description of the scenario is presented in Listing 4.1. In particular, POIs  $a, b, c, d$  are added in Fig. 4.2 to indicate special positions of the environment where the robot or the uncontrolled agent can pass through. The rest of the section focuses on the different aspects of the RAUA application that can be expressed with PuRSUE-ML.

Listing 4.1– Description for the Drug Delivery example in PuRSUE-ML .

```

1 //locations
2 poi "medicine"
3 poi "room"
4 poi "door1", "door2", "door3"
5 poi "a", "b", "c", "d"
6
7 //connections
8 connect a and door1 distance 6
9 connect door1 and b distance 13
10 connect b and door2 distance 8
11 connect door2 and c distance 20
12 connect c and door3 distance 7
13 connect door3 and d distance 12
14 connect d and medicine distance 14
15 connect medicine and a distance 14
16
17
18 //events
19 event "giveMedicine1" location
    door1 duration 3
20 event "giveMedicine2" location
    door2 duration 3
21 event "giveMedicine3" location
    door3 duration 3
22 event "takeMedicine" location
    medicine duration 2
23 event "confirmDelivery"
24 event "openDoor1" location room
    duration 2
25 event "openDoor2" location room
    duration 2
26 event "openDoor3" location room
    duration 2
27 event "crossRoom" location room
    duration 20
28 event "closeDoor1" location door1
    duration 2
29 event "closeDoor2" location door2
    duration 2
30 event "closeDoor3" location door3
    duration 2
31
32 //rules
33 rule "nurseBehaviour" : ((
    openDoor1 or openDoor2) or
    openDoor3) before crossRoom
34 rule "robotTask" : takeMedicine
    before ( ( giveMedicine1 or
    giveMedicine2) or
    giveMedicine3 ) before
    confirmDelivery)
35
36 //states and dependencies
37 state "door1open": initially false
    , true_if openDoor1 false_if
    closeDoor1
38 state "door2open": initially false
    , true_if openDoor2 false_if
    closeDoor2
39 state "door3open": initially false
    , true_if openDoor3 false_if
    closeDoor3
40
41 stateDependency: giveMedicine1
    only_if door1open is_true
42 stateDependency: giveMedicine2
    only_if door2open is_true
43 stateDependency: giveMedicine3
    only_if door3open is_true
44
45 //agents
46 agent "medBot" controllable mobile
    1 location a can_do
    giveMedicine1, giveMedicine2,
    giveMedicine3, takeMedicine,
    confirmDelivery
47 agent "nurse" controllable
    location room can_do openDoor1
    , openDoor2, openDoor3,
    crossRoom
48 agent "stretcher" mobile 1
    location c can_do closeDoor1,
    closeDoor2 ,closeDoor3
49
50 //objectives
51 reach_objective: do
    confirmDelivery
52 objective: medBot never_with
    stretcher
53
54

```

### 4.2.1 Environment

The physical environment in which the robotic application is deployed is defined by the user through a series of Points Of Interest (POI) and connections among these POI.

- *POI* are used to represent high level physical locations through a textual representation. They represent locations of the environment that are relevant for the behavior of the RAUA applications and that can be occupied by the agents while they move in the environment. They are generic and designed to identify different types of locations, such as buildings, rooms, logical areas or  $(x, y, z)$  points coordinates, according to the considered scenarios. Each POI is defined through the keyword `poi` as follows:

`poi x`

Where  $x$  is the identifier of the POI.

Table 4.1: POIs for the Drug Delivery case

Name	Description
medicine	the storage room where the medicine is located
room	the emergency room where the nurse is located and where the medicine should be delivered
door1, door2, door3	the doors where the medBot can go to deliver the medicine
a, b, c, d	virtual points added to better define the topology of the environment

**Example.** *In 4.1 the locations relevant to the Drug Delivery case are listed with their respective significance for the modeling of the scenario.*

- *Connections* describe how agents can move among POI, i.e., they describe physical links that allow agents to move from one POI to another. They can for instance represent corridors, elevators or any other sort of physical or logical connection among POI.



Connections are labeled with a meta information, that is the length of the path between a pair of POIs, key feature to model scenarios with real-time concerns. A connection is defined through the use of the keyword `connect`, as follows:

```
connect x and y distance n (unidirectional)?
```

Where  $x$  and  $y$  are the two locations to be connected and  $n$  is the distance between them. Connections are bidirectional by default, but unidirectional connections can also be modeled by adding the keyword `unidirectional` at the end of the definition. Here and for the remainder of this section, the question mark symbol is used to mark optional keywords in the definition of a construct. While connections must be textually specified by the user, future extensions may integrate PuRSUE with graphical interfaces and techniques that map a graphical representation of the locations to the textual representation provided by PuRSUE.

**Example.** *In the Drug Delivery case, in Listing 4.1, Lines 7-15 describe in PuRSUE-ML how these locations are connected.*

## 4.2.2 Events

Events represent atomic behaviors that agents can execute through the usage of appropriate actuators and that modify the configuration of the environment. All events are considered to be observable by both the controller and the opponents. Indeed, as we are interested in generating a controller capable of selecting an action given the full configuration of the system, we need to assume for the controller to have full observability of said system.

Every event, in order to be triggered, is associated with an agent that is able to perform it. In case an event is not associated with any agents in the environment, e.g. a blackout, a dummy agent with the sole function of triggering it can be defined with no loss of generality. Events are defined by means of the keyword `event` as follows:

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

The keywords have the following meanings:

- **collaborative:** it defines an event that requires collaboration between two agents when triggered. Intuitively, if in the DD scenario, the

medBot can only fetch a medicine from another agent, e.g., a medicine dispenser capable of moving through the environment, then the event representing the “medicine retrieval” has to be modeled as collaborative. The semantic of collaborative events is better explained in 4.2.5.

- **location  $x$** : it indicates that the event can only occur when the agent performing it is in POI  $x$ .
- **duration  $n$** : defines an event modeling a task whose execution requires  $n$  time units. When reasoning on temporal properties, it is very useful to be able to analyze how the controller generated changes according to the duration of the events performed by the agents. This allows the designer to explicitly model events that take the physical agents some time to perform, e.g. the time to raise the robotic arm and grab a medicine for the `takeMedicine` event.

**Example.** *The events relevant to the Drug Delivery case are the ones depicted in Table 4.2.*

Another set of events, related to the movement between through POIs and connections can be automatically defined by PuRSUE. This is done for all agents defined by the designer as capable of moving, concept which will be further explained in 4.2.5. To prevent an agent from moving between two POIs, PuRSUE-ML includes keyword `prevent`:

`prevent a moving_between x and y (unidirectional)?`

If the keyword `unidirectional` is used, then the order between  $x$  and  $y$  determines the direction that is not allowed.

### 4.2.3 Rules

Rules allow the user to model constraints in the evolution of the environment through the description of possible behaviors of its subsystems. The evolution is expressed in terms of temporal relationships (i.e., precedence) between events. In other words, rules express any sequence of events that could be described as a regular expression having as alphabet the set of events of the system/scenario. In the current implementation, a subset of regular expressions is used: only the concatenation operator, `before`) and the `(or)` operator are available, as with these two operators we could capture all the scenarios considered in this work. The reason motivating this design choice is twofold: (a) regular expressions are a formalism that designers generally

Table 4.2: Events Relevant to the Drug Delivery case

Name	coll.	location	dur.	description
giveMedicine1, giveMedicine2, giveMedicine3	✗	door1, door2, door3	3	the delivery of the medicine on respectively table1, table2 or table3
takeMedicine	✗	medicine	3	picking up the medicine from the storage room
confirmDelivery	✗	✗	✗	signal that the medicine has been delivered
openDoor1, openDoor2, openDoor3	✗	room	2	the opening of respectively door1, door2 and door3 (from the emergency room)
crossRoom	✗	room	20	a general representation of the nurse being busy between opening of doors
closeDoor1, closeDoor2, closeDoor3	✗	door1, door2, door3	2	the closing of respectively door1, door2 or door3 (from the outside)

know and can easily use; (b) methods to translate regular expressions into FSA are currently available in the literature. This is very valuable to us, as in the PuRSUE framework the model as expressed in the PuRSUE-ML will be translated into a TGA (as shown in Sec. 4.3).

All rules are cyclical, that is once the last action of the rule is completed, the initial action returns available for triggering. Cyclic rules are useful to

model several behaviors of the environment which by nature don't have a terminal conditions, e.g. you need to pick up coffee before serving it, at which point you can pick it up again.

Rules are defined by means of the keyword `rule` as follows:

`rule r: rule`

Where  $r$  is the identifier of the rule, and  $rule$  is the regular expression defined through the aforementioned operators. Some examples of expressions are here presented:

- Events A, B and C must happen in order:

$(A \text{ before } B) \text{ before } C$

or equivalently

$A \text{ before } (B \text{ before } C)$

- Event A must be followed by either event B, C or D

$A \text{ before } (B \text{ or } (C \text{ or } D))$

or equivalently

$A \text{ before } ((B \text{ or } C) \text{ or } D)$

Observe that rules have a blocking property on the agents: if an event is included in a rule but not available for triggering in the current state of the rule, the event cannot be triggered by any agent. However, rules do not enforce the triggering of any of the events included in them, as such an agent could decide to stay indefinitely in any given state thus stalling the evolution of the system. For instance, rule  $(A \text{ before } B) \text{ before } C$  does not impose/enforce the eventual triggering of event  $C$ , but it only prescribes that, if  $C$  is triggered, then event  $B$  must have been triggered earlier than  $C$ . The designer should keep this in mind, to avoid ill-defined scenarios.

To explain why incorrect modeling might derive from the wrong use of rule, we use a simple scenario: consider a controllable robot that has to deliver coffee to an uncontrollable human and that the mission of the system is expressed by means of event `takeCoffee` that, when triggered, certifies the winning of the game. Moreover, consider that `takeCoffee` can only be triggered by the uncontrollable agent, i.e. the human that takes the coffee. In such a scenario, the designer might be tempted to set a rule which prescribes `bringCoffee before takeCoffee`, expecting the agent human to eventually trigger the event `takeCoffee`. However, the agent human could never trigger the event thus making the game impossible to win.

**Example.** In the Drug Delivery case, there are two rules defined. The first (Line 33) is *nurseBehaviour*, this rule models how the nurse should do the event *crossRoom*, a fake event act to modeling the nurse not being able to continuously open doors, between any two events of opening doors.

The second rule (Line 34) is *robotTask*, this rule describes how, in order to be able to trigger the event *confirmDelivery*, one of the three events related to medicine delivery should be performed, and again how in order to perform this events, the event *takeMedicine* needs to take place.

#### 4.2.4 States and State Dependencies

States are a modeling construct useful to represent the configuration of some entities of the environment, either physical or abstract, whose state affects the overall evolution of the system and changes according to the triggering of certain events. States are defined using the keyword `state`, as follows:

```
state s initially (true/false), true_if e1,true, ..., en,true false_if
e1,false, ..., em,false
```

where  $s$  is the identifier for the state, `initially` is used to set the initial condition of the state, either *true* or *false*, and  $e_{1,true}, \dots, e_{n,true}$  and  $e_{1,false}, \dots, e_{m,false}$  are two disjoint sets that identify the events that change the state of  $s$ , respectively, to *true* or *false*, during the evolution of the system. States can trivially be extended beyond the Boolean definition used in this thesis. However, the Boolean domain turned out to be enough to capture all the case studies considered in the work.

State dependencies describe the necessary conditions that must hold to allow certain events to occur. A state dependency is defined through keyword `stateDependency`, for any event  $e$  and any Boolean formula  $exp$  defined on state variables, as follows:

```
stateDependency e only_if exp
```

**Example.** In the Drug Delivery case, there are three states defined (Lines 36-40), describing whether the doors are open or closed, the initial value is false for all three states, that is, all doors are closed at the beginning of the execution. The three state dependencies at at Lines 41-44, describing how the medicine can only be delivered at each POI when the corresponding door is open.

#### 4.2.5 Agents

Agents are used to describe the entities interacting in the environment, whose behavior is defined by the sequence of actions that they perform over the time.

The set of the agents contains both *controllable* and *uncontrollable* agents. An agent is controllable when its behavior can be controlled by a controller that determines the actions it has to perform based on the current system configuration, encoded as the current state of the state machine representing the system. An agent is uncontrollable when it spontaneously moves or performs actions.

Agents are defined using the keyword **agent**, as follows:

```
agent a: (controllable)? (mobile sp)? location x (can_do  $e_1, \dots, e_n$ )?
      (reacts_to  $e_1, \dots, e_m$ )?
```

Where *a* is the identifier of the agent, and the keywords `controllable`, `mobile`, `location`, `can_do` have the following meanings:

- `controllable`: it is used to define controllable agents, as explained in the above paragraph.
- `mobile sp` : It is used to define agents that can move through the user defined POIs. The `mobile` keyword must be followed by an integer *sp* representing the time in which the agent covers a distance unit as defined in the connections (4.2.1).
- `location x` : It defines the initial POI, *x*, where the agent resides at the beginning of the execution of the system.
- `can_do  $e_1, \dots, e_n$`  : It defines a set of *n* events that a certain agent is able to perform.
- `reacts_to  $e_1, \dots, e_m$`  : It defines a set of *m* events that a certain agent enables for collaboration. A collaborative event is an event that requires the presence of two agents to be triggered. When an event is defined as `collaborative`, both agents capable of performing it (`can_do`) and agents capable of reacting to it (`reacts_to`) should be defined. The event will be available for triggering when at least one acting and one reacting agents for that event are in the same POI. If an event is defined as `collaborative` but no reacting agents are defined, the event will always be allowed to be triggered.

**Example.** *In the Drug Delivery case, there are three agents. The two controllable ones are `medBot`, representing the robot we wish to control, and `nurse`, a nurse that we assume is acting in order to receive the medicine. `MedBot` is capable of picking up the medicine (`takeMedicine`) and delivering it to one of the three tables (`giveMedicine`). `Nurse` is capable of opening the*

Table 4.3: Available Objectives

Name	Type	Syntax
Reaction	safety	if $e_1$ then $e_2$ within $n$
Event Avoidance	safety	avoid $e$
Execution	reachability	do $e$ (after $n$ )?
Positional Avoidance	safety	$a_1$ never_with $a_2$

three doors. The event *crossRoom*, emulates the busy work of the nurse, this combined with the rule *nurseBehaviour* simulates how the nurse needs to perform several tasks and can not spend the whole executing time opening doors. The nurse is not modeled as a mobile agent, as to the end of the model it is confined to the emergency room. Finally we have one uncontrollable agent, *stretcher*. This is a mobile agent that represents a patient being carried on a stretcher.

#### 4.2.6 Objectives

An objective describes a desired goal or a behavior of the system that the controlled agents have to achieve. Objectives can be expressed by means of different constructs, some including the use of real-time constraints. The semantics of objectives supported by PuRSUE-ML are presented in Table 4.3, where every safety objective is preceded by the keyword `objective:` and every reachability objective is preceded by the keyword `reach_objective:`. Their meaning is described hereafter. For any event  $e$ ,  $e_1$ ,  $e_2$ :

**Reaction** : If  $e_1$  happens, then event  $e_2$  must be triggered within  $n$  time units.

**Event Avoidance** :  $e$  should never happen.

**Execution** : after  $n$  time units from the initial time instance,  $e$  must eventually happen. If the second part of the syntax is omitted,  $n$  will be considered zero.

**Positional Avoidance** : at no time during the execution agents  $a_1$  and  $a_2$  can be in the same POI or traveling over the same connection linking two POIs but towards opposite directions.

It is possible to expand objectives to include more complex missions for the controllable agents. This could for instance be done by including the representation of several patterns. The work in [35] shows a collection of patterns which could be implementable to enrich the representative capabilities of PuRSUE, this was not done as the missions defined resulted sufficient, in combination with the other constructs, to represent the scenarios taken into consideration.

**Example.** *In the Drug Delivery case, the mission is, to deliver the medicine to the emergency room, while avoiding to obstruct the stretcher. That is, to manage to trigger the event `confirmDelivery` while avoiding for agents `stretcher` and `medBot` to ever be in the same location or traveling in opposite directions between the same two locations. This is translated through the use of an Execution objective and a Positional Avoidance objective.*

### 4.3 Intermediate TGA

This section elaborates on the design of the automata that are automatically generated by PuRSUE and used for the the generation of the controller. In particular, the descriptive model of RAUA applications, that are described through the PuRSUE-ML, are translated into a network of TGA, that properly captures all the aspects characterizing the evolution of the modeled scenario over the time. Therefore, all the aspects that are discussed in Section 4.2 will be mirrored in the TGA network by means of specific constructs or resources, such as, for instance, automata, integer values or synchronization channels.

The TGA model has been designed in order to properly model the behavior of the system and the following assumptions:

- **A1:** Once an agent is committed to an event, whether it is a movement or the performing of an event with duration, it cannot do anything else until that event is completed.
- **A2:** A collaborative event with or without duration is considered available for triggering if the following condition is verified: acting and reacting agent are in the same location. In case of a collaborative event without duration, the event is considered available for triggering also if acting and reacting agent are traveling between the two same locations in opposite directions.
- **A3:** Once an agent reaches a location, it needs at least one time unit of inactivity to pass before performing events or movements.



- **A4**: give priority to events performed by uncontrollable agents over events performed by controllable ones.

**A1** represents the assumption that events are atomic components of the system modeled, in order to allow an agent to interrupt during the execution of an event or a movement, the designer should define more events or POIs, thus changing the granularity of the model.

We consider **A2** to be a reasonable assumption, it is up to the designer to keep this in mind, when defining the system. It is also reasonable to assume that two agents traveling in different directions between the same two locations will meet at some point, it is again important that the designer keeps in mind how this mechanic works when modeling the scenario. **A3** is a very important assumption for our model, as without it, due to **A4**, uncontrollable agents could avoid cooperative events, at a model level, by spending zero time at any given location. It is also a reasonable assumption, as any agent occupying a physical space will need a finite amount of time to cross it and/or start any task in it. **A4** is necessary in order to provide robustness to the plans, it would not be recommendable to deploy a plan which capability to meet the mission requirements relies on acting before an opposing agent when the system constraints would allow both agents to act. The fulfillment of **A4** depends on the model checked selected to generate the controller (Sec. 4.5).

The final network of TGA representing a RAUA application is composed of the following automata:

- State automaton (Section 4.3.2): it performs the update of the variables related to the states. The state automaton is composed by one state and a series of transitions performing the necessary variable updates.
- Rule automata (Section 4.3.3): the automata representing the rules, as defined in PuRSUE-ML. A rule automaton is composed of transitions labeled by the events that occur in the rule. A rule automaton associated with a rule can only accept a sequence of events as accepted by the rule. The synchronization of this automaton with the system ensures that the whole system can only evolve through patterns accepted by said rule.
- Agent automata (Section 4.3.4): the automata representing the agents of the system. An agent automaton is composed of a location for every POI in which the agent can move and locations representing the agent being busy performing certain events or movements.

- Objective automata (Section 4.3.5): the automata that, together with the automatically generated logic formula, defines the mission of the system. An objective automaton composition depends on the specific mission being represented.

### Implementation of Extended Broadcast Channels in UPPAAL-TiGA.

The channels as offered by UPPAAL-TiGA do not provide the feature of both having blocking capability, as do normal channels, and allowing synchronization between more than two automata, as do broadcast channels. We wish have both these features, e.g. to have an event  $e$ , that can be triggered by automaton  $A_m$  (*Master*), only when the event is available for triggering in both  $A_{s1}$  and  $A_{s2}$  (*Blocking Slaves*), and when the event  $e$  is triggered, both  $A_{s1}$  and  $A_{s2}$  should evolve accordingly. To do so, we declare the channel  $e$  as broadcast; we then endow a transition in  $A_m$  with the label  $e!$  thus allowing the automaton to trigger the transition, and endow transitions in  $A_{s1}$  and  $A_{s2}$  with the  $e?$  label; as such, when  $A_m$  triggers the transition, the transition will also be triggered in  $A_{s1}$  and  $A_{s2}$ . Furthermore, guards are set on the transition in  $A_m$ , so that the transition can only be triggered when  $A_{s1}$  and  $A_{s2}$  are both in a location from which they can trigger a transition labeled with  $e?$ . This ensures that  $A_m$  can only trigger the transition labeled with  $e!$  only when all the automata containing a transition labeled  $e?$  are available to be slaves to that transition, thus effectively implementing the blocking property of normal channels. Furthermore, it is still possible to define further automata that synchronize with the channel by do not block its execution simply by not implementing said guard (*Simple Slaves*).

#### 4.3.1 Variables declaration

To correctly model the evolution of the environment, i.e. the coordinated evolution of all the automata modeling the interaction of the agents in the environment, the network of TGA is endowed with integer variables, clocks and channels that are use to model the progress of the agents performing actions in the environment, the time elapsing with respect to relevant events occurring in the environment and the coordinated evolution of two, or more, automata on the occurrence of certain events in the environment. The declaration of said variables for the Drug Delivery case is reported in Listing 4.2.

Listing 4.2– Variable declaration for the Drug Delivery case

```
//events

//movements
broadcast chan medBot_a2door1, medBot_door12a, medBot_door12b,
    medBot_b2door1, medBot_b2door2, medBot_door22b, medBot_door22c,
    medBot_c2door2, medBot_c2door3, medBot_door32c, medBot_door32d,
    medBot_d2door3, medBot_d2medicine, medBot_medicine2d,
    medBot_medicine2a, medBot_a2medicine, stretcher_a2door1,
    stretcher_door12a, stretcher_door12b, stretcher_b2door1,
    stretcher_b2door2, stretcher_door22b, stretcher_door22c,
    stretcher_c2door2, stretcher_c2door3, stretcher_door32c,
    stretcher_door32d, stretcher_d2door3, stretcher_d2medicine,
    stretcher_medicine2d, stretcher_medicine2a, stretcher_a2medicine ;

//actions
broadcast chan giveMedicine1, giveMedicine1DONE , giveMedicine2,
    giveMedicine2DONE , giveMedicine3, giveMedicine3DONE , takeMedicine,
    takeMedicineDONE , bumpInto, confirmDelivery, openDoor1,
    openDoor1DONE , openDoor2, openDoor2DONE , openDoor3, openDoor3DONE
    , crossRoom, crossRoomDONE , closeDoor1, closeDoor1DONE , closeDoor2
    , closeDoor2DONE , closeDoor3, closeDoor3DONE ;

int PnurseBehaviour=0, ProbotTask=0;

//Agents
clock CmedBot, Cnurse, Cstretcher;
int PmedBot=6, Pnurse=2, Pstretcher=8;
int Sdoor1open=0, Sdoor2open=0, Sdoor3open=0;
```

The elements generated are now presented.

### Channels.

Channels model the movement of an agent from a POI to another one and to model the occurrence of an event of the environment. Channels are necessary to represent events happening in the system, thus allowing for synchronization between automata in the model, as well as representing the triggering of an event in the run-time architecture when the transition is sent to the robot as an execution choice by the controller.

- For any pair of connected POI  $x$  and  $y$  and for all the agents  $a$  in the environment defined with the keyword `mobile`, two broadcast channels called  $a\_y2x$  and  $a\_x2y$  are introduced. The semantics of a synchronization event occurring through such channels is that “agent  $a$  begins the moving from location  $x$  to location  $y$ ” or vice versa.

- For every event  $e$ , a broadcast channel called  $e$  is introduced. The semantics of a synchronization event occurring through such channels is that “ the execution of event  $e$  has started”. Furthermore, for every event  $e$  defined with the keyword `duration`, a channel called `eDONE` is introduced as well. The semantics of a synchronization event occurring through such channels is that “ the execution of event  $e$  has ended”. When an event with duration is triggered, the triggering agent will go to through the triggering of  $e$  in a location marking it as busy, and will be allowed to leave such location after the appropriate time through the triggering of `eDONE`.

**Example.** Channel `medBot_a2door1` is triggered anytime agent `medBot` starts moving from POI `a` to POI `door1`. Channel `giveMedicine1` is triggered whenever an agent starts performing the event `giveMedicine1`, while `giveMedicine1DONE` is triggered when the agent is done performing it.

### Clocks.

For every agent  $a$  in the environment a clock  $C_a$  is introduced. Clock  $C_a$  is used to keep track of the time elapsed from the beginning of a movement or an event execution of agent  $a$ , and it is reset every time a new movement or event is executed.

Moreover, for every objective including real-time constraints  $o$ , as defined in Sec. 4.2.6, a clock  $C_{obj}$  is introduced. Clock  $C_{obj}$  is used to keep track of the passing of time and it is reset according to the dependency of the specific mission on time, as presented in Sec. 4.3. How these resets are handled is explained for each objective construct in Sec. 4.3.5.

**Example.** Clocks `Cmed`, `Cnurse` and `Cstretcher` are used to keep track respectively of the three agents. No clock is declared for the objectives.

### Integer variables.

They are used to model collaboration among agents and the implementation of extended broadcast channels as explained in 4.3.

- For every rule  $r$  in the set a *rule state* variable  $P_r$  is introduced. The value of  $P_r$  ranges over a finite domain  $\{0, \dots, M_r - 1\}$  of integers, where  $M_r$  is the number of locations of the automaton representing rule  $r$  and every location  $q$  of the rule is associated with an integer  $h(q) = i$ . This allows to have global variables keep track of the evolution of every

rule automaton. These variables are then used to ensure that agents only evolve through sequences of events allowed by the rules.

- For every agent  $a$ , an integer  $P_a$  is introduced. Value  $P_a$  is set to a unique identifier defined internally every time agent  $a$  reaches a POI, starts a movement between two connected POIs or is busy performing an event at a certain POI. Let  $P$  be the set of POIs in the scenario. Every POI is associated with an Integer in the domain  $\{1, M_l\}$ , where  $M_l$  is the cardinality  $|P|$  of set  $P$ . For every  $p \in P$ , let  $l(p)$  be the integer value associated with  $p$ . We now define operator  $\circ$ : given two integer values  $n$  and  $m$ , the value  $n \circ m$  is equal to  $n \times 10 + m$ , e.g.  $1 \circ 3 = 13$ ,  $3 \circ 1 = 31$ . The value of  $P_a$  is defined as follows:
  - When agent  $a$  is in POI  $p$  not performing an event, then  $P_a$  is set to  $l(p)$ .
  - When agent  $a$  is performing an event in POI  $p$  then  $P_a$  is set to  $-l(p) \circ l(p)$ .
  - When agent  $a$  is moving from POI  $p$  to  $p'$  then  $P_a = -l(p) \circ l(p')$ ; whereas if  $a$  moving from POI  $p'$  to  $p$  then  $P_a = -l(p') \circ l(p)$ .

These variables allow the system to keep track of where agents are and whether they are busy or not, this information is then used to set appropriate guards to implement collaboration.

**Example.**  *$P_{medBot}$  is the variable associated with the location of  $medBot$ . When it is set to 3, it means  $medBot$  is at  $door1$ , if it is set to  $-33$ , it means it is busy in  $door1$ , if it is set to  $-36$ , it means  $medBot$  is going from  $door1$  to  $a$ , if it is set to  $-64$ , it means  $medBot$  is going from  $a$  to  $door1$ .*

- For every state  $s$ , an integer  $S_s$  is introduced. The value of  $S_s$  can be either 0 or 1, and is updated according to the logic defined by the user through PuRSUE-ML as explained in sect. 4.2.4. Conditions are then set on these variables to implement State Dependencies. The decision to implement this value as an Integer and not as a Boolean is dictated by the fact that states could be easily expanded to include any number of values.

**Example.** *Variable  $S_{door1open}$  is associated with the state of  $door1$ , which can be open or close. Its value is set to 0 when event  $closeDoor1$  is triggered, and to 1 when  $openDoor1$  is triggered.*

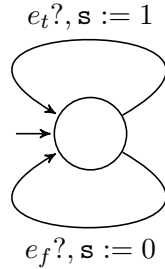


Figure 4.3: State automaton with two transitions updating the value of a state  $s$ .

### 4.3.2 State Automaton

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 RAUA application only one state automaton is introduced. By definition, every state  $s$  defined in the environment is associated with two sets of events  $E_{true}$  and  $E_{false}$  that determine a modification of its value (see Sec. 4.2.4). The state automaton has only one location. Moreover, for every state  $s$ , the state automaton is endowed with two transitions, one for each element of  $E_{true}$  and  $E_{false}$ . In particular, a transition is labeled with a synchronization guard  $e_{i,true}?$  and assignment  $S_s = 1$  for every element of set  $E_{true}$ ; one is labeled with a synchronization guard  $e_{i,false}?$  and assignment  $S_s = 0$  for every element of set  $E_{false}$ . If the same event is tied to the change of value of several state constructs, only one transition per event is kept and an update (according to the previously explained logic) is added for each state the event is associated with. This allows us to update the value of the states whenever the corresponding event is triggered by any agent. Figure 4.3 exemplifies the case of a state  $s$  of the form  $(0, \{e_t\}, \{e_f\})$ . In case event  $e$  has a duration, the event  $e_{DONE}$  is used instead, as we assume that the effect of a durable event on the environment is determined at its conclusion.

**Example.** *The state automaton for the DD example is depicted in Fig. 4.4. Once the event `openDoor1DONE` is triggered, the variable `Sdoor1open` is set to 1, while if `closeDoor1DONE` is triggered, the same variable is set to 0. This in turn allows us to set conditions making it possible to trigger the event `giveMedicine1` only when `Sdoor1open` is set to 1, that is, when the event `openDoor1` was the last one triggered of the two, so we assume the door to be open.*

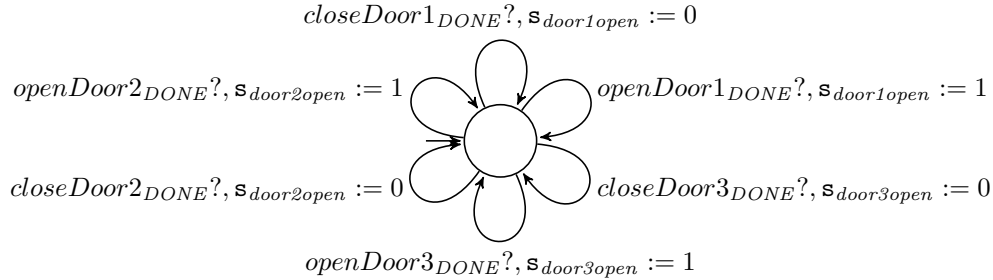


Figure 4.4: Example of a state automaton for the Drug Delivery case

### 4.3.3 Rules Automata

A rule automaton represents a single rule operating in the environment. Since a rule expresses a constraint on a subset of events of the environment, a rule automaton simply determines the set of allowed sequences of events described by the associated rule. For this reason, a rule automaton does not use clocks and location invariants, as no real-time constraints are encoded in the rules. A rule automaton can be built piecwisely. Two are the patterns allowed to appear in a rule and each one is translated into a suitable sub-automaton that can be combined piecwisely with others to capture the whole rule. Every rule  $r$  is of the form  $(a \text{ before } b)$  or  $(a \text{ or } b)$  where  $a$  and  $b$  can be either an event  $e$  or recursively a rule  $r_1$ . We will refer to as “union” of two locations  $l_1$  and  $l_2$  when two locations are replaced by a one,  $l_3$ , having as incoming transitions all the incoming transitions to  $l_1$  and  $l_2$  and, as out coming transitions, all the outcoming transitions of  $l_1$  and  $l_2$ .

- An event  $e$  is translated by means of a transition, labeled with  $e$ , which connects two locations  $l_0$  and  $l_1$ . We call  $l_0$  and  $l_1$  as “initial” and “final” location, respectively.
- $r_1 \text{ before } r_2$  is defined through the union of the final location of  $r_1$  and the initial location of  $r_2$  (Fig. 4.5)
- $r_1 \text{ or } r_2$  is defined through the union of the initial locations of  $r_1$  and  $r_2$ , and the union of final locations of  $r_1$  and  $r_2$  (Fig. 4.6)

Every rule is cyclic. This means that once the last event accepted by the rule has been triggered, the first one returns available for triggering. To implement such a behavior, after the rule is composed as explained, a union is performed between the final and initial locations of the rule.

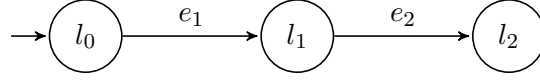


Figure 4.5: Automaton representing the statement “ $e_1$  before  $e_2$ ”

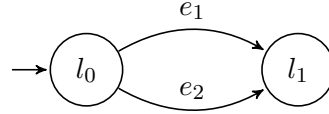


Figure 4.6: Automaton representing the statement “ $e_1$  or  $e_2$ ”

Furthermore, every rule  $r$  is associated with an integer  $P_r$  as explained in 4.3.1.  $P_r$  is assigned an integer value uniquely associated with every location of the rule, so every location  $q$  of the rule is associated with an integer  $h(q) = i$ , as such it is possible to define guards on agent automata based on the location of a rule automaton. These values are assigned by PuRSUE at design time and depend on the order of declaration of the locations, the initial location being always 0.

Events with duration  $e_d$ , defined with the keyword `duration`, used to model events that take a finite amount of time units for the agent to perform, can be included in rules as well. PuRSUE will automatically implement it as:

$$e_d \text{ before } e_{d,DONE}$$

The rule automaton so implemented will synchronize with channels of both beginning and end of  $e_d$ . This ensures that, if a designer defines a rule prescribing that picking up the coffee, event with a finite duration, happens before serving the coffee, the event of serving the coffee becomes available after the transition related to the ending of picking up the coffee ( $e_{d,DONE}$ ) has been fired.

**Example.** The automaton representing the rule `robotTask`, can be seen in Figure 4.7, where the labels are instantiated as follows: regarding synchronization labels,  $e_1$  is `takeMedicine`,  $e_2$  is `giveMedicine1`,  $e_3$  is `giveMedicine2`,  $e_4$  is `giveMedicine3`,  $e_5$  is `confirmDelivery`, while completion events (e.g. `takeMedicineDONE`) are instantiated as  $e_{i,D}$  (e.g.  $e_{1,D}$ ). Regarding update labels,  $u_0$  is  $P_{robotTask} := 0$ ,  $u_1$  is  $P_{robotTask} := 1$ ,  $u_2$  is  $P_{robotTask} := 2$ ,  $u_3$  is  $P_{robotTask} := 3$ ,  $u_4$  is  $P_{robotTask} := 4$ ,  $u_5$  is  $P_{robotTask} := 5$  and  $u_6$  is  $P_{robotTask} := 6$ .

The automaton allows one of the three `giveMedicine` events only once the event `takeMedicine` has been completed. Since the `takeMedicine` events



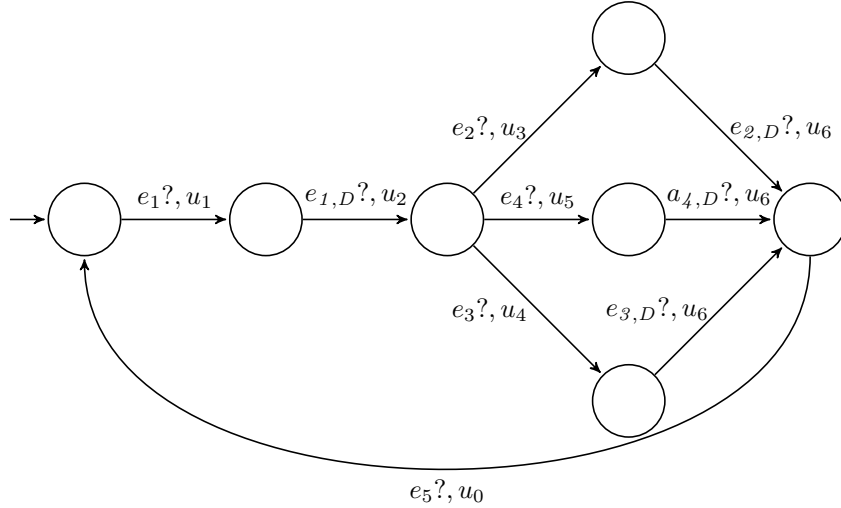


Figure 4.7: Automaton representing the rule `robotTask`

have a duration, their are modeled as two consecutive events, *takeMedicine* and *takeMedicineDONE* for each one of them. In the same way, the event *confirmDelivery* can only be triggered once one of the *threegiveMedicine* has been completed. Once *confirmDelivery* is triggered, we can see how  $P_r$  is set back to 0, value always corresponding to the initial position of the rule.  $P_r$  is then set to 1 when the first transition is triggered and so on, always keeping track of the automaton state.

#### 4.3.4 Agent Automata

An agent automaton represents an agent in the environment.

For any agent  $a$ , the corresponding automaton is composed by several elements that model the following aspects: (i) the position of  $a$ , if  $a$  is idle, (ii) the act of motion of  $a$  or the act of performing of an event and (iii) whether  $a$  is capable of triggering a certain event or not, according to the constraints defined by the user.

These aspects are modeled through these portions of the agent automata:

- **POI locations:** they model the presence of an agent in a certain POI of the environment and that the agent resides there for a non null period of time. If an agent is “mobile” (i.e., it is defined by using keyword `mobile` of PuRSUE-ML) then for every POI  $x$  in the environment, the

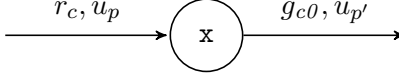


Figure 4.8: POI location with one incoming and one outgoing transition

corresponding POI location  $\mathbf{x}$  is introduced in the set of locations of the automaton. Otherwise, the automaton for an agent that is not defined with the keyword `mobile` only contains the unique POI location identified with the keyword `location`.

As already discussed in Sec. 4.2.1, every POI is an abstraction of a physical space in a realistic world that has no geometric characterization and dimension. However, modeling realistic behavior of agents requires an assumption on the physical space occupied by the POI. This assumption entails that an agent passing through a POI, while moving towards another position, takes at least a certain positive amount of time, being every POI associated with a physical location with non null size.

For this reason, all incoming transitions resets the clock  $\mathcal{C}_a(r_c)$ , and all outgoing transitions have a guard requiring  $\mathcal{C}_a > 1$  ( $g_{c0}$ ), which will be referred to from now on as “base clock guard”. This is to ensure that the system maintains a physically feasible behavior. This might happen due to the fact that uncontrollable agents are assumed to have the possibility to always act first when both a controllable and uncontrollable transitions are available; as such, an uncontrollable agent could move to the same POI of a controllable one and immediately move to another location, not allowing the controllable agent to trigger any collaborative event in the meantime, which is a clearly unfeasible behavior.

Every transition updates the value of  $\mathbf{P}_a$  according to the system explained in 4.3.1. All incoming transition to a location  $x$  sets  $\mathbf{P}_a = l(x)$  ( $u_p$ ). An outgoing transition going to the movement location modeling the movement from  $x$  to  $y$  sets  $\mathbf{P}_a = -l(x) \circ l(y)$  ( $u_{p'}$ ). The resulting automaton is reported in Fig 4.8.

**Example.** In Figure 4.9, a snippet of the automaton representing the location a agent `medBot` is presented, considering its connection to the movement location towards `door1`. The incoming transitions set the value of  $\mathbf{P}_{medBot}$  to  $l(\mathbf{a}) = 6$ , as  $l(\mathbf{a}) = 6$  and resets the clock. The outgoing transition present the

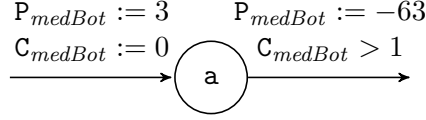


Figure 4.9: example of a POI location of agent `turtleBot`

guard  $g_{c0}$ , and it set the value of  $P_{medBot}$  to  $-63$ , as the movement location is towards `door1`, location associated with value 3 (i.e.  $l(\text{door1}) = 3$ ).

- **Movement locations:** these locations represent the robot being in the act of moving from one POI to another. Given two connected POIs  $x$  and  $y$ , the two corresponding movement locations generated are respectively `going_x_to_y` and `going_y_to_x`. Every agent set as mobile will have two movement locations per every connection defined in the PuRSUE-ML. For the location `going_x_to_y`, the incoming transition is labeled with the synchronization  $a_x2y!$  ( $a_{x,y}!$ ), while the incoming transition into the location named `going_y_to_x`, the transition will be labeled  $a_y2x!$  ( $a_{y,x}!$ ). The timing of the movement needs to be modeled, as this allows the designer to reason on different solutions depending on different topologies of the environment and speeds of the robots.

Given a mobile agent with speed  $sp$  ( the speed parameters is defined as the amount of time units necessary to cover a distance unit as discussed in Sec. 4.2.5) and two locations of distance  $dist$ , the movement between them is handled as follows.

Upon entering a movement location, the clock  $C_a$  is reset ( $r_c$ ), the location has the invariant  $C_a \leq (sp \times dist) + 1$  ( $inv_{mov}$ ), and the transition going from the movement location to the target location has a “movement duration guard”  $g_{dm}$ , defined as  $C_a > sp \times dist$ .

This ensures that the agent will spend no less than  $int \times distance$  and no more than  $(int \times distance) + 1$ . The time unit is considered a reasonable margin as it is the smallest unit of time that the model can handle.

Furthermore, all transitions coming from a POI location have the clock base guard  $g_{c0}$  For each of the two locations, the transition updates the value of  $P_a$  according to the system explained in 4.3.1. The incoming transition sets  $P_a := -l(x) \circ l(y)$ , value corresponding

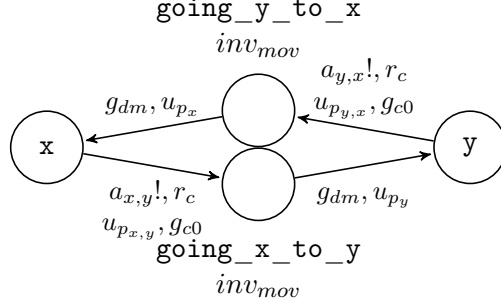


Figure 4.10: a couple of movement locations associated to a single connection POIs

to the movement location between  $x$  and  $y(u_{p_{x,y}})$ , the same happens for the other movement location, where  $P_a := g = -l(y) \circ -(x) (u_{p_{y,x}})$ . The outgoing transition from the movement locations set  $P_a =: l(x) (u_{p_x})$  and  $P_a := l(y) (u_{p_y})$ . This is to update  $P_a$  to the value of the corresponding POI.

a general example is shown in Fig 4.10.

**Example.** Taking into consideration the movement locations between  $a$  ( $x$ ) and  $door1$  ( $y$ ), for agent `medBot` the labels in Figure 4.10 are instantiated as follows:  $m!$  is `medBot_a2door1!`,  $m!$  is `medBot_door12a!`, the guards  $g_{dm}$  are  $mathhtt{C}_{medBot} > 6$  while the invariant  $inv_{mov}$  is  $\mathbf{C}_{medBot} \leq 7$ ;  $u_{p_{x,y}}$  is  $P_{medBot} := -63$ ,  $u_{p_{y,x}}$  is  $P_{medBot} := -36$ ,  $u_{p_x}$  is  $P_{medBot} := 6$  and  $u_{p_y}$  is  $P_{medBot} := 3$ .

- **Durable events locations:** they model when an agent is busy performing a certain event, as well as whether an event is available for triggering in the current state of the agent automaton.

As discussed in Sec. 4.2.5, an agent  $a$  can perform event  $e$ , in the definition of  $a$ , event  $e$  is declared after the keyword `can_do`. If  $e$  is defined as POI specific, using the keyword `location` followed by POI  $x$ ,  $e$  is only available for triggering when the automaton is at the location corresponding to  $x$ , otherwise it will be available at all POI locations.

For every POI  $x$  in which the agent can perform event  $e$ , if the event is defined with a duration of  $n$  time units, a durable event location is defined with name `doing_e_in_x`, always having one incoming transi-

tion from the POI location and one outgoing back to the same POI location.

The timing of the duration is handled like the timing of movements. The clock is reset upon entering the durable event location ( $r_c$ ), the invariant  $\mathbb{C}_a \leq n + 1$  ( $inv_{ev}$ ) checks that the agent does not stay in the durable event location longer than the defined duration of the event plus one time unit, and the “event duration guard”  $g_{de}$  defined as  $\mathbb{C}_a > n$  ensures that it does not stay less than  $n$  time units, thus effectively modeling the passing of  $n$  time units while the agent is busy performing the event.

The transition going from the POI location to the durable event location is endowed with an “event guard”  $g_{e,x}$ , defined as  $g_{e,x} = g_s \wedge g_{coll,x} \wedge g_r \wedge g_{c0}$ , where  $g_s$ ,  $g_{coll,x}$ ,  $g_r$  and  $g_{c0}$  are respectively defined as follows:

- $g_s$ , state dependency guard: in case  $e$  is associated with a state dependency, a guard is added checking whether this condition on the state is respected in order to trigger the event. For instance, if the event can only be triggered if state  $s$  is set to *true*, a guard  $\mathbb{S}_s == 1$  is added, if the event can only be triggered if state  $s$  is set to *false*, a guard  $\mathbb{S}_s == 0$  is added instead. In case several state dependencies  $s_1, \dots, s_m$  are tied to the same event, the state dependency guard will be composed as follows:  $g_s = g_{s_1} \wedge g_{s_2} \cdots \wedge g_{s_m}$ ; this models how all the state dependencies must be satisfied in order for the event to be triggered.
- $g_{coll,x}$ , collaboration guard: in case  $e$  is defined as collaborative through the use of the keyword `collaborative`, a guard checking for the presence of at least one reacting agent in the current POI  $x$  is also introduced (reacting agent for  $e$  being an agent that is defined with  $e$  in its `reacts_to` set). For instance, if event  $e$  is a collaborative event which can be performed by agent  $a$  and agent  $b$  is a reacting agent for it, the guard  $\mathbb{P}_b == l(x)$  is added to the transition going from a POI  $x$  location to the durable event location. In case several agents  $b_1, \dots, b_m$  are defined as reacting to event  $e$ , the collaboration guard will be composed as  $g_{coll,x} = g_{coll_{b_1},x} \vee g_{coll_{b_2},x} \cdots \vee g_{coll_{b_m},x}$ , thus modeling how it is sufficient for one agent to be present for the event to be triggered.
- $g_r$ , rule guard: in case  $e$  is included in a rule  $r$ , a guard is introduced ensuring that the automaton associated with  $r$  is in a location where  $e$  can be triggered. In case event  $e$  is included in

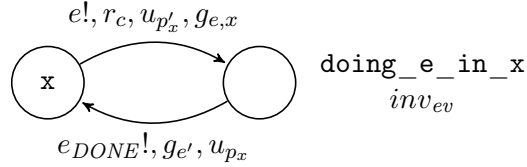


Figure 4.11: A durable event location, modeling agent  $a$  performing  $e$  in POI  $x$

several rules  $r_1, \dots, r_m$ , the rule guard will be composed as follows:  
 $g_r = g_{r_1} \wedge g_{r_2} \cdots \wedge g_{r_m}$ ; this models how all the guards must be satisfied in order for the event to be triggered.

- $g_{c0}$  clock base guard: as for every transition outcoming from any POI location, the guard  $\mathbf{C}_a > 1$  is added.

The transition going from the durable event location to the POI location is endowed with a guard ( $g_{e'}$ ) defined as  $g_{e'} = g_r \wedge g_{de}$ , where  $g_r$  and  $g_{de}$  are the rule and duration event guards previously defined.

The value of  $\mathbf{P}_a$  is updated according to the system explained in 4.3.1. The transition going from the POI location  $x$  to the durable event location will have the update  $\mathbf{P}_a := -l(x) \circ l(x) (u_{p'_x})$ , while the transition going to the POI location will have the update  $\mathbf{P}_a := l(x) (u_{p_x})$ .

Finally, the transition from the POI location to the durable event location has the synchronization label  $e!$ , while the transition going from the durable event location to the POI location is labeled  $e_{DONE}!$ . In the UPPAAL-TiGA semantics, once a channel is defined (4.3.1), only the transitions labeled with the  $!$  symbol are capable of triggering its transitions, while the ones labeled with the  $?$  appendix must synchronize with the transition if it is available. Only agents have transitions labeled with the  $!$  appendix, as such, only agents are capable of triggering transitions.

a general example is shown in Fig 4.11.

**Example.** In Figure 4.12, we focus on the location modeling the performing of event *giveMedicine1* in location *door1* for agent *medBot*. The incoming transition has the guards “ProbotTask==2”, to ensure synchronization with the corresponding rule, and “Sdoor1==1”, so to allow the triggering of the event only when the state “door1” is set to 1.

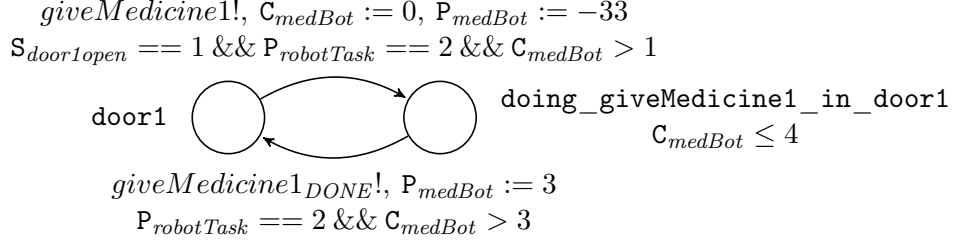


Figure 4.12: example of a durable event location for agent `turtleBot`

- **Instantaneous events self-loops:** they model whether an event that has a null duration, i.e. that is defined with no use of keyword `duration`, is available for triggering in the current state of the automaton.

As for events with duration, If  $e$  is defined as POI specific, using the keyword `location` followed by POI  $x$ ,  $e$  is only available for triggering when the automaton is at the location corresponding to  $x$ , otherwise it will be available at all POI locations. Unlike events with duration, non-location specific instantaneous events are also available for triggering in movement locations, that is, while performing movement. This models the assumption that the agent does not need to stop a movement to perform an instantaneous event.

The guards on the self-loop related to the instantaneous event  $e$  in location  $x$  are identical to the ones on the transition going from a POI location to a durable event location. The event guard  $g_{e,x}$  is defined as  $g_{e,x} = g_s \wedge g_{coll,x} \wedge g_r \wedge g_{c0}$ , where  $g_s$ ,  $g_{coll,x}$ ,  $g_r$  and  $g_{c0}$  are as defined in the previous point. Furthermore, as for every transition incoming in a POI location, they present the clock reset ( $r_c$ ), for the reasons explained earlier.

The guards on the self-loop related to the instantaneous event  $e$  in a transitional location present some differences. The movement event guard  $g_{e,x,y}$  is defined as  $g_{e,x,y} = g_s \wedge g_{coll,x,y} \wedge g_r$ , where  $g_s$ ,  $g_r$  are as defined in the previous point, while  $g_{coll,x,y}$ , in case of a movement between POI  $x$  and  $y$ , is defined as  $P_b == -l(y) \circ l(x)$ ,  $b$  being an agent reacting to event  $e$ . This models, according to the positioning rules as defined in 4.3.1, how for collaboration to take place during a movement, the two agents need to be moving between two POIs in different directions. This modeling choice is done under the assumption that the two agents, moving along the same path between two POIs

but with opposite directions, will surely meet at some point during the transition. Furthermore, the guard  $g_{c0}$  is missing, as the assumption that the base clock guard is design to model is reserved to POI locations. Finally, these transitions perform no updates to  $P_a$ . This is because self-loops do not change the location of the automaton.

A general example of a instantaneous event-self loop in a POI location Fig 4.13a, while Fig. 4.13b represents a instantaneous event self-loop on a movement location, where labels not directly related to the instantaneous event-self loops have been removed for simplicity.

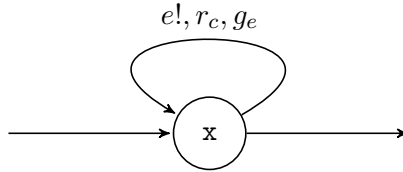
It is to be observed that, by construction, the transition exemplified in Fig. 4.13b does not enforce any check on the number of times event  $e$  might be triggered at any time instance, as there are no clock guards on it. This means that event  $e$ , without any additional guards, could be triggered an infinite amount of times without any time passing, thus blocking the evolution of the system. As such, the designer should be careful to introduce appropriate rules to ensure that this does not happen, or that the instantaneous event is a terminal condition for the scenario.

For example, let an event **beep** be introduced, instantaneous event modeling the robot emitting a noise, not location specific and not included in any rule, and let this event being assigned to an uncontrollable agent  $b$ ; in case of a reachability type of mission (e.g. controller must execute event  $e$ ), a valid strategy for  $b$  to stop the controller from winning would be to go into a movement location and trigger **beep** an infinite amount of times, never letting time pass and effectively stopping the controller from ever triggering  $e$ . This can be solved by either making **beep** location specific, or by introducing it into a rule that binds its execution with other, non-instantaneous, events e.g. “**chargeBeeper** before **beep**”.

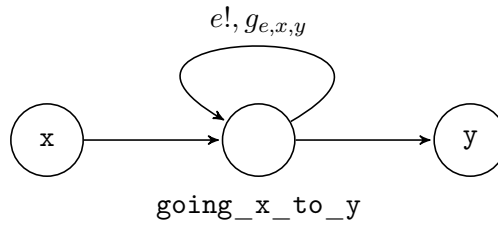
**Example.** *Figures 4.14b and 4.14a represent two instantaneous event self-loops as instantiated in the automa representing **medBot**. Event **confirmDelivery** is present on both POI locations and movement locations, as it is not defined as location specific nor with a duration. The guards only include the rule guard  $g_r$ , as the event is not included in state dependencies nor is it collaborative.*

Finally, in case the agent is not defined as controllable, the model generated is identical, however all the transitions will be set as uncontrollable.



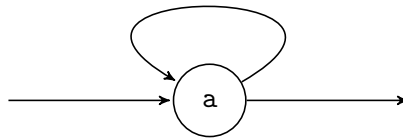


(a) Self-loop on location  $x$  dealing with the occurrence of an instantaneous event  $e$  in location  $x$



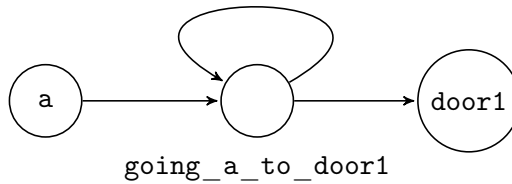
(b) Self-loop dealing with the occurrence of instantaneous event  $e$  in a movement location between  $x$  and  $y$

$confirmDelivery!, C_{medBot} := 0$   
 $P_{robotTask} == 6 \ \&\& \ C_{medBot} > 1$



(a) example of a self-loop in location  $a$  for agent `medBot`

$confirmDelivery!$   
 $P_{robotTask} == 6$



(b) example of a self-loop in a transitional location for agent `medBot`

In order to have all agents behave accordingly to all the rules defined, a synchronization between several agents is needed, feature which is not available in UPPAAL-TiGA.

As explained in 4.3, we could circumvent this problem by implementing ourselves the construct, indeed the guards as defined in the previous sections defined agents as *masters* of extended broadcast channels, rules as *blocking slaves* and state automaton and objective automata (which we discuss in 4.3.5) as *simple slaves*.

### 4.3.5 Objective Automata

An objective automaton implements an (objective) element of the kind Reaction, Event Avoidance and Execution (see Sec. 4.2.6).

Every automaton is mission specific as it depends on the objective it represents. The objective of the kind Positional Avoidance is encoded through a formula alone instead of an automaton.

The mission in a scenario is modeled by using a specific temporal formula that includes the objective formulae and that constraints the executions of the corresponding objective automata.

As with the state automaton (Sec. 4.3.2), since events included in the objective definition might be instantaneous or durable, in the following constructions if an event  $e$  is instantaneous then the considered label is  $e?$ , otherwise it is  $e_{DONE}$ ? (the effect of a durable event on the environment is determined at its conclusion).

The available missions in PuRSUE are the ones presented in Table 4.3:

- Reaction : it models the mission that, in case event  $e_1$  is triggered, event  $e_2$  must be triggered within  $n$  time units. It is translated by means of three locations *idle*, *atRisk*, *lose* and three transitions, one going from *idle* to *atRisk* labeled  $e_1?$ , one going from *atRisk* to *idle* labeled  $e_2?$  and one going from *atRisk* to *lose*. When the automaton is in state *idle*, it means that  $e_1$  hasn't been triggered, when the automaton is in *atRisk*, it means that  $e_1$  has been triggered and the system should act in order to trigger  $e_2$  before  $n$  time units have passed, finally if the automaton is in state *lose* it means that  $e_2$  has not been triggered in time and the controller lost the game. Moreover, the automaton is equipped with a clock  $C_{obj}$  that is used to measure the time elapsing between  $e_1$  and  $e_2$ . The clock  $C_{obj}$  is set to zero when the automaton moves from *idle* to *atRisk* ( $u_c$ ). The invariant  $C_{obj} \leq n$  (*inv*) of location *atRisk* ensures that not more than  $n$  time units pass between the triggering of  $e_1$  and

that of  $e_2$ , hence  $e_2$  is a proper reaction to  $e_1$  occurring by  $n$  time units; otherwise, the automaton evolves to the *lose* location, capturing a wrong execution such that the reaction to  $e_1$  has not been triggered on time. The automata is depicted in Fig. 4.15. This is combined with a formula prescribing the avoidance of location *lose*, in order to fully represent the mission.

- **Event Avoidance:** it models the mission of avoiding, at any time, the triggering of event  $e$ . It is translated by means of two locations, *idle* and *lose*, and a transition going from one to the other, labeled  $e?$ . The automaton is in location *idle* from the beginning of the execution, if the automaton is in location *lose*, it means the controller has lost the game. The automaton will go to location *lose* if  $e$  is triggered at any moment, thus capturing a wrong execution such that event  $e$  has been triggered. The automata is depicted in Fig: 4.16. This is combined with a formula prescribing the avoidance of location *lose*, in order to fully represent the mission.
- **Execution:** It models the mission of executing event  $e$ , after at least  $n$  time units have passed. the automaton is translated by means of three locations, *initialLocation*, *idle*, *win*, and two transitions, one going from *initialLocation* to *idle*, and one going from *idle* to *win*, labeled with  $e$ . The automaton is in location *initialLocation* at the beginning of the execution, in this location the triggering of  $e$  will not result in reaching location *win*. When it is in *idle*, a triggering of  $e$  will result in the transition to location *win*. Finally, if the automaton is in location *win*, it means the controller has won the game. Furthermore a guard defined as  $C_{exeObj} > n$  is set on the first transition, ensuring that at least  $n$  time units pass before the event can be triggered to reach location *win*. The automata is depicted in Fig. 4.17. In case no time units have been defined by the user, the automaton will be translated only in the two locations *idle* and *win*. It evolves with the same mechanisms just explained, having as starting location *idle*. This is combined with a formula prescribing to reach location *win*, in order to fully represent the mission.
- **Positional Avoidance:** it models the mission of two agents ,  $a_1$  and  $a_2$ , never being in the same POI or traveling over the same connection linking two POIs but towards opposite directions. It is translated by prescribing in the logic formula that, for every POI  $x$ , if  $a_1$  is in  $x$ ,  $a_2$  can not be in  $x$  or in any durable event location directly connected

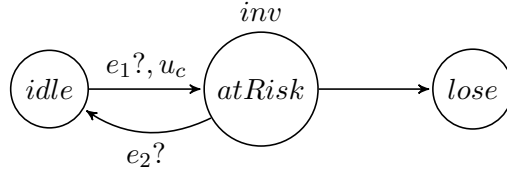


Figure 4.15: Automaton representing the Reaction mission.

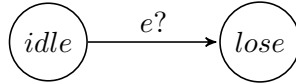


Figure 4.16: Automaton representing the Avoidance mission.

to  $x$ , that is, performing an event in  $x$ . Furthermore, for any pair of connected POIs  $x$  and  $y$ , if  $a_1$  is in the movement location between POI  $x$  and  $y$ ,  $a_2$  can not be in the movement location between  $y$  and  $x$ .

**Example.** Figure 4.18 represents the execution objective defined for the Drug Delivery case. The automaton will evolve into the location *win* when the event *confirmDelivery* is triggered. Furthermore, it needs be prescribed in the logic formula to reach location *win*. To model the positional avoidance objective, it is coded into the logic formula that the two agents can never be in the same POI at the same time. The resulting formula is depicted in Listing 4.3.

Listing 4.3– Logic formula generated for the Drug Delivery case

```

1 control : A[not((stretcher.a and medBot.a) or (stretcher.b and medBot.b)
or (stretcher.c and medBot.c) or (stretcher.d and medBot.d) or (
stretcher.medicine and medBot.medicine) or (stretcher.medicine and
medBot.doing_takeMedicine_in_medicine) or (stretcher.door1 and
medBot.door1) or (stretcher.door1 and medBot.
doing_giveMedicine1_in_door1) or (stretcher.door2 and medBot.door2)
or (stretcher.door2 and medBot.doing_giveMedicine2_in_door2) or (
stretcher.door3 and medBot.door3) or (stretcher.door3 and medBot.
doing_giveMedicine3_in_door3) or (stretcher.going_a_to_door1 and

```

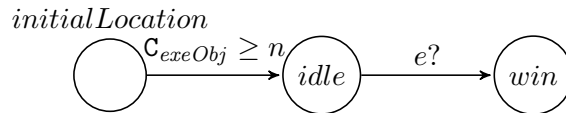


Figure 4.17: Automaton representing the Execution mission.



Figure 4.18: Execution Mission automaton in the Drug Delivery case

```

medBot.going_door1_to_a) or (medBot.going_a_to_door1 and stretcher.
going_b_to_door1) or (stretcher.going_door1_to_b and medBot.
going_b_to_door1) or (medBot.going_door1_to_b and stretcher.
going_b_to_door1) or (stretcher.going_b_to_door2 and medBot.
going_door2_to_b) or (medBot.going_b_to_door2 and stretcher.
going_door2_to_b) or (stretcher.going_door2_to_c and medBot.
going_c_to_door2) or (medBot.going_door2_to_c and stretcher.
going_c_to_door2) or (stretcher.going_c_to_door3 and medBot.
going_door3_to_c) or (medBot.going_c_to_door3 and stretcher.
going_door3_to_c) or (stretcher.going_door3_to_d and medBot.
going_d_to_door3) or (medBot.going_door3_to_d and stretcher.
going_d_to_door3) or (stretcher.going_d_to_medicine and medBot.
going_medicine_to_d) or (medBot.going_d_to_medicine and stretcher.
going_medicine_to_d) or (stretcher.going_medicine_to_a and medBot.
going_a_to_medicine) or (medBot.going_medicine_to_a and stretcher.
going_a_to_medicine)) U (reachObj.win)]
  
```

## 4.4 DSL to TGA translation

In this section, we present a PuRSUE-ML model in abstract form and then the logic used to translate this model into the TGA presented in 4.3.

### 4.4.1 Abstract Representation of the PuRSUE-ML Model

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

- $P$  is a finite set of POI defined through the keyword **poi**.
- $C$  is a multiset of elements of elements in  $P \times P \times \mathbb{N}$ , defined through the keyword **connection** and distance. A triple  $(p_i, p_j, n) \in C$  is a connection from  $p_i$  to  $p_j$  with distance  $n$ . If a connection is bidirectional then  $(p_i, p_j, n) \in C$  and  $(p_j, p_i, n) \in C$  whereas only one of the two is in  $C$  if the connection is unidirectional.
- $E$  is a multiset of elements in  $\{0, 1\} \times \mathbb{N} \times P \cup \{\#\}$  defined through the keyword **event**. A tuple  $(b, n, p) \in E$  is a collaborative event if  $b = 1$

and it has duration  $n$ . The event is bound to POI  $p$  if  $p \in P$ , otherwise  $p = \#$ .

- $F$  is a set of elements in  $P \times P \times A$  defined through the keyword **prevent**. A tuple  $(i, j, a) \in F$  is a prohibition for agent  $a$  of moving from POI  $i$  to POI  $j$ .
- $R$  is the set of regular expressions defined with keyword **rule**;
- $S$  is a set of elements in  $\{0, 1\} \times 2^E \times 2^E$  such that, 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**).
- $D$  is a set of elements in  $E \times \Psi$ , where  $\Psi$  is the set of propositional formulae that are built through conjunction and negation of formulae and state variables, that are atomic formulae in  $\Psi$ . A pair  $(e, \psi) \in D$  is state dependency constraining the occurrence of  $e$  with  $\psi$ .
- $A$  is a multiset of elements in  $\{0, 1\} \times \mathbb{N} \times P \times 2^E \times 2^E$ . A tuple  $(b, sp, x, \{e_{act,1}, \dots, e_{act,n}\}, \{e_{react,1}, \dots, e_{react,m}\})$  is an agent that is controllable if  $b = 1$  (keyword **controllable**), it is mobile if the velocity  $sp > 0$  (**mobile**), it is initially located in  $x$  (keyword **location**), it can perform the actions associated with events  $\{e_1, \dots, e_n\}$  (keyword **can\_do**) and it can react to events in  $\{e_1, \dots, e_m\}$  (keyword **reacts\_to**).
- $O$  is a set of objectives, that are elements of the form:
  - $(e_1, e_2, n)$ , with  $e_1, e_2 \in E$  and  $n > 0$ , for Reaction objectives,
  - $(e)$ , with  $e \in E$ , for Event avoidance objectives,
  - $(e, n)$ , with  $e \in E$  and  $n \geq 0$ , for Execution objectives,
  - $(a_1, a_2)$ , with  $a_1, a_2 \in A$ , for Positional avoidance and

The following section refers to an instance model of the form  $(P, C, E, F, R, S, A, O)$ .

#### 4.4.2 Transformation between the PuRSUE-ML Model and TGA

The declaration of the variables of the system is generated as follows:

1. For all the connection  $(x, y, n) \in C$  and for all the agents  $a \in A$  the corresponding broadcast channels called  $\mathbf{a\_x2y}$  is introduced, unless  $(x, y, a)$  is a prohibition in  $F$ .
2. For every event  $e \in E$ , the corresponding broadcast channel  $\mathbf{e}$  is introduced. Furthermore, if  $e = (b, n, p)$  is such that  $n > 0$ , then the channel  $\mathbf{eDONE}$  is introduced.
3. For every rule  $r \in R$ , variable  $\mathbf{P}_r$  is introduced.
4. For every state  $s \in S$ , an integer  $\mathbf{S}_s$  is introduced.
5. For every agent  $a \in A$ , clock  $\mathbf{C}_a$  and a bounded integer  $\mathbf{P}_a$  is introduced.
6. For every element  $o$  of  $O$  including a property with real-time constraints, as defined in Sec. 4.2.6, a clock  $\mathbf{C}_{obj}$  is introduced.

The logic used to generate the automata composing the TGA network is the following:

1. For every rule  $r \in R$ , the automaton  $\mathcal{A}_r$  representing the corresponding regular expression is generated. Since a rule  $r$  is expressed by means of a regular expression,  $\mathcal{A}_r$  is simply an acceptor of a regular language that can be built by using well-known procedures in the theory of Formal Languages. All the transitions of  $\mathcal{A}_r$  obtained from a rule  $r$  are synchronized with specific channels (only events in  $E$  are allowed on transitions) and perform specific updates. If  $e \in E$  is the event associated with a transition then the transition is synchronized with  $e?$ . Let  $Q$  be the set of states of  $\mathcal{A}_r$  and  $h$  be a bijection from  $Q$  to  $\{1, \dots, M_r\}$ . All the incoming transitions in a location  $q$  update  $\mathbf{P}_r$  with the update  $\mathbf{P}_r := h(q)$ .
2. For every state  $s \in S$  of the form  $(b, E_t, E_f)$ , the state automaton is endowed with  $|E_t| + |E_f|$  distinct transitions, one for each event  $e \in E_t \cup E_f$ . In particular, every transition is labeled with a synchronization guard of the form  $e?$  and the assignment  $\mathbf{S}_s := 1$ , if  $e \in E_t$ , or assignment  $\mathbf{S}_s := 0$  otherwise. Furthermore, if a transition labeled with event  $e$  is already present in the automaton (because it was present in another state), only the corresponding assignment is added to that same transition.
3. An agent automaton  $\mathcal{A}_a$  represents an agent  $a = (b, sp, x, E_{act}, E_{react}) \in A$ .

- If agent  $a$  is “mobile” (i.e.,  $sp > 0$ ) then  $Q_{\text{loc}}^a = \{q_x \mid x \in P\}$  (for every POI  $x$  in the environment, the corresponding POI location  $q_x$  is introduced in the set of locations of the automaton). Otherwise,  $Q_{\text{loc}}^a = \{q_x\}$ . The initial state of  $\mathcal{A}_a$  is  $q_x$ .
- For every  $(x, y, d) \in C$ , set  $Q_{\text{mov}}^a$  is  $\{q_{(x,y,d)} \mid (x, y, a) \notin F\}$ . Every location  $q_{(x,y,d)} \in Q_{\text{mov}}^a$  is connected with  $q_x$  and  $q_y$  by means of a transition from  $q_x$  to  $q_{(x,y,d)}$  and from  $q_{(x,y,d)}$  to  $q_y$  such that:
  - The incoming transition to  $q_{(x,y,d)}$  is endowed with the synchronization label,  $a_{x,y}!$ , clock reset  $r_c$ , the update  $u_{p_{x,y}}$  and the guard  $g_{c0}$ .
  - The outgoing transition from  $q_{(x,y,d)}$  is labeled with the guard  $g_{dm}$ , and the update  $u_{p_y}$
- Set  $Q_{\text{ev}}^a$  is  $\{q_{e,x} \mid e = (b, d, \#) \in E_1 \cup E_2 \text{ and } x \in P \text{ and } d > 0\} \cup \{q_{e,x} \mid e = (b, d, x) \in E_1 \cup E_2 \text{ and } d > 0\}$  In other words, for every event  $e \in E$  with non null duration:
  - If  $e = (b, d, \#)$  and  $e \in E_1$  then a location  $q_{e,x}$  is introduced in  $Q_{\text{ev}}^a$  for all the POI  $x \in P$  ( $e$  is not bound to a specific POI and agent  $a$  can perform it in every POI).
  - Otherwise  $e = (b, d, x)$  and  $e \in E_1$  then a location  $q_{e,x}$  is introduced in  $Q_{\text{ev}}^a$  only for POI  $x \in P$  ( $e$  is bound to POI  $x$  and agent  $a$  can perform it only in POI  $x$ ).

Every location  $q_{e,x} \in Q_{\text{ev}}^a$  have one incoming transition from the POI location  $q_x$  and one outgoing transition towards POI location  $q_x$  such that:

- The incoming transition to  $q_{e,x}$  is endowed with the synchronization label  $e!$ , clock reset  $r_c$ , clock update  $u_{p'_x}$ , and guard  $g_{e,x}$ .
- The outgoing transition from  $q_{e,x}$  is labeled with the synchronization label  $e_{\text{DONE}}!$ , the update of  $u_{p_x}$  and guard  $g_{e'}$ .
- Instantaneous events transitions model agent actions that have a null duration, i.e., that are associated with an event  $e = (b, 0, p)$  for some  $b \in \{0, 1\}$  and  $p \in \{P \cup \#\}$ . For every instantaneous event  $e \in E$ :
  - If  $e = (b, 0, x)$  then a transition from  $q_x$  to  $q_x$  is introduced. It is endowed with the synchronization label is  $e!$ , the guard  $g_{e,x}$  and the clock reset  $r_c$ .



- Otherwise, if  $e = (b, 0, \#)$ , a transition from the POI location  $q_x$  to  $q_x$  is introduced, for all the POI  $x \in P$ , and labeled with the same guard and synchronization as the ones used in the previous case.

Moreover, a transition from the movement location  $q_{x,y}$  to  $q_{x,y}$ , for all  $x, y \in P$ , is introduced. The synchronization label is  $e!$  and the guard  $g_{e,x,y}$ .

4. For every objective  $o \in O$ , the corresponding automaton as presented in Sec. 4.3.5 is generated:

- Reaction: The automaton representing  $o$  is the one depicted in Fig. 4.15, where elements  $e_1$ ,  $e_2$  and  $n$  are instantiated as defined in the tuple  $(e_1, e_2, n)$ .
- Event avoidance: The automaton representing  $o$  is the one depicted in Fig. 4.16, where element  $e$  is instantiated as defined in the only element of  $o$ :  $e$ .
- Execution: the automaton representing  $o$  is the one depicted in Fig. 4.17, where elements  $e$  and  $n$  are instantiated as defined in the tuple  $(e, n)$ .
- Positional Avoidance: Positional avoidance for two agents  $a, a' \in A$  is translated into the following two formulae (and no automaton is introduced), where  $q_x \in Q_{\text{POI}}^a$  and  $q'_x \in Q_{\text{POI}}^{a'}$  are POI locations of agents  $a$  and  $a'$ , respectively, and  $q_{x,y} \in Q_{\text{mov}}^a$  and  $q'_{x,y} \in Q_{\text{mov}}^{a'}$  are movement locations of agents  $a$  and  $a'$ , respectively, for any  $x, y \in P$ :

$$\phi_{\text{same-pos}}(a, a') = \bigvee_{x \in P} (q_x \wedge q'_x)$$

$$\phi_{\text{same-move}}(a, a') = \bigvee_{x, y \in P} (q_{x,y} \wedge q'_{y,x})$$

Formula  $\phi_{\text{same-pos}}(a, a')$  encodes the conditions that hold when the agents  $a$  and  $a'$  reside on the same POI, i.e., when the current configuration of the automata  $(q, v)$  and  $(q', v')$  is such that the locations  $q$  and  $q'$  refers to the same POI  $x \in P$ . Similarly, formula  $\phi_{\text{same-move}}(a, a')$  encodes the conditions that holds when the agents  $a$  and  $a'$  are traversing the same path between POI  $x, y \in P$  towards opposite directions.

5. The objective automata and formulae defined in Sec. 4.3.5 are used to characterize the reachability (control) game  $RG\langle A, Init, Safe, Goal \rangle$ , for TGA  $A$  obtained by combining all the agent automata, the rule automata, the state automaton and the objective automata modeling the scenario. The sets  $Safe$  and  $Goal$  are sets of configurations of  $A$  that are determined by means of the following formulae  $\phi_{Safe}$  and  $\phi_{Goal}$  encoding, respectively, the set of “safe” configurations that  $N$  has to exhibit along all the executions starting from the an initial configuration in  $Init$  and possibly leading to some “goal” (or target) configuration in  $Goal$ . Let  $o$  be an objective in  $O$  of the form  $(e_1, e_2, n)$ , with  $e_1, e_2 \in E$ , and  $n > 0$ , or of the form  $e \in E$ , or of the form  $(e, n)$ , and  $\mathcal{A}_o$  be the corresponding objective automaton; or  $o$  be a tuple  $(a, a')$ , with  $a, a' \in A$ . With abuse of notation, we write  $q \in \mathcal{A}_o$  to indicate a location  $q$  of automaton  $\mathcal{A}_o$  and  $q_{win}, q_{lose}$  to indicate locations that are called *win*, *lose* in Sec. 4.3.5.

$$\phi_{Safe} := \bigwedge_{(a,a') \in O} \neg\phi_{\text{same-move}}(a, a') \wedge \neg\phi_{\text{same-pos}}(a, a') \wedge \quad (4.1)$$

$$\bigwedge_{\substack{q_{lose} \in \mathcal{A}_o \text{ s.t. } o \in O \\ o = (e_1, e_2, n) \text{ or } o = e}} \neg q_{lose} \quad (4.2)$$

$$\phi_{Goal} := \bigwedge_{\substack{q_{win} \in \mathcal{A}_o \text{ s.t. } o \in O \\ o = (e, n)}} q_{win} \quad (4.3)$$

Formula  $\phi_{Safe}$  encodes the Positional avoidance, the Reaction and the State avoidance objectives. The two subformulae in  $\phi_{Safe}$  have the following meaning: (i) the first formula (4.1) holds if all the pair of agents  $(a, a') \in O$  do not reside in the same POI or they do not traverse the same path between two POI with opposite directions; (ii) the second formula (4.2) holds if the current location of all the automata  $\mathcal{A}_o$  for objectives of the form  $(e_1, e_2, n)$  or  $o = e$  is not  $q_{lose}$ .

Formula  $\phi_{Goal}$  encodes the Execution objectives. The formula holds when the current location of all the automata  $\mathcal{A}_o$  for objectives of the form  $(e, n)$  is  $q_{win}$ .

Finally, the reachability (control) game aims at defining a strategy such that a certain *mission formula* is satisfied by all the executions of  $A$

where the actions performed by all the controllable agents comply with the strategy.

## 4.5 Controller Generation

In order to generate the controller for the system, we need to use a model checker capable of (i) accepting models expressed as TGA, (ii) generate a controller for controllable transitions in order to satisfy the provided formula, provided one exists; we also require it, in the evolution of the system, to (iii) give priority to actions performed by uncontrollable agents over actions performed by controllable ones.

It would furthermore be desirable to have (iv) a feature allowing the designer to test the controller in a theoretical environment, in order to check for its correct functioning before deployment on the robotic platform.

Among the model checkers available, we decided to use UPPAAL-TiGA.

In order to include UPPAAL-TiGA in the PuRSUE framework and thus generate plans in an automatized manner, we take advantage of the executable file *verifytga.exe*.

Once the framework has generated the model in the UPPAAL-TiGA encoding, that is a model composed of (a) a TGA in the UPPAAL encoding named "*UPPAAL\_model.xml*" and (b) a logic property named "*UPPAAL\_model.q*", the controller is obtained with the command:

```
./verifytga -t0 UPPAAL_model.xml
```

Where *UPPAAL\_model* is the specific name given to the file containing the model.

**Example.** *The plan generated for the Drug Delivery case is composed of 5376 States and 9768 Execution Choices (Sec. 3.3), and it was generated in 32680 ms. A snippet of the plan is reported in Listing 4.4.*

*When the system is in the state defined in Line 1, that is, when all the variables have the prescribed values and all the automaton are in the prescribed states, the controller is going to wait as long as the clock conditions defined in Line 2 hold, while it is going to trigger the transition between the locations `going_medicine_to_d` and `d` of agent `medBot` when the clock conditions in Line 3 hold true.*

Listing 4.4– Snippet of the plan generated for the Drug Delivery case

```
1 State: ( states.base medBot.going_medicine_to_d nurse.room stretcher.  
    going_medicine_to_d reachObj.idle nurseBehaviour._nurseBehaviour0  
    robotTask._robotTask0 ) PnurseBehaviour=4 ProbotTask=2 PmedBot=-19  
    Pnurse=2 Pstretcher=-19 Sdoor1open=1 Sdoor2open=0 Sdoor3open=1  
2 While you are in (0<Creach && CmedBot<=14 && Cstretcher-CmedBot<-13),  
    wait.  
3 When you are in (14<CmedBot && 0<Creach && CmedBot<=15 && Cstretcher<1),  
    take transition medBot.going_medicine_to_d->medBot.d { CmedBot > 14,  
    tau, PmedBot := 9, CmedBot := 0 }
```

# Chapter 5

## Implementation

In this section we present the implementation of PuRSUE. First, Section 5.1 describes the tool support provided by PuRSUE in designing a robotic application. Then, Section 5.2 presents the run-time environment used to control the robot according to the run-time controller generated by PuRSUE.

### 5.1 Design-Time Support

The design time workflow of PuRSUE is depicted in Figure 5.1, and is composed of the following elements:

- PuRSUE Eclipse Plug-In: Figure 5.2 presents a screenshot of the interface provided by the PuRSUE-ML plug-in. The plug-in provides static grammar validation (i.e. validation of the grammar while writing it) on the basic constructs of the grammar.
- PuRSUE main pc-side: it coordinates the interaction between all the other components and aims at generating the plan that will be executed by the robot and sending it to the Turtlebot using REST2ROS (component that acts as a bridge between the run-time and design-time environments.) [31]. This component takes as input the grammar as a *.pur* file, runs in order the parsers presented in the upcoming sections and finally sends the run-time components to the TurtleBot.
- PuRSUE sender: it sends the run-time components to the TurtleBot.
- PuRSUE-ML 2 UPPAAL: it takes as input the model in PuRSUE-ML and generates the TGA model according to the transformation described in 4.4. The generated model is expressed as an *.xml* file representing

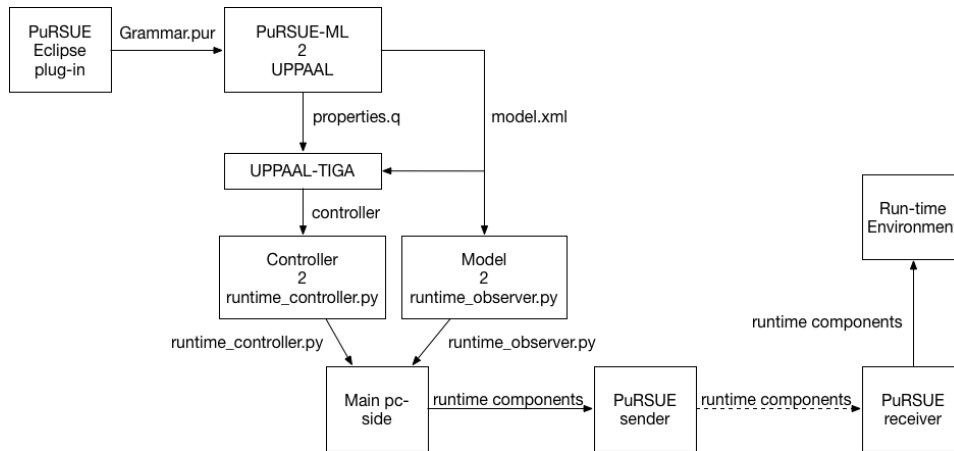


Figure 5.1: PuRSUE: design time workflow

```

//locations
poi "a"
poi "b"
poi "c"
poi "d"

//connections
connect a and b distance 10
connect b and c distance 10
connect c and d distance 10
connect d and a distance 10

//events
event "catch" collaborative

//agents
agent "police" controllable mobile 1 location a can_do catch
agent "thief" mobile 2 location c reacts_to catch

//objectives
reach_objective: do catch
  
```

Figure 5.2: ScreenShot of the Eclipse Plug-In

the automata in UPPAAL-TiGA language and a *.q* containing the properties, expressed as temporal logic; indeed as was explained in 4.3.5, the objective of the system is translated into a temporal property which needs to be placed in a different file.

The current implementation of the parser is missing the following constructs presented in Chapter 4:

- Keyword `prevent`.
  - Keyword `unidirectional`.
  - State Dependencies can only handle a single state, rather than a Boolean expression.
- 
- UPPAAL-TiGA: It takes as input the TGA model and property and generates a controller for the problem at hand, if one exists, as explained in Sec. 4.5.
  - Controller 2 `Runtime_Controller.py`: it takes as input the controller generated by UPPAAL-TiGA (expressed through the formalism explained in Sec. 3.3 and generates the python script of the controller that will be executed at run-time, called *runtime\_controller.py*.
  - Model 2 `Runtime_Observer.py`: it takes as input the model of the system as TGA represented in the *.xml* used as input by UPPAAL-TiGA and generates a component that will monitor the environment at run-time, called *runtime\_observer.py*. As described in Sec. 4.5, the controller constrains the behavior of the robot depending on the behavior of the other agents in the environment and the resulting state of the system, for this reason run-time monitoring is necessary.
  - PuRSUE Receiver: it takes care of the reception of *runtime\_controller.py* and *runtime\_observer.py* and places them in the appropriate folders in the run-time platform, i.e. the TurtleBot.

## 5.2 Run-Time Support

The run-time architecture of PuRSUE is presented in Figure 5.3. Its components are detailed in the following.

- **Observer Node.** It updates the current instance of the TGA used by the controller based on the occurrence of events within the system. It

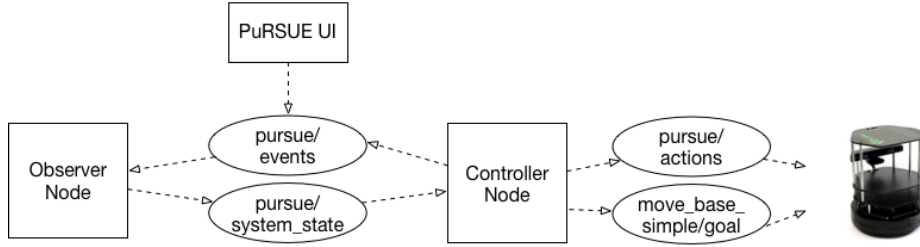


Figure 5.3: Run-time architecture

does so by executing the script *runtime\_observer.py* generated during design time (Sec. 5.1). The observer is subscribed to the topic “*pursue/events*” and publishes to the topic “*pursue/system\_state*”. The current configuration of the model is updated by means of the events that the Observer can read in “*pursue/events*”. When an event is published on the topic “*pursue/events*”, the automata modeling the system are updated accordingly, as well as the clocks of the system. The updated state is then published to the topic “*pursue/system\_state*”. Note that the “*pursue/system\_state*” topic has a queue of one, this ensures that any subscriber to the topic can only read the latest published state.

The node implements at runtime the automata as defined in 4.3 using the python library “transitions” [38].

In a system composed of the network of automata  $A$  and their respective clocks  $C_a$ , upon the reception of the event  $e$ , the following happens:

```

 $T_{elapsed} \leftarrow (T_{current} - T_{start})$ 
 $T_{start} \leftarrow T_{current}$ 
for all  $a \in A$  do
   $a.trigger(e)$ 
  if  $a.needReset$  then
     $C_a \leftarrow 0$ 
  end if
   $C_a \leftarrow C_a + T_{elapsed}$ 
end for

```

Where the function  $a.trigger$  triggers the evolution of the automaton if the triggering of event  $e$  is available, and the Boolean  $a.needReset$  informs the system of whether the later transition caused a reset to that automaton clock.

- **Controller Node:** It defines the operating logic of all the controllable



components, through the triggering of actions as a consequence of the of the current state and the strategy it is implementing. It does so by running the received component *runtime\_controller.py*, as well as *pursue\_lubrary.py*, *exeggutor.py*, *action\_sender.py*, *move\_command\_sender.py* and *transition\_sender.py*, which interact as shown in Fig. 5.4.

The controller node reads the state of the environment, chooses the action to take according to the strategy generated by PuRSUE and sends the corresponding command to the appropriate topics. The controller node is subscribed to the topic “*pursue/system\_state*”, as well as “*pursue/events*”, the latter only to read the start command at the beginning of the execution. It publishes on the topics “*pursue/events*”, “*pursue/actions*” and “*move\_base\_simple/goal*”.

The controller node reads the system state from the topic “*pursue/system\_state*”. according to the system state and clock values, it selects whether the robot should perform an action or wait, and if so, how long it should wait. This is done according to the strategy that is obtained by UPPAAL-TiGA, discussed in Sec. 4.5. It also takes care of the interface between the events and POIs as defined at high level in the PuRSUE-ML and their low level implementation, that is, the association of a POI as a set of coordinates in the environment and the association of the triggering of an event with a string that signals to the appropriate actuator an action to perform. When an action has to be performed, the controller node sends the appropriate message either to the “*move\_base\_simple/goal*” topic, in case of movement actions, or to “*pursue/actions*” topic, in case of any other actions.

Furthermore, all events triggered by the controller are sent to the topic “*pursue/events*”. While events on this topic should ideally be published by sensors that detect the events in the system in order to have a real feedback controller; we have them as published by the controller as a system capable of detecting events in the environment was not available at the time of implementation of this thesis.

The node is composed of the following major components, depicted in the class diagram in Figure 5.4:

1. *Runtime\_controller.py*: this block is generated at design-time from PuRSUE. It implements the controller generated as shown in Section 4.5. It reads from the topic “*pursue/system\_state*” the state of the system, selects whether to wait or trigger the execution of an event and then invokes *exeggutor.py* to send the trigger to

the appropriate channels.

2. *Pursue\_library.py*: this library provides the standard functions used by the controller.
  3. *Exegutor.py*: it takes as input the events to be triggered as sent from the *runtime\_controller.py* and handles their deployment to the robot. It is the interface between the events as modeled in PuRSUE-ML and the robot's API, as such it handles the association of POIs with their correspondent location, as well as the actions with their actuators. A designer wishing to implement the system on different hardware should change this block accordingly.
  4. *Move\_command\_sender.py*, *action\_sender.py*, *transition\_sender.py*: they handle the composition of the messages and the publishing to the appropriate topics.
- **PuRSUE UI**: It allows the developer to read from all the topics relevant to the system, as well as sending signals to the topic "*pursue/events*" in order to emulate the triggering of uncontrollable events. It allows the designer to select the topics it desires to read and either start a predetermined sequence of timed events or select an event to be triggered during the execution time.

Figure 5.5 is a sequence/communication diagram referring to the following scenario/use-case:

1. The system is started from the UI.
2. The Observer publishes the initial state.
3. The controller node enters a state, waits and then pings the observer to check that the state hasn't changed during the wait.
4. The controller node sends the command to move to a target location to the Turtlebot.

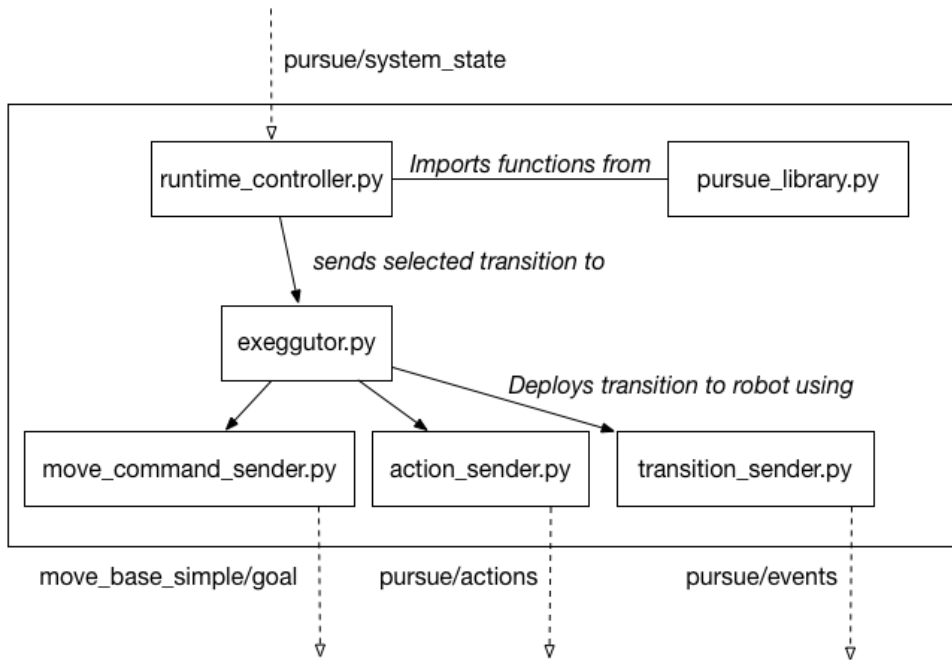


Figure 5.4: modules of runtime controller

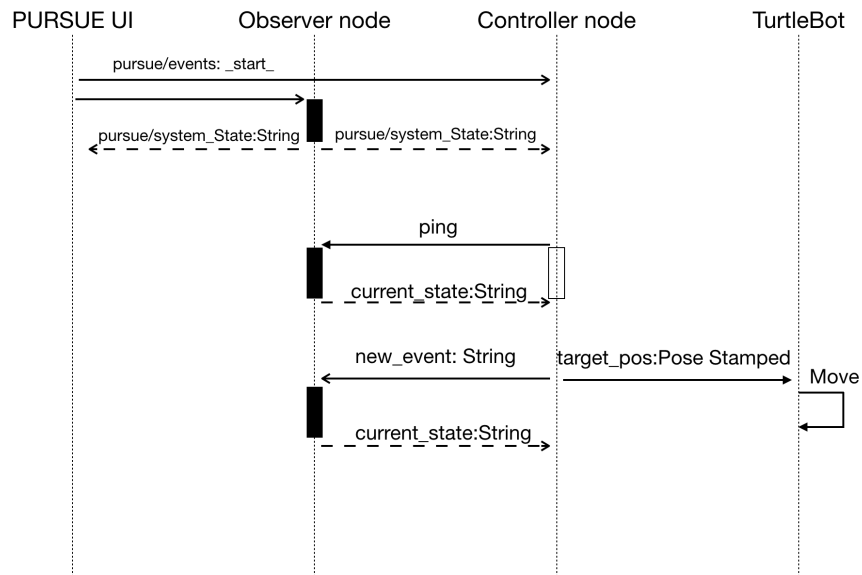


Figure 5.5: run-time communication diagram

## Chapter 6

# Evaluation

In this chapter we evaluate PuRSUE by answering the following research questions:

- **RQ1:** *does PuRSUE offer a more compact way than TGA to model RAUA applications?* We check if PuRSUE allows modeling (complex) RAUA applications and variations on them, in presence of uncontrolled agents. We also compare the size of the PuRSUE-ML model compared to the size of the generated TGA.
- **RQ2:** *how does PuRSUE support designers in generating controllers for RAUA applications?* We would like to evaluate whether PuRSUE allows to effectively compute the run-time controllers for the considered applications. Furthermore, we would like to estimate the development time that is saved for designing controllers through the usage of PuRSUE. To do so we evaluate the time taken to generate compute the controller by PuRSUE. We report the size of the generated controller and show that PuRSUE allows easily generating complex controllers.
- **RQ3:** *is the control strategy generated by PuRSUE effectively implementable on actual robots?* We deploy controllers generated by PuRSUE on an actual robot and check whether the robot behaves as expected.

To answer these questions we considered three RAUA applications: Catch the Thief (*CT*), Work Cell (*WC*) and EcoBot (*EB*), further described in Section 6.1. The applications considered were inspired by case studies found in the literature, respectively in [24], [3] and [46]. From these applications, 13 variants (scenarios) were extrapolated, by changing the specifications of

the application (6 for *CT*, 3 for *WC* and 4 for *EB*). These scenarios are used in Sec. 6.2, 6.3, 6.4 to answer respectively **RQ1**, **RQ2** and **RQ3**.

## 6.1 Scenarios

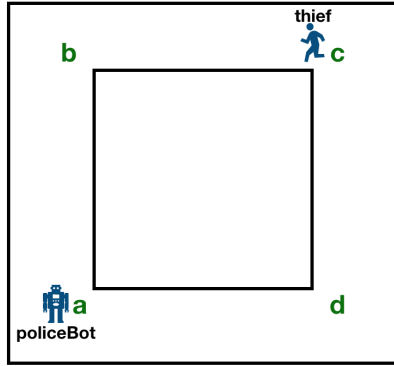
### 6.1.1 Catch the Thief (CT)

A robot-cop (`policeBot`) and a human (`thief`) are both located in an 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.

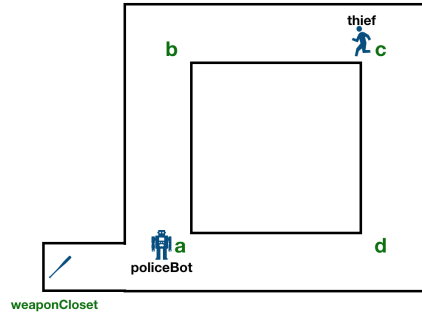
Table 6.1: Variations of the Catch the Thief application.

Sc.	Env.	Description
<i>CT1</i>	E1	Base scenario
<i>CT2</i>	E2	The base scenario is enriched with an additional constraint: the robot needs first to pick up his weapon in a weapon closet before catching the thief.
<i>CT3</i>	E1	There are two ( <i>CT3a</i> ) or three ( <i>CT3b</i> ) <code>policeBot</code> robots that are active in seizing the <code>thief</code> . Moreover, the <code>policeBot</code> will move at the same speed as the <code>thief</code> .
<i>CT4</i>	E3	The base scenario but in a more complex environment.
<i>CT5</i>	E4	The <code>thief</code> robot can steal objects, located in three locations( <code>a</code> , <code>b</code> and <code>d</code> ) and then escape through either a <code>window</code> or the <code>stairs</code> . The goal of the <code>policeBot</code> robot is to catch the <code>thief</code> before the <code>thief</code> leaves the area after a successful theft.
<i>CT6</i>	E4	The same scenario as <i>5</i> where however controllable and non-controllable agents play an exchanged role. The controllable robot has to steal from the office, while an uncontrollable security agent has to stop it. The speeds of the two agents are set equal so that the thief objective is achievable.

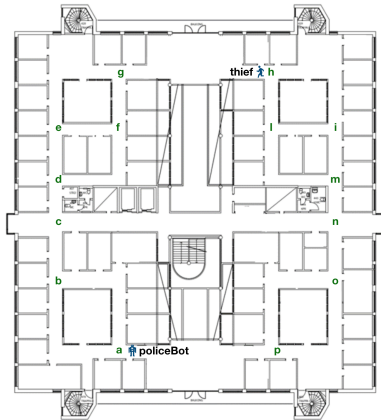
The robot is assumed to be faster than the human, otherwise the scenario is trivially unfeasible and no strategy exists for the robot. In such a case, PuRSUE would not be able to generate a controller for it; indeed, the key



(a) A graphical representation of the environment E1 used in scenarios CT1 and CT3.



(b) A graphical representation of the environment E2 used in scenario CT2.



(c) A graphical representation of the environment E3 used in scenario CT4.



(d) A graphical representation of the environment E4 used in scenarios CT5 and CT6.

Figure 6.1: Graphical representation of the environments used in the different scenarios.

variables that the designer needs to take into consideration when reasoning on the possibility of the controller to achieve its goal, for this application, are the topology of the environment and the relative speeds of the two agent.

The environments taken into consideration are the following (Fig. 6.1):

- E1: A room of the Jupiter building of the University of Goteborg that

is surrounded by a corridor is considered as the environment in which the application is deployed (Figure 6.1a).

- E2: The same room of the Jupiter building as in E1, with the addition of a weapon closet located next to it (Fig. 6.1b).
- E3: The third floor of the Jupiter building of the University of Goteborg is considered (Fig. 6.1c).
- E4: The same floor of the Jupiter building as in E3, with the addition of windows and stairs in the model (Fig. 6.1d).

The variations of this scenario considered in this work are reported in Table 6.1, 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.

### 6.1.2 Work Cell (WC)

A work cell, in its simplest form, 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, then jigs are to be screwed in it, then the box is put back on the conveyor belt. This must be done within a time limit set by the factory. A robotic assistant (**asisstBot**) is provided to the human to support him in performing said 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 overtake any task that human delays in performing. To tackle this issue, one must design a controller taking into consideration all the possible ways in which controllable and uncontrollable agents might cooperate, which is in general a non-trivial task. The key parameter the designer needs to take into consideration in this application is the duration of the tasks to be performed in respect to that of the time limit assigned by the factory to the work cell.

The variations of this scenario considered in this work are reported in Table 6.2, where column **Sc.** is the acronym identifying the scenario and **Description** is a short summary of what characterizes this variation.

Table 6.2: Variations of the Work Cell application.

Sc.	Description
<i>WC1</i>	We assume the presence of a structure where the box can be placed for operation of screwing. As such, a single agent could independently perform all the three tasks required from the station
<i>WC2</i>	We consider that, if an agent picks up the box, it will be considered busy holding it until the box is put down, this implies that another agent will have to screw the jigs in. There can be one ( <i>WC2a</i> ) or two ( <i>WC2b</i> ) robots.

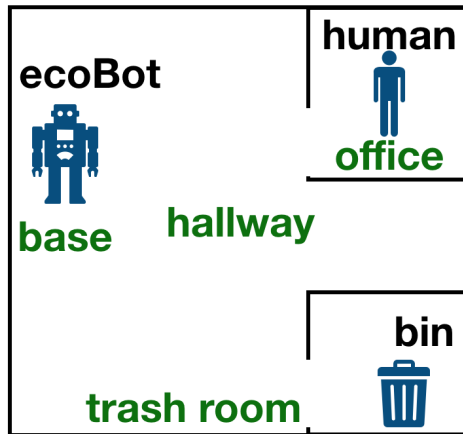


Figure 6.2: A graphical representation of the EB example

### 6.1.3 EcoBot (EB)

A robot (*ecoBot*) has the task of collecting the trash from an office (*office*) and throwing it into the trash bin (*bin*), located in a separated area (*trash room*). The robot is initially located in another room, *base*. The three aforementioned locations are connected through the hallways of the building (*hallway*). A human (*human*), located in *office*, can activate the robot, hence, starting a cleaning task in *office*. Upon calling, the robot is expected to collect the trash from the office withing a defined amount of time units. The robot can only carry one piece of trash at the time. Finally, it is assumed that any two consecutive requests from the human are always separated by a positive time delay.

The key variables of this scenario are represented by the topology of



the environment and the duration of events, as compared to the time limit to achieve the mission as specified by the designer and the positive delay between two requests from the human.

The scenario is set in a portion of a floor of the Jupiter building of the University of Goteborg, depicted in Figure 6.2.

The variations of this scenario considered in this work are reported in Table 6.3, where column **Sc.** is the acronym identifying the scenario and **Description** is a short summary of what characterizes this variation.

Table 6.3: Variations of the EcoBot application.

<b>Sc.</b>	<b>Description</b>
<i>EB1</i>	The base scenario as presented
<i>EB2</i>	The bin can be moved in the environment
<i>EB3</i>	The human must specify whether he/she needs to throw paper or plastic, and the bot should go to the correspondent bin. The robot can hold one piece of trash per type.
<i>EB4</i>	A combination of <i>EB2</i> and <i>EB3</i> . There are two distinct bins for paper and plastic and both can be moved in the environment

## 6.2 Modeling support (RQ1)

To answer **RQ1**, we evaluated the ability of PuRSUE to model the scenarios depicted in Sec. 6.1. We checked whether PuRSUE-ML was able to describe all the considered scenarios (Sec. 6.2.1). To estimate the effort needed for using PuRSUE-ML w.r.t. manually developing the model, we compared the number of constructs used to model the RAUA application in PuRSUE-ML and the number of states, transitions and variables that are necessary to model our problem using TGA (Sec. 6.2.2).

### 6.2.1 Effectiveness in Encoding the Considered Scenarios

In this section we show how we could model the scenarios using PuRSUE-ML.

The *CT* scenarios were encoded as follows:

- *CT1*: the environment is modeled through four POIs (a, b, c and d). The agents acting in it are modeled through the two agents `policeBot`

and **thief**. The act of catching is modeled through a collaborative event **catch**, which has **policeBot** as acting agent and **thief** as reacting agent. Finally, the mission of the control system is modeled with the **reach\_objective: do catch** construct. This encoding of the mission event is kept identical for all the following scenarios if not specified differently. The encoding is reported in Listing 7.1.

- *CT2* : the environment is modeled through the same four POIs as *CT1*, with the addition of the **WeaponCloset** location, which is connected to **a**. The agents are the same of *CT1*. The additional constraint is implemented in two different ways: through a rule (*CT2a*), and through states (*CT2b*). The first implementation is done by enforcing that the event of catching the thief (**catch**) can only be triggered after the event of picking up the weapon (**pickUpBaton**) has been triggered. In the second implementation, a state is added (**hasBaton**), this state, initially false, is set to true if the event **pickUpBaton** is triggered. Furthermore, a state dependency is defined, ensuring that the event **catch** can only be triggered when the state **hasBaton** is set to true. The encodings are reported in Listings 7.2 and 7.3.
- *CT3*: The environment is modeled as in *CT1*. The agents are the same of *CT1*, with the difference that two (*CT3a*) or three (*CT3b*) **policeBot** agents are declared. Moreover, the **policeBot** will move at the same speed as the **thief**. The encodings are reported in Listings 7.4 and 7.5.
- *CT4*: The environment is modeled by defining POIs in every location in which an agent could change direction on the map. The rest of the definition is identical to that of *CT1*. The encoding is reported in Listing 7.6.
- *CT5*: the environment is modeled as in *CT4*, with three additional POIs: **window1**, **window2** and **stairs**, located as shown in Figure 6.1d. To model the capability of the thief to enter or exit the building, state **away** is added; when **away** is *true*, **thief** is outside the building, when it is *false*, **thief** is inside. Accordingly, location specific events **leave1**, **leave2** and **leave3** have been defined to change the state of **away** to *true*, and location specific event **enter** has been defined to change the state of **away** to *false*.

To model the capability of the thief to steal objects in the three locations, three location specific events have been defined (**steal1**, **steal2** and

`steal3`).

These events are included in state dependencies, stating that it is only possible for `thief` to perform the three `steal` events if `away` is *false*.

To model the objective of the control system, the event `stolen` is introduced. Event `stolen` represents the act of the thief to successfully steal an object and leave the floor. This event is included on the rule `stealing`, which prescribes that after one of the three `steal` events has been performed, either event `stolen` or event `catch` can be performed. Furthermore, `stolen` can only be triggered when `away` is *true*, while `catch` can only happen if `away` is *false*. Finally, the construct used to define the mission is `objective: avoid stolen`. The encoding is reported in Listing 7.7.

- *CT6*: This scenario is modeled as *CT5*, with the difference that `thief` is the controllable agent and `policeBot` the uncontrollable one. The speeds of the two agents are set equal so that the objective is achievable. Finally, the mission is defined with the construct `reach_objective: do stolen`. The encoding is reported in Listing 7.8.

The *WC* scenarios can be encoded as follows:

- *WC1*: As the environment is not relevant in the considered example, it is modeled as a single POI (`station`). The agents in the model are `human` and `assistBot`. One event is defined for every task (`pickUpBox`, `screw` and `putDownBox`), as well as an event modeling the arrival of a new box (`newBox`). The three task related events are assigned to both agents as all of them should be able to perform all tasks, while it is up to the human to signal the arrival of a `newBox`. The rule `workFlow` describes the expected sequence of events. Finally, the mission is defined with the construct `objective: if newBox then putDownBox within 30`, thus ensuring that the box is put back on the conveyor belt within 30 seconds of its arrival. The encoding is reported in Appendix (Listing 7.9).
- *WC2*: To model one of the agents being busy, we need to distinguish at a modeling level which agent performs every action. As such, events are defined for every task and for every agent (e.g. for `human`, `pickUpBoxH`, `screwH` and `putDownBoxH`). Furthermore a state for every agent is added, (e.g. `busyH`) modeling whether an agent is busy holding the box or not. This state is initially *false*, becomes *true* if the agent picks up the box,

and *false* if the agents puts it back down. One state dependency per agent ensures that the act of screwing jigs can only be done if the agent is not busy (e.g. `screwH only_if busyH is_false`). For every agent, a rule is defined ensuring that they only put down a box after they have picked it up and vice-versa (e.g. `pickUpBoxH before putDownBoxH`). The rule `workFlow` is similar to the one presented before, but considering that any agent performing the event should forward the workflow, and with the addition of an event, `done`, used to signal that the procedure is complete (as there are several `putDownBox` events). Finally, the mission is defined with the construct `objective: if newBox then done within 30`, thus ensuring that the box is put back on the conveyor belt within 30 seconds of its arrival. The scenario is implemented with one (*WC2a*) and two (*WC2b*) robots. The encodings are reported in Listings 7.10 and 7.11.

The *EB* scenarios can be encoded as follows:

- *EB1*: The environment is modeled with 3 POIs (`room`, `base`, `trash`) all connected to a fourth POI (`hallway`). The agents are `human`, `ecoBot` and `bin`. `Bin` is a non mobile agent located in `trash`. The events defined to describe the system are `callBot`, to model when `human` calls for a cleaning task in the `office`, `getTrash`, to model the act of the robot of picking up trash, and `throwTrash`, a collaborative event having as acting agent the robot and reacting the bin, to model the act of throwing the trash. The event `officeClean` is also defined, modeling the robot notifying that it has completed its task. The behavior of the system is modeled through the following rules. Rule `getTrash`, states that the events of calling the bot (`callBot`), is followed by that of the robot picking up the trash (`pickUpTrash`) and confirming the completion of the mission (`officeClean`). Rule `TrowingTrash` states that after `getTrash` is triggered, `throwTrash` must be triggered before `getTrash` is available again for triggering, this models how the robot can only hold one piece of trash at the time. To model the delay between consecutive summonings of the robot, i.e. triggering of the event `callBot`, an event with duration as long as the requested delay is defined (`wait`); the rule `makingTrash` is also introduced, which states that event `wait` must be triggered between two consecutive triggerings of `callBot`.

The mission is defined with the construct `objective: if callBot then officeClean within 20`, thus ensuring that the trash is retrieved

within 20 second of calling the robot. The encoding is reported in Listing 7.12.

- *EB2*: The encoding is identical to the one of *EB1*, with the uncontrollable agent `bin` set as mobile, this allows to model the agent being moved around by other entities. The encoding is reported in Listing 7.13.
- *EB3*: The presence of two bins is modeled by the presence of two non-mobile bin agents (`paperBin` and `plasticBin`). The state (`ispaperOrPlastic`) is introduced. It is *true* if the new event `isPaper` is triggered, and *false* if the new event `isPlastic` occurs; these events model `human` selecting whether he/she needs to throw paper or plastic. These events are added to the rule `makingTrash`, forcing `human` to select a type of trash before summoning the robot. Furthermore, all the rules and events explained in *EB1* are extended to cover two different types of trash (e.g. instead of only having `getTrash` two events are defined, `getPaperTrash` and `getPlasticTrash`). Finally, a state dependency is set so that the robot can trigger `getPaperTrash` only if `ispaperOrPlastic` is *true* and `getPlasticTrash` only if `ispaperOrPlastic` is *false*. The encoding is reported in Listing 7.14.
- *EB4*: The encoding is identical to the one of *EB3*, but with the two bin agents set as mobile. The encoding is reported in Listing 7.15.

We can conclude that we were able to model all the considered scenarios through PuRSUE-ML.

### 6.2.2 Assessing the Design Effort Saved by the Usage of PuRSUE-ML

To assess the effort of using PuRSUE-ML in real applications compared with manually developing the model, we compared the number of constructs used to model the RAUA application in PuRSUE-ML and the number of states, transitions and variables that are necessary to model our problem using the TGA. Specifically:

- The size of the description of the scenario in PuRSUE-ML, is measured by computing the sum of the constructs (locations, connections, events, rules, states, state dependencies, agents and objectives) used in the PuRSUE-ML specification.

- The size of the TGA is measured by computing the sum of locations, transitions and number of variables (clocks and Integers) of the TGA specification. Note that, in our case the TGA specification is obtained by using the PuRSUE-ML to TGA procedure presented in Section 4.4.

Table 6.4: comparison of scenarios to answer **RQ1**

	PuRSUE-ML	UPPAAL-TIGA			
scenario	constructs	variables	locations	transitions	total
<i>CT1</i>	12	4	26	45	75
<i>CT2a</i>	16	4	36	61	101
<i>CT2b</i>	18	5	35	62	102
<i>CT3a</i>	13	6	38	73	117
<i>CT3b</i>	14	8	50	101	159
<i>CT4</i>	36	4	98	185	287
<i>CT5</i>	56	7	121	291	419
<i>CT6</i>	56	6	121	291	418
<i>WC1</i>	9	6	19	23	48
<i>WC2a</i>	19	10	31	44	85
<i>WC2b</i>	26	14	42	63	119
<i>EB1</i>	19	10	36	40	86
<i>EB2</i>	19	10	45	52	107
<i>EB3</i>	28	14	49	62	125
<i>EB4</i>	28	14	63	86	163

The presented scenarios are confronted according to these parameters in Table 6.4.

The size of the corresponding TGA specification ranges from a minimum value of 48 to a maximum value of 419. PuRSUE allows the designer to define very complex automata through the use of a few lines of code, ranging from 9 (19% of the size of the PuRSUE-ML model) to a maximum of 56 (13% of the size of the PuRSUE-ML model).

It is worth noting how 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. Furthermore, the proposed language allows handling different scenarios with simple changes. Consider for example the scenarios *EB1* and *EB2*, a simple

change in one of the keywords allows the designer to model two conceptually very different scenarios. The same task would require the addition of 9 states and 12 transitions in the TGA.

We can conclude that PuRSUE is effectively providing a more compact way to formally model RAUA applications compared to a manual encoding in TGA.

### 6.3 Automatic Controller Generation (RQ2)

To evaluate whether PuRSUE allows effectively computing the run-time controllers for the considered RAUA applications, we considered the scenarios presented in Section 6.1 and evaluated whether PuRSUE was effectively able to compute the run-time controllers. To estimate the development time that is saved for designing controllers through the usage of PuRSUE we evaluated the time taken to generate the controller by PuRSUE and report the size of the generated controller to provide an estimation on how complex they would be to be manually developed. The time required to generate the controller includes the time needed to (**Step1**) translate the PuRSUE-ML specification into a TGA as specified in Section 4.4 and to (**Step2**) compute the run-time controller from the TGA as specified in Section 4.5. The time needed to perform **Step1** and **Step2** is indicated in the following as **time m** and **time c**, while the total time needed to compute the controller is indicated as **time**. These three columns report the time elapsed during the corresponding step in milliseconds. The size of the generated controller is obtained by computing the sum of the number of states (**states**) and execution choices (**when**) of the computed controller. The total size of the controller, computed as the sum of the two, is reported in **size**. Table 6.5 contains the results of our experiment.

The average size of the generated controller is 4603, while the average time required for computing the controller is 1083 milliseconds. The capability of the designer to simply reason on complex temporal properties in RAUA scenarios is a key feature of PuRSUE. To show this we use the *EB* scenarios. Assuming all other variables to be set (that is the distances between POIs, their connections, the speed of agents and duration of all other events), the value assigned to the positive delay between requests from the human (i.e. the duration of the event **wait**) determines whether the robot is capable of fulfilling the task or not; indeed if it is set too low, the robot will not have time to throw the trash in the bin between request thus making the scenario unfeasible. To quantify the exact amount of time units to assign to

Table 6.5: comparison of scenarios to answer **RQ2**, time reported in milliseconds

scenario	time m	states	when	size	time c	time
<i>CT1</i>	1231	164	194	358	83	1314
<i>CT2a</i>	1639	588	751	1339	207	1846
<i>CT2b</i>	1238	617	800	1417	240	1478
<i>CT3a</i>	1238	<b>X</b>	<b>X</b>	<b>X</b>	unfeasible	<b>X</b>
<i>CT3b</i>	1319	2301	3022	5323	815	2134
<i>CT5</i>	1307	7989	9005	16994	915	2222
<i>CT6</i>	1018	9547	12896	22443	2399	3417
<i>WC1</i>	963	10	6	16	58	1021
<i>WC2a</i>	985	<b>X</b>	<b>X</b>	<b>X</b>	unfeasible	<b>X</b>
<i>WC2b</i>	1015	16	14	30	78	1093
<i>EB1</i>	1020	77	82	159	86	1106
<i>EB2</i>	1002	887	987	1874	6850	7852
<i>EB3</i>	1021	332	348	680	182	1203
<i>EB4</i>	1287	-	-	-	out of memory	-

the duration of `wait` in order to make the scenario feasible, the PuRSUE framework was run several times with different values assigned to said variable in the PuRSUE-ML. In *EB1*, the minimum duration of event `wait` required is 37. In *EB2*, the minimum duration of event `wait` required varies according to the speed at which we assume the bin is moving. If we set it to 2, the duration of `wait` needs to be 60 in order for the scenario to be feasible, while if we set it to 5, the minimum `wait` is 65, this is because the robot in the worst case scenario needs to wait for the bin to finish transiting before it can throw the trash in it, a slower bin will lead to a longer `wait`. In *EB3*, the minimum duration of event `wait` required is 36. The time unit of difference when compared to *EB1* is due to the fact that the human needs to select the type of trash before calling the bot, thus intrinsically allowing it one extra time unit to throw the trash and come back. The encodings reported in 7 used for the evaluation of the framework are modeled with the aforementioned values. PuRSUE allows designers to easily perform this type of temporal reasoning, which is highly non-trivial in an arbitrary complex environment, through the change of a few lines in the PuRSUE-ML.

No plan was generated for *CT3a*, this can be explained by the fact that



**thief** can wait indefinitely until a robot tries to reach it at its location, at that point, **thief** can leave said location as the robot prepares to catch it (as we remember, it is a modeling assumption that every agent needs to wait a time unit before performing any even upon arrival in a location). Furthermore, given the topology of the environment, all POIs are connected to two other POIs, as a consequence as one of the two robots goes to catch it, one of the two connected POIs will always be free. This is solved in *CT3b*, where three robots are provided to accomplish the task, this will allow two robots to stay in both the POIs thief could move towards, while the third goes to the POI it is located at.

No plan can be generated for *WC2a* either. Similarly to the aforementioned scenario, since the robot depends on the human to fulfill the task, and the human could decide to indefinitely wait, the scenario results unfeasible. In *WC2b* instead, if the human decides to no perform a task, the other robot can perform it, thus ensuring that the mission is always achievable regardless of the strategy of uncontrollable agents.

Both the previous problems could be solved through the implementation of a feature to force the uncontrollable agents to move from a location (or perform an event) within a given amount of time units, which will be better explained in the Future Work in Chapter 7.

No plan was generated for *EB4* either. However, in this case, the analysis performed by UPPAAL-TiGA has not been completed because of the exhaustion of the address space available with the 32-bit addressing. Unfortunately, since the version of UPPAAL-TiGA supporting 64-bit architectures is not publicly available, we were not able to conclude the analysis of the *EB4* case.

An estimation of the development time that is saved for designing controllers through the usage of PuRSUE is shown by the results in Table 6.5. The size of the generated controllers (states and when clauses), clearly evidence that a manual design of such controllers is far for being trivial and feasible.

We can conclude that PuRSUE is effectively supporting designers in the generation of controllers for the models of RAUA applications.

## 6.4 Experimental Evaluation (RQ3)

To answer **RQ3**, we evaluated whether the control strategy generated by PuRSUE is effective when deployed on real robots. We generated, sent and executed the controller for slightly modified versions of scenarios *EB2* and *EB3* described in Sec. 6.1.3 by exploiting the PuRSUE implementation

support described in Sec. 5. Our controller was executed by the TurtleBot robot (Sec. 3.4.3). We checked whether the robot was behaving as expected over the considered scenarios. Videos of the experimental evaluation are available online<sup>1</sup>.

#### 6.4.1 EB2 Scenario

The `ecoBot` robot is asked to retrieve the trash and notify the use that the office is clean (`officeClean`) within 40 seconds of being called by the `human` (`callBot`). Furthermore, the `bin` can be relocated in other POIs by another person. The PuRSUE-ML model of the scenario as reported in Listing 7.16 is used to generate the controller which is then deployed on the TurtleBot, then the PuRSUE UI is used to emulate the reading of events happening in the environment.

We observed that the TurtleBot performed the expected actions according to the uncontrollable events triggered by the environment. For example, in the considered case, after the start signal is sent, `callBot` is triggered immediately, as a consequence the TurtleBot starts moving towards `office`, passing through `hallway`. In the meanwhile, the bin is being moved from its original location (`trashRoom`) to `base`, passing through `hallway`. Once the Turtlebot reaches the location `office`, it sends on the topic *"pursue/actions"* the event `takeTrash`, triggering the human in the office to put to trash on the TurtleBot. Then the robot sends the event `officeClean`, confirming the cleaning of the office, and starts moving towards the bin, which has now completed its movement towards `base`; the movement again passes through `hallway`. After its arrival in `base` the robot sends the event `throwTrash` on topic *"pursue/actions"*, thus causing the human to take the trash from the robot and throw it in the bin. Whenever a movement transition is performed by an agent, the corresponding signal is published on topic *"pursue/events"*.

#### 6.4.2 EB3 Scenario

The `ecoBot` robot is asked to retrieve the trash and notify that the office is clean (`officeClean`) within 40 seconds of being called by the `human` (`callBot`). The trash can be of two types, paper or plastic, which need to be thrown in the corresponding bins, `plasticBin` located in `base` or `paperBin` located in `trashRoom`. The bins cannot be moved from their locations. The PuRSUE-ML model used is reported in Listing 7.17.

---

<sup>1</sup><https://youtu.be/w6SfCmgdCsk>, <https://youtu.be/Xm0Y-urEDD0>

After the start signal, the human immediately defines the types of trash he wants to throw with the event `isPaper` and then calls the robot (`callBot`). The robot moves to `office`, going through `hallway`, and triggers the human to give him the trash with the event `takePaperTrash`. Then the robot notifies the human that the office is clean (event `officeClean`) and moves towards the corresponding bin, which is in location `trashRoom`. Once it reaches the bin, the event `throwPaper` is published on topic *"pursue/actions"*, causing the human to throw the trash in the bin. While the robot is performing `throwPaper`, `human` triggers the events `isPlastic` and `callBot`, thus starting another cleaning task. The task is performed identically to the first one, but the robot then heads to location `base`, where `plasticBin` is located, to perform event `throwPlastic`. As in the previous case, whenever a movement transition is performed by an agent, the corresponding signal is published on topic *"pursue/events"*.

The robot did indeed behave as expected in the considered scenarios. We can conclude that the plans generated with PuRSUE are indeed effective when deployed on actual robots.

## Chapter 7

# Conclusions and future work

This thesis aimed at extending the support provided to developers in the creation of robotic applications. The analysis of the literature showed the absence of a general framework able to address current robotic developers needs, as evidenced in the Co4robots project [44], within which this thesis has been developed. The analysis of the state of the art showed the absence of an approach capable of (i) supporting a systematic and rigorous design of robotic applications through a high-level structured semantic; (ii) enabling the automatic synthesis of controllers that allow robots to achieve their missions; and (iii) allowing the designer to reason on real-time properties.

This thesis developed PuRSUE, a comprehensive framework that supports designers in the creation of controllers for robotic applications. Specifically, the contribution of this thesis can be summarized as follows:

- PuRSUE provides a high-level language called PuRSUE-ML, that allows high-level modeling of robotic applications. PuRSUE-ML allows designers to describe a robotic application in terms of its key locations as well as how they are connected, the relevant events agents can trigger in it and the agent themselves. Furthermore constructs such as rules and states allow the designer to better describe how the agents and the environment interact. Finally, the objective constructs of PuRSUE-ML allow the designer to provide the mission the controller should achieve.
- PuRSUE automatically generates a model of the robotic application as a TGA. This model allows for the computation of a controller for the controllable agents included in the scenario prescribing a sequence of actions ensuring that the controlled agents achieve their missions regardless of the behavior of the uncontrollable agents. This allows

the designers to easily reason on how different configurations of the environment allow for the accomplishment of the provided missions, as well as whether a mission can be accomplished regardless of uncontrollable agents behavior. Finally, PuRSUE allows designers to deploy the controller on the target hardware.

- We evaluated PuRSUE considering three different aspects:
  1. We evaluated whether PuRSUE offers a more compact way than TGA to model robotic applications by trying to encode several robotic applications using PuRSUE-ML and compare the sizes of the model in PuRSUE-ML with that of the TGA model generated. We were capable of modeling all the robotic applications considered using PuRSUE-ML.
  2. We evaluated the capability of PuRSUE in supporting designers in the generation of controllers and reasoning on time-explicit properties for robotic applications. To do so, we generated controllers using PuRSUE for the scenarios considered. While for most scenarios we were able to generate a controller, some scenarios resulted impossible to solve regardless of the behavior of the controllable agents. This showed how PuRSUE is indeed capable of generating controllers for robotic applications described with PuRSUE-ML, as well as its capability to support developers in reasoning on real-time properties.
  3. We evaluate whether the generated controllers were effective once deployed on a robotic platform. To do so we deployed the controllers generated for 2 of the considered robotic applications on the TurtleBot robot in the Jupiter building of Chalmers University. The robot behaved as expected and achieved the mission in both the scenarios, showing that the controllers generated by PuRSUE are effective when deployed on actual robots.

This work opens several directions of future work:

- Improving the *implementation* of PuRSUE. As mentioned in Chapter 5, there is a small number of features presented in Chapter 4 that were not implemented, as not relevant for the evaluation of this work. These include
  - The keyword `prevent`.
  - The keyword `unidirectional`.

- The inclusion of general Boolean expression in State Dependencies.
- Improving the *robustness* of the generated *controller*. The current implementation of the controller assumes that the real system behaves as expected, e.g. uncontrollable agent cannot perform actions that are not modeled and change in the state of the system only occurs as a consequence of the modeled actions. While this is an assumption for our work, in order to allow designers to the deploy the controllers generated by PuRSUE in real-life scenarios, the controller generated needs to be able to handle these uncertainties.
- Expanding the *expressiveness* of PuRSUE-ML. There are many features that can be added to PuRSUE-ML. These include:
  - Developing the *user-interface* of PuRSUE. While an Eclipse plug-in is available for the generation of a PuRSUE-ML model, work can be done in the direction of providing PuRSUE as an off-the-shelf product. A visual interface allowing the designer to define the POIs and their connections with more ease should be developed, as well as an automatic system to link these POIs with the coordinates on the robots internal map.  
The same should be done with signal from sensors in the environment and actuators of the robot, this would ultimately allow the designer to model and deploy robotic applications without having any expertise on ROS, thus rendering PuRSUE available not only for robotic application designers, but for anyone interested in deploying a controller on a robot.
  - The inclusions of more constructs used to specify the mission of the controller in the robotic application. Patterns provide solutions for recurrent specification problems, they allow designers to more easily and clearly define robotic missions. The patterns to be implemented could be taken from work done in [35], where a number of key patterns for robotic applications have been analyzed and classified.
  - The addition of a construct forcing an uncontrollable agent to leave a location after it has stayed there for a certain amount of time units. The PuRSUE framework generates a controller only when one can be found that is capable of winning, regardless of the strategy of the opponent. This makes it difficult to model some scenarios, e.g. a robot has the task to pick up an object in a

POI, without interfering with the movement of other uncontrollable agents in the application. When encoding this scenario, the designer might run into the situation in which an uncontrollable agent chooses, as strategy, to spend an indefinite amount of time in the POI where the object is placed, thus making the control problem unfeasible. This is however an unrealistic behavior in some scenarios, e.g. a human in an office will likely not spend his entire day occupying the POI where the object is located. A construct forcing an agent to leave any POI after a certain amount of time could solve this modeling issue, thus greatly increasing PuRSUE-ML's expressive power.

- The explicit inclusion of forbidden location for certain agents, thus automatically preventing them from ever entering them.
- The extension of event constructs to include events with uncertain duration, in which the designer can provide the minimum and maximum duration of the event.
- The modeling of an agent being busy while collaborating with another agent. In the current implementation of the system, collaboration is defined as the need of an agent to be in a location in order to allow another agent to trigger a certain event. In many robotic applications it might be important to model a collaborative action that keeps both agents busy for the duration of the event.
- The extension of rules to include any general expression. As previously discussed, only a subset of regular expressions was included in rules for the scope of this thesis, but it would indeed greatly increase the expressiveness of the model to include any general expression in rules, as they would allow designers to implement many different constraints on the evolution of the system. This would be trivial, as transformation between regular expressions and TA are available in the literature.
- The extension of states to include any number of states. In this work, only boolean states have been considered. It would however be simple to implement constructs allowing user to define generally complex states, which could be very useful to model a more complex behavior of the environment.

# APPENDIX

## 7.1 Scenarios from Evaluation modeled in PuRSUE-ML

Listing 7.1– Description for the CT1 scenario in PuRSUE-ML .

```
1 //locations
2 poi "a"
3 poi "b"
4 poi "c"
5 poi "d"
6
7 //connections
8 connect a and b distance 10
9 connect b and c distance 10
10 connect c and d distance 10
11 connect d and a distance 10
12
13 //events
14 event "catch" collaborative
15
16 //agents
17 agent "police" controllable mobile
18   1 location a can_do catch
19 agent "thief" mobile 2 location c
20   reacts_to catch
21
22 //objectives
23 reach_objective: do catch
```

Listing 7.2– Description for the CT2a scenario in PuRSUE-ML .

```
1 //locations
2 as 7.1, (2-5)
3 poi "weaponCloset"
4
5 //connections
6 as 7.1, (8-11)
7 connect weaponCloset and a
8   distance 3
9
10 //events
11 event "catch" collaborative
12 event "pickUpBaton" location
13   weaponCloset duration 5
14
15 //rule
16 rule "howToCatch": pickUpBaton
17   before catch
18
19 //agents
20 agent "police" controllable mobile
21   1 location a can_do catch,
22   pickUpBaton
23 agent "thief" mobile 2 location c
24   reacts_to catch
25
26 //objectives
27 reach_objective: do catch
```



Listing 7.3– Description for the CT2b scenario in PuRSUE-ML .

```

1 //locations
2 as 7.1, (2-5)
3 poi "weaponCloset"
4
5 //connections
6 as 7.1, (8-11)
7 connect weaponCloset and a
   distance 3
8
9 //events
10 event "catch" collaborative
11 event "pickUpBaton" location
   weaponCloset duration 5
12 event "putDownBaton" location
   weaponCloset duration 5
13
14 //states
15 State "hasBaton": initially false,
   true_if pickUpBaton false_if
   putDownBaton
16
17 //state dependencies
18 stateDependency: catch only_if
   hasBaton is_true
19
20 //agents
21 agent "police" controllable mobile
   1 location a can_do catch,
   pickUpBaton, putDownBaton
22 agent "thief" mobile 2 location c
   reacts_to catch
23
24 //objectives
25 reach_objective: do catch

```

Listing 7.4– Description for the CT3a scenario in PuRSUE-ML .

```

1 //locations
2 as 7.1, (2-5)
3
4 //connections
5 as 7.1, (8-11)
6
7 //events
8 event "catch" collaborative
9
10 //agents
11 agent "police1" controllable
   mobile 1 location a can_do
   catch
12 agent "police2" controllable
   mobile 1 location a can_do
   catch
13 agent "thief" mobile 1 location c
   reacts_to catch
14
15 //objectives
16 reach_objective: do catch after 0

```

Listing 7.5– Description for the CT3b scenario in PuRSUE-ML .

```

1 //locations
2 as 7.1, (2-5)
3
4 //connections
5 as 7.1, (8-11)
6
7 //events
8 event "catch" collaborative
9
10 //agents
11 agent "police1" controllable
   mobile 1 location a can_do
   catch
12 agent "police2" controllable
   mobile 1 location a can_do
   catch
13 agent "police3" controllable
   mobile 1 location a can_do
   catch

```

```

14 agent "thief" mobile 1 location c 16 //objectives
    reacts_to catch                    17 reach_objective: do catch
15

```

Listing 7.6– Description for the CT4 scenario in PuRSUE-ML .

```

1 //locations
2 poi "a"
3 poi "b"
4 poi "c"
5 poi "d"
6 poi "e"
7 poi "f"
8 poi "g"
9 poi "h"
10 poi "i"
11 poi "l"
12 poi "m"
13 poi "n"
14 poi "o"
15 poi "p"
16
17 //connections
18 connect a and p distance 11
19 connect g and h distance 11
20 connect c and n distance 16
21 connect a and b distance 10
22 connect p and o distance 10
23 connect b and c distance 4
24 connect o and n distance 4
25 connect c and d distance 3
26 connect n and m distance 3
27 connect d and e distance 4
28 connect m and i distance 4
29 connect d and f distance 9
30 connect m and l distance 9
31 connect f and g distance 4
32 connect l and h distance 4
33 connect e and g distance 9
34 connect i and h distance 9
35
36 //events
37 event "catch" collaborative
38
39 //agents
40 as 7.1, (17-18)
41
42 //objectives
43 reach_objective: do catch

```

Listing 7.7– Description for the CT5 scenario in PuRSUE-ML .

```

1 //locations
2 as 7.6, (2-15)
3 poi "stairs"
4 poi "window1"
5 poi "window2"
6
7 //connections
8 connect a and window1 distance 5
9 connect window1 and p distance 5
10 connect g and window2 distance 5
11 connect window2 and h distance 11
12 connect c and stairs distance 8
13 connect stairs and n distance 8
14 as 7.6, (21-34)
15
16 //events
17 event "catch" collaborative
18 event "steal1" location a
19 event "steal2" location b
20 event "steal3" location g
21 event "stolen"
22 event "leave1" location window1
23 event "leave2" location window2
24 event "leave3" location stairs
25 event "enter" location stairs
26
27 //Rule
28 rule "stealing": ((steal1 or
    steal2) or steal3) before (
    stolen or catch)
29

```

```

30 //states
31 state "away" : initially false,
    true_if leave1, leave2, leave3
    false_if enter
32
33 //state dependencies
34 stateDependency: catch only_if
    away is_false
35 stateDependency: steal1 only_if
    away is_false
36 stateDependency: steal2 only_if
    away is_false
37 stateDependency: steal3 only_if
    away is_false
38 stateDependency: stolen only_if
    away is_true
39
40 //agents
41 agent "police" controllable mobile
    1 location a can_do catch
42 agent "thief" mobile 2 location
    stairs can_do steal1, steal2,
    steal3, leave1, leave2, leave3
    , enter, stolen reacts_to
    catch
43
44 //objectives
45 avoid stolen

```

Listing 7.8– Description for the CT6 scenario in PuRSUE-ML .

```

1 as 7.7, (1-38)
2 //agents
3 agent "police" mobile 1 location a
    can_do catch
4 agent "thief" controllable mobile
    1 location stairs can_do
5
6 //objectives
7 reach_objective: do stolen
    steal1, steal2, steal3, leave1
    , leave2, leave3, enter,
    stolen reacts_to catch

```

Listing 7.9– Description for the WC1 scenario in PuRSUE-ML .

```

1 //locations
2 poi "station"
3
4 //events
5 event "newBox"
6 event "pickUpBox" location station
    duration 2
7 event "screw" location station
    duration 10
8 event "putDownBox" location
    station duration 2
9
10 //rules
11 rule "workFlow": (newBox before
    pickUpBox ) before (screw
    before putDownBox )
12
13 //agents
14 agent "bot" controllable location
    station can_do pickUpBox,
    screw, putDownBox
15 agent "human" location station
    can_do pickUpBox, screw,
    putDownBox, newBox
16
17 //objectives
18 objective: if newBox then
    putDownBox within 30

```

Listing 7.10– Description for the WC2a scenario in PuRSUE-ML .

```

1 //locations
2 poi "station"
3
4 //events
5 event "newBox"
6 event "pickUpBoxR1" location
  station duration 2
7 event "pickUpBoxH" location
  station duration 2
8 event "screwR1" location station
  duration 10
9 event "screwH" location station
  duration 10
10 event "putDownBoxR1" location
  station duration 2
11 event "putDownBoxH" location
  station duration 2
12 event "done"
13
14 //rules
15 rule "workFlow": ((newBox before (
  pickUpBoxR1 or pickUpBoxH ))
  before (screwR1 or screwH ))
  before ((putDownBoxR1 or
  putDownBoxH ) before done)
16 rule "pickUp1" : pickUpBoxR1
  before putDownBoxR1
17 rule "pickUpH" : pickUpBoxH before
  putDownBoxH
18 state "busyR1" : initially false,
  true_if pickUpBoxR1 false_if
  putDownBoxR1
19 state "busyH" : initially false,
  true_if pickUpBoxH false_if
  putDownBoxH
20 stateDependency: screwR1 only_if
  busyR1 is_false
21 stateDependency: screwH only_if
  busyH is_false
22
23 //agents
24 agent "bot1" controllable location
  station can_do pickUpBoxR1,
  screwR1, putDownBoxR1, done
25 agent "human" location station
  can_do pickUpBoxH, screwH,
  putDownBoxH, done, newBox
26
27 //objectives
28 objective: if newBox then done
  within 30

```

Listing 7.11– Description for the WC2b scenario in PuRSUE-ML .

```

1 //locations
2 poi "station"
3
4 //events
5 event "newBox"
6 event "pickUpBoxR1" location
  station duration 2
7 event "pickUpBoxR2" location
  station duration 2
8 event "pickUpBoxH" location
  station duration 2
9 event "screwR1" location station
  duration 10
10 event "screwR2" location station
  duration 10
11 event "screwH" location station
  duration 10
12 event "putDownBoxR1" location
  station duration 2
13 event "putDownBoxR2" location
  station duration 2
14 event "putDownBoxH" location
  station duration 2
15 event "done"
16
17 //rules
18 rule "workFlow": (newBox before (
  (pickUpBoxR1 or pickUpBoxR2)
  or pickUpBoxH ) ) before (((
  screwR1 or screwR2 ) or screwH
  ) before ((putDownBoxR1 or
  putDownBoxR2) or putDownBoxH )

```

```

    ) before done)
19 rule "pickUp1" : pickUpBoxR1
    before putDownBoxR1
20 rule "pickUp2" : pickUpBoxR2
    before putDownBoxR2
21 rule "pickUpH" : pickUpBoxH before
    putDownBoxH
22 state "busyR1" : initially false,
    true_if pickUpBoxR1 false_if
    putDownBoxR1
23 state "busyR2" : initially false,
    true_if pickUpBoxR2 false_if
    putDownBoxR2
24 state "busyH" : initially false,
    true_if pickUpBoxH false_if
    putDownBoxH
25 stateDependency: screwR1 only_if
    busyR1 is_false
26 stateDependency: screwR2 only_if
    busyR2 is_false
27 stateDependency: screwH only_if
    busyH is_false
28
29 //agents
30 agent "bot1" controllable location
    station can_do pickUpBoxR1,
    screwR1, putDownBoxR1, done
31 agent "bot2" controllable location
    station can_do pickUpBoxR2,
    screwR2, putDownBoxR2, done
32 agent "human" location station
    can_do pickUpBoxH, screwH,
    putDownBoxH, done, newBox
33
34 //objectives
35 objective: if newBox then done
    within 30

```

Listing 7.12– Description for the EB1 scenario in PuRSUE-ML .

```

1 //locations
2 poi "office"
3 poi "base"
4 poi "hallway"
5 poi "trashRoom"
6
7 //connections
8 connect office and hallway
    distance 5
9 connect base and hallway distance
    2
10 connect trashRoom and hallway
    distance 7
11
12 //events
13 event "throwTrash" collaborative
    duration 3
14 event "getTrash" location office
    duration 5
15 event "callBot"
16 event "wait" duration 37
17 event "officeClean" location
    office
18
19 //Rule
20 rule "makingTrash" : callBot
    before wait
21 rule "pickingUp" : (callBot before
    trash) before officeClean
22 rule "throwingTrash" : getTrash
    before throwTrash
23
24 //agents
25 agent "ecoBot" controllable mobile
    1 location base can_do trash,
    throwTrash, officeClean
26 agent "human" location office
    can_do callBot, wait
27 agent "bin" location trashRoom
    reacts_to throwTrash
28
29 //objectives
30 objective: if callBot then
    officeClean within 20

```

Listing 7.13– Description for the EB2 scenario in PuRSUE-ML .

```

1 as 7.12, (1-26)
2 agent "bin" mobile 2 location
  trashRoom reacts_to throwTrash
3
4 //objectives
5 objective: if callBot then
  officeClean within 20

```

Listing 7.14– Description for the EB3 scenario in PuRSUE-ML .

```

1
2 as 7.12, (1-10)
3
4 //events
5 event "throwPaper" collaborative
  duration 3
6 event "throwPlastic" collaborative
  duration 3
7 event "getPlasticTrash" location
  office duration 5
8 event "getPaperTrash" location
  office duration 5
9 event "callBot"
10 event "wait" duration 36
11 event "isPaper"
12 event "isPlastic"
13 event "officeClean" location
  office
14
15 //Rule
16 rule "makingTrash" : (isPaper or
  isPlastic) before (callBot
  before wait)
17 rule "pickingUp" : (callBot before
  (getPlasticTrash or
  getPaperTrash)) before
  officeClean
18 rule "throwingPaper" :
  getPaperTrash before
  throwPaper
19 rule "throwingPlastic" :
  getPlasticTrash before
  throwPlastic
20
21 //states
22 state "paperOrPlastic": initially
  true, true_if isPaper false_if
  isPlastic
23
24 //stateDependencies
25 stateDependency: getPaperTrash
  only_if paperOrPlastic is_true
26 stateDependency: getPlasticTrash
  only_if paperOrPlastic
  is_false
27
28 //agents
29 agent "ecoBot" controllable mobile
  1 location base can_do
  getPlasticTrash, getPaperTrash
  , throwPlastic, throwPaper,
  officeClean
30 agent "human" location office
  can_do callBot, wait, isPaper,
  isPlastic
31 agent "plasticBin" location
  trashRoom reacts_to
  throwPlastic
32 agent "paperBin" location
  trashRoom reacts_to throwPaper
33
34 //objectives
35 objective: if callBot then
  officeClean within 20

```

Listing 7.15– Description for the EB4 scenario in PuRSUE-ML .

```

1 as 7.12, (2-30)
2 agent "plasticBin" mobile 2
   location trashRoom reacts_to
   throwPlastic
3 agent "paperBin" mobile 2 location
   trashRoom reacts_to
   throwPaper
4
5 //objectives
6 objective: if callBot then
   officeClean within 20

```

Listing 7.16– Description for the EB2 experimental scenario in PuRSUE-ML .

```

1 //locations
2 poi "office"
3 poi "base"
4 poi "hallway"
5 poi "trashRoom"
6
7 //connections
8 connect office and hallway
   distance 3
9 connect base and hallway distance
   4
10 connect trashRoom and hallway
   distance 4
11
12 //events
13 event "throwTrash" collaborative
   duration 3
14 event "takeTrash" location office
   duration 5
15 event "callBot"
16 event "wait" duration 111
17 event "officeClean" location
   office
18
19 //Rule
20 rule "robotCallBuffer" : callBot
   before wait
21 rule "pickingUp" : (callBot before
   takeTrash) before officeClean
22 rule "throwingTrash" : getTrash
   before throwTrash
23
24 //agents
25 agent "ecoBot" controllable mobile
   3 location base can_do
   takeTrash, throwTrash,
   officeClean
26 agent "human" location office
   can_do callBot, wait
27 agent "bin" mobile 4 location
   trashRoom reacts_to throwTrash
28
29 //objectives
30 objective: if callBot then
   officeClean within 40

```

Listing 7.17– Description for the EB3 experimental scenario in PuRSUE-ML .

```

1 //locations
2 poi "office"
3 poi "base"
4 poi "hallway"
5 poi "trashRoom"
6
7 //connections
8 connect office and hallway
   distance 3
9 connect base and hallway distance
   4
10 connect trashRoom and hallway
   distance 4

```

```

11                                     28
12 //events                           29 //states
13 event "throwPaper" collaborative 30 state "paperOrPlastic": initially
    duration 3                        true, true_if isPaper false_if
14 event "throwPlastic" collaborative 31
    duration 3                        isPlastic
15 event "takePlasticTrash" location 32 //stateDependencies
    office duration 5                 33 stateDependency: paperTrash
16 event "takePaperTrash" location    only_if paperOrPlastic is_true
    office duration 5                 34 stateDependency: plasticTrash
17 event "callBot"                    only_if paperOrPlastic
18 event "wait" duration 54            is_false
19 event "isPaper"                     35
20 event "isPlastic"                   36 //agents
21 event "officeClean" location        37 agent "ecoBot" controllable mobile
    office                             3 location base can_do
22                                     takePlasticTrash,
23 //Rule                               takePaperTrash, throwPlastic,
24 rule "makingTrash" : (isPaper or    throwPaper, officeClean
    isPlastic) before (callBot
    before wait)                       38 agent "human" location office
25 rule "pickingUp" : (callBot before  can_do callBot, wait, isPaper,
    (takePlasticTrash or              isPlastic
    takePaperTrash)) before           39 agent "plasticBin" location base
    officeClean                       reacts_to throwPlastic
26 rule "throwingPaper" :              40 agent "paperBin" location
    takePaperTrash before              trashRoom reacts_to throwPaper
    throwPaper                          41
27 rule "throwingPlastic" :            42 //objectives
    takePlasticTrash before            43 objective: if callBot then
    throwPlastic                       officeClean within 40

```



# Bibliography

- [1] *International Federation of Robotics*.
- [2] Rajeev Alur and David L Dill. A theory of timed automata. *Theoretical computer science*, 126(2):183–235, 1994.
- [3] Mehrnoosh Askarpour, Dino Mandrioli, Matteo Rossi, and Federico Vicentini. Formal model of human erroneous behavior for safety analysis in collaborative robotics. *Robotics and Computer-Integrated Manufacturing*, 57:465 – 476, 2019.
- [4] Marco Autili, Lars Grunske, Markus Lumpe, Patrizio Pelliccione, and Antony Tang. Aligning qualitative, real-time, and probabilistic property specification patterns using a structured english grammar. *IEEE Transactions on Software Engineering*, 41(7):620–638, 2015.
- [5] Gerd Behrmann, Agnes Cougnard, Alexandre David, Emmanuel Fleury, Kim G Larsen, and Didier Lime. Uppaal tiga user-manual, 2007.
- [6] Gerd Behrmann, Alexandre David, and Kim G Larsen. A tutorial on uppaal. In *Formal methods for the design of real-time systems*, pages 200–236. Springer, 2004.
- [7] Franck Cassez, Kim Larsen, Jean-François Raskin, and Pierre-Alain Reynier. Timed controller synthesis: An industrial case study. *Deliverable no.: D5. 12 Title of Deliverable: Industrial Handbook*, page 150, 2011.
- [8] A. Ceballos, Saddek Bensalem, André Cesta, Lavindra de Silva, Simone Fratini, Félix Ingrand, Jorge Ocón, Andrea Orlandini, Frederic Py, Kaushik Rajan, Riccardo Rasconi, and Michel van Winnendael. A goal oriented autonomous controller for space exploration. 2011.
- [9] Amedeo Cesta, Andrea Orlandini, and Alessandro Umbrico. Toward a general purpose software environment for timeline-based planning. In

*International Workshop on Experimental Evaluation of Algorithms for solving problems with combinatorial explosion*, 2013.

- [10] A. Cimatti, E.M. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: a new Symbolic Model Verifier. In N. Halbwachs and D. Peled, editors, *Computer-Aided Verification (CAV'99)*, number 1633 in Lecture Notes in Computer Science, pages 495–499, Trento, Italy, July 1999. Springer.
- [11] Alessandro Cimatti, Luke Hunsberger, Andrea Micheli, and Marco Roveri. Using timed game automata to synthesize execution strategies for simple temporal networks with uncertainty. In *AAAI*, pages 2242–2249, 2014.
- [12] Bruno Damas and Pedro Lima. Stochastic discrete event model of a multi-robot team playing an adversarial game. *IFAC Proceedings Volumes*, 37(8):974–979, 2004.
- [13] Cameron Finucane, Gangyuan Jing, and Hadas Kress-Gazit. LTLMoP: Experimenting with language, temporal logic and robot control. In *Intelligent Robots and Systems (IROS)*, pages 1988–1993. IEEE, 2010.
- [14] Michael Fisher, Louise Dennis, and Matt Webster. Verifying autonomous systems. *Communications of the ACM*, 56(9):84–93, 2013.
- [15] Tully Foote and Melonee Wise. Turtlebot home page, Last access 27 March 2019, <https://www.turtlebot.com/>.
- [16] Paul Gainer, Clare Dixon, Kerstin Dautenhahn, Michael Fisher, Ullrich Hustadt, Joe Saunders, and Matt Webster. Cruton: Automatic verification of a robotic assistant’s behaviours. In *Critical Systems: Formal Methods and Automated Verification*, pages 119–133. Springer, 2017.
- [17] Nicola Gigante, Angelo Montanari, Marta Cialdea Mayer, Andrea Orlandini, and Mark Reynolds. A game-theoretic approach to timeline-based planning with uncertainty. *arXiv preprint arXiv:1807.04837*, 2018.
- [18] Sebastian Götz, Max Leuthäuser, Jan Reimann, Julia Schroeter, Christian Wende, Claas Wilke, and Uwe Aßmann. A role-based language for collaborative robot applications. In *International Symposium On Leveraging Applications of Formal Methods, Verification and Validation*, pages 1–15. Springer, 2011.
- [19] Jan Jakoh Jessen, Jacob Illum Rasmussen, Kim G Larsen, and Alexandre David. Guided controller synthesis for climate controller using uppaal

- tiga. In *Formal Modeling and Analysis of Timed Systems*, pages 227–240. Springer, 2007.
- [20] Savas Konur, Clare Dixon, and Michael Fisher. Formal verification of probabilistic swarm behaviours. In *International Conference on Swarm Intelligence*, pages 440–447. Springer, 2010.
- [21] Lars Kunze, Tobias Roehm, and Michael Beetz. Towards semantic robot description languages. In *International Conference on Robotics and Automation (ICRA)*, pages 5589–5595. IEEE, 2011.
- [22] M. Kwiatkowska, G. Norman, and D. Parker. PRISM 4.0: Verification of probabilistic real-time systems. In G. Gopalakrishnan and S. Qadeer, editors, *Computer Aided Verification*, volume 6806 of *LNCS*, pages 585–591. Springer, 2011.
- [23] Christine Largouët, Omar Krichen, and Yulong Zhao. Temporal planning with extended timed automata. In *28th International Conference on Tools with Artificial Intelligence (ICTAI 2016)*, 2016.
- [24] Alberto Quattrini Li, Ra aele Fioratto, Francesco Amigoni, and Volkan Isler. A search-based approach to solve pursuit-evasion games with limited visibility in polygonal environments.
- [25] Constantine Lignos, Vasumathi Raman, Cameron Finucane, Mitchell Marcus, and Hadas Kress-Gazit. Provably correct reactive control from natural language. *Autonomous Robots*, 38(1):89–105, 2015.
- [26] Martin Loetzsch, Max Risler, and Matthias Jünger. Xabsl-a pragmatic approach to behavior engineering. In *IROS*, pages 5124–5129, 2006.
- [27] Yuri K Lopes, Stefan M Trenkwalder, André B Leal, Tony J Dodd, and Roderich Groß. Supervisory control theory applied to swarm robotics. *Swarm Intelligence*, 10(1):65–97, 2016.
- [28] Shahar Maoz and Jan Oliver Ringert. Spectra. available at <http://smlab.cs.tau.ac.il/syntech/spectra/userguide.pdf>.
- [29] Marta Cialdea Mayer and Andrea Orlandini. An executable semantics of flexible plans in terms of timed game automata. In *Temporal Representation and Reasoning (TIME), 2015 22nd International Symposium on*, pages 160–169. IEEE, 2015.

- [30] C. Menghi, C. Tsigkanos, T. Berger, P. Pelliccione, and C. Ghezzi. Poster: Property specification patterns for robotic missions. In *International Conference on Software Engineering: Companion Proceedings*, pages 434–435, May 2018.
- [31] Claudio Menghi. Rest2ros. available at <https://github.com/claudiomenghi/Rest2Ros>.
- [32] Claudio Menghi, Sergio Garcia, Patrizio Pelliccione, and Jana Tumova. Multi-robot LTL planning under uncertainty. In *International Symposium on Formal Methods*, pages 399–417. Springer, 2018.
- [33] Claudio Menghi, Sergio García, Patrizio Pelliccione, and Jana Tumova. Towards multi-robot applications planning under uncertainty. In *International Conference on Software Engineering: Companion Proceedings*, pages 438–439. ACM, 2018.
- [34] Claudio Menghi, Christos Tsigkanos, Thorsten Berger, and Patrizio Pelliccione. PsALM: Specification of dependable robotic missions. In *International Conference on Software Engineering (ICSE): Companion Proceedings*, 2019.
- [35] Claudio Menghi, Christos Tsigkanos, Patrizio Pelliccione, Carlo Ghezzi, and Thorsten Berger. Specification patterns for robotic missions. *CoRR*, abs/1901.02077, 2019.
- [36] Alvaro Miyazawa, Pedro Ribeiro, Wei Li, Ana Cavalcanti, and Jon Timmis. Automatic property checking of robotic applications. In *International Conference on Intelligent Robots and Systems (IROS)*, pages 3869–3876. IEEE, 2017.
- [37] Jeremy Morse, Dejanira Araiza-Illan, Jonathan Lawry, Arthur Richards, and Kerstin Eder. Formal specification and analysis of autonomous systems under partial compliance. *arXiv preprint arXiv:1603.01082*, 2016.
- [38] Alexander Neumann. transitions. available at <https://github.com/pytransitions/transitions>.
- [39] Arne Nordmann, Nico Hochgeschwender, and Sebastian Wrede. A survey on domain-specific languages in robotics. In *International Conference on Simulation, Modeling, and Programming for Autonomous Robots*, pages 195–206. Springer, 2014.

- [40] Andrea Orlandini, Alberto Finzi, Amedeo Cesta, Simone Fratini, and Enrico Tronci. Enriching apsi with validation capabilities: The keen environment and its use in robotics. In *Advanced Space Technologies in Robotics and Automation (ASTRA)*, 2011.
- [41] Lucas Preischadt Pinheiro, Yuri Kaszubowski Lopes, André Bittencourt Leal, RSU Rosso, and LP Pinheiro. Nadzoru: A software tool for supervisory control of discrete event systems. In *International Workshop on Dependable Control of Discrete Systems (DCDS)*, volume 5, 2015.
- [42] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y Ng. Ros: an open-source robot operating system. In *ICRA workshop on open source software*, volume 3, page 5. Kobe, Japan, 2009.
- [43] Neil Rugg-Gunn and Stephen Cameron. A formal semantics for multiple vehicle task and motion planning. In *International Conference on Robotics and Automation*, pages 2464–2469. IEEE, 1994.
- [44] The European Commission. *EU H2020 Research and Innovation Programme under GA No. 731869 (Co4Robots)*. The European Commission, 2016-2017.
- [45] Jana Tumova and Dimos V Dimarogonas. Multi-agent planning under local LTL specifications and event-based synchronization. *Automatica*, 70:239–248, 2016.
- [46] Tatiya Padang Tunggal, Andi Supriyanto, Ibnu Faishal, Imam Pambudi, et al. Pursuit algorithm for robot trash can based on fuzzy-cell decomposition. *International Journal of Electrical & Computer Engineering (2088-8708)*, 6(6), 2016.
- [47] J. Tůmová and D. V. Dimarogonas. A receding horizon approach to multi-agent planning from local LTL specifications. In *American Control Conference*, pages 1775–1780, June 2014.