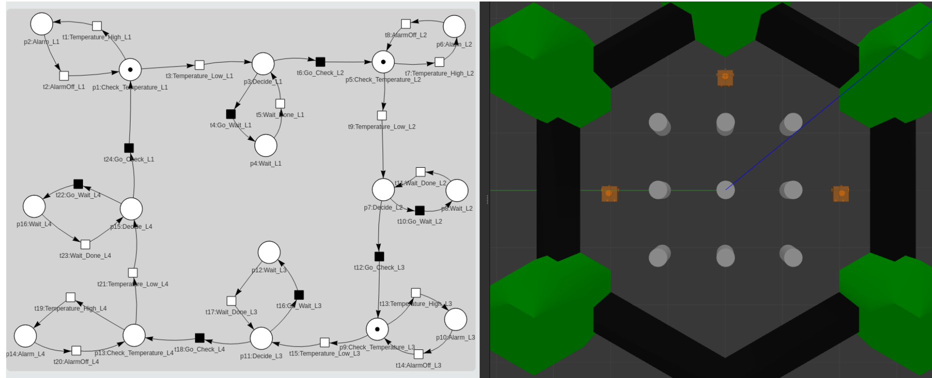
TÉCNICO
LISBOA



A Multi-Robot GSPN Software Framework to Execute and Visualize Plans

Pedro Alegre Caldeira

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisor(s): Prof. Pedro Manuel Urbano de Almeida Lima

Examination Committee

Chairperson: Luís Manuel Antunes Veiga
Supervisor: Pedro Manuel Urbano de Almeida Lima
Member of the Committee: Hugo Filipe Costelha de Castro

November 2020

Dedicated to my grandpa, António Alegre, who taught me how to dream.

Acknowledgments

Developing this master's thesis was by far the hardest project I've ever been a part of.

Deadline-induced stress, many days without real advancements and other things which I eventually lost track of, filled my 9 months while developing this work. But somehow, I eventually managed to reach the end of this chapter in my life and, in my eyes, did a good job, taking into account the uncountable variables that were at play.

However, I didn't do this alone.

By my side I always had my parents and my brother, listening to my never ending rants about robots and their interesting approach to Newton's first law, about how the algorithms weren't working as they should and about my many worries while developing this piece of software. Their ears and sensibility helped me heavily in achieving my goals and reaching the finish line of my master's degree.

In a more personal aspect, I had my girlfriend, whose help and enlightenment was crucial to the development of this "short" report and associated framework. Everyday, she motivated me and kept me afloat, always helping me see the positive side of things. She always said that I was going to make it through, and although I don't like pointing out that she is right, she truly was this time.

I would also like to thank my supervisor, Pedro Lima for the support and guidance and especially Carlos Azevedo, who also accompanied me throughout the whole process, always nudging me in the right direction and helping whenever he could. Although the truth was sometimes hard to hear, it kept me grounded and focused during these months. I hope both of you are as satisfied with my work as I am and I hope our paths cross again in the future.

Finally, I would like to thank the inanimate object that housed me for the past 5 years, Instituto Superior Técnico. Although at times you made me feel more like a hostage than a student, you shaped me into a thougher human-being and an overall great engineer who is ready to take on the world and bring more pride to this incredible faculty.

Resumo

Atualmente não existe uma ferramenta que, de forma integral, inclua modelação, visualização, análise lógica e de desempenho e execução. Redes de Petri estocásticas (RPE) são um modelo matemático que se tem provado como uma ferramenta eficiente para modelar sistemas multi-robô devido à sua forma compacta, execução assíncrona e capacidade para capturar incerteza temporal. Sendo assim, o nosso trabalho incide sobre a expansão de uma ferramenta de modelação e de análise que utiliza RPEs.

O objetivo principal desta tese de mestrado é portanto o desenvolvimento de um pacote de software que permita executar e visualizar planos em sistemas multi-robô e em sistemas multi-agente. Para isso criámos dois módulos novos: um módulo de execução e um módulo de visualização. O módulo de execução é responsável por executar a RPE fornecida como input enquanto que o módulo de visualização tem como objetivo simular ou visualizar a execução de uma RPE. Apesar de este trabalho ter sido feito com o intuito de utilizar ambos os módulos em conjunto, é possível utilizá-los em separado, consoante a necessidade do utilizador. A integração com robôs é assegurada através do middleware robot operating system (ROS).

De forma a testar a nossa ferramenta, corremos uma série de testes na parte que foi integrada com o ROS. No primeiro conjunto de testes, corremos apenas a nossa ferramenta e no segundo, utilizámo-la em conjunto com o simulador Gazebo.

Os resultados obtidos para o conjunto de testes em que só corremos a nossa ferramenta demonstram que, para um sistema com apenas um robô, é possível executar e visualizar redes consideravelmente grandes. Por outro lado, quando utilizámos o simulador Gazebo, confirmámos que é possível correr com sucesso sistemas com um máximo de três robôs simulados, no entanto, a nossa ferramenta não suporta a carga computacional envolvida na simulação de uma equipa de quatro robôs.

Palavras-chave: Redes de Petri estocásticas, Sistema multi-robot, Execução de planos, Visualização de planos

Abstract

Currently one lacks a tool that integrates modelling, visualization, logic and performance analysis and execution. Generalized stochastic Petri nets (GSPNs) are a mathematical model that have proven to be efficient for modelling homogeneous multi-robot systems due to their compact form, asynchronous execution and ability to capture temporal uncertainty. And so, our work extended an existing multi-robot modelling and task analysis tool that uses GSPNs.

The main goal of this master's thesis is to develop a software package that allows the execution of GSPN plans in multi-robot systems and in multi-agent systems. In order to do so, we created two new modules: an execution and a visualization module. The execution module is responsible for executing the input GSPN while the visualization module's goal is to simulate or visualize the execution of the GSPN, depending on the intentions of the user. Although the objective of this work is to use both modules as a whole, it is possible to use both of them independently. The integration with robots is assured by the robot operating system middleware (ROS).

In order to test our framework, we ran a series of tests on the part of the framework that was integrated with ROS. On the first set of tests, we only ran our framework and on the second, we used our framework alongside with the Gazebo simulator.

The obtained results for the tests where we only ran our framework show that, for a system with only one robot, it is possible to execute and visualize considerably large networks. On the other hand, when we used the simulator, we verified that it is possible to successfully run multi-robot systems with up to three virtually simulated robots, however, our tool did not support the computational overhead involved in the simulation of a team of four robots or more.

Keywords: Generalized stochastic Petri nets, Multi-robot systems, Plan execution, Plan visualization

Contents

Acknowledgments v
Resumo vii
Abstract ix
List of Figures xiii

1 Introduction **1**
1.1 Motivation 1
1.2 Objectives 1
1.3 Thesis Outline 3

2 Background **5**
2.1 Generalized Stochastic Petri Nets 5
2.2 GSPNs and multi-robot systems 7
2.3 Policies 8
2.4 Petri net properties 9

3 Related Work **11**
3.1 Finite State Automata Approaches 11
3.2 Belief-Desire-Intention Approaches 12
3.3 Petri Net Approaches 13
3.4 Software tools 15

4 The Execution Module **17**
4.1 Execution Module: standalone implementation 17
4.1.1 Allowing parallelism in our system 19
4.1.2 Agent states 20
4.1.3 Defining the user input 22
4.2 A summary of the standalone execution module 23
4.3 Execution Module: ROS integration 24
4.3.1 General architecture of our system 25
4.3.2 Inner robot communication: Using ROS actions to execute functions 25
4.3.3 Outer robot communication: Using ROS topics and services to change the GSPN 26

4.3.4 Resource tokens and resource places . . . 31
4.3.5 Limiting the possible input GSPNs . . . 32
4.3.6 Defining the user input . . . 32
4.4 A summary of the ROS execution module . . . 33

5 The Visualization Module **35**
5.1 The visualization module architecture . . . 35
5.2 Visualizing GSPNs . . . 36
5.3 Offline visualization . . . 37
5.4 Online visualization . . . 37
5.4.1 Online visualization: standalone implementation . . . 37
5.4.2 Online visualization: ROS integration . . . 38
5.5 A summary of the visualization module . . . 39

6 Results **41**
6.1 Testing the framework . . . 41
6.1.1 Increasing the number of tokens . . . 42
6.1.2 Increasing the number of places . . . 42
6.1.3 Increasing the number of places with only one token . . . 44
6.2 Testing the framework with simulated robots . . . 44
6.2.1 Problem Description . . . 44
6.2.2 Creating a GSPN for the plan . . . 47
6.2.3 Results . . . 48
6.3 Discussion . . . 49
6.3.1 Testing the framework with no robots . . . 49
6.3.2 Testing the framework with robots on Gazebo . . . 49

7 Conclusions **51**
7.1 Achievements . . . 51
7.2 Future Work . . . 51
7.2.1 Execution Module . . . 51
7.2.2 Visualization Module . . . 52

Bibliography **53**

List of Figures

1.1 Framework initial architecture. 2
1.2 Framework final architecture. 3
1.3 Visualization and execution module architecture. 3

2.1 A marked GSPN with an immediate (T1) and a exponential transition (T2). 6
2.2 Evolution of network and corresponding marking graph. 7
2.3 A GSPN with the corresponding multi-robot system. 8

3.1 FSA example 12

4.1 Execution process for agents/robots. 18
4.2 High level execution. 19
4.3 Standalone execution module inputs and outputs. 19
4.4 Synchronization and inactive cases. 20
4.5 The agents' possible states. 21
4.6 Fork case. 22
4.7 A GSPN and an example of the corresponding JSON input. 23
4.8 ROS execution module inputs and outputs. 24
4.9 ROS action client and servers and corresponding GSPN. 26
4.10 Moments where robots need to communicate. 28
4.11 Execution module outer communication. 29
4.12 Service Process. 31
4.13 Resources example. 32
4.14 Algorithm to limit possible input GSPNs. 33

5.1 Visualization module inner architecture. 36
5.2 Using Vis.js to visualize a GSPN model. 36
5.3 Visualization module input screen. 37
5.4 GSPN visualization module in Python 3. 38
5.5 Interaction with the standalone online visualization. 38
5.6 Interaction with ROS online visualization. 39
5.7 GSPN visualization module in ROS. 39

6.1 GSPN to test the increase of tokens. 42
6.2 Results for an increasing number of tokens. 43
6.3 Initial GSPN to test increase of places. 43
6.4 Results for an increasing number of places. 44
6.5 Results for an increasing number of action servers. 45
6.6 Results for an increasing number of places with only one token. 45
6.7 Critical room viewed from the top. 46
6.8 The metric map of our environment (a) and our topological map (b). 47
6.9 Temperature patrol designed GSPN. 48

Chapter 1

Introduction

1.1 Motivation

It has long been recognized that there are several tasks that can be performed more efficiently and robustly using multiple robots. Applications such as: inspection [1], surveillance, search and rescue [2], mapping of unknown or partially known environments [3] or transportation of large objects greatly benefit from the use of multi-robot systems.

Presently, an issue of this area is the fact that the solutions for these applications tend to be hand crafted almost every time a new problem occurs and on most cases, they don't assure any formal guarantees. Besides this, if the problem in hand is too complex, the designer will surely have difficulties coming up with an adequate solution, which most times leads to inefficient and unreliable results.

A good way to solve the mentioned issues is by using a formal model, such as generalized stochastic Petri nets (GSPNs), because it provides methods to synthesize policies to coordinate the multi-robot system that respect formal requirements. Besides this, they also present great advantages in the modelling of homogeneous robotic systems with its intuitive analysis of flow of information and control.

In most cases, the process of building a GSPN starts out by iteratively building a model and analyzing it until the obtained one formally guarantees the existence of certain properties. Next, a policy is obtained by an optimization method and finally, the plan is executed, taking into account the acquired policy and the built network.

Nowadays, one lacks a tool that unifies the above mentioned process [4], and consequently, the user has to implement interfaces that connect distinct 3rd-party tools, many of which implemented in different programming languages, which obviously is a cumbersome task.

1.2 Objectives

The main objective of this master thesis, is to improve on an already implemented GSPN software framework developed in [5], in order to allow the visual representation of the model and the execution of GSPN plans. On Figure 1.1 we included the framework's architecture before we started developing our

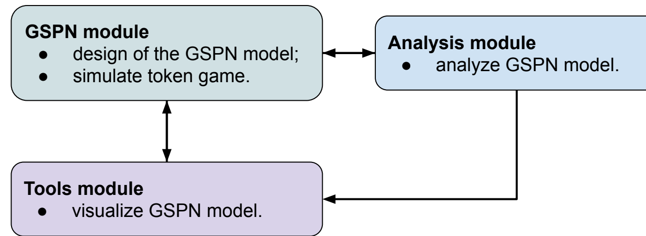


Figure 1.1: Framework initial architecture.

work (the arrows represent data flow). The GSPN module allows the user to manually design multi-robot system as a GSPN using the xml language. The tools module can be used to parse the GSPN in the xml language and translate it into an object that can be understood by the rest of the framework. Besides this, it also allows the user to manually draw other models such as a CTMC. Finally, the analysis module allows the user to directly verify a set of formal properties of the built GSPN. These formal properties will be introduced later on.

The main two components that we added are the execution module and the visualization module. The former allows the user to execute a GSPN plan, which represents a set of actions that an agent or a robot must perform. To execute a GSPN is to complete the mentioned actions. The latter is a graphical front-end, that provides an user friendly interface for the visualization of the GSPN model, the execution progress and the properties obtained from the model analysis.

On Figure 1.2 we included the framework's final architecture where the arrows represent data flow. The execution module uses the GSPN created with the GSPN module and sends the changes of the marking of the GSPN into the visualization module, which are posteriorly reflected on the visualization module's interface. The visualization module also uses the tools module to parse the GSPN into an object that it can understand.

Our framework is integrated with the Robot Operating System (ROS) middleware which enables the execution of plans in real robots or in simulated environments, such as the ones provided by Gazebo [1], but we also built a standalone version which does not need ROS to be used. On Figure 1.3, we included a detailed version of the architecture of the two modules that we introduced. The Figure is divided in half to explicitly show the two different implementations. On the left side, we have the ROS integrated version, where the ROS execution module communicates with the ROS online visualization module. As portrayed by the Figure, the ROS integrated version does not have an offline visualization. On the right side, we have the standalone version, where the standalone execution module communicates with the standalone online visualization module. On the other hand, the offline standalone visualization module is not connected with the execution module because it is used to simulate a GSPN, instead of executing it. An important note to take is that both the standalone offline and online visualizations are accessed through the same interface: the selection of one or the other is performed through the click of two different buttons.

[1] http://gazebosim.org/tutorials?tut=ros_overview

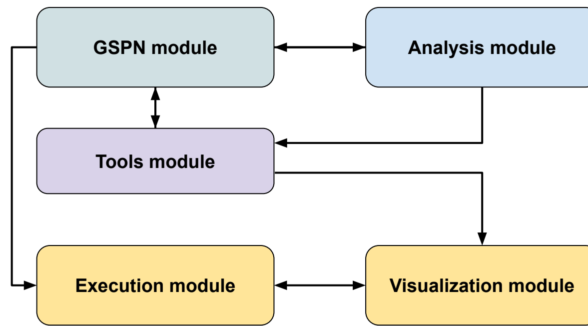


Figure 1.2: Framework final architecture.

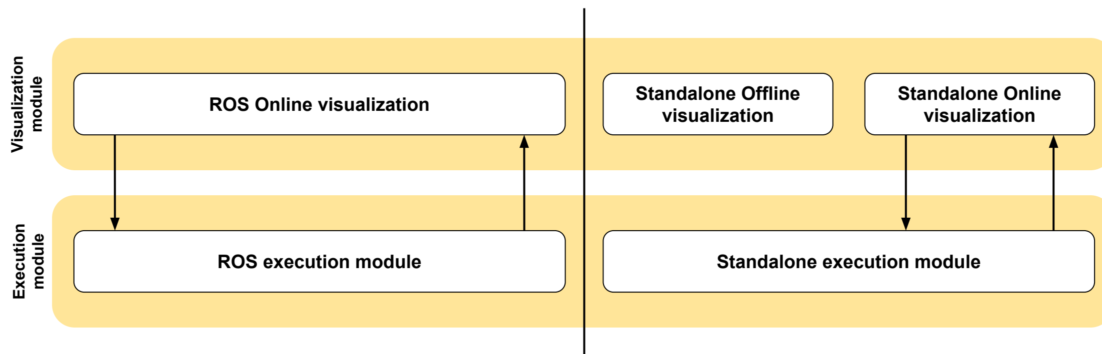


Figure 1.3: Visualization and execution module architecture.

1.3 Thesis Outline

On Chapter 2, the relevant theoretical concepts for the understanding of this work are presented.

On Chapter 3, we go through literature that we consider relevant concerning execution, analysis and visualization of single and multi-robot systems.

On Chapter 4 and Chapter 5, we explain how we implemented the execution and the visualization module respectively, both for the standalone version and the ROS integrated version.

Closer to the end, on Chapter 6, we present the group of tests that we chose to perform in order to understand the limits of our framework and discuss the obtained results.

Finally, on Chapter 7 we present the final overview of our achievements, our final conclusions and a list of suggestions of future work that other developers might find interesting in order to further improve this framework.

Chapter 2

Background

This section introduces the necessary formalisms and notations to understand the rest of this master's thesis. We start by introducing generalized stochastic Petri nets (GSPNs) and afterwards, explain how we will see them in the context of our framework. We then introduce the definition of policies and finally we present the set of formal guarantees that make the use of GSPNs advantageous.

2.1 Generalized Stochastic Petri Nets

Definition 1. *Formally, a GSPN can be defined by the following tuple:*

$$GSPN = (P, T_I, T_E, F, W^-, W^+, m_0, Z_I, R) \tag{2.1}$$

- *P is a finite set of places;*
- T_I *is the set of immediate transitions and* T_E *is the set of exponential transitions where* $T = T_I \cup T_E$*. Immediate transitions model activities that can occur in the system and fire as soon as they have the necessary number of tokens. Exponential transitions are characterized by an exponential distribution which models the elapsed time until firing;*
- F *is the set of arcs where* $F = (P \times T) \cup (T \times P)$*;*
- $W^- : P \times T \to \mathbb{N}$ *and* $W^+ : T \times P \to \mathbb{N}$ *are input and output arc weight functions, respectively. Input arcs go from places into transitions and output arcs go from transitions into places;*
- $m_0 : P \to \mathbb{N}$ *is the initial marking, which can be represented by a vector with a size corresponding to the number of places;*
- $Z_I : T_I \to [0, 1]$ *is the weight of each immediate transition, which means that on the case of having two or more enabled transitions, this value will determine the probability of each transition being fired;*
- $R : T_E \to \mathbb{R}{>}\ 0$ *is a function that associates each exponential transition with a rate.*

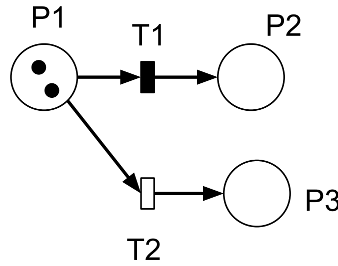


Figure 2.1: A marked GSPN with an immediate (T1) and a exponential transition (T2).

On Figure 2.1 we included an example of a GSPN. More intuitively, a GSPN is composed by a net structure and a marking. The net structure is a bipartite graph built with four elements: places, immediate transitions, exponential transitions and arcs. Places are represented by circles, and the transitions are represented by rectangles. Places can be either input or output, depending on their interaction with a transition. Considering $T = T_I \cup T_E$, $t \in T$ and $p \in P$, the set of input places of t can be defined as $IN = \{p|(p,t) \in F\}$ and the set of output places of t can be defined as $OUT = \{p|(t,p) \in F\}$. The net structure of a GSPN will remain constant all throughout the execution.

The marking, on the other hand, is the discrete number of tokens inside each place and is defined by $S : P \to \mathbb{N}$. The marking is visually represented by small dots inside each place. The only way a transition is fired is if the said transition is enabled. A transition is considered to be enabled if each input place IN is marked with at least W^- tokens. If the transition is enabled, then it can be fired, removing W^- tokens from the input places and adding W^+ tokens to the output places. The tokens' configuration will change all throughout the execution and they're what empowers this tool: with tokens we can observe the evolution of the model during the passing of time.

An important feature of GSPNs is the *marking graph* which formally can be defined as $< S, E >$ considering $E : S \times S \to T \cup \emptyset$, where T is a transition. We consider that E can be null in order to cover the case where we have two markings, that are not connected by a transition.

A more intuitive explanation is the following: on every level of the graph, we observe the possible markings of the network. Consider the following example, which is illustrated on Figure 2.2, where we have a network with 3 places and a total of 2 tokens. Initially the tokens are all in place P1 and as such, their marking will be [2,0,0], meaning that the head of our *marking graph* will be this marking. However, as time progresses, the transitions will fire and the tokens will start moving onto different places. On our simple example, one of the tokens can move onto place 2 while the other stays behind or one of them goes to place 3 while the other stays behind. The markings that represent the positions of the tokens are [1,1,0] and [1,0,1] respectively, meaning that attached to the head of the graph, we will have two children nodes. This process is iterated until no more markings are achieved. On the left of Figure 2.2, we have our initial Petri net and the resulting two children and on the right, we have the corresponding marking graph.

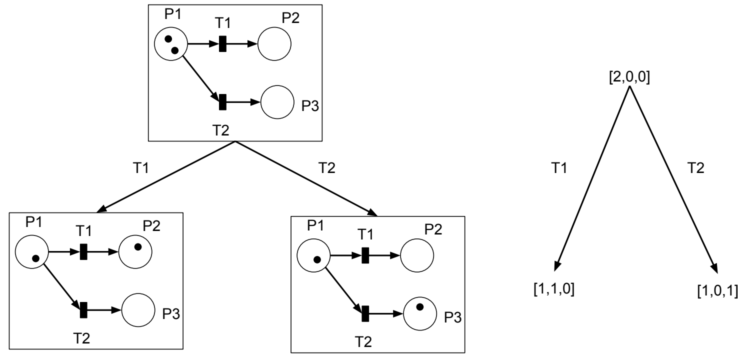


Figure 2.2: Evolution of network and corresponding marking graph.

2.2 GSPNs and multi-robot systems

A multi-robot system is a group of two or more robots which have the ability to communicate with each other and achieve goals as a team.

In our context, the tokens of a GPSN represent two different elements, depending on the places where they are: robots or counters. Tokens will be seen as robots when they are in a place where an action (such as moving into a specific room) can be performed (action places). Besides this, these kinds of tokens are always associated with a physical or simulated robot. On the other hand, tokens will be seen as counters when they are in a place where no action is being performed (resource places), meaning that their existence is merely informative to the designer and to the analysis and synthesis algorithms. These counters will be useful to count, for example, the number of robots that went through a certain place.

Immediate transitions model action selection while exponential transitions model uncontrollable events which can represent the reaction of the environment towards an action executed by the robot or an internal change.

As an example, consider the GSPN and the robot system illustrated on Figure 2.3. On the upper part, the reader can observe the GSPN with five places, four exponential transitions, two immediate transitions and three tokens. One of the tokens is on P1, another one is on P3 and the third one is on P5. On the lower part, we have an illustration of the multi-robot system associated with this specific GSPN.

The GSPN represents a system where two robots must measure the temperature of a critical area. If the temperature is higher than a predefined threshold, an alarm will be activated. It is advantageous to use a multi-robot system because by doing so, we can have a more efficient monitorization of the critical system: when one of the robots is taking measurements, the other one is rebooting its system in order to avoid working for long periods of time. All places except P3 represent action places, where a specific action is being performed, while P3 represents a resource place, where no action is being performed. As such, both the tokens on P1 and P5 represent robots, while the token on P3 represents a counter. This token is not associated with any robot and on the case where the user checks the current marking, by checking the number of tokens inside P3, it will be clear that the number of measurements done is 1.

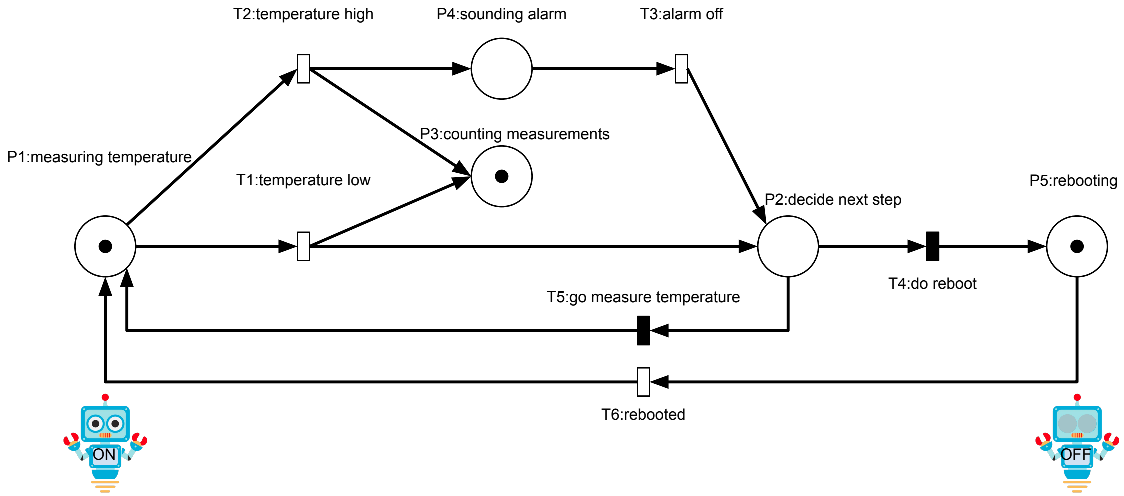


Figure 2.3: A GSPN with the corresponding multi-robot system.

Regarding the transitions, T4 and T5 are immediate transitions and represent the selection of an action, meaning that when a robot enters P2, it will decide its next step based on the current marking. The remaining transitions are exponential because we do not know how long it will take to accomplish any of the actions associated with them. For instance, when on P1, we do not know how long it will take a robot to check the temperature of the critical area, and as such, the associated transitions are exponential.

The illustrated case is cyclic, which means that while the user does not interrupt the execution, the number of measurements done will continue to increase, which of course means that the number of tokens inside P3 will also continue to increase. In order to avoid having robots inside a resource place, we built an algorithm to distribute the tokens during the execution, which will be further explained.

2.3 Policies

A policy defines how a certain system behaves at a given time. To be simply put, a policy dictates what should be done in each state. A state in GSPNs is the current marking of our GSPN and what should be done is a subset of immediate transitions, t_i, where $t_i \in T_i$.

Definition 2. *A policy is formally defined as:* $\pi : ST \rightarrow A$, *where* ST *is a set of states and* A *is a set of actions. In the context of GSPNs, an action* A *can be defined as a set of transitions* T *and associated probabilities* Pr, (T, Pr). *We can have four different forms of policies (which are inter-changeable), such as deterministic, stochastic, stationary and non-stationary. However, we will only focus our attention on stationary and deterministic policies since this will be the kind that we will use further on. Deterministic are the simplest cases of policies and there are no uncertainties to which action the system will execute on each state. Stationary policies are policies that don't change over time.*

2.4 Petri net properties

Petri nets (PNs) are the predecessor of GSPNs and as such, their properties are passed onto the latter. The usage of PNs or GSPNs has many advantages but the main ones are related to the fact that they are mathematical models that can be analyzed to check for properties of the modeled system. Besides this, GSPNs are flexible to the point where they can be converted to other graph-like representations, such as Markov Decision Processes (MDPs), Discrete Time Markov Chains (DTMCs) and Continuous Time Markov Chains (CTMCs) under some assumptions.

PN properties can be divided into two main groups [6]: qualitative or logical, which have to do with the structure of the PN and quantitative or performance analysis, which have to do with their performance.

The properties are listed and shortly described below but for more information about them, consult [7] and [8].

- Qualitative Properties:
  - Reachability and coverability: This property allows us to assess if the system moves into undesired states. A marking m_n is considered reachable from m_0 if there is a certain sequence of firings that transforms m_0 into m_n. Associated with reachability, we have reversibility which states the possibility of returning to m_0. On the other hand, coverability is the property that states if a certain state is covered or not;
  - Boundedness: If the GSPN is bounded, the reachability graph is finite;
  - Safety: if every place of a Petri net has at most *k* tokens for all reachable markings, then the net is *k-safe*;
  - Liveness and Deadlock: A transition is said to be live if it can be fired at least once for every marking of the model. If the net is not live, then it's dead. Besides this, if at least one transition is live, then the net cannot deadlock, which is a state where it's not possible to fire any transition.
- Quantitative Properties:
  - Transient: During the transient phase, the GSPN state suffers various alterations until the steady state is reached. The user can analyze how the probability distribution of tokens per place evolves until the steady-state is reached. Besides this, the time that it takes to reach the steady-state can also be analyzed.
  - Steady-state: The PN reaches a steady state when a certain balance is achieved. The user can analyze data related to the efficiency of the net such as the mean time to reach a certain state, the mean time spent on each state, the average number of tokens in each place or the average throughput of each transition.

Chapter 3

Related Work

On a survey made in 2019, Luckcuck et. al have parsed the state of the art of formal specification and verification models [4]. The survey presents three research questions: (i) what are the difficulties in specifying and verifying the behaviour or robotic systems, (ii) what are the best tools and approaches to these systems and finally, (iii) what are the current limitations of the best practices identified before.

Concerning the second question, the answer provided indicates that temporal logics, discrete event systems and model-checkers are currently the best practice due to the existence of formal guarantees, which essentially indicates that GSPNs (which are discrete event systems) are a good approach. On the other hand, the answer to the third question presents the problem that we aim to solve: the main issue behind the use of several formal specification and verification frameworks is that there is always the need of using some sort of translation between every tool, since there isn't any unified way of doing so.

Many ways of modelling multi-robot systems have been proposed but as of today, the main ones are Finite State Automata (FSA), Belief-Desire-Intention Systems (BDI) and Petri nets (PN).

3.1 Finite State Automata Approaches

A Finite State Automata (FSA) is composed by nodes, which represent states, and arcs, which represent transitions between the nodes. In Figure 3.1 we have included a short example of a FSA.

FSA based approaches have been very successful in modelling robotic systems and tasks because of their intuitive approach to design: FSAs only have two building blocks (nodes and transitions) and they are relatively simple to build.

In [9] the authors introduce a FSA-based approach to use a multi-robot system to patrol a certain perimeter. Based on the Null-Space-based behavior control [10], which essentially is a way of combining multiple smaller behaviors into a single complex one, the system creates actions using several of the elementary behaviours. The supervisor, which is a FSA, selects the intended actions and orders the robots in the system to complete them. The supervisor has the job of deciding what each robot should do and controlling the robots' whereabouts. This can be advantageous because all possible transitions

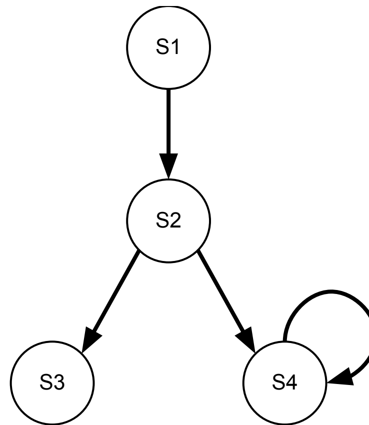


Figure 3.1: FSA example

between actions can be explicitly encoded but on the other hand, this can be hard if a large volume of changes happens in the environment. This work's intention is to create a way of delegating actions using a FSA. In this case, there is no apparent need for time monitoring and as such, FSA are a considerable model to solve this issue.

Although FSA doesn't directly allow the modelling of systems where time is relevant, several approaches have tried to manipulate its definition in order to have some kind of time variable. The most popular one is presented as a framework based on timed automata [11]. Timed automata extends the definition of an automaton with a group of real-valued clock variables. Based on the definition presented on [12], the timed automaton is a tuple $A = (L, l_0, E, I, V)$ where L is a finite set of locations, l_0 is the initial location, E is a finite set of edges, I is a function that assigns a clock constraint to every location and finally V is a function that assigns a set of true atomic propositions. The robots' world is defined as a grid and the clock measures the amount of time spent in a location since it was last reset, which happens every time the robot enters a new location. To verify the properties of the multi-robot system, the authors use UPPAAL, [13], which is a tool box used for modeling, simulation and verification of real-time systems.

When comparing a FSA to a GSPN (or even a PN), GSPNs are usually smaller and although the growth of the size of the marking process is exponential for both models, in the case of GSPNs, it is possible to model, in a finite way, FSAs that are theoretically infinite. Although the construction of a FSA can be very intuitive and simple, most systems built with this approach tend to represent single-robot systems, since the existing mechanisms to build concurrency are not very intuitive and less explicit than on GSPNs.

3.2 Belief-Desire-Intention Approaches

Belief-Desire-Intention systems, or BDI, appeared as an alternative to FSA-based approaches. To be simply put, BDI [14] is an architecture where the agent has three main components:

- Beliefs, which represent what the agent knows about the environment, which is usually imperfect since it depends on the list provided by the programmer;
- Desires, which represent the agent's goals and as such, some can represent a possible end state. Most of the desires will be sub-goals that the agent will need to accomplish in order to achieve his final goal;
- Intentions, which represent the selected behaviours to be executed. This is useful on the case where the environment changes and the agent has to change its desires frequently.

Besides these three components, the system also has a predefined library of plans where the procedural knowledge is stored.

The main reason why this approach is an evolution in comparison with the previous one is that with BDI, the programmer is not forced to create a very solid list of tasks to be completed. Instead, the robot itself has a certain degree of freedom to choose what he feels is more adequate to execute at the time.

In terms of BDI approaches, the most relevant applications are BITE [15] and STEAM [16]

BITE (Bar Ilan Teamwork Engine) is a behaviour-based teamwork architecture that aims to simplify and automate collaboration among the robots of a multi-robot system. To achieve this, BITE has an organization hierarchy, a task/sub-task behaviour graph and a library of social interaction behaviours. The main goal of BITE is to create a teamwork architecture that provides integrated synchronization and allocation and allows to combine different synchronization and allocation protocols.

STEAM, on the other hand, is based on the Joint Intentions Theory [17] which is based on the definition of a common goal for every member of a team, and whose main focus is that the goal has not been reached yet. While this is maintained, the team wont stop trying to achieve it, unless it is considered impossible to be achieved or irrelevant by every member. Moreover, STEAM aims to create a model of teamwork where agents are able to overcome changes in their acting environments, communicating through each team member's commitment in joint intentions and reorganizing it if necessary. On the mentioned paper, the authors apply STEAM in virtual environments of events such as battlefield combat and robot soccer but never on real robots.

A variant of BDI systems was explored on [18], which adds an extra layer to the definition of BDI, specifying two different types of agents, depending on their degree of commitment. If the agent will try anything to achieve his goals, he is described as a *fanatic*. On the other hand, the *relatively committed* has the choice to stop his actions if some other conditions are met. This allows for the user to define two degrees of agent motivation, creating more dynamic and realistic systems.

3.3 Petri Net Approaches

In comparison with the previously mentioned approaches, PNs seem like a more viable approach due to their capability of representing concurrent systems and also verification of formal properties on the performance of the system.

In [19] the authors define a framework based on a modular approach to represent robot task plans using PNs. The building blocks of the framework don't add any changes in PN definitions but are essential in order to simplify the modelling process. These building blocks include predicate places, used to check for a certain condition; action places, which is where a task is performed meaning that the world can be changed; task places, which are modeled as PNs and represent sets of actions to be done; memory places which is another designation for a place that isn't any of the previous ones. Adding to this, the authors define three layers: environment layer, which models the external environment; action execution layer, which includes the PNs that represent actions; action coordination layer, which includes PNs that represent compositions of actions. With these elements, the authors propose a set of algorithms to transform all of these PNs into a single one so that the analysis and the execution are simplified. The main contribution of this framework is the possibility of representing everything related to a robot through PNs, making it easier to analyze and execute the desired system. Furthermore, its design-analysis-design approach improves significantly the resulting models and by separating the PNs into different layers, the complexity of modelling is reduced significantly.

Another approach taken in [20] uses PNs as a language to program the high-level behavior of multi-robot systems. The main goal behind it is creating a rich and intuitive modelling language to support robot developers in designing and implementing high-level robot and multi-robot behaviors. This framework takes inspiration from action languages, which are languages where the program is defined with actions and so, the flow of the program will be composed by several PNs, where each node corresponds to an action. An important note is that, since we're working with robots, we must consider that these actions take time to execute the necessary actions. With this framework we see a different way of modelling multi-robot systems. On the previously mentioned approach, the authors modeled both the environment and the robot in the same PN model, whereas in this paper, the authors simply modelled the tasks that the robot has to finish using PNs. Although it has many advantages, one should note that the main issue with this framework is the fact that the user is constrained to the framework's building blocks, meaning that the framework itself doesn't allow much deviation from its primary objective.

More recently, [21] have created a framework that uses PNs and safe linear temporal logic (LTL) to model the multi-robot system's behaviour and to specify rules that control the coordination, respectively. Besides this, it also uses a PN as a supervisor for the robots' behaviours, restricting what they can do. As opposed to remaining papers presented so far, in this framework the work behaviours are synthesized from the PN while using LTL to specify behaviours in a user-friendly and intuitive way.

On [22], the authors aim to solve the problem of planning a team of robots using a Boolean-based specification to model regions of interest. Although the environment is known and considered to be static, the specification, which is known by every member of the team, imposes Boolean requirements on the locations visited by the robots during the model's execution. The presented solution models the team's movement and the satisfaction of regions with a PN model in order to avoid a state-space explosion. The robot's mission is converted into a set of linear inequalities and later into PN markings, in order to finally be able to solve an integer linear programming problem.

3.4 Software tools

There are many tools with the purpose of modelling, visualizing and analyzing PNs. However, the main limitation of these tools is the fact that they don't do all of these tasks in an integrated way, causing the programmer's job to be more difficult than it has to be.

PRISM [23] is an open-source probabilistic model-checker. It provides mechanisms to build and analyse DTMCs, CTMCs, MDPs and the extensions of these models with rewards. A PRISM model is composed by various modules whose states are represented by a finite set of variables and is based on the Reactive Modules [24] formalism with the intention of being able to describe the various modules that compose the model. Other DTMC and CTMC analysis tools are available but unlike PRISM, they do not allow logic specifications. However, being focused on model-checking, PRISM does not allow execution of plans.

STORM [25] is another probabilistic model checker like PRISM, but since it was developed more recently, it's more optimized than the former. However, STORM doesn't support LTL model checking and doesn't support the PRISM features such as probabilistic timed automata and multi-objective model checking.

SMACH [26] is a ROS-integrated Python framework used for modelling of robotic systems based on FSA. It was created with simplicity in mind so that any programmer who needs a small state machine to define the behavior of its robot can build it rapidly and intuitively. Additionally, the creators of the framework made it very simple for the robot to recover from error situations, since this can be a very tricky event to program. Unlike usual FSAs, SMACH allows parallel execution, using the execution policy *Concurrence*. Nevertheless, this concurrency is not very explicit because of the way it is presented. Moreover, it doesn't show the passing of time, which can be an essential metric do analyze in a multi-robot system. Adding to this, this tool doesn't allow formal analysis, which makes it impossible to obtain formal guarantees from the model.

Pipe [1] is a tool to design GSPNs and PNs. It has as simple user interface and moreover, you can also simulate the token game, which means that we can observe the tokens being exchanged between the places. However, it doesn't allow an execution of the network, meaning that although the user can see the token's movements, there aren't any functions being executed on real robots.

GreatSPN [27] is another tool created with the purpose of analysing Discrete Event Dynamic Systems (DEDS), or more concretely, GSPNs. As of today, the tool allows the user to build the GSPN, visualize it in a graphical way, evaluate the network's qualitative and quantitative properties and finally, visualize the obtained results. Although this tool allows the user to model and analyse a certain GSPN, it doesn't allow its execution.

Contrary to the previously mentioned tools, Petri net plans [20] can execute a PN. However, they are based on PNs, which we can be considered as a slightly less expressive model when compared to GSPNs since they don't take into account uncertainty. Besides this, Petri net plans aren't very flexible in the sense that the user has to use a series of predefined building blocks that compose the framework

[1] `http://pipe2.sourceforge.net`

and although these building blocks are a good way of creating a robust model, this limits the user's possibilities.

On Table 3.1, we summarize the purpose of each tool and its main characteristics.

Tool name	Purpose	Modelling	Analysis	Execution
PRISM [23]	probabilistic model-checker	models DTMCs, CTMCs and MDPs	Logic and Performance	No
STORM [25]	probabilistic model-checker	models DTMCs, CTMCs and MDPs	Logic and Performance	No
SMACH [26]	design of FSAs	models FSAs	No	No
PIPE [2]	design and simulation of PNs	models PNs and GSPNs	Logic and Performance	No
GreatSPN [27]	design and analysis of GSPNs and SWNs	models DEDS	Performance	No
PN Plans [20]	design analysis and execution of PNs	models PNs	Logic	Yes
Our Framework	design, analysis and execution of GSPNs	models GSPNs	Logic and Performance	Yes

Table 3.1: Summary of mentioned tools.

Chapter 4

The Execution Module

The execution module's main goal is to execute the action plan of a GSPN. This module was firstly developed in a standalone version without ROS, using only Python 3, where agents are seen as tokens and functions are seen as places. This version can be used by anyone who is interested in working with a GSPN tool to model and execute plans without robots. Afterwards the module was integrated with ROS, where robots are seen as tokens and actions are seen as places, in order to accomplish the final goal of using it with simulated robots. The purpose of this integration is to allow robot developers to use our tool in a real robot system.

Taking all this into account, by using and executing plans of GSPNs, we are creating a generally transparent system where the user can easily understand what each agent/robot is doing at every point in time because each place is directly associated with a specific function/action. If an agent/robot is, for instance, stuck in place P1, then the programmer can have the intuition that the reason can be inside the code of the function/action.

Generally, the algorithm created to perform the execution takes as input a GSPN, a policy and a mapping between function/action and each place. When an agent/robot, which is represented by a token, enters a place, it will start executing the corresponding function/action. When it is done with it, the function/action outputs either the transition to fire in the GSPN or a flag that requires the algorithm to check the input policy in order to determine the next transition that should be fired. Afterwards, the transition is fired and the agent/robot moves into the next place, starting this process once again. The process where the function/action outputs a transition is described on Figure 4.1.

4.1 Execution Module: standalone implementation

In this section we will not be talking about robots and instead, we will refer to the tokens as agents because the standalone implementation's objective is to coordinate virtual agents instead of multi-robot teams.

In the scope of this master's thesis, an agent is represented by a token with an identifier and a state attached to it internally. The purpose of the id is to distinguish each agent, since once they enter a

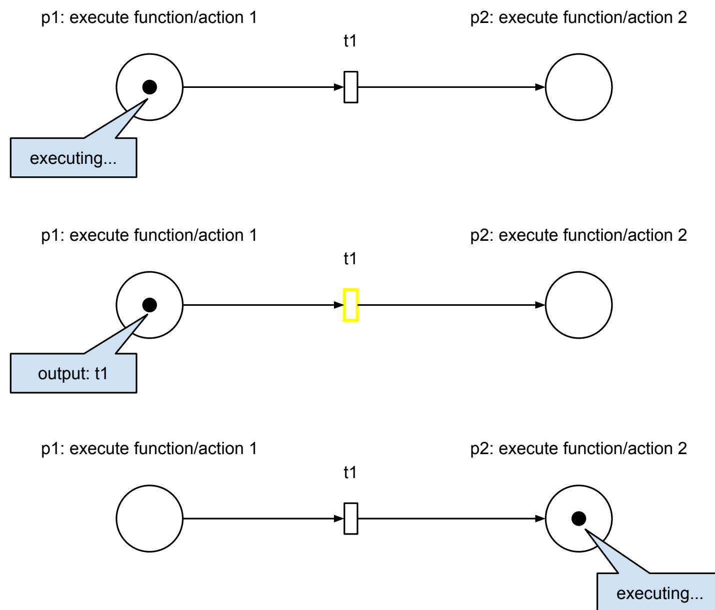


Figure 4.1: Execution process for agents/robots.

different place, a new action will have to be executed (the assignment of the ids is performed automatically by our algorithm). On the other hand, the purpose of the agents' states is to inform the execution algorithm about what an agent is doing in its current place. This is necessary because the marking for a system where an agent has already executed a function is the same as the marking where an agent is still executing it.

The state of each agent is stored as a string inside a Python list, where the order is directly related to the agents' ids: in order to read, for example, the state of the agent with id number 3, we must access the third element of the list. The current place of each agent is stored in a different Python list and its access is performed in the same way as before. This means that the length of each list will be the same as the total number of agents. On Figure 4.2 we included an example of how the lists of agent states and agent places look like, when compared with a GSPN. The main cycle will go through the list with the agent states and taking the state into account, will decide what the agent should do next.

In a very high level overview, we created a system that at all times checks the progress of each agents' execution, checks if they are done with their action or not and decides what they should do next. This cycle is infinite unless the user explicitly stops it or on the case where all agents reach a place without any output arcs.

The main idea of this module is to receive as input a policy, a GSPN model and the mapping between each place and the function that is supposed to be executed there. The output is the token game and the output of the functions that are executed. The graphical representation of this exchange is represented on Figure 4.3.

Agent states: ['Free', Done, 'Free']

Agent places: ['P1', 'P2', 'P2']

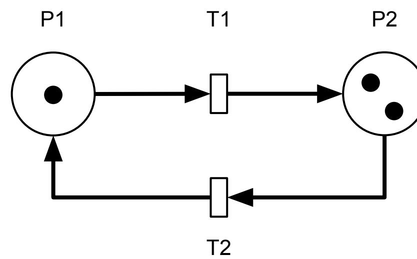


Figure 4.2: High level execution.

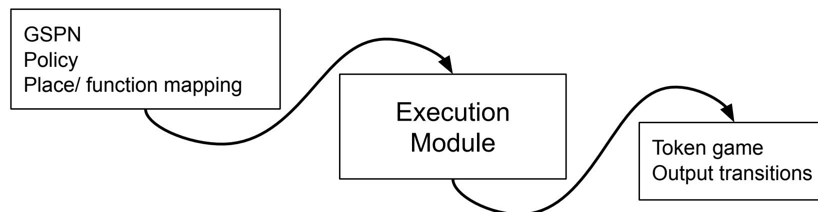


Figure 4.3: Standalone execution module inputs and outputs.

4.1.1 Allowing parallelism in our system

The mentioned cycle requires a high level of parallelism because it must guarantee that at all times, every agent is busy. Regarding parallelism we essentially have two approaches: we either use processes or threads. We chose to use threads because of its lightweight, when compared to processes. This way we can operate systems with much more agents.

By using Python 3, we had at our disposal a Python library called concurrent futures [1] which provides the same functionality as threads but in a more programming-friendly way. Concurrent futures are built over the same concepts that threads are, but they have several built-in methods that make certain operations easier. For instance, threads don't directly allow the return of function outputs, meaning that if a thread executes a certain function, its output can't be directly returned by it. Concurrent futures, on the other hand, allow functions to return outputs, which is advantageous to our implementation because after the completion of an action, the resulting transition should be returned as output.

Each agent will be associated with one concurrent future which will make the agent execute the functions corresponding to the place where it is presently. After being finished with the function, the concurrent future moves the agent into the next place and once again makes him execute the corresponding function. This cycle continues until either the execution is interrupted or the agent moves into a place with no output arcs.

[1] https://docs.python.org/3/library/concurrent.futures.html

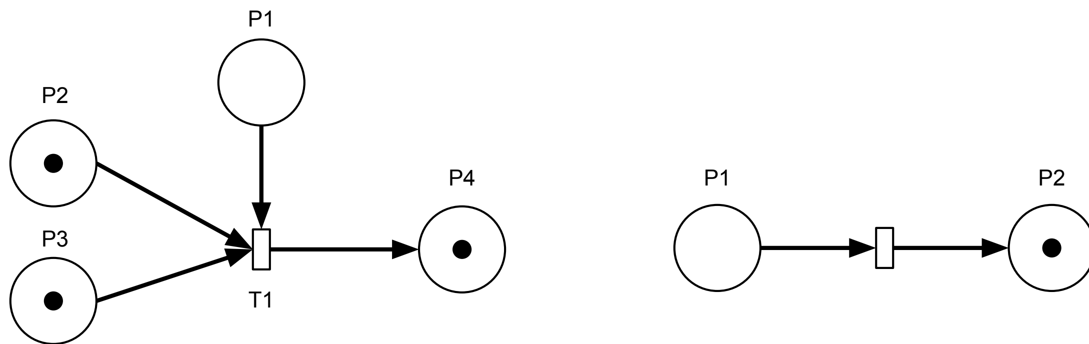


Figure 4.4: Synchronization and inactive cases.

4.1.2 Agent states

In order for our cycle to know the whereabouts of each agent, we created a mechanism to differentiate the several stages where the agents can find themselves. One should not confuse the agent states with the agent identifiers. The states represent what the agents are doing, while the identifiers are a sort of name to the agents, with the purpose of distinguishing each one.

Each agent will have 6 different states which can be attained at any point of the plan's execution:

- Free: indicates that the agent is available to start working. In the beginning of the execution, every single agent will have this state;
- Occupied: indicates that the agent is occupied with a certain action;
- Done: indicates that the agent finished the action and is ready to move onto the next place;
- Waiting: in synchronization cases, such as the one on the left side of Figure 4.4, the agents will have to wait for every agent involved to be finished. For example, on the left side of Figure 4.4, both tokens on P1 and P2 will be on this state after completing the functions of P1 and P2 because there are no tokens on P3. T1 only fires when the three places have agents in it and have finished their actions;
- Inactive: used for the cases where the agents reach a place with no output transitions. For example, on the right side of Figure 4.4, once the agent finishes the action of P2, it reaches the Inactive state because it doesn't have any output arc;
- Void: used for cases where we have ”destruction” of agents. Taking the synchronization example again on the left side of Figure 4.4, once T1 fires, the tokens on P1, P2 and P3 will be consumed and only one will be created on P4. Taking into account that we are using concurrent futures, we came to the conclusion that the deletion of these objects is quite difficult and if we were to delete them mid-execution, then many problems would arise with the cycle that goes through all concurrent futures. And so, we decided to create this new state to describe agents that aren't considered anymore.

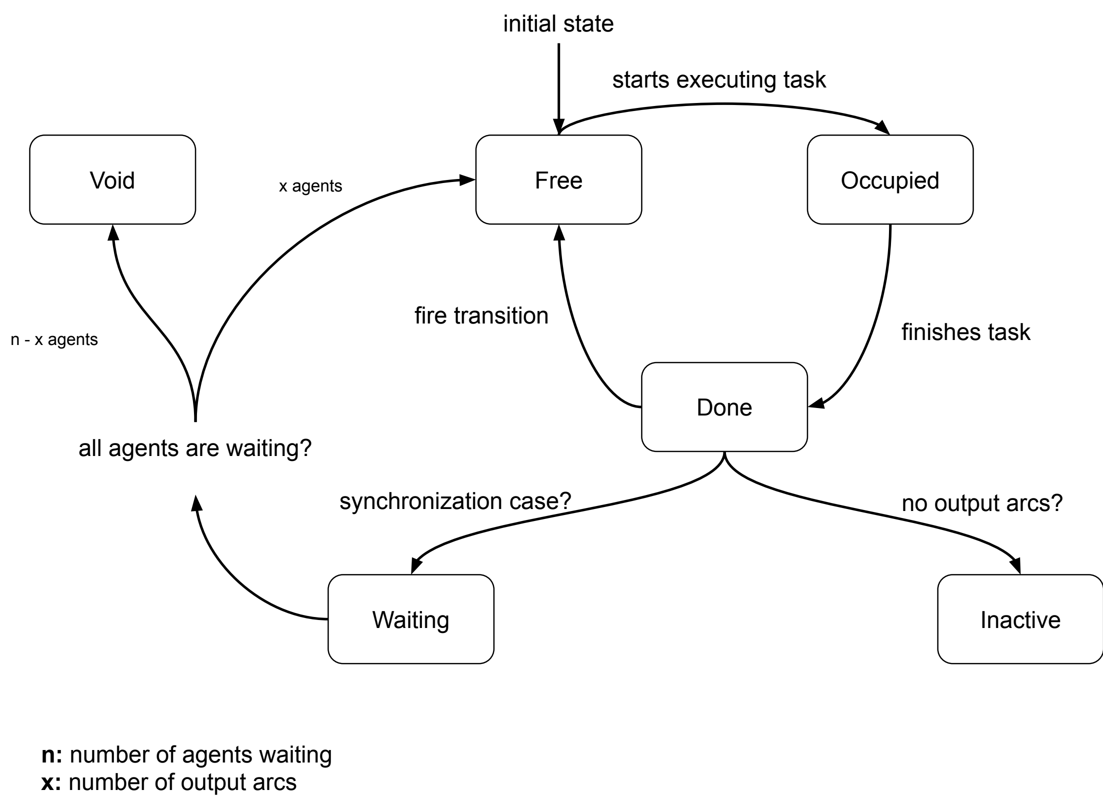


Figure 4.5: The agents' possible states.

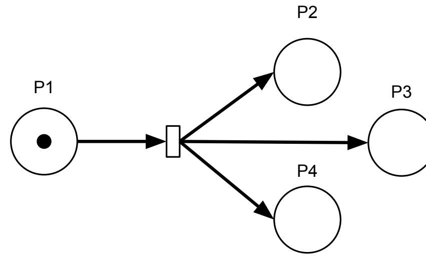


Figure 4.6: Fork case.

In order to fully understand how the states are modified during the execution, we included Figure 4.5. An agent starts the execution with *Free*. Afterwards, it will start executing the action that it is assigned to and since it is occupied, its state is altered to *Occupied*. When it finishes its action, its state is altered to *Done*. By confirming that a certain agent has *Done*, the number of output arcs is checked. If we have zero output arcs, then the agent's state is updated to *Inactive* and its execution has come to an end. If we are not in a synchronization case (cases where we have only one input arc), then its state is updated to *Free*. On the other hand, if we have a synchronization case, then the state is altered to *Waiting*. Once every agent's state is labeled as *Waiting*, then all of them are fired and once again created on the output places. In this case, if the number of agents waiting, n, is the same as the number of output arcs x, then all the agents will be labeled as *Free*. However, if n is larger than the number of output arcs x, then some of the agents will be randomly picked to be labeled as *Void*. On the case where n is smaller than the number of output arcs x, then new tokens are created into the execution and labeled as *Free*.

Done and *Free* can be confused very easily due to their similarity in wording, however, they are quite different because if *Free*, then the agent has already moved to a new place, meaning that the transition has already fired, while if *Done*, the agent still has to go through this process.

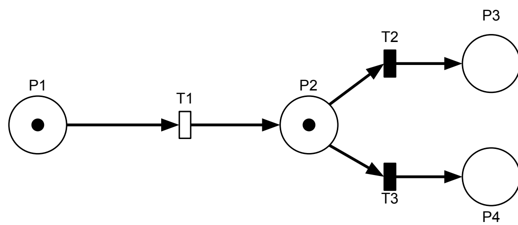
In order to cover fork cases, where we have a growth of the number of agents in the marking, such as the one illustrated on Figure 4.6, we decided that instead of creating exactly n concurrent futures for n initial agents in the beginning of the execution, we would create $3n$ concurrent futures. Since these elements are rather lightweight, we did not heavily compromise our execution.

4.1.3 Defining the user input

We simplified the user input as much as possible. The input will need to include the following elements:

- The path to the project folder;
- The GSPN model;
- The policy;
- The mapping between each place and function.

We decided that the input was to be provided through a JSON file, which is a lightweight data-interchange format. Although being highly important, this JSON file can be placed anywhere because




```
{ "project_path":"/home/execution_functions/",

  "gspn": "gspn_4_places.xml",

  "place_to_function_mapping": "{'p1': 'functions.scan_room',
                                  'p2': 'functions.decide',
                                  'p3': 'functions.pick_object',
                                  'p4': 'functions.go_to_base'}",

  "places_tuple": "('p1', 'p2', 'p3', 'p4')",

  "policy_dictionary": "{ (1,1,0,0): {'t2':0.6, 't3':0.4}, (0,1,1,0): {'t2':1} }" }
```

Figure 4.7: A GSPN and an example of the corresponding JSON input.

once the user starts the execution module, it is queried about its location. On Figure 4.7 we included a simple GSPN and an example of the corresponding input file.

The first input that should be provided is the path to the folder where the functions to execute are located. Inside this folder should also be included the GSPN represented in the xml language, which is provided through the "gspn" field. The translation from xml to a data structure that the module can understand and manipulate is possible through an xml parser which was already implemented on [5].

The mapping between each place and function is a Python dictionary (which are unordered, changeable and indexed collections with keys and values) where the key is the name of the place and the value is the name of the file followed by the name of the function to be executed. This file must be placed inside the project path that was previously defined. This mapping establishes the direct connection between the GSPN and the functions to be executed.

Finally, regarding the policy, we defined a policy Python class once again with the help of Python dictionaries. This class is composed by two elements: The policy dictionary, which is a dictionary where the key is a tuple with the marking and the value is another dictionary where the key is a transition and the value is the probability of being fired; The places tuple, which is a tuple with the order of the places that are represented in the previously mentioned dictionary.

4.2 A summary of the standalone execution module

On the following list we included a summary of the most relevant features and elements of the standalone execution module:

- The Python 3 library concurrent futures is used to allow parallelism in this module;
- All agents are the same visually, but inside the source code, each one has a different integer to identify them;
- Each agent has six different possible states to distinguish in what phase of the execution each one is currently;
- In order to use this module, the user must input the project file, the GSPN model in xml, the calculated policy and the mapping between each place and function;

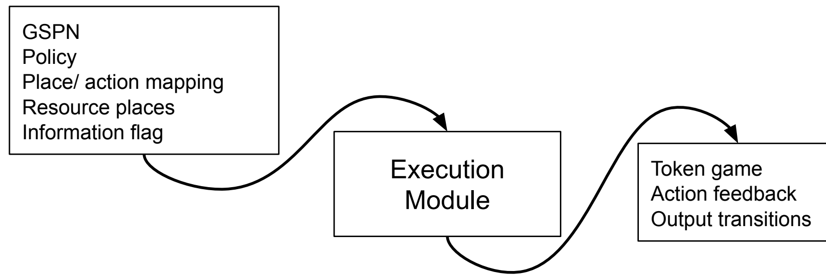


Figure 4.8: ROS execution module inputs and outputs.

- The execution module's essence lies on a single file with a main infinite cycle that constantly checks the current states of each agent and moves them accordingly.

4.3 Execution Module: ROS integration

The following section will introduce the integration of the execution module with ROS. This module was developed to be used with the latest distribution of ROS, Noetic, which implies that it was developed and tested with Ubuntu 20.

We decided to use this distribution and subsequent Ubuntu version because some of the libraries that we based our code on were only available for Python 3. Besides this, there was no reason to keep using Python 2, since it lost its support earlier this year, meaning that there wont be any more bug fixes for it.

Although on Section 4.1 we viewed agents as tokens, on this Section we describe them as robots, since our main goal is to coordinate multi-robot teams. However, since each robot is still a separate entity that will move and execute functions independently from the rest of the team members, they also need to be tagged with a unique identifier.

Instead of using token states as the main focus of the execution module, with the ROS integration we used the built-in states that are provided with ROS actions. In a high level overview, this module will take a GSPN, and for each one of its places, it will run an action server whose result will be a transition to be fired. As robots finish actions, they take the output transition and fire them, entering into a new place and running new action servers, restarting this process. The changes on the marking of the GSPNs are transmitted through either a service or a topic, depending on the quantity of information that the user wants to receive.

As inputs, the user must provide the GSPN model, the policy, the mapping between each place and action, a list with the names of the resource places and a flag that indicates the the desired quantity of information received. As outputs, the user receives the token game, the action's feedback and the output transitions (Figure 4.8).

4.3.1 General architecture of our system

Regarding the general system architecture, two options were considered:

- A centralized architecture: where there is a single entity that is responsible for each robot and the actions that they are carrying out. This single entity would also be responsible for the update of any internal structures (such as the marking, for instance);
- A decentralized architecture: where a peer-to-peer communication based protocol would be implemented in order to make the robots coordinate among themselves and execute the actions. On this architecture, there is no central entity that has knowledge of the full system. Instead, each robot contains the algorithms and the tools necessary to carry on the execution. Every time a robot changes an internal structure (such as the marking), it should be able to communicate the change to the remaining members of the network.

We chose to use a decentralized architecture due to the nature of the problem we need to solve: our framework's main goal is to be used with multi-robot systems and as such, it is not a wise choice to concentrate the entirety of our algorithm into a single point of failure. However, this choice also means that the synchronization task falls upon each member of the robot team and it also demands a larger processing capacity from each robot, which would not happen in a centralized architecture.

A final note on the architecture of our system that should be pointed out is that inside each robot we will have an execution node running inside it, which means that for each robot, we will have the same number of execution modules running. Since the execution module is the one responsible for the modification of the GSPN's marking, this means that extra steps were necessary in order to guarantee the synchronization of the GSPNs inside each member of the multi-robot team.

4.3.2 Inner robot communication: Using ROS actions to execute functions

The inner robot communication defines how each robot is able to execute the many actions of its plan. Our framework uses the actionlib [2], which is based on a client-server model where the action client communicates with an action server. Actions consist of three separate parts: a goal, which is firstly sent to the server, a result, which represents the output of the function that was executed and a feedback which can be provided to understand the system's progression.

Consider Figure 4.9, which represents a robot's inner structure. On the left side, we have the robot's action client and servers, while on the right side we have the current GSPN. For each place of the GSPN, we have one action server, which are named according to the places they represent. For instance, place P1 is associated with the P1 action server. Taking into account that the robot is on P1, then it will connect its action client to the P1 action server. This connection is represented by the bi-directional arrow between the action client and the P1 action server. In general, after completing the action, the server will either return a transition (if it is an exponential transition) or a flag that requires the policy and the current marking to be checked (if it is an immediate transition). Afterwards, the transition is fired, the

[2] `http://wiki.ros.org/actionlib`

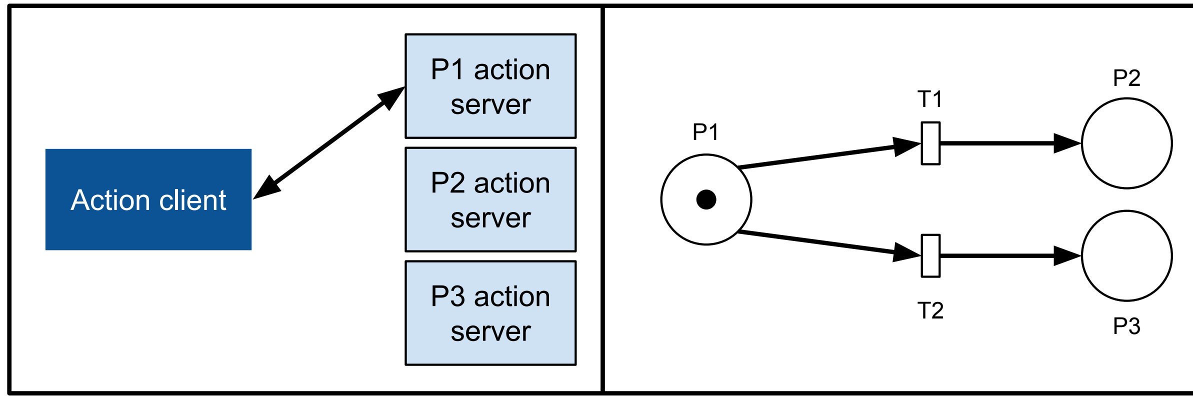


Figure 4.9: ROS action client and servers and corresponding GSPN.

robot will disconnect its action client from the current action server and connect it to the action server of the new current place. This process repeats itself until the robot's current place has no output arcs or until the user explicitly stops the execution.

Taking into account that every robot will have to run inside it one server per place of the GSPN, this means that for example, a GSPN with three places and three robots, a total of nine action servers will be created.

For a better understanding of the algorithm behind the execution, please consider Algorithm 1. The red and blue portions will be introduced on the next Section. For now, only consider the black colored portions of the pseudo-code. The algorithm takes as inputs a GSPN, the policy, the mapping between each place and server and a flag to decide whether the outer communication is performed via topic or service. The list of resource places is not used directly in this algorithm and as such, we chose to omit it. The first step is to know the current place of the robot, so that it can map it to the corresponding action server. Afterwards, the action is executed and once it is finished, the algorithm checks whether the robot's current place has output arcs or not. If not, then the execution is finished for the robot. On the other hand, if there are output arcs, then the algorithm checks the result provided by the action server. If the result equals to the string "None", this means that the transition to be fired is immediate. As such, the algorithm must take the current marking, check the policy that was provided and fire the resulting transition. On the other hand, if the result is a concrete transition, this means that the transition to be fired is exponential and as such, it is simply fired. The remaining parts of the algorithm will be explained on the next Section.

4.3.3 Outer robot communication: Using ROS topics and services to change the GSPN

In order to guarantee a robust exchange of information between every robot, such as the exchange of the GSPN's current marking, we chose to use ROS topics [3]. In its essence, a topic is a way of communicating between nodes through ROS using a publisher/subscriber architecture. When a message is

[3] http://wiki.ros.org/Topics

Algorithm 1: ROS execution algorithm.

```
Input: GSPN, policy, mapping between each place and server, communication flag to decide
       between topic and service
Output: Execution of actions
1  while True do
2      Get the robot's current place, connect to the corresponding action server and start executing
        action;
3      if The action server is done then
4          Check GSPN and get the output arcs of the current place;
5          if There are any output arcs then
6              Get the result returned by the action server;
7              if Result is immediate transition then
8                  Check the communication flag;
9                  if Communication flag is topic then
10                     Check transition to fire in the policy and fire it;
11                     Share the firing information with the topic;
12                 else if Communication flag is service then
13                     Get the other robots' current places, check the policy and get the other robots'
                        current states;
14                     if All robots are in state 'Done' then
15                         Fire the transition obtained from the policy;
16                         Share the firing information with the topic;
17                     else
18                         Wait
19             else
20                 Fire the transition obtained from the action server;
21                 Share the firing information with the topic;
22         else
23             End execution of this robot.
24     else
25         Wait for result
```

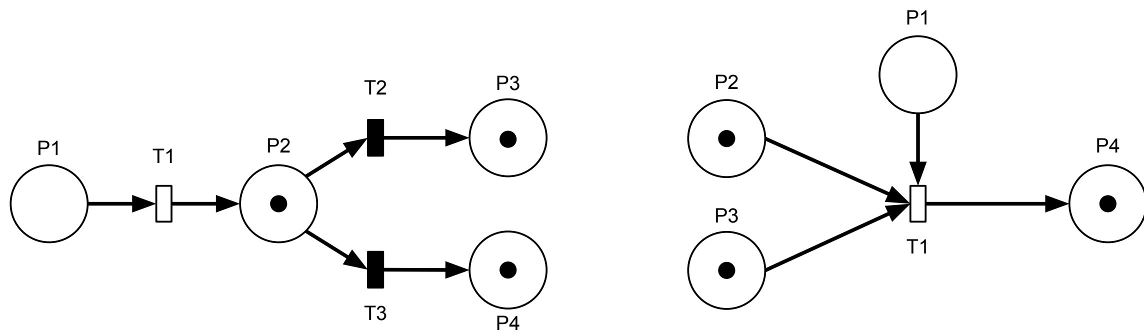


Figure 4.10: Moments where robots need to communicate.

published to a topic, every subscriber of the said topic executes a listener callback function. This architecture is also supported by other callback functions such as feedback and cancel callback functions. In our specific case, every robot is both a publisher and a subscriber to the exact same topic, which simplifies the task of synchronizing the system but can lead to unwanted behaviours, which we will further get into.

Communicating through topics can, however, make the understanding of the network much harder since every time there is a change, a message is published to the topic. We concluded that there are only two moments when the robots actually need to communicate their current place, which is the moment when a robot has to choose which immediate transition it should fire (such as the robot on P2 on the left side of Figure 4.10) and the moment when a robot is on a synchronization case (such as the robots on P2 and P3 on the right side of Figure 4.10). Taking this into account, we decided to create an extra communication channel, through ROS services [4], which instead of being called every time someone publishes to it, is only executed when it is necessary.

Using ROS topics to communicate changes

Every time a robot finishes executing an action, it publishes the changes to a topic and the other robots listen to it, in order to also apply those same changes into their own inner structures. Every publisher publishes the fired transition, the id of the firing robot, the resulting marking and a timestamp of the time when the publishing occurred. The reason why we chose each one of these elements is summarized in the following list:

- fired transition: to inform the remaining team members of the changes that occurred to the network so that they can also apply them to their version of the marking;
- resulting marking: to confirm how the marking should look like after the transition was fired;
- firing robot id: to avoid firing the same transition twice and to know which token has to be consumed and created. Taking into account what was said before about the publisher/subscriber architecture, our listener callback function takes this id and on the case where it corresponds to the id of the

[4] `http://wiki.ros.org/Services`

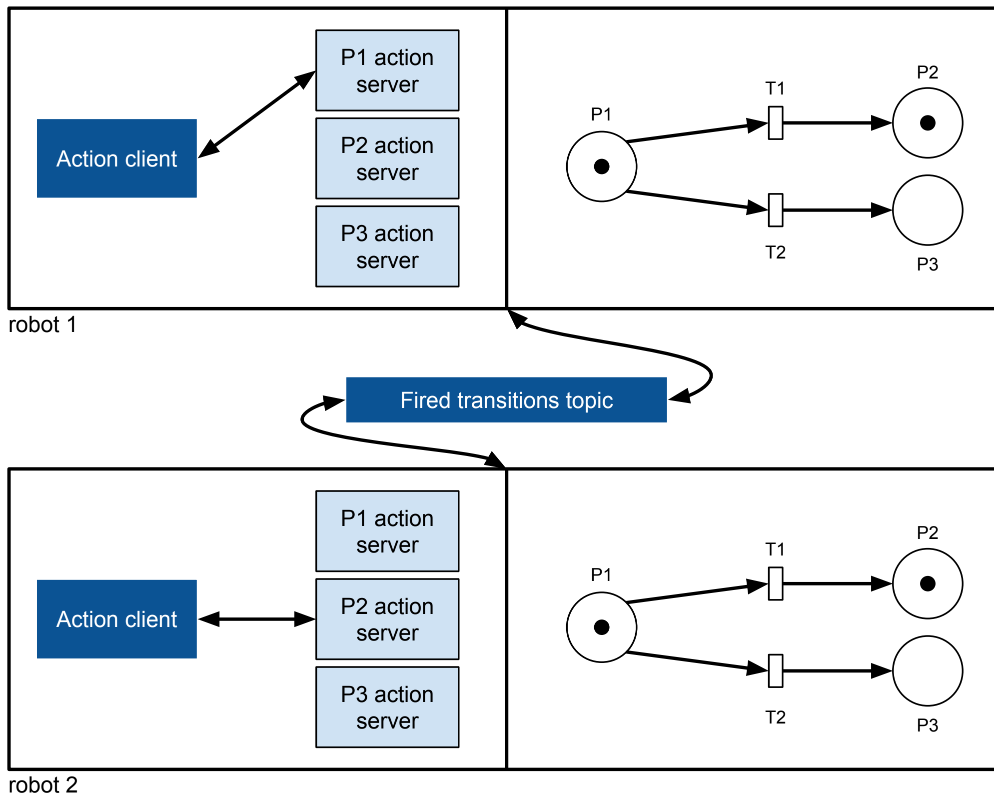


Figure 4.11: Execution module outer communication.

robot that is listening, then the message is ignored. However, if the id is different from the id of the robot listening, the transition on the messaged is fired in the current robot's GSPN;

- firing timestamp: for debug purposes and to take metrics from the execution.

On Figure 4.11 you can analyze the outer communication model. In this example we have a multi-robot system composed by two robots and a GSPN with three places, which translates into three servers for each robot. Robot 1 is on P1 and as such, it connects its action client to the P1 action server. On the other hand, robot 2 is on P2 and as such, it connects its action client to the P2 action server. If one of the robots finishes the action, they will fire a transition and as such, the marking will change. These changes are published to the fired transitions topic and each subscriber of the same topic receives them. On Algorithm 1, the topic communication is performed by the red colored portions of the pseudo-code. Every time a robot fires a transition, it will have to publish to the topic the information that was previously mentioned, so that the remaining members of the team can update their local GSPNs.

Using ROS services to communicate changes

On Sub Section 4.3.3 we briefly explained that there are only two moments when the robots need to explicitly communicate with each other:

- when the robot has to choose which immediate transition it will fire, because when an immediate transition has to be fired, the policy has to be checked. As defined in Definition 2, a policy corre-

sponds a set of states to a set of actions. The states are, in our context, a marking. And so, for each marking, there will be a resulting action. Since the marking is essentially what decides which transition will be fired, then this means that the key point of the immediate transition is to know the configuration of the marking;

- when the robot is on a synchronization case, because on this case, the robots need to know exactly whether every other input place has at least one robot or not.

To solve the first bullet point, we defined a very simple service for each robot which returns its current place. Every time a robot needs to make a decision regarding which Immediate transition will be fired, it calls the service of every remaining member of the team and registers their current places. After having the quorum from every member, the robot builds the current marking of the GSPN and makes a decision regarding the transition to fire, based on the obtained marking.

This implementation has two main advantages:

- It requires less computational power from the robots which in the long run means that our solution becomes scalable and allows for the user to test plans with a bigger number of robots;
- Every time a robot publishes to the topic or calls a service, an output message is generated to inform the user. Since services are called less often than topics (because they are only called on the two cases presented on Figure 4.10), than this means that the number of output messages for the user will be much smaller which implies that the system will also be easier to follow and understand.

Figure 4.12 includes a general explanation of how the service procedure works. Firstly, the robot that needs to make a decision calls the services of the remaining members of the team, which returns the current place of the said members. Finally, the robot builds a marking with the information it received and makes a decision regarding the transition that it will fire.

To solve the second bullet point, we used the previously introduced service and two new ones: a service to return the robot's current activity state and a service to change the robot's current place. Both these values are defined as global variables and the robot's current activity state is either *Done* or *Doing*, depending on whether it is done with its current action or not.

If the robot encounters a synchronization case, the first thing it does is check whether the current marking is the same as the one that will allow the transition to fire. If it is not, then the robot stays blocked and waiting for a change in its state. On the other hand, if it does correspond to the necessary marking, then we check where each robot is with the current place service and whether it finished its action or not. If some of the robots are still *Doing*, then the current robot stays blocked and waiting for a change in its state. If all robots are *Done*, this means that the current robot is the last one to be ready and as such, it will be responsible of connecting to the service used to change the current place of every robot and calculating their new places. After assigning new places to every member of the team, the last robot simply fires the transition and assigns its own place as well.

Considering Algorithm 1, the blue colored lines represent, in a very simplified manner, the lines that were introduced to allow this communication mechanism's functionality. If the communication flag is

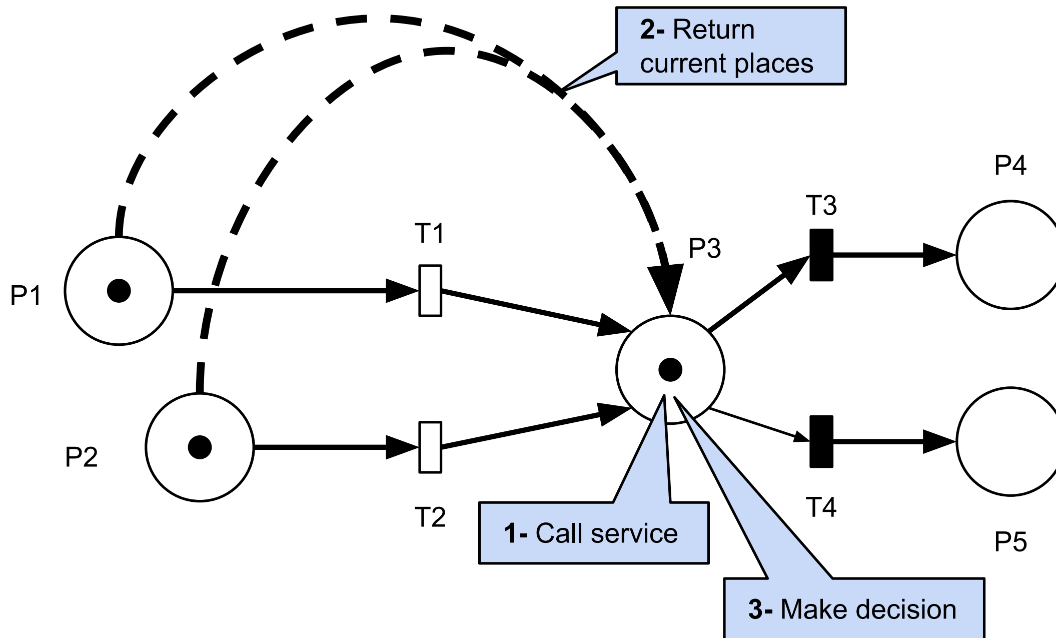


Figure 4.12: Service Process.

set to service, then this means that the outer communication is performed via service. As such, the algorithm must get the other robots' current places, check the input policy and get the other robots' current states. If all robots are done, then the transition is fired, however, if some robots are still in the middle of the execution of an action, then it will wait for them to finish.

4.3.4 Resource tokens and resource places

Besides having robots on our GSPNs, we also have the possibility of having resource tokens and resource places. In order to allow their existence, we must first define them:

Definition 3. *A resource place is a place of the GSPN where no action is executed and where instead of robots, we have resource tokens. These tokens can be used to take metrics from the execution of a GSPN.*

Considering Figure 4.13, **P2: Counting measurements done** is a resource place and as such, the tokens inside it do not represent any robot and therefore are not associated with the execution of any action. These tokens exist in this place in order to inform the user of how many measurements were done, through the analysis of the GSPN's marking. Every time transition **t1: Temperature measured** is fired, a new token is created both on **P2** and **P3**.

The user should provide a list with all the resource places of the GSPN, so that the execution algorithm, when confronted with such a place, knows that a real robot should never be placed on that particular place.

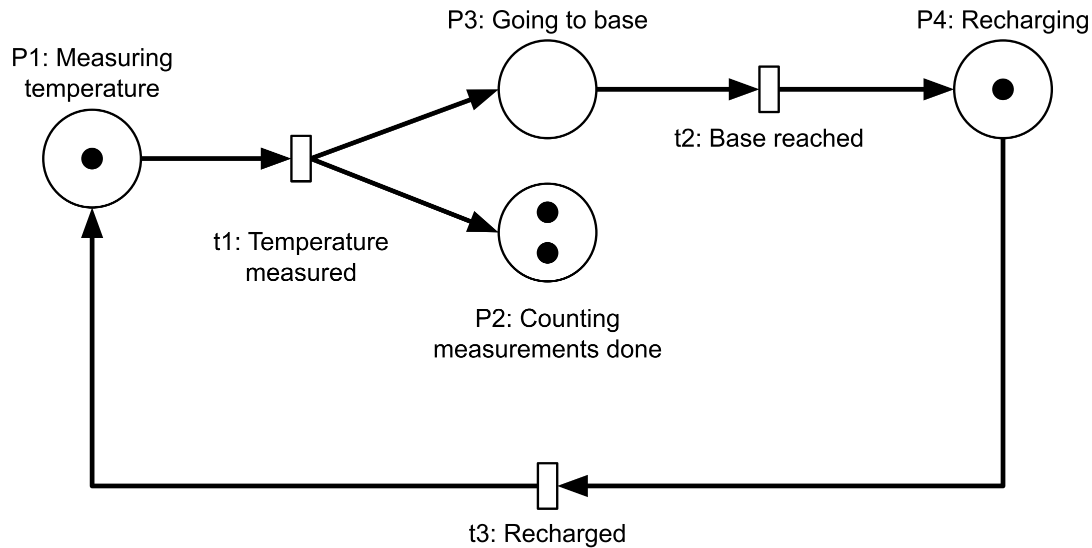


Figure 4.13: Resources example.

4.3.5 Limiting the possible input GSPNs

Before the execution starts, our algorithm analyzes the GSPN that the user provided as input (see Figure 4.14). This analysis depends on the list of resource places that the user also provides in the beginning of the execution. The algorithm creates a temporary GSPN, equal to the one provided and removes the resource places by using a ”delete place” function from the GSPN module. Afterwards, it removes the transitions that were left unconnected with a ”delete transition” function also from the GSPN module. Next, it obtains the reachability graph for this newly created GSPN and counts the number of tokens in each node of the graph. If the number of token sums changes in any part of it, then there is either creation or destruction of tokens, which cannot be allowed since we are working with robots, which are physical entities, and we reject the input GSPN. However, if the number stays constant throughout the reachability graph, we accept the GSPN.

4.3.6 Defining the user input

The input will need to include several elements such as:

- The GSPN model;
- The policy;
- The mapping between the place, the server name and the action type;
- A flag that indicates whether we want to use the topic (full synchronization) or the service (partial synchronization) for the outer communication;
- A list with the names of the places that are considered to be resources.

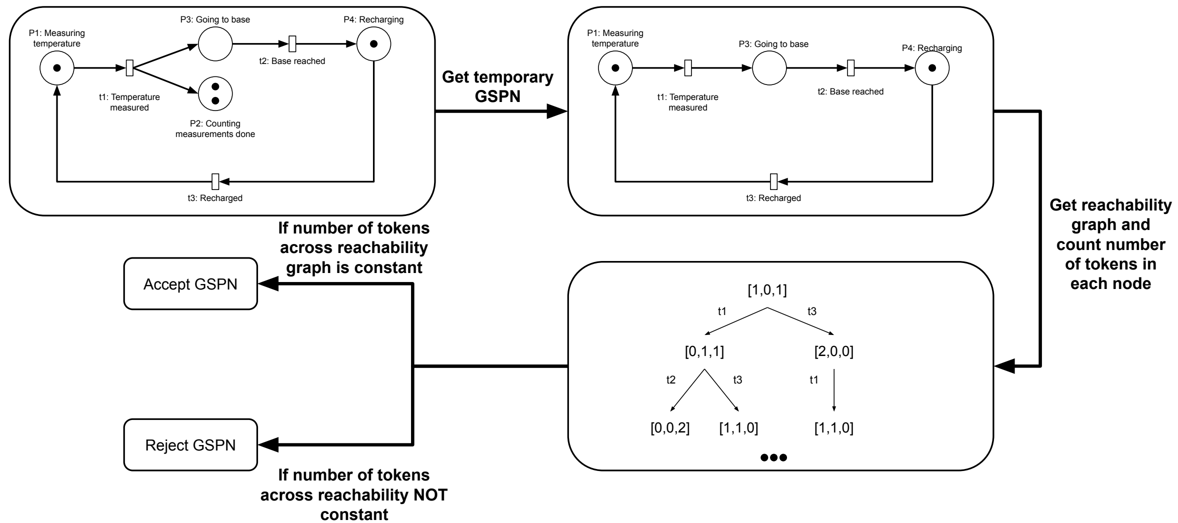


Figure 4.14: Algorithm to limit possible input GSPNs.

Regarding the GSPN model and the calculated policy, the definitions remain the same as the ones presented on Section 4.1.3. The mapping between the place and the name of the file to be executed there is simply a dictionary where the key is the name of the place and the value is a list with the type of action and the server name. To determine whether the user intends to use full or partial synchronization, a boolean variable should also be included. And finally, the list of resource places will help our framework determine whether the input GSPN is valid in our scope or not.

4.4 A summary of the ROS execution module

On the following list we summarize the main characteristics of the execution module in ROS.

- Our system architecture is decentralized in order to make our framework as robust as possible to robot related flaws. This way, if one of the robot stops working spontaneously, the system stays up and running since every single robot knows the configuration of the network;
- The inner communication is performed using ROS actions. Each robot has an action client that connects itself to different action servers, depending on the place where it is presently. Each robot has one action server for each place of the GSPN and each robot connects to its group of action servers. The encapsulation is achieved by using different *namespaces* for each robot;
- The outer communication is performed using a ROS topic or service, depending on the chosen synchronization type. If the synchronization type is full, the robots publish and subscribe to the same topic and every time a transition is fired, a new message is published to the topic. On the other hand, if the synchronization type is partial, the robots only communicate with each other through a service when they are before an immediate transition or when they are in a synchronization case;

- We defined a concept known as *resource places* which represents places that are used to count elements from the execution and inside them we have resource tokens;
- We created an algorithm to reduce the number of accepted GSPNs in order to avoid unwanted cases.

Chapter 5

The Visualization Module

The visualization module's main goal is to visualize a GSPN and its execution. This visualization can be both offline, where the user has the chance of simulating the built GSPN, and online, where the user can monitor the progress of the execution of a GSPN.

On the case where the user only intends to visualize and simulate a GSPN, the offline visualization is used. On the other hand, if the user wants to execute a plan, the online visualization is used, which communicates with the execution module.

While on the execution of a plan, the algorithm runs either a function or an action for each place, on a simulation, the algorithm simply fires a number of random enabled transitions with the purpose of illustrating how the designed GSPN will work in a real execution.

Although we created the online visualization for both the standalone and ROS integrated versions, the offline visualization was only created in the standalone version. We chose to do this because the idea behind the visualization module was to create a single interface for everything and we did not want to overwhelm the ROS visualization module's interface. However, as we explain further down, a robot developer can always use the offline visualization from the standalone version because the necessary input is rather simple.

5.1 The visualization module architecture

The visualization module's architecture is composed by three different elements, which are very common in today's web-apps: a backend, a frontend and a web framework tool to allow the flow of information between the two previous elements.

Since our whole framework was already built on Python 3, we decided to use this language for our backend as well. Regarding frontend, we chose to use a combination of HTML, CSS and Javascript, since our application will be very simple-looking. Another element we incorporated into the frontend was a framework to build graphs, Vis.js [1]. Finally, we chose to use Flask [2] in order to allow the communication

[1] `https://visjs.org/`
[2] `https://flask.palletsprojects.com/en/1.1.x/`

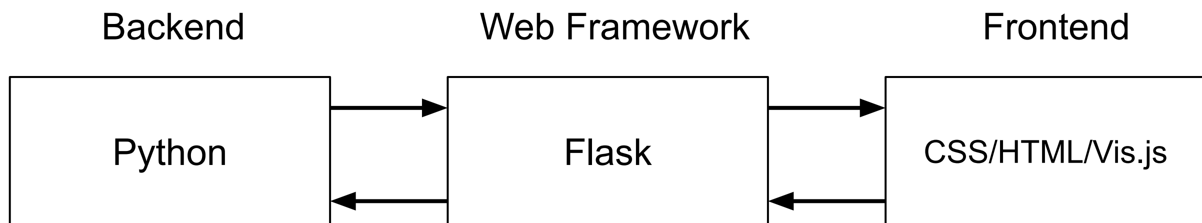


Figure 5.1: Visualization module inner architecture.

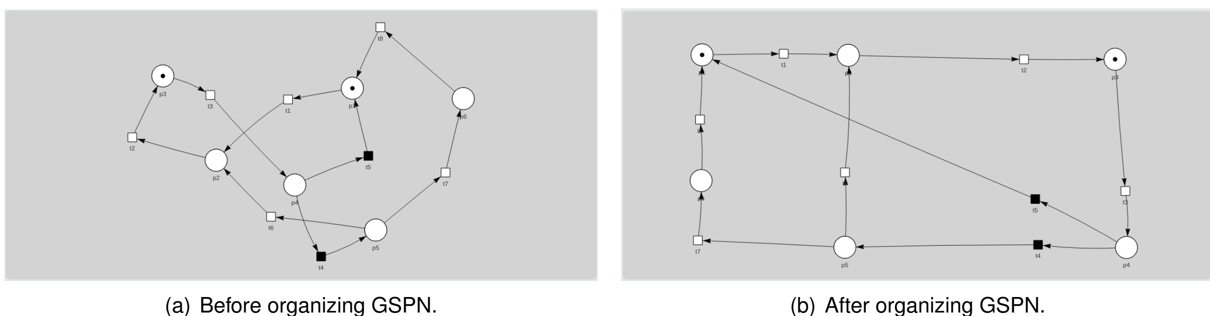
between the frontend and the backend. On Figure 5.1 you can analyze the inner architecture of the visualization module. The arrows represent data flow.

To make the mechanics of our architecture clear, suppose the following simple example: Imagine an interface with only a button that when clicked, prints a message on the screen. When a user clicks that button, its ID is sent through Flask into the backend, where it is processed. After this, the backend returns another message through flask that instructs the frontend to display a message and finally the message is shown into the user's screen.

5.2 Visualizing GSPNs

For simplicity reasons, we minimized a GSPN's visual complexity into a graph with two different types of nodes (transitions and places) and directed edges. With this in mind, we decided to use a Javascript modelling tool, Vis.js. Our task was to create a parser that would take a GSPN represented in xml and render it as a Vis.js graph.

An interesting characteristic of Vis.js is that although the programmer can choose the absolute position of each node of the graph, it can also simply place it randomly. However, Vis.js is malleable to the point that after rendering the graph, the user can actually move and select each node and zoom in or out on the rendered area. This means that after generating a GSPN, the user of our framework will be able to move around each place and each transition until reaching a configuration that is intuitive. An example of this is presented in Figure 5.2.



(a) Before organizing GSPN. (b) After organizing GSPN.

Figure 5.2: Using Vis.js to visualize a GSPN model.

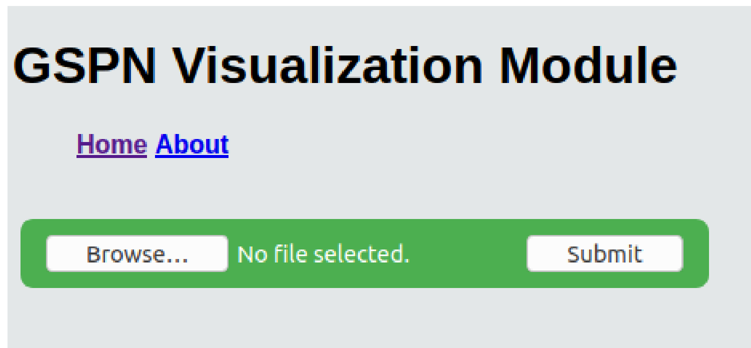


Figure 5.3: Visualization module input screen.

5.3 Offline visualization

The offline visualization is not integrated with ROS, however, if a robot developer wants to visualize its GSPN, all that is necessary is an input with the same configuration as the one mentioned in Sub Section 4.1.3 (since the robot developer will not execute the GSPN using this module, it is not necessary to provide a real policy or a real mapping between each place and function) with the intended GSPN. The input is provided through the screen presented on Figure 5.3.

This module also allows the simulation of one or n steps of the token game. Notice that this is not an execution, instead it is only a simulation that can be used to understand how the GSPN would react in an execution scenario. We define one step as one transition being fired. On the case where we have two possible exponential transitions, one of them will be chosen randomly or according to the firing probabilities set by the user.

Other features of this module include the possibility of firing a chosen transition (if it is enabled), the possibility of restarting the simulation and the highlight of transitions that are about to be fired. On Figure 5.4 we included the final version of the visualization module's interface. The simulation tools are available with "simulate 1 step", "reset simulation", "clear network", "fire chosen transition" and "simulate n steps". On the right panel, the user can take some metrics from the input GSPN, however, most of the metrics are not available at the present time.

5.4 Online visualization

Similarly to the execution module, the online visualization section is composed by two different implementations: a standalone version and the one integrated with ROS, both to be used with the corresponding execution modules.

5.4.1 Online visualization: standalone implementation

Considering Figure 5.4, the online visualization can be initiated and stopped by pressing "start execution" and "stop execution" respectively. When the former button is pressed, the execution module is initiated by the backend of the visualization module and on the frontend, a timer is set to make sure that every

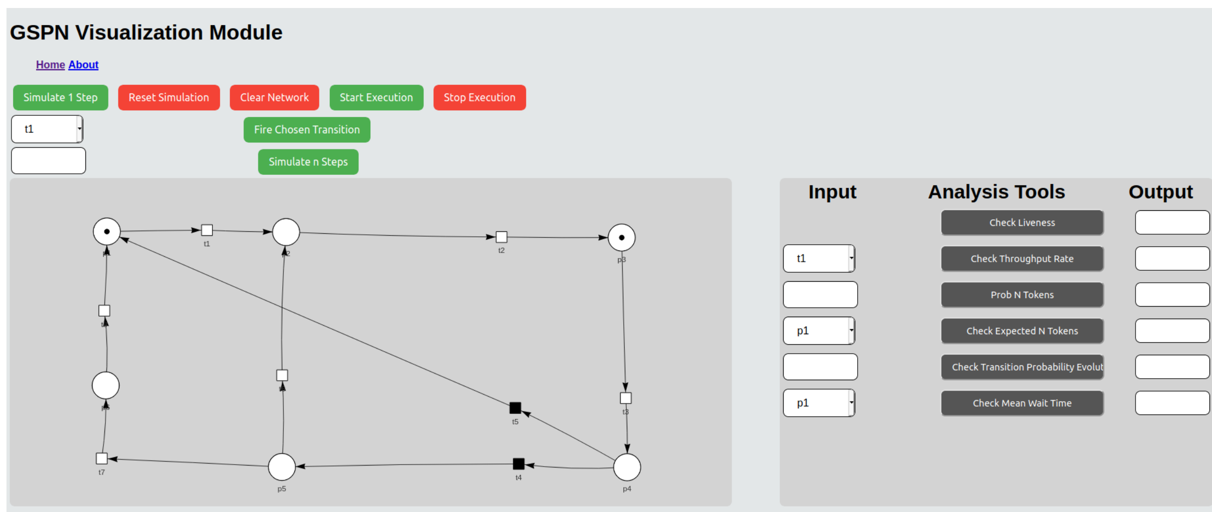


Figure 5.4: GSPN visualization module in Python 3.

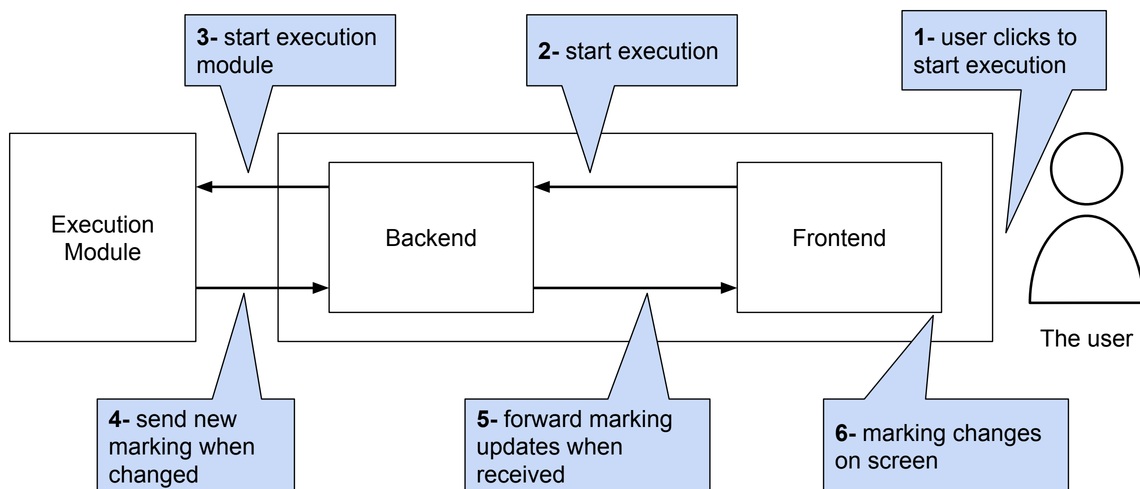


Figure 5.5: Interaction with the standalone online visualization.

second the marking updates are sent over from the backend into the frontend. This process is described on Figure 5.5 in a visual way. We chose to omit the middle man, Flask, in order to simplify the Figure. Steps 4, 5 and 6 occur every time the marking changes until the execution reaches its end.

5.4.2 Online visualization: ROS integration

By using ROS, we were able to take advantage of all its built-in functions and components. Topics and topic callbacks were crucial while building this module. Whenever a publisher of the topic mentioned in Section 4.3.3 publishes to it, the backend of the visualization module registers this change and applies it to the GSPN presented in the frontend. This mechanism is illustrated on Figure 5.6 and in order to achieve it, we created a new subscriber to the above mentioned topic in the backend of the visualization module. We then associated a callback function to it, which saves the fired transition and the resulting marking into a Python list. On the frontend side, a periodic function that fetches the saved updates is

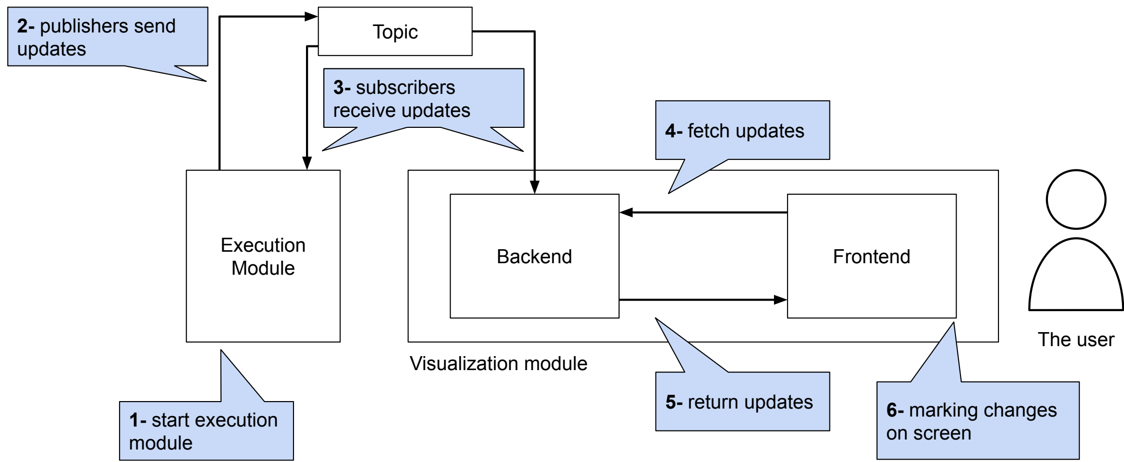


Figure 5.6: Interaction with ROS online visualization.

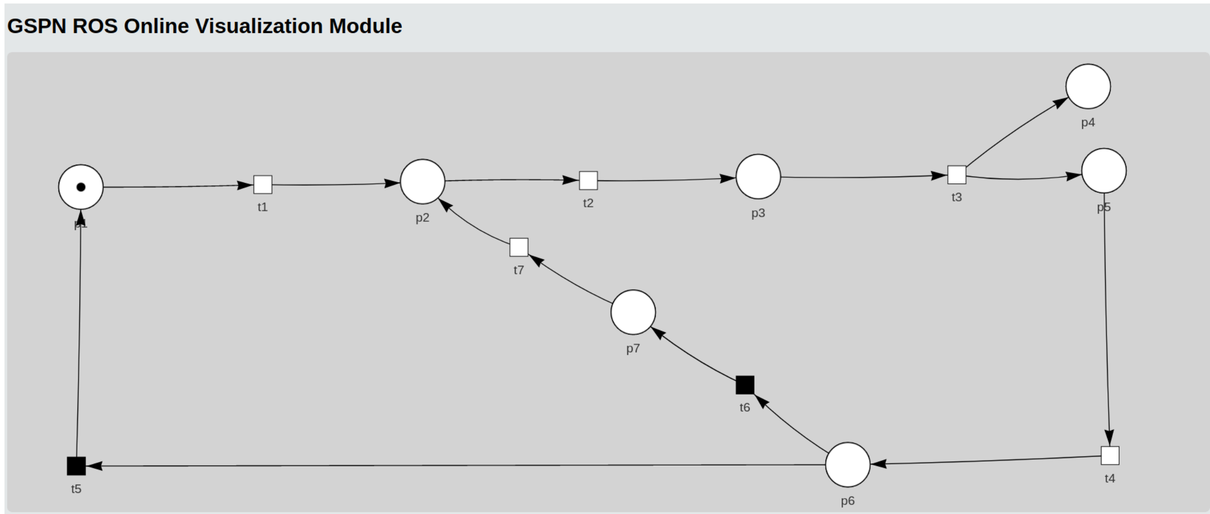


Figure 5.7: GSPN visualization module in ROS.

executed every second and applies them.

And so, the only connection between the execution module and the visualization module is the topic that is used to divulge the transitions that were fired. This abstraction makes these two models more independent from each other than on the standalone version of our framework.

Although elegant, this solution has a slight limitation: on Sub Section 4.3.3 we mentioned the option of executing the plan with services (instead of topics) to improve efficiency. However, while using the online visualization, the user is obliged to use the communication with the topic, since the whole module depends on it.

The final version of the online visualization module interface is presented on Figure 5.7.

5.5 A summary of the visualization module

The following list presents the most important topics that were discussed in this Chapter:

- The visualization module architecture is composed by a backend in Python 3, a frontend in CSS and HTML and the communication between both elements is made with a Python based framework, Flask;
- The visualization module uses a Javascript framework to design graphs, Vis.js in order to be able to design GSPNs;
- The offline visualization can be used to visualize and simulate a GSPN;
- Generally speaking, the online visualization module communicates with the execution module in order to be able to reflect the occurring changes;
- The online visualization in ROS is based on the topics that were already built in the execution module. When a transition is fired on the execution module, a message is published to the topic and the visualization module, which is a subscriber to this topic, applies the necessary changes in the visualized GSPN.

Chapter 6

Results

We created two different types of tests, both built solely for the modules integrated with ROS: on the first batch of tests, we decided to test the scalability of our framework *without* simulated robots, meaning that we did not use the Gazebo simulator. On the second batch of tests, our goal was precisely to test our framework with a multi-robot simulated system on Gazebo, with a scenario as realistic as possible, to illustrate how our framework could be used in the future.

The setup that was used to test this framework consists of a laptop with an Intel Core i7-8550U CPU and 11 Gb of RAM.

6.1 Testing the framework

On this section we aimed at testing only our framework. We created three different types of tests:

1. The number of robots is increased while the number of actions that they can execute is kept constant;

2. The number of robots is kept constant while the number of actions that they can execute is increased;

3. The number of robots is kept equal to one while the number of actions that it can execute is increased.

In the three tests we measured the percentage of CPU and virtual memory (not including swap) that was consumed throughout the execution by using two functions from the python library, psutil [1], cpu_percent and virtual_memory respectively. For each moment of the test, we took a sample of five measurements and calculated the average of them. These values should be taken carefully, since CPU and memory consumption are variable, depending on the moment they are analyzed. However, for a better contextualization, the baseline of each value was 0,8% of CPU and 25,9% of virtual memory.

[1] https://pypi.org/project/psutil/

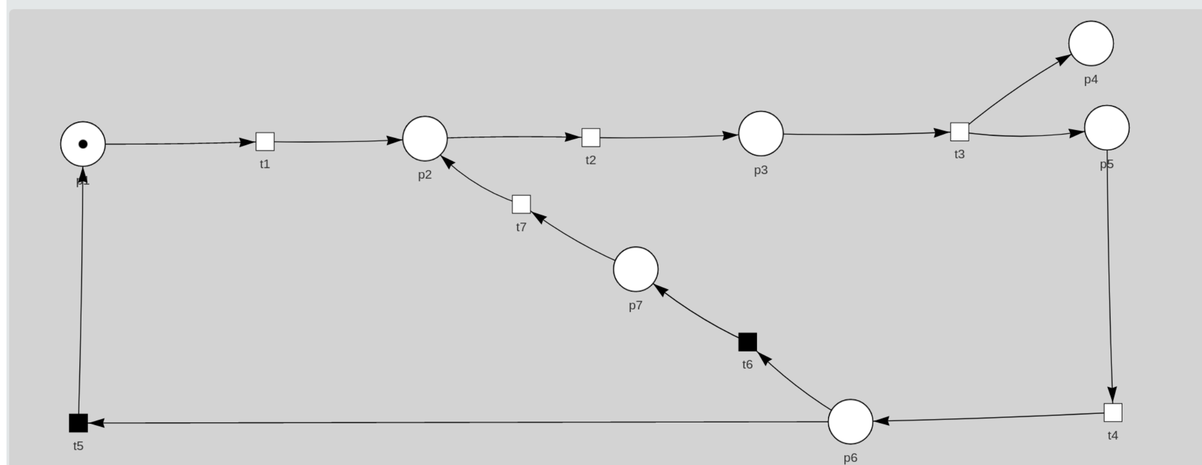


Figure 6.1: GSPN to test the increase of tokens.

6.1.1 Increasing the number of tokens

The GSPN we chose is presented in Figure 6.1 and in order to simplify it as much as possible, we chose to shorten the names of each place and transition. Besides this, inside each place, each identified token simply executes a *for-loop* with an addition operation and a sleep for 0.2 seconds inside it.

The results can be analyzed in Figure 6.2 and as expected, both values increase with an increasing number of tokens. Besides the increase on consumed resources, the time that the system took to start the execution also increased with the addition of more tokens, creating wait times of around 20 minutes, for more than 13 tokens.

6.1.2 Increasing the number of places

Next, we tested how our system would react to an increase in the number of places. We decided to start with a circular GSPN with only two places. The initial GSPN can be analyzed in Figure 6.3 and as we added new places (and transitions of course), we kept the circular structure. However, the number of tokens remained equal to five all throughout the execution.

The results are presented in Figure 6.4 and as expected, the percentage of CPU and virtual memory both increased with the increase of the number of places. Also in this test, as we inserted more places, the time it took the system to start the execution also took longer. And at times we had to wait around 20 minutes for the execution to begin.

We took the percentage of CPU used on the first test and the percentage of CPU used on the second test and related both these values to the total number of action-servers and obtained the results on Figure 6.5. Taking into account the growth in action servers of each test, we can easily see that although on both cases the percentage of CPU increases heavily, on the test where we are increasing the number of tokens, the percentage quickly surpasses the percentage of the test where we are adding places.

Although in some points, a smaller number of action-servers corresponds to a bigger percentage of

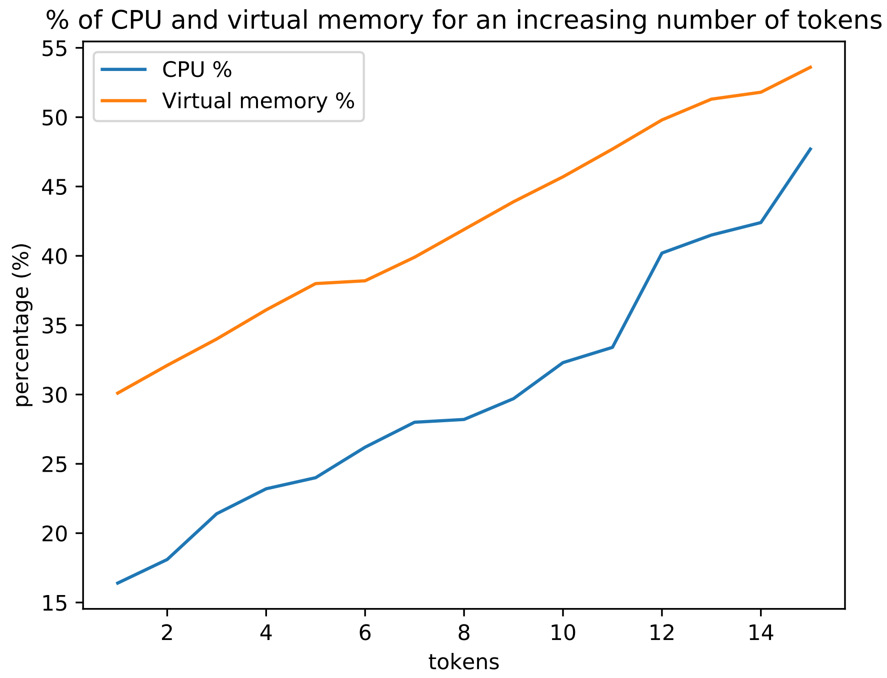


Figure 6.2: Results for an increasing number of tokens.

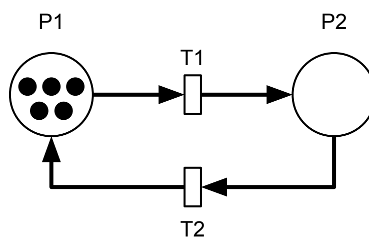


Figure 6.3: Initial GSPN to test increase of places.

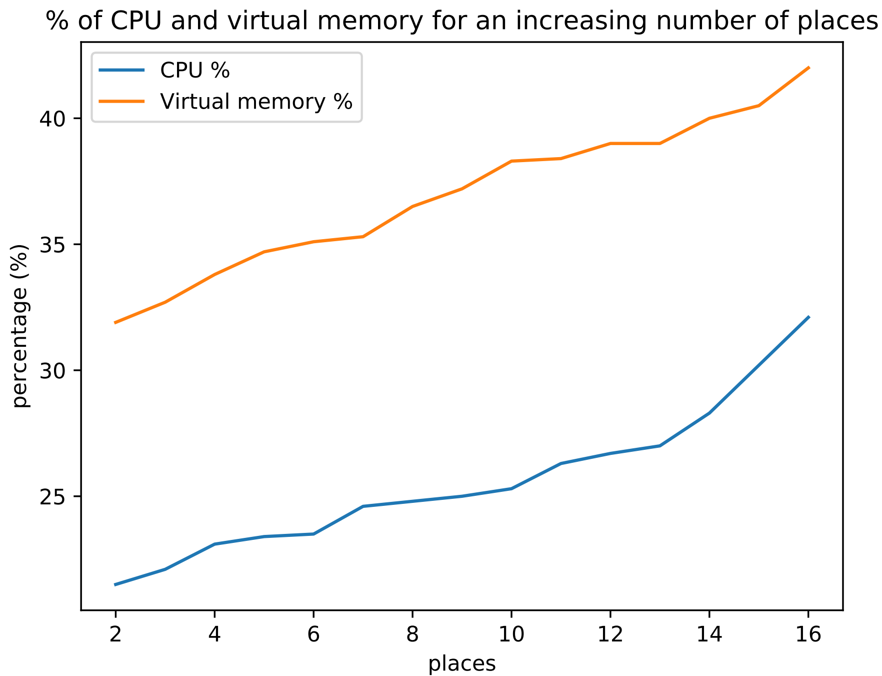


Figure 6.4: Results for an increasing number of places.

CPU, we can see that every time we add an action-server, the percentage of CPU increases.

6.1.3 Increasing the number of places with only one token

In order to test how our system would react in a truly decentralized architecture, where each robot only runs their action servers, we ran a third set of tests, very similar to the previous one, but with the exception of having only one token to execute the plan.

We started this test with two places, two transitions, but only one token, and went up to 35 places and afterwards we went up to 100, counting 5 by 5. The results are presented in Figure 6.6 and they are very similar to the ones obtained previously. However, we registered a major difference in terms of the time it took to initiate the execution. On the two previous tests, we pointed out that there was a major time gap between the moment we launched the command to start the execution and the moment the execution started. On this third test, the time gap was very close to zero.

6.2 Testing the framework with simulated robots

6.2.1 Problem Description

Consider the turtlebot basic environment (Figure 6.7) and suppose that it represents a room that needs 24/7 surveillance on the temperature of four different areas. The temperatures should not go above a certain value because if they do, systems can fail spontaneously. A team of mobile robots is a good option because it automates a monotonous but important task and, besides this, the more robots we

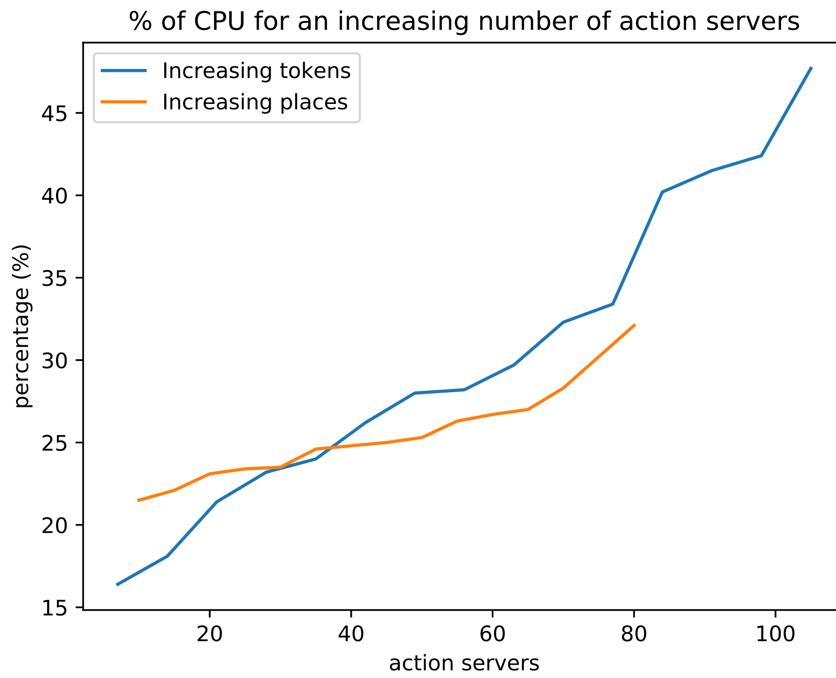


Figure 6.5: Results for an increasing number of action servers.

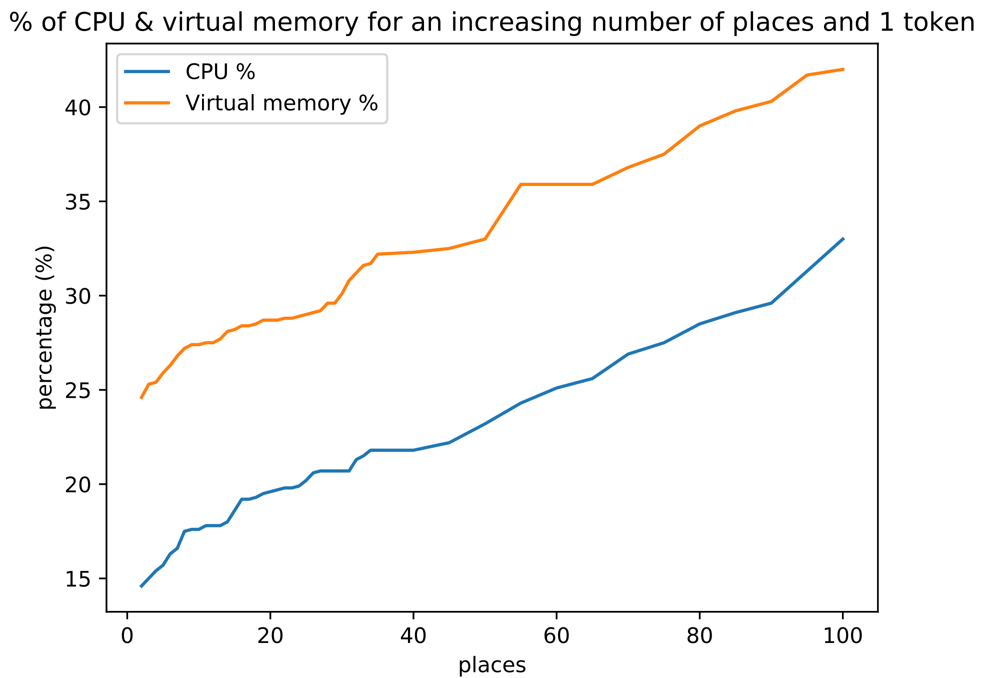


Figure 6.6: Results for an increasing number of places with only one token.

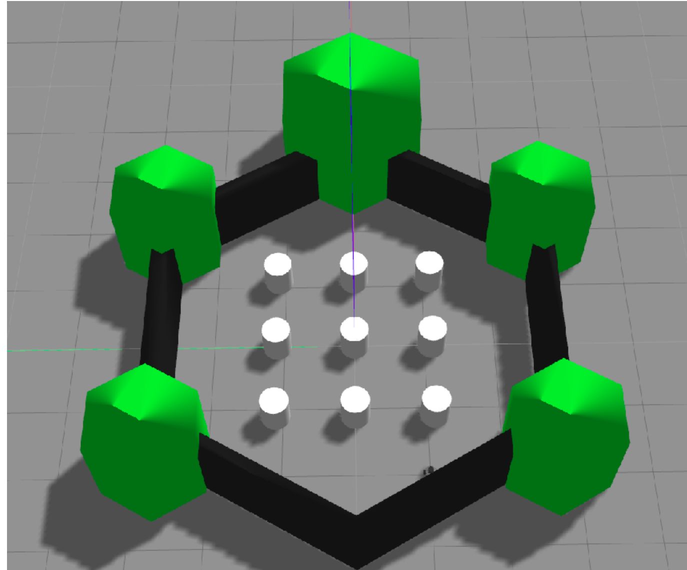


Figure 6.7: Critical room viewed from the top.

have, the less area each one will have to cover, which of course means that each area can be covered more quickly, consequently, improving the efficiency of the system.

The robots will move around the area inside the black barriers and they the white cylinders represent obstacles that must be avoided.

The goal of the team of robots is to go around the critical room and perform a quick sweep of the room, meaning that depending on the size of the team, each robot will have to move more (if the team is composed by few members) or less (if the team is composed by many members).

We took the environment that was presented previously and divided it into four different locations: L1, L2, L3 and L4. With these four locations, we designed a topological map and obtained the one presented on Figure 6.8. With simplicity in mind, we created a system where the robots can only move into the next location or stay in their current one.

Besides the ones that were already mentioned on the previous Chapters, we used extra ROS packages such as move_base [2] (to plan movement), map_server [3] (which offers map data as a ROS service), amcl [4] (a localization package based on the adaptive Monte Carlo algorithm), tf2_ros [5] (which keeps track of multiple coordinate frames over time) and gazebo_ros [6] (which allows the simulation). Furthermore, we also used several configuration files specific to the turtlebot3 robot from packages such as turtlebot3_navigation [7] turtlebot3_gazebo [8] and turtlebot3_description [9].

[2] http://wiki.ros.org/move_base
[3] http://wiki.ros.org/map_server
[4] http://wiki.ros.org/amcl
[5] https://wiki.ros.org/tf2
[6] http://gazebosim.org/tutorials?tut=ros_overview
[7] http://wiki.ros.org/turtlebot3_navigation
[8] http://wiki.ros.org/turtlebot3_gazebo
[9] http://wiki.ros.org/turtlebot3_description

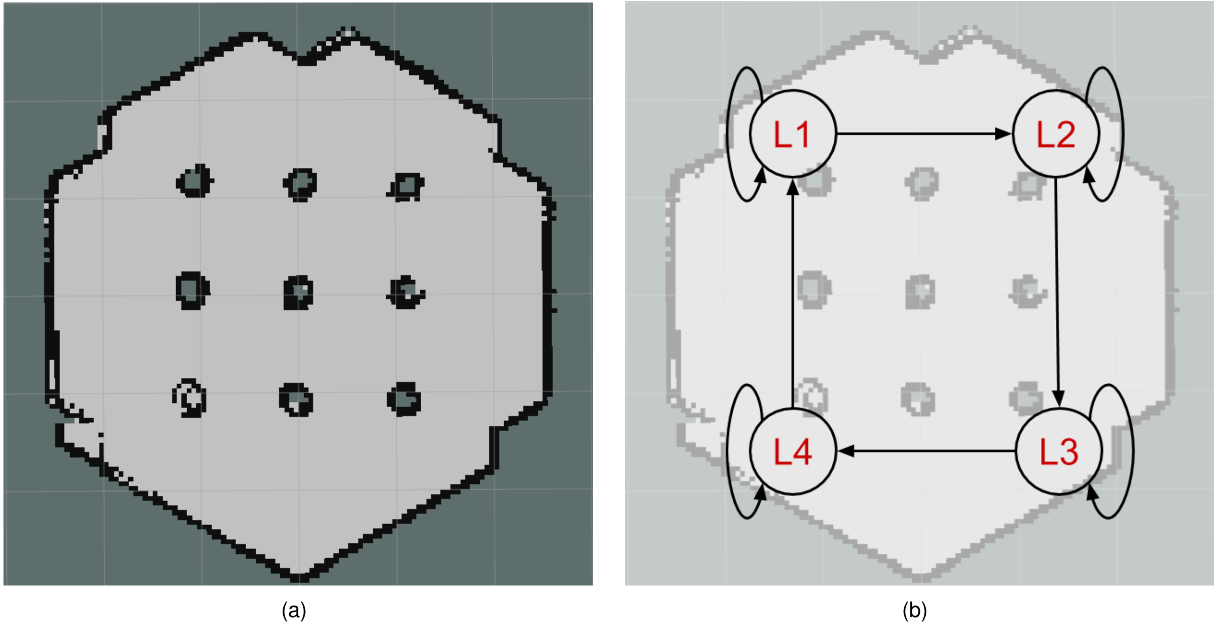


Figure 6.8: The metric map of our environment (a) and our topological map (b).

6.2.2 Creating a GSPN for the plan

The obtained GSPN is presented on Figure 6.9. The Figure is a screenshot taken from the ROS online visualization module.

The designed GSPN is very symmetrical and has four kinds of places: places where the robots are taking measurements from a specific location (p1:Check_Temperature_L1, p5:Check_Temperature_L2, p9:Check_Temperature_L3, p13:Check_Temperature_L4), places where the robots are deciding what they should do next (p3:Decide_L1, p7:Decide_L2, p11:Decide_L3, p15:Decide_L4), places where the alarm of high temperatures is activated (p2:Alarm_L1, p6:Alarm_L2, p10:Alarm_L3, p14:Alarm_L4) and places where they are waiting (p4:Wait_L1, p8:Wait_L2, p12:Wait_L3, p16:Wait_L4). On the other hand, we use both immediate and exponential transitions. The immediate transitions (t4:Go_Wait_L1, t6:Go_Check_L2, t10:Go_Wait_L2, t12:Go_Check_L3, t16:Go_Wait_L3, t18:Go_Check_L4, t22:Go_Wait_L4, t24:Go_Check_L1) model action selection while the exponential transitions (t1:Temperature_High_L1, t2:AlarmOff_L1, t3:Temperature_Low_L1, t5:Wait_Done_L1, t7:Temperature_High_L2, t8:AlarmOff_L2, t9:Temperature_Low_L2, t11:Wait_Done_L2, t13:Temperature_High_L3, t14:AlarmOff_L3, t15:Temperature_Low_L3, t17:Wait_Done_L3, t19:Temperature_High_L4, t20:AlarmOff_L4, t21:Temperature_Low_L4, t23:Wait_Done_L4) model the outputs of the various actions that are executed all throughout the execution.

An important fact that we should point out is that the robots are not actually taking measurements from a location, they are only moving along the said location. We chose to create this wrap-around the execution in order to make it more interesting.

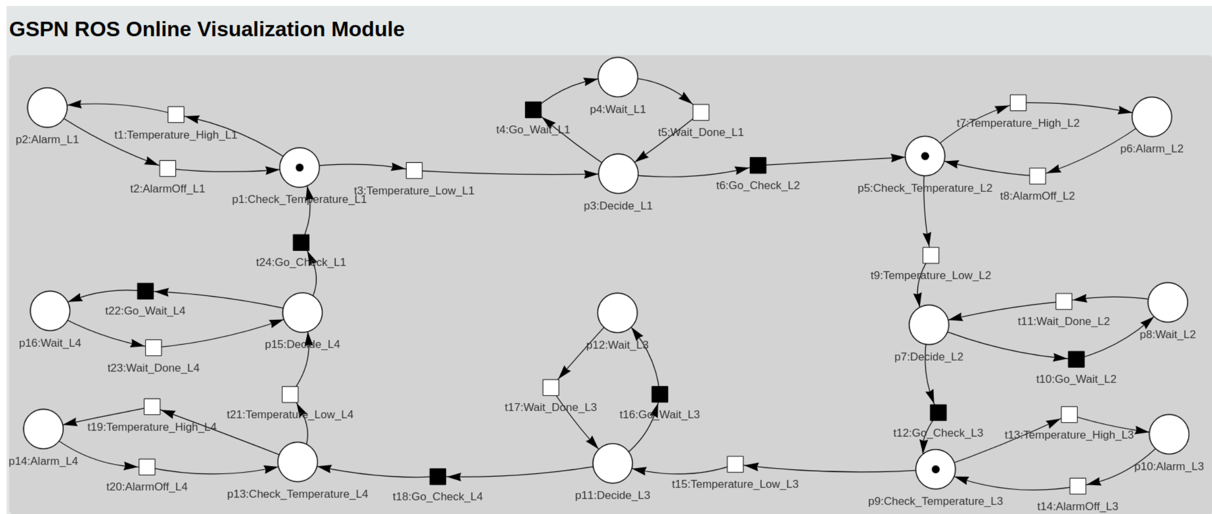


Figure 6.9: Temperature patrol designed GSPN.

6.2.3 Results

We ran our setup with one, two, three and four robots. For the first three tests we managed to run an execution successfully, however, the test with four robots led to errors and was not successful. For each test, we filmed a short video and made it available through the following link:

`https://drive.google.com/drive/folders/1DuohmLUU7pl7-_JWS_TGHT3f48blpQ3T?usp=sharing`

The names of the videos are numerated from 1 to 4, depending on the number or robots used.

When testing the system with one robot, the execution was both smooth and relatively fast.

With an increasing complexity, our system started having some performance issues. When we tried running the test for the first time, we realized that instead of moving, the second robot would initiate its recovery behaviour, which is an embedded mechanism that attempts to clear the space around the robot. Although disappointing, we understood that we could bypass this issue by manually clearing the robot's costmaps. We had to do this in the beginning of the execution but after doing it a couple of times, the system was able to continue autonomously.

When running the test with three robots, we faced similar issues to the previous test, where only one of the robots managed to start moving autonomously. The remaining two got stuck in the beginning of the execution and we had to manually clear their costmaps. The movement of the simulated robots was generally not very smooth.

An important note to take is that on some cases, our framework would not apply the input policy when confronted with an immediate transition and the simulated robots would keep waiting for the corresponding action server indefinitely, which of course is an unwanted behaviour.

Finally, we achieved the bottleneck of our tool on the test with four robots. On most cases, the robots would stop moving and not even the costmap clearing service would be able to recover them. Besides this, being a cyclic environment, on the case where one of the robots stopped moving, the others would stay waiting behind.

6.3 Discussion

6.3.1 Testing the framework with no robots

In order to simplify the identification of each test, we will refer to the test of Sub Section 6.1.1 as test 1, the test of Sub Section 6.1.2 as test 2 and the test of Sub Section 6.1.3 as test 3.

The first result we would like to talk about is the increase of both computational power and memory usage when we increased the number of tokens and the number of places. This is due to the fact that the more elements we introduce into our plan, the more resources will need to be provided by the host computer. The values for the virtual memory increased in a much more linear way, when compared to the increases of the processing power, because adding a new place or adding a new token (for example) usually takes almost the same amount of memory every time (in the case of our framework, we are using sparse matrices, and as such, this means that a new connection is created when we add a new place). When talking about the percentage of CPU, the values tend to reflect a bigger variation because as we add more tokens, more actions are being executed at the same time.

The most interesting and surprising result was the wait time necessary to start the execution that we mentioned for test 1 and 2. This increase is mainly due to the time that the processor requires to assign each task. Increasing the initial number of tokens means an increase on the number of action clients to match with action servers. Increasing the initial number of places, adds more action servers that can be paired up with action clients. This does not occur with test 3 because we only had one token to execute the plan, which means that the processor is only assigning one task at a time, even though it has increasingly more action servers active.

Still regarding action servers and their overall impact on the system, we would like to discuss the result obtained in Figure 6.5, when we directly compared the percentage of CPU of test 1 and 2 with the number of action servers available in each one. In the case of test 1, by adding a new token, we were essentially adding seven new action servers (one for each place) while in test 2, when we added a new place, we were adding five new action servers (one for each token). The results obtained show that the action servers directly impact the computational overhead because test 1 starts with a lower percentage of used CPU than test 2, however, as we add more elements into each one, since test 1 has a higher order of growth (seven), its computational overhead quickly surpasses the second one's (which has an order of growth of five).

Regarding test 3, we managed to run the execution with a high number of places and taking this into account, we can conclude that in a truly decentralized system, our framework would work well.

6.3.2 Testing the framework with robots on Gazebo

When we ran the test with one and two simulated robots, the results were adequate, not considering the issue that was mentioned with the costmaps. Nevertheless, when running the test with three and four robots, we started to register some flaws with the developed framework. One of the flaws is of course the sporadic case where the policy is not applied and the involved simulated robot's movement is

brought to a halt. Although we tried understanding the reason behind this issue, we were not successful due to the randomness of the event. However, we concluded that the system worked best when the places and the transitions represented on the visualization module were not moved around, hence the strangely organized left side of the video for the simulation with three robots.

On the other hand, the simulation with four robots showed that our framework is only feasible to use with up to three robots, due to the high computational overhead (which means that for a setup with more processing capacity, the results might differ). On the presented video, the execution takes very long to start itself (it only starts around timestamp 2:42) and some robots eventually get stuck in the environment and can never recover from that (not even with the help of the already mentioned service). When this happens, move base automatically discards the current objective and considers it done. Although this is unwanted, an even worse issue arises at times: for instance, on timestamp 6:17, we see robot 3 surpassing robot 4, which is not something that should happen due to the nature of our GSPN. This happens because robot 3 spent too much time on L1 without moving and at some point move base discarded that goal and with that, both robot 3 and robot 4 got entangled into the waiting area and eventually robot 3 was sent into L2 (while robot 4 stayed in the waiting area), moving in a horizontal manner and completely surpassing robot 4. In the end of the video, every robot was stuck and not moving, which of course led to the premature end of the execution of the plan.

Chapter 7

Conclusions

7.1 Achievements

The developed tool is available on:

`https://github.com/PedroACaldeira/gspn_framework_package.git`

We were able to successfully lay down the groundwork for a multi-robot GSPN software framework to execute and visualize plans both in ROS and in systems without robots. Besides this, we were also able to develop a graphical visualization tool for the execution and simulation of GSPNs.

Although the results were obtained from a simulated environment, taking into account that a vast majority of the framework's code was integrated with the ROS middleware and that when we tested a truly decentralized system, everything ran fairly well, we believe that, with more future work, this framework will work well in a multi-robot system with real robots.

Even though the visualization module is operable and accomplishes its mission, it does not have the most user-friendly interface, which was one of the objectives that was layed down.

7.2 Future Work

The following list includes some elements that might be interesting to pursue in the future. Some changes are related to the most basic foundations of our framework, while others are less disruptive, however, all of them can improve the framework's overall quality and usability.

7.2.1 Execution Module

On the standalone execution side of the framework, a new approach should be explored in order to avoid concurrent futures and use a different tool that allows the algorithm to create threads during the execution. This way, the efficiency will improve since we would not be creating three times the necessary concurrent futures (as mentioned in Section 4.1.1).

Regarding the ROS integrated execution, after having the necessary updates to the involved packages, one should try to get our framework working in real robots, instead of simulated ones in order to

truly understand the framework's limitations and improve them. It could also be interesting to deploy this framework with a team of different robots from the turtlebots. Besides this, one can focus their attentions in correcting the issue of the execution with four robots. An interesting first step would be to create a setup with a bigger environment that would allow more time to clear the costmaps and to avoid having robots collide into each other.

Finally, regarding both sides of the execution module, a new way of checking and applying the policy should be implemented. The current one works relatively well for small systems (the ones that were tested worked well, for instance), however, the more complex a system is, the longer it takes to check and apply a policy, which can be a caveat for systems where a user wants an immediate response. A possible solution would be to create a "shortcut" to the policy, meaning that each place would have an inner structure associated with it that included the parts of the policy that involved transitions linked to it. This way, the algorithm would first check this list and on the case where the current marking does not match any marking available in it, the full policy would be checked.

Besides this, presently, our framework is not integrated with any tool that allows policies to be automatically generated, which of course can be very cumbersome and inefficient.

7.2.2 Visualization Module

On the visualization module, it would be interesting to try a new approach that avoided using Vis.js (see Section 5.2). Since this tool was created to generate graphs, it does not allow the direct creation of tokens inside each place. However, it allows the user to put images inside each node and as such, the solution we came up with was to place an image with the number of tokens inside each place, which of course means that our framework is limited to the number of images created to represent the marking. Although this solution works well with small systems, once the marking of a specific place reaches a value larger than the maximum number of available images, the system will not be able to represent the marking. It will not crash, but instead of presenting an image with the marking, it will simply present a black image inside the place in question, which is of course unwanted behaviour.

Finally, on the offline part of the module, the analysis panel of the right side of the visualization module interface is not complete because we did not implement the necessary algorithms in order to make it functional. This could be interesting to complete because it can improve the user's workflow. Besides this, the final interface is not very adequate and as such, it should be redesigned and tested along with users in order to build it in a user-centered way.

Bibliography

[1] K. Jose and D. K. Pratihar. Task allocation and collision-free path planning of centralized multi-robots system for industrial plant inspection using heuristic methods. *Robotics and Autonomous Systems*, 80:34–42, 2016.

[2] R. Mendonça, M. M. Marques, F. Marques, A. Lourenco, E. Pinto, P. Santana, F. Coito, V. Lobo, and J. Barata. A cooperative multi-robot team for the surveillance of shipwreck survivors at sea. In *OCEANS 2016 MTS/IEEE Monterey*, pages 1–6. IEEE, 2016.

[3] L. Carlone, M. K. Ng, J. Du, B. Bona, and M. Indri. Simultaneous localization and mapping using rao-blackwellized particle filters in multi robot systems. *Journal of Intelligent & Robotic Systems*, 63 (2):283–307, 2011.

[4] M. Luckcuck, M. Farrell, L. A. Dennis, C. Dixon, and M. Fisher. Formal specification and verification of autonomous robotic systems: A survey. *ACM Computing Surveys (CSUR)*, 52(5):100, 2019.

[5] C. Azevedo and P. U. Lima. A gspn software framework to model and analyze robot tasks. In *2019 IEEE International Conference on Autonomous Robot Systems and Competitions (ICARSC)*, pages 1–6. IEEE, 2019.

[6] H. F. C. de Castro. *Robotic tasks modelling and analysis based on discrete event systems*. PhD thesis, Dissertation to obtain the Doctorate Degree in Electrical and Computer . . . , 2010.

[7] G. Balbo. Introduction to generalized stochastic petri nets. In *International School on Formal Methods for the Design of Computer, Communication and Software Systems*, pages 83–131. Springer, 2007.

[8] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, 1989.

[9] A. Marino, L. Parker, G. Antonelli, and F. Caccavale. Behavioral control for multi-robot perimeter patrol: A finite state automata approach. In *2009 IEEE International Conference on Robotics and Automation*, pages 831–836. IEEE, 2009.

[10] G. Antonelli, F. Arrichiello, and S. Chiaverini. The null-space-based behavioral control for autonomous robotic systems. *Intelligent Service Robotics*, 1(1):27–39, 2008.

[11] M. M. Quottrup, T. Bak, and R. Zamanabadi. Multi-robot planning: A timed automata approach. In *IEEE International Conference on Robotics and Automation, 2004. Proceedings. ICRA'04. 2004*, volume 5, pages 4417–4422. IEEE, 2004.

[12] K. G. Larsen, P. Pettersson, and W. Yi. Model-checking for real-time systems. In *International Symposium on Fundamentals of Computation Theory*, pages 62–88. Springer, 1995.

[13] K. G. Larsen, P. Pettersson, and W. Yi. Uppaal in a nutshell. *International Journal on Software Tools for Technology Transfer (STTT)*, 1(1):134–152, 1997.

[14] M. Georgeff, B. Pell, M. Pollack, M. Tambe, and M. Wooldridge. The belief-desire-intention model of agency. In *International workshop on agent theories, architectures, and languages*, pages 1–10. Springer, 1998.

[15] G. A. Kaminka and I. Frenkel. Flexible teamwork in behavior-based robots. In *Proceedings Of The National Conference On Artificial Intelligence*, volume 20, page 108. Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999, 2005.

[16] M. Tambe. Towards flexible teamwork. *Journal of artificial intelligence research*, 7:83–124, 1997.

[17] P. R. Cohen and H. J. Levesque. Teamwork. *Nous*, 25(4):487–512, 1991.

[18] A. Rao and M. Georgeff. Modelling rational agents within a bdi architecture, prin. of knowl. *Rep. & Reas., San Mateo, CA*, 1991.

[19] H. Costelha and P. Lima. Robot task plan representation by petri nets: modelling, identification, analysis and execution. *Autonomous Robots*, 33(4):337–360, 2012.

[20] V. A. Ziparo and L. Iocchi. Petri net plans. In *Proceedings of Fourth International Workshop on Modelling of Objects, Components, and Agents (MOCA)*, pages 267–290, 2006.

[21] B. Lacerda and P. U. Lima. Petri net based multi-robot task coordination from temporal logic specifications. *Robotics and Autonomous Systems*, 122:103289, 2019.

[22] C. Mahulea and M. Kloetzer. Robot planning based on boolean specifications using petri net models. *IEEE Transactions on Automatic Control*, 63(7):2218–2225, 2017.

[23] M. Kwiatkowska, G. Norman, and D. Parker. Prism: Probabilistic symbolic model checker. In *International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, pages 200–204. Springer, 2002.

[24] R. Alur and T. A. Henzinger. Reactive modules. *Formal methods in system design*, 15(1):7–48, 1999.

[25] C. Dehnert, S. Junges, J.-P. Katoen, and M. Volk. A storm is coming: A modern probabilistic model checker. In *International Conference on Computer Aided Verification*, pages 592–600. Springer, 2017.

[26] J. Bohren and S. Cousins. The smach high-level executive [ros news]. *IEEE Robotics & Automation Magazine*, 17(4):18–20, 2010.

[27] E. G. Amparore, G. Balbo, M. Beccuti, S. Donatelli, and G. Franceschinis. 30 years of greatspn. In *Principles of Performance and Reliability Modeling and Evaluation*, pages 227–254. Springer, 2016.