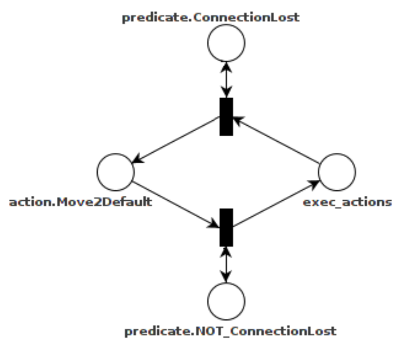


PN-RTE
Petri Net Robot Task Execution

Pedro Miguel Pina dos Santos

Thesis to obtain the Master of Science Degree in

Electrical and Computer Engineering

Supervisors:	Doctor Pedro Manuel Urbano de Almeida Lima
	Doctor Hugo Filipe Costelha de Castro

Examination Committee

Chairperson:	Doctor João Fernando Cardoso Silva Sequeira
Supervisor:	Doctor Hugo Filipe Costelha de Castro
Member of the Committee:	Doctor Carlos Baptista Cardeira

May 2016

Acknowledgments

Em primeiro lugar quero agradecer ao Prof. Pedro Lima e ao Prof. Hugo Costelha pela orientação, apoio e disponibilidade ao longo deste trabalho. Gostaria ainda de reconhecer a importância das oportunidades dadas pelo Prof. Pedro Lima, que me permitiu fazer parte da equipa do SocRob, desde os tempos do futebol robótico. Foram vários anos a aprender e a crescer devido a todo o ambiente que caracteriza a equipa e as competições.

Aproveito para agradecer a todos os membros que fazem/fizeram parte da equipa do SocRob. Foram diversas noites mal dormidas, muitas linhas de código escritas, muita cola quente e inúmeros desafios para resolver...mais importante que isso tudo, um grande sentimento de apoio e camaradagem, Obrigado a todos.

Quero deixar um agradecimento especial ao Pedro Resende e ao Diogo Pires, pois para além do SocRob e dos trabalhos para as diferentes disciplinas, são amigos a quem eu recorri sempre que precisei, Obrigado, tem sido um prazer conhecer-vos.

Um grande abraço ao João Reis, Obrigado pelo apoio e por teres sido incansável em todas as explicações sobre programação e afins.

Aos meus amigos, tanto os da Guarda como os conheci ao longo destes últimos anos, um Obrigado por todas as vivencias que partilhamos.

Aos meus pais, Obrigado por todo o apoio, esforço e dedicação ao longo destes anos. Ao meu irmão e à minha irmã, Obrigado por terem sido compreensivos e por serem tão importantes na minha vida.

Aos meus avós, tios e primos, vocês sabem o quão importantes São para mim, Obrigado por terem sido sempre capazes de me motivar, divertir e apoiar.

A ti Raquel, Obrigado pela paciência, por estares sempre presente, por me fazeres acreditar nas minhas capacidades, por tudo!

Resumo

Esta tese introduz um sistema de desenvolvimento que permite executar tarefas robóticas definidas através de um formalismo baseado em redes de Petri. A solução proposta é composta por três componentes: PetriNetExecution, PrimitiveActionManager e PredicateManager, que endereçam os problemas da tomada de decisão, actuação e percepção.

Ao implementar e integrar a solução através do Robot Operating System, ROS, e retirando-se partido dos seus conceitos e funcionalidades, alcançou-se uma solução flexível e modular, capaz de executar em tempo real diferentes tarefas, incluindo casos em que as mesmas são definidas hierarquicamente.

Para além disso, o sistema de desenvolvimento proposto fornece uma clara e simples definição dos métodos, verificações, bem como informação de todos os componentes, permitindo uma simples e rápida depuração e análise da tarefa enquanto a mesma está a ser executada.

Por fim, a ideia que suportou o desenvolvimento e a implementação da solução permite que cada biblioteca desenvolvida seja substituída, expandida ou utilizada em diferentes âmbitos.

Palavras-chave: Agentes Autónomos, Execução de tarefas, Redes de Petri, ROS, Sistemas Robóticos, PN-RTE

Abstract

This thesis introduces a software framework which allows the execution of a robot task defined by a formalism based on Marked Petri nets. The proposed solution is composed by three components: a Petri net execution, a primitive action manager and a predicate manager, which address the problems of decision-making, actuation and perception.

By implementing and integrating the framework via the Robot Operating System, and withdrawing advantage of its concepts and functionalities, a modular and flexible solution was achieved, capable of realtime execution of different tasks, including hierarchically defined cases.

Additionally, the framework provides simple and clear methods definitions, verifications and indepth informations which allows for simple and fast debugs of the task during execution.

Finally, the idea that supported the solution architecture implementation and development allows each package to be replaced, extended or used in different scopes.

Keywords: Autonomous Agents, Petri nets, Robotic Systems, ROS, Task Execution, PN-RTE

Contents

List of Figures

Chapter 1

Introduction

Robots, or artificial intelligent autonomous agents as they are often called, are being developed to accomplish complex sets of different assignments in dynamic, partially observable and unpredictable environments, from the living room of a domestic house, a full crowded museum atrium, a hospital corridor or even extreme conditions environments as the outer space, underwater regions or the Mars planet [2, 3, 4, 5].

It is clear that to be able to perform complex behaviours properly, a robot must be able to perceive and interact with the environment. Most robots employ high-level decision layers which deliberate or evaluate the sensed environment and, through a robot task plan, decide the action or set of actions to execute in order to achieve the expected goal. Therefore, a robot task plan is a description of what actions or/and how a robot needs to perform in order to complete the desired task.

However, the design of a robot task plan that could perform a given task in a correct, intelligent and feasible way is a common problem which has been addressed in the literature and can be separated in three main approaches:

1. Manually written, directly programmed in the robot and tailored to the tasks, without using any formal or explicit representation;

2. Manually written based on task representation formalisms;

3. Automatically generated based on the description of the goals and capabilities of the system.

The first approach is highly limited to the expertise and free will of the programmer responsible for the implementation of such tasks, leading to task plans with few actions or too complex or confusing implementations, which make the tasks very difficult to debug, modify and reuse in different scopes.

The second approach is limited by the expressiveness of the representation formalism and capabilities of the designer, although most formalisms offer systematic and consistent modeling methods, which not only allow to verify and/or ensure a task is efficient and feasible, but also leads to richer task models, pruning design errors. Additionally, the formalisms usually include analysis and evaluation methods that allow to have an insight of the task performance.

The last approach consists in automatic task planning and, although it is more advantageous, the necessity to express all features of interest, allied with the complexity of real applications, introduces a limitation to the application of such approach. Additionally, the tasks obtained from this approach can be expressed by the representation formalisms described in the second approach.

This thesis concerns the problem of the execution of a robot task representation formalism framework based on the second approach, being the main focus the ability to easily create, design and execute a robot task and the respective actions based on the world perception, which is essential in most real case applications where there are specific tasks to be performed.

As result of this thesis a software framework was developed, based on the Robot Operating System (ROS) [6] as a middleware and a collection of packages that simplify the development of robot applications. The proposed software framework is available in a public git repository [7] and was tested using both simulated and real robot scenarios.

1.1 Motivation

The development and execution of a robot task plan and the respective actions should be simple enough regarding the design and implementation, in order to allow that not only roboticists can program them, but also to be powerful enough to be used in real and complex case applications.

Despite our daily life being invaded by robots, creating robust and general purpose robots is still a very challenging procedure. With the fast grow of the robot market across the world, and as more generic purposed robots start to appear in both commercial and research markets, e.g., new robots as Pepper [8] and Buddy [9], it is expected that the tasks they can perform also increase, in number and complexity, which makes clear the importance of having well defined robot task plans and software frameworks capable of executing them.

The Robot Operating System has become the popular robot platform for the robotics research and development, thanks to an agnostic framework with a modular and distributed nature, as well as a strong and active community that contributes with several packages that implement relevant software algorithms, namely task plan executors.

Taking in consideration all characteristics identified, the work developed in this thesis should contribute to the ROS with a framework capable of execution a robot task plan represented by a Marked Petri net and the respective embedded actions.

1.2 State of the Art

The concept of a robot completing a task involves several fields working together, control theory, computer vision, electronics, software communication, and concepts as concurrency, synchronism, parallelism, loops and hierarchy. Clearly, robot task representations need to be able to handle and represent such concepts in an explicit and efficient way, just as the software responsible for their execution .

Traditionally, task plans are represented and implemented using discrete events system based approaches, as Finite State Machines [10], or Petri Nets [11].

Finite State Machines are a mathematical model composed by discrete states and transitions between states. In Finite State Machines, each state is a unique representation of the world, evolving from one state to the others through transitions, driven by the execution of actions, or according to inputs or events received. Application of such approach can be found in most robotic competitions teams as [12, 13], where the tasks the robot needs to perform are usually well defined.

Petri nets are a powerful mathematical and graphical modeling language widely used for design, model and analysis of discrete event systems [14]. Petri nets formalism allows to graphically model aspects such as synchronism, parallelism, concurrency and have a larger modeling power when compared to Finite State Machines. In fact, Finite State Machines are a specific subset of Petri nets. Given its formalism and capabilities, Petri nets that model and execute a robot task is not only a subject discussed in the literature, [15, 16], but also applied in several environments [17, 18].

Additionally, and from the ROS standpoint, SMACH [19], and PNP-ROS [20, 1, 21], are two packages available that allow to model and execute robot tasks using the stated approaches, and will be described in Chapter 2.

1.3 Goals and Objectives

The goal of the work developed in this thesis is to create a software framework for ROS capable of executing a robot task plan described by a Marked Petri net, following the formal model introduced in [22].

To achieve the goal, some objectives were established corresponding to contributions and/or evaluations of the framework implementation. The first objective was to identify the requirements needed to implement a software framework capable of execution the robot task representation described in [22].

From the requirements identification, three new objectives were defined which correspond to the three modules needed for a complete execution framework: a Petri net executor, a primitive action manager and a predicate manager. Each module is described in Chapter 3. Since there is a Predicate Manager package [23] available in ROS which completely suits the requirements needed for that module, the decision was to integrate it instead of implement one.

Finally, the last objective defined was to test each implemented module and the framework as a whole using simulated and real robot.

1.4 Dissertation Outline

The remainder of this dissertation is organized as follows:

- **Chapter 2 - Background and Related work**: This Chapter describes not only the main concepts and an overview of the background required to understand the work done in this thesis, but also relevant related work that was already done in the field.

- **Chapter 3 - The PN-RTE framework**: This Chapter introduces the work developed in this thesis: Petri net robot task execution (PN-RTE), a software framework that allows the execution of a Robotic Task plan modeled by Marked Petri Net in the ROS environment.
- **Chapter 4 - Experiments and Results**: This Chapter presents a test scenario, proof-of-concept examples using the proposed representation and framework and the result of the implemented features.
- **Chapter 4 - Conclusions**: The final Chapter of the thesis presents a review and summary of the work developed, and additionally lists future research and development work.

Chapter 2

Background and Related Work

The work developed along the thesis was based on the robot task model for plan representation and execution using a Petri net based framework proposed in [22] which allows a systematic approach for the correct modeling and execution of a Robotic Task plan.

In this chapter we first review the Petri net formal definitions and the framework proposed in [22], and later we go over the two ROS packages named before, SMACH and PNP-ROS, in order to identify characteristics that are relevant in the architecture and implementation of the solution.

2.1 Robot Task Plan representation by Petri nets

Petri nets, defined by Carl Adam Petri during the 1960s, are widely used as a modeling language for dynamic systems. According to [24] and [15], the well defined syntax and capability to model relevant aspects such as synchronization, parallelism, concurrency and decision making, makes Petri nets a good approach to model Robot task plans.

2.1.1 Petri Nets

A Petri net consists of weighted, directed, bipartite graphs composed by a set of places, transitions and arcs. According to [25]:

Definition 1. *A Petri net is a four-tuple P,T,Pre,Post, where:*

- $P = \{p_1, p_2, \ldots, p_n\}$*, is a finite, non-empty, set of places;*
- $T = \{t_1, t_2, \ldots, t_m\}$*, is a finite set of transitions;*
- $P \cap T = \emptyset$*;*
- $Pre : P \times T$*, is a matrix which represents the set of arcs from places to transitions, such that* $Pre(p_i, t_j)$ *is equal to the weight value of that arc, if there is an arc from* p_i *to* t_j*, or 0 otherwise;*
- $Post : T \times P$*, is a matrix which represents the set of arcs from transitions to places, such that* $Post(t_i, p_j)$ *is equal to the weight value of that arc, if there is an arc from* t_i *to* p_j*, or 0 otherwise;*

A simple Petri net is depicted in Figure 2.1, where graphically:

- **Places** are represented by **circles**,
- **Transitions** are represented by filled **rectangles**,
- Directed **Arcs** are represented by **arrows**

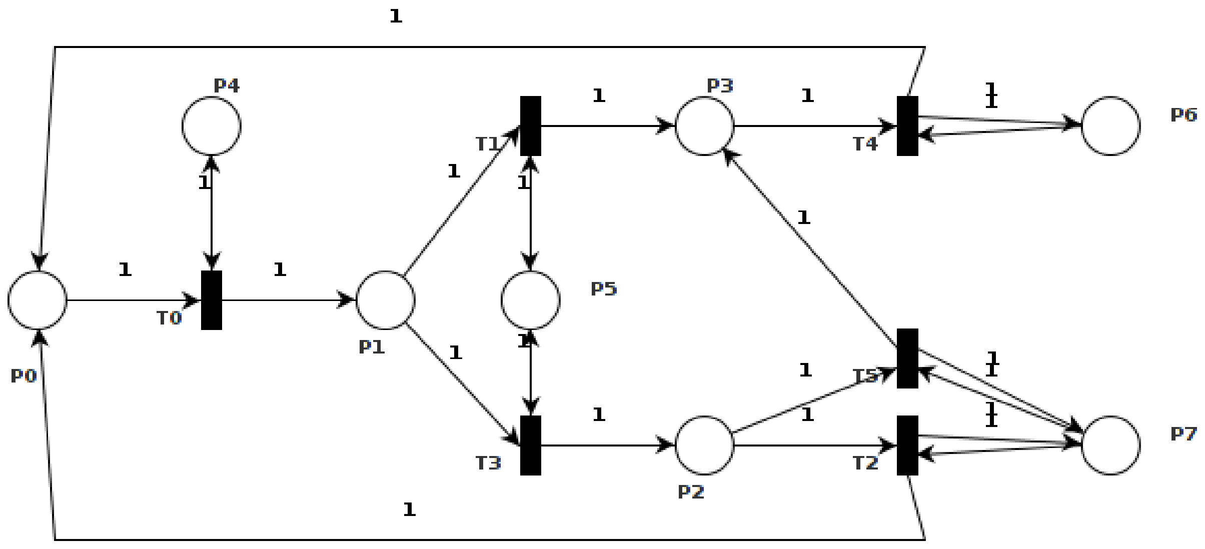


Figure 2.1: Simple Petri net.

Marked Petri net is a Petri net with marking tokens that represent the state of the net.

Definition 2. *A Marked Petri net is a couple R,M, where:*

- *R is a Petri net;*
- $M = [m_1, \ldots, m_n]$*, is the marking of the net and represents the state of the net, where* $m_n = q$ *means that are* q *tokens in a place* p_n*;*
- *The initial marking of the Petri net is usually denoted as* M_0*, and specifies the initial number of marking tokens for all places.*

Note: Although some authors use the Petri net to refer to a Marked Petri net the notation of this thesis follows the one present in [25]. Using this notation was useful for the implementation of Definitions 1 and 2, as one can see in the implementation Chapter 3.

Graphically the marking tokens are represented by the an integer number of dots inside each place, corresponding to the integer value of the marking in that place. For instance, using the Petri net shown in Figure 2.1 with the initial marking, $M_0 = [0\ 1\ 0\ 0\ 0\ 1\ 0\ 0]$, one would obtain the Marked Petri net shown in Figure 2.2 and could represent it using the tuple (P,T,Pre,Post,M), where:

- $P = \{P0, P1, P2, P3, P4, P5, P6, P7\}$
- $T = \{T0, T1, T2, T3, T4, T5\}$

- With Pre and Post matrices as one can see on table 2.1
- $M_0 = [0\ 1\ 0\ 0\ 0\ 1\ 0\ 0]$

	Pre Matrix						Post Matrix					
	T0	T1	T2	T3	T4	T5	T0	T1	T2	T3	T4	T5
P0	1	0	0	0	0	0	0	0	1	0	1	0
P1	0	1	0	1	0	0	1	0	0	0	0	0
P2	0	0	1	0	0	1	0	0	0	1	0	0
P3	0	0	0	0	1	0	0	1	0	0	0	1
P4	1	0	0	0	0	0	1	0	0	0	0	0
P5	0	1	0	1	0	0	0	1	0	1	0	0
P6	0	0	0	0	1	0	0	0	0	0	1	0
P7	0	0	1	0	0	1	0	0	1	0	0	1

Table 2.1: Pre and Post Matrix from the Petri net present on Figure 2.1.

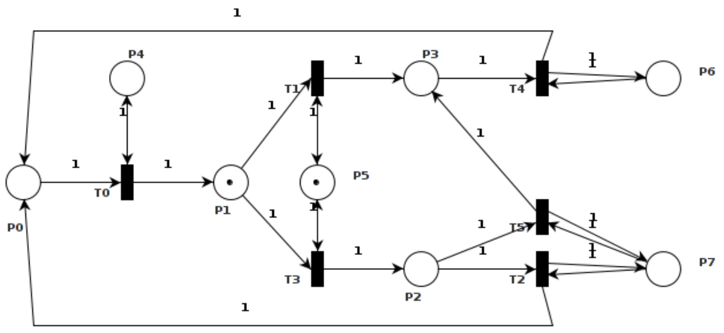


Figure 2.2: A Marked Petri net.

The evolution through different states (or markings) of the Marked Petri net is achieved by firing enabled transitions. A transition is enabled if all its input places have at least the number of tokens required by the weight of the arcs connecting them to the transition. Recalling Figure 2.2, the enabled transitions given the current marking are T1 and T3. Upon firing an enabled transition, all its input places get their number or tokens reduced by the weight of the arc between the place and the transition, while all its output places have their number of tokens increased by the weight of the arc between the fired transition and the place.

Definition 3. *For a marking M, a transition t_i is said to be enabled, if and only if:*
$\forall p \in P : m_p \geq Pre(p, t_i)$

Definition 4. *For a given marking M, any enabled transition t_i can be fired and lead to a marking M', defined by:*
$\forall p \in P, m'_p = m_p - Pre(p, t_i) + Post(p, t_i)$

It is important to notice that Marked Petri net transitions are always immediate, meaning that once they are fired a new marking is immediately reached. Additionally, even if two or more transitions are enabled for the same marking only one will fire. Going back to the example of the Figure 2.2, firing transition T1 would result in the marking $M0 = [0\ 0\ 0\ 1\ 0\ 1\ 0\ 0]$, whereas firing T3 would lead to $M0 = [0\ 0\ 1\ 0\ 0\ 1\ 0\ 0]$. The two different output are shown in Figure 2.3 and 2.4, respectively.

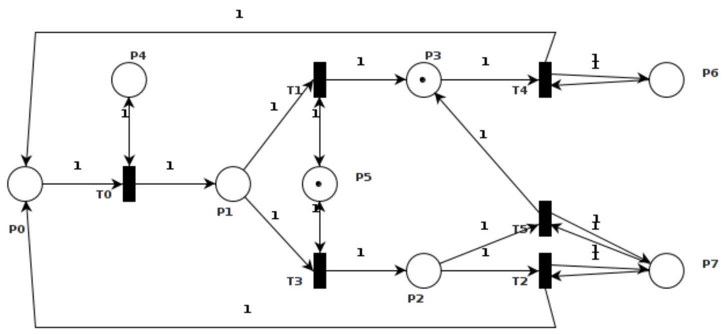


Figure 2.3: Result from firing transition T1.

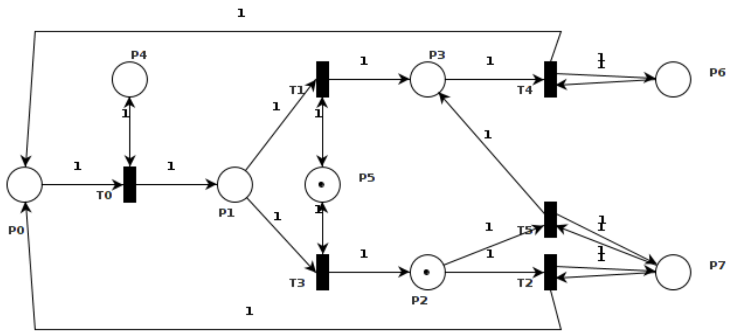


Figure 2.4: Result from firing transition T3.

Furthermore, Marked Petri nets are often simplified by additional notations, called abbreviations. In the scope of this thesis was also crucial to add the definition of a place capacity and how to solve/expand both matrices, Pre and Post.

Definition 5. *The capacity of a place defines the number of n tokens a place p can hold, meaning that if a transition t would add an $Post(p,t)$ tokens to the place p and $m_p + Post(p,t) > n$ it should not be enabled. Usually it is graphically represented using a thickest border for place p and/or using a lowercase $k = n$ near the place p.*

For example, consider the Marked Petri net depicted in Figure 2.5 where:

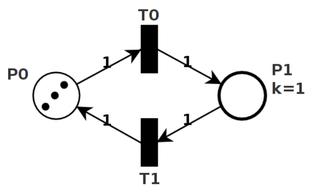


Figure 2.5: Marked Petri net where a place has a restricted capacity.

- $P = \{P0, P1\}$
- $T = \{T0, T1\}$
- With Pre and Post matrices as one can see on table 2.2
- $M_0 = [3\ 0]$

	Pre Matrix		Post Matrix	
	T0	T1	T0	T1
P0	1	0	0	1
P1	0	1	1	0

Table 2.2: Pre and Post Matrix from the Petri net present on Figure 2.5.

It is important to notice that the described Marked Petri net does not take in account the capacity of the place $P1$, which makes Definitions 3 and 4 invalid. However, it is possible to expand both matrices, Pre and Post, with a complement place p' for each place p that has a restricted capacity and correctly assign the initial marking and arcs connections for p', in order to obtain again a Marked Petri net where any place does not have a bounded capacity and where definitions 3 and 4 can be correctly applied again.

Definition 6. *For each place p which is bounded by a capacity c, a new place p' is added to the set of Places P. For each transition t that is not connected to p by a loop ($Pre(p,t) = Post(p,t) \neq 0$) an additional arc is added:*

- $ifPre(p,t) \neq 0$ *: new arc from p' to t where $Post(p',t) = Pre(p,t)$*
- $ifPost(p,t) \neq 0$ *: new arc from t to p' where $Pre(p',t) = Post(p,t)$*

The initial marking of p' is given by $m_{p'} = c - m_p$

Using again the Marked Petri net from Figure 2.5 and applying the Definition 6, a new Marked Petri net is obtained, as depicted in Figure 2.6, where:

- $P = \{P0, P1, P1'\}$
- $T = \{T0, T1\}$
- With Pre and Post matrices as one can see on table 2.3
- $M_0 = [3\ 0\ 1]$

	Pre Matrix		Post Matrix	
	T0	T1	T0	T1
P0	1	0	0	1
P1	0	1	1	0
P1'	1	0	0	1

Table 2.3: Pre and Post Matrix from the Petri net present on Figure 2.6.

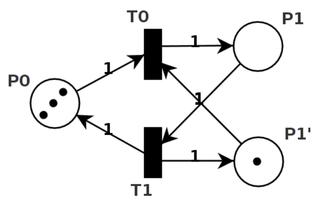


Figure 2.6: The Marked Petri net obtained after applying the Definition 6 to the Marked Petri net from Figure 2.5.

It is also useful to review how Marked Petri nets can represent different execution polices, such as:

• **Sequential**, where a transition t_{j+1} only fires after transition t_j

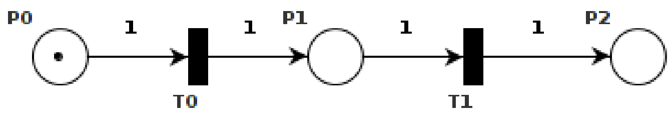


Figure 2.7: Sequential, $T1$ only fires after $T0$.

• **Fork**, transitions t_j has multiple output places arcs, meaning that after t_j fires, all the output places of t_j will increase their numbers of tokens,

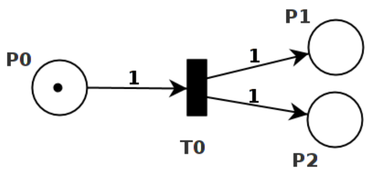


Figure 2.8: Fork, if transition $T0$ fires both places $P1$ and $P2$ will have one token.

• **Concurrency**, given that tokens may be placed on different places, situation that happens after a fork,

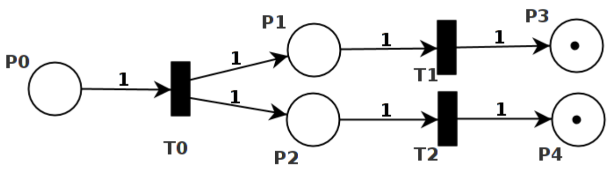


Figure 2.9: $P3$ and $P4$ are concurrent places.

• **Synchronism and Union**, given that a transition t_j may have several incoming arcs and is only enabled when each correspondent place have at least the number of tokens required by the arc weight,

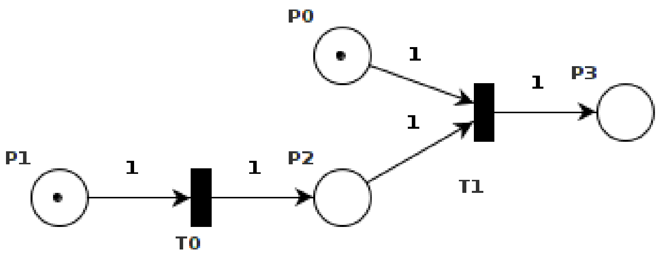


Figure 2.10: Synchronism transition $T1$ will only be enabled when $P0$ and $P2$ have at least one token.

• **Conflict**, when multiple transitions $t_j, \ldots, t_z$ are enabled and ready to fire and the result from firing t_j disables all the other transitions,

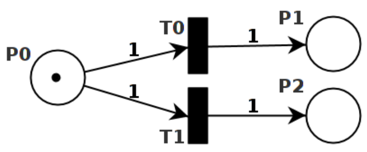


Figure 2.11: Conflict transitions $T0$ and $T1$ are both enabled, firing one will disable the other.

2.1.2 Framework

Beyond the formal definition of a Marked Petri net, and from the standpoint a robot task plan based in Petri nets [15] and in [22], an extension the Marked Petri net model is described, where place labels are used to distinguish between different types of places, such as: regular, predicate, action and task. These places do not introduce changes on the Marked Petri nets definitions, but increase the analysis and the design power of a robot task plan.

The following definitions, from [22], describe the properties of each kind of place:

Definition 7. *A regular place is a normal Petri net place, without any other special property. Could be useful as a counter or a memory place, for example.*

Definition 8. *A predicate place represents a logic predicate value, having always one or zero tokens, respectively to the predicate being true or false. A Predicate place label has prefix "predicate." or "p."*

- *If an arc from a predicate place p_n to a transition t_m exists, then there is an arc from the t_m back to p_n.*

A negated predicate has, in addition to the "predicate." (or "p."), the prefix "NOT_" before the name, e.g. "predicate.NOT_IsInCoffeeRoom";

Definition 9. *An action place represents a primitive action, an elementary block on the execution of a task by a robot. An action label has prefix "action." or "a." and, when it has at least one token, it means the action is being executed by the robot, whereas zero tokens means the action is not running.*

Definition 10. *A task place acts as a Marked Petri net macro place, which is used to create hierarchical Petri nets. A task place label has a prefix "task." or "t." and, similar to the actions places, having at least one token in a task place means that the task is running whereas zero means otherwise.*
A task place embodies a task plan, which is in fact a Marked Petri net, where transitions match to predicate based decisions which trigger the execution of actions and/or task plans.

For a better understanding on these definitions, the Marked Petri net in Figure 2.2 was extended to represent an example of a Robotic Task plan, Figure 2.12.

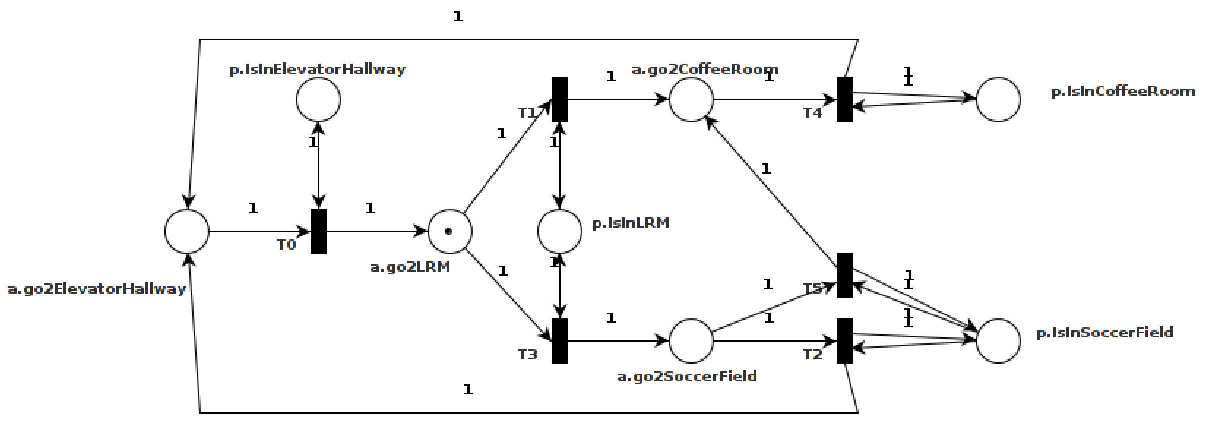


Figure 2.12: A robot task plan modeled by a Marked Petri Net.

This plan corresponds to a patrol between four different locations, composed by:

Actions:	Predicates:
Move2LRM	IsInLRM
Move2CoffeeRoom	IsInCoffeeRoom
Move2SoccerField	IsInSoccerField
Move2ElevatorHallway	IsInElevatorHallway

Is it also important to notice the crucial role of the initial marking in a task plan model since it determines which actions, and/or tasks, should run when the task is started and, subsequently, the rest of the plan. Using the patrol task of Figure 2.12 one could build another task using that task as a place, as is depicted in Figure 2.13.

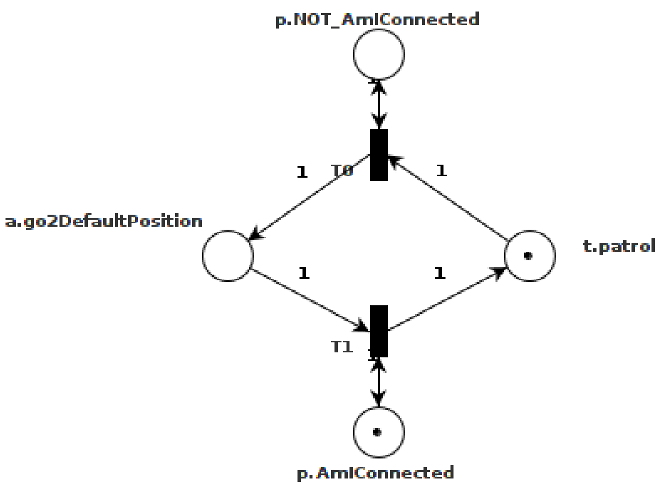


Figure 2.13: A Robotic task with a task place on it.

Additionally the framework presents action and environment Petri net layers that model the robot's response to environment and the response of the environment to the robot's actions. The layers will not be directly used for the execution of the task plan by the robot, but allow to obtain a complete task model, closing the loop for simulation purposes and allowing to perform qualitative analysis, to determine Petri net properties as: Boundedness, Safety, Blocking, Conservation, Coverability, Reachability and Liveness, and quantitative analysis, which allows to analyse the task as a Markov Decision Problem and to take advantage of all the existing Markov Decision Theory to determine several quantitative properties.

2.2 Frameworks to execute and plan a Robot Task on ROS

ROS framework is currently the de-facto standard choice for the research and development of robotic applications, providing a vast collection of packages that implement different robot functionalities. Along this section, a review of the SMACH and PNP-ROS packages, which enable designing a robot task plan represented by Finite State Machines and Petri nets, respectively, is presented, focusing on the definitions and functionalities of each package.

2.2.1 SMACH

The SMACH package is a Python API that allows the design and execution of Finite State Machines, and is a default package from ROS full desktop installer. In its core, SMACH provides an easy to use interface that enables the design of complex robot tasks/behaviours, based on hierarchical state machines. According to [19], SMACH provides two main interfaces:

- **State**: represent states of execution;
- **Container**: collections of one or more states which implement some execution policy;

It is important to notice that SMACH implements different containers with different properties that allows to overcome limitations and ease the modeling of a task, being the following the most relevant containers:

- State machine, simplest execution policy where only one state can be executed at a given time;
- Concurrence, execution policy that allows to execute multiple states simultaneously.

At this point, it is important to notice that a SMACH state does not exclusively describe a representation of the world, but rather has a more broader definition. A SMACH state is defined by its execution and set of possible outcomes, and can either be a representation of knowledge about the world, a robot behavior, task, a simple system action or a mixture of those. Each outcome connects to another SMACH state, representing each outcome a transition between one SMACH state and the other.

In fact, SMACH ”state machines” or ”concurrencies” are also states, they have outcomes of their own that can be used as an end outcome or as a transition to other states, meaning SMACH ”state machines” or ”concurrencies” can be composed hierarchically, allowing to create complex models while maintaining a clear and simple implementation of each state.

Besides the base SMACH state class that needs to be extended by a developer, executes arbitrary Python code and has no predefined outcomes, SMACH also provides different parametrized states classes that simplify the developer task of creating low level systems, being the following some of the states provided:

- ServiceState, which is a state that acts like a proxy to a ROS Service, has three outcomes: succeeded, preempted, aborted;
- MonitorState, which is a state that subscribes a ROS topic and defines its outcome based on the received information, with three predefined outcomes: valid, invalid, preempted;
- ConditionState, which is a state that checks a condition and defines its outcome based on it, with two outcomes: true, false;
- SimpleActionState, which is a state with the same outcomes as the ServiceState but that acts like a proxy to a simple actionlib[1] action.

[1] Actionlib [26] is a ROS package that extends the notation of a ROS service, providing a client-server structure where the client can set a goal to the server, ask feedback about the execution or cancel a request.

A finite state machine representing a possible SMACH implementation of a task equivalent to the one presented in Figure 2.12 is depicted in Figure 2.14, where ovals represent states and outcomes are portrayed as arrows with the outcome's name.

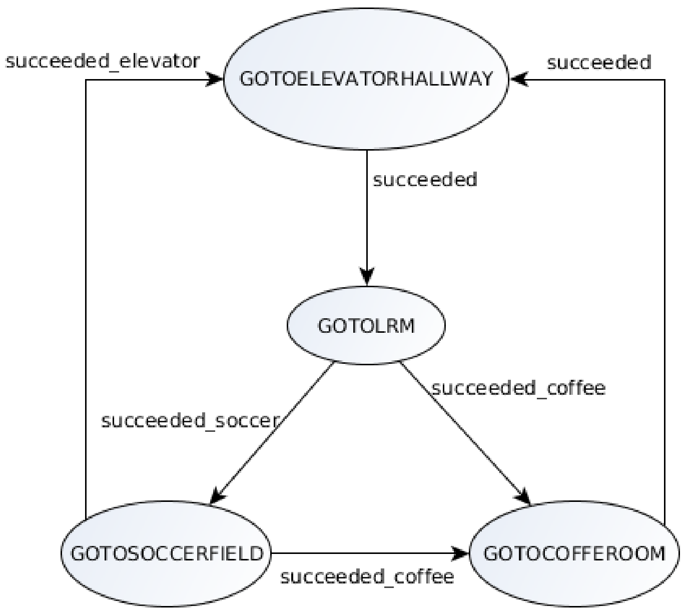


Figure 2.14: Robotic patrol task represented using SMACH states

Despite the graphical representation, it is important to notice that SMACH does not provide any method to graphically create/model a state machine, although a introspection server can be executed in order to visualize and debug an already running state machine. Besides allowing to visualize the state machine and quickly identify the running state, it also enables the visualization of the userdata defined inside each state, making this a powerful tool in terms of debugging and analysis of a running task plan.

Although the representation in Figure 2.14 is simpler when compared to the Marked Petri net in Figure 2.12, it is important to understand that they do not provide the same amount of information. While the framework proposed by [22] explicitly describes the composition of the task using the different places definitions, allowing to quickly understand in which conditions each transition can be enabled in terms of the task model, in SMACH those details are deeply immersed in the implementation of the user.

Additionally, SMACH as a framework is self-contained, meaning the implementation and execution of the task and the states is done side-by-side, not directly allowing to disconnect each component.

Lastly, although it is possible to reuse states in SMACH, each state carry its own defined outcomes, introducing a clear limitation to their reuse.

2.2.2 PNP-ROS

The PNP-ROS package implements a bridge between the Petri net Plans library and ROS, that allows Petri net models to be executed in the external library, and actions and conditions in ROS using the actionlib package. The scheme depicted in Figure 2.15. displays the bridge between PNP and ROS and was obtained from [1].

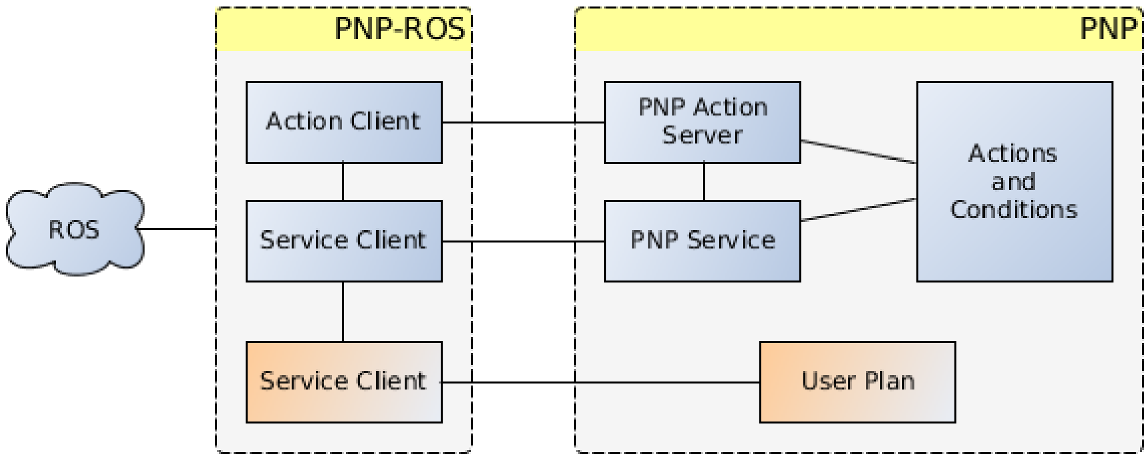


Figure 2.15: Scheme of the PNP-ROS connection, reprinted from [1]

Petri Net Plans, introduced by [16], propose a modeling language using Petri nets and formally defines a robot task plan by a set of elementary structures (no-action, ordinary action, sensing action) and the combination among those structures using control structures (operators), such as sequences, loops, interrupts and concurrent execution operators.

Before defining each elementary structures it is important to review the definition of a Petri Net Plan as a Marked Petri net and the extensions and constrains it adds. The following definitions were extracted from [16].

Definition 11. *A Petri net Plan is a Marked Petri net, see Definition 2, with the following characteristics:*

1. *Places represent execution phases of actions: initiation, execution, termination;*
2. *Transitions represent events and may be labeled with conditions that control their firing;*
3. *Transitions are grouped according to categories: action starting, action terminating, action interrupts and control;*
4. *All arcs have a weight value of one;*

Elementary PNPs are defined as:

1. **no-action**, which is a PNP with a single place and no transitions. The place is both initial and terminating place.
2. **Ordinary action**, which is a PNP defined by three places and two transitions, as depicted in Figure 2.16. Where:
   - p_i is the initial place;
   - p_o is the terminating place;
   - p_e is the execution place;
   - t_s is the transition that triggers the action start;

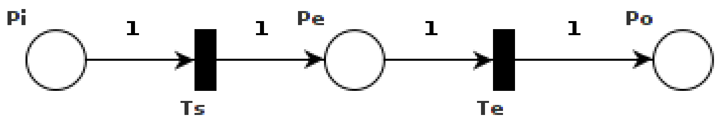


Figure 2.16: PNP ordinary action.

- t_e is the transition that triggers the action end;

3. **Sensing action**, which is a PNP defined by four places and three transitions, as depicted in Figure 2.17. Here:

   - p_i, p_e and t_s are the same as in ordinary places;
   - p_{o_t} is the terminating place if the sensed property was true;
   - p_{o_f} is the terminating place if the sensed property was false;
   - t_{e_t} is the transition that triggers the action end if the sensed property was true;
   - t_{e_f} is the transition that triggers the action end if the sensed property was false;

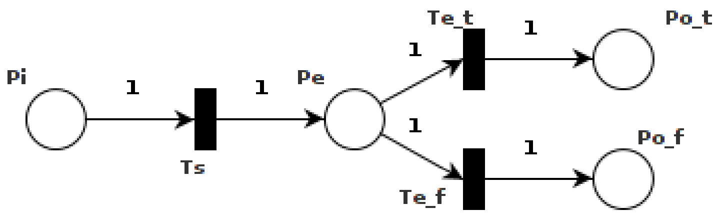


Figure 2.17: PNP sensing action.

As stated, a Petri Net Plan that represents and can execute a robot behavior or plan is achieved by using the three elementary PNP models and combining them using operators. An equivalent task modeled by a PNP to the one depicted in Figure 2.12 is shown in Figure 2.18.

The model obtained is not only different in terms of the places and transitions labels and properties but also in the way the Petri net model is executed.

First, transitions not only have labels of four different types: start, end, interrupt or standard, but are also the elements responsible for the control and execution of the actions. Second, Petri Net Plans transitions are event based, which means Petri Net Plans uses the Definition 3 to evaluate if a transition is enabled, but only fire them given the associated event. Petri Net Plans rely on a knowledge base, composed by the terms and formulas of the environment, which is used to verify the conditions in order for the related event to occur. At last, upon firing a transition a routine is called to handle the execution of the transition type, meaning that the places in this formalism are only used as representation of the state of the execution.

The formalism is able to represent explicitly the robot task and allows to perform in-advance analysis of the tasks designed. However, and as stated, the formalism is only partially implemented on ROS up to date, it relies on the Petri Net Plan external library in order to execute the Petri net model and the

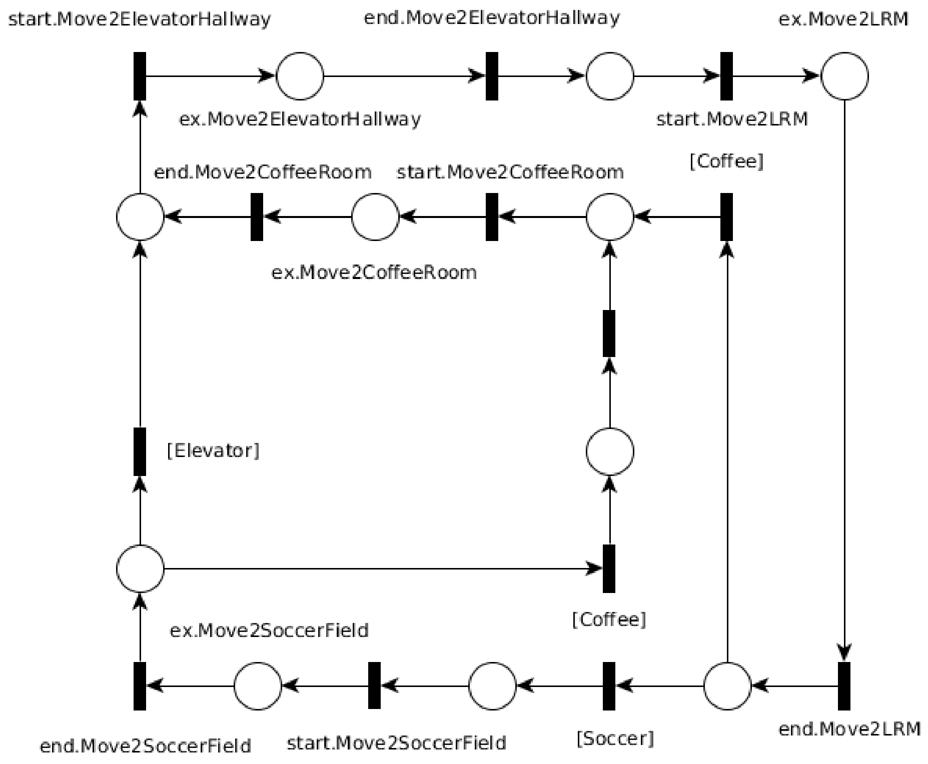


Figure 2.18: Robotic patrol task represented using PetriNetPlans elements.

knowledge base, while the ROS counterpart is required for the implementation of actions and conditions using the actionlib package.

Chapter 3

Petri net Robot Task Execution

This Chapter describes PN-RTE, Petri net Robot Task Execution, a Robot Operating System framework capable of execute a Petri net model that represents a robot task plan and the actions present on each task. It comprises four sections. In the first, a brief overview of the framework as a whole is presented, defining not only the framework architecture but also the requirements and environment, while Sections 3.2, 3.3 and 3.4, present each component of the framework individually, describing the implementation and key-features of each package.

3.1 Overview

As stated, the main objective of this thesis is to implement a framework capable of executing robot task plans and actions, based on the formalism defined in Section 2.1.2. The framework aims to achieve a development environment where a user can define and execute a robot task plan, while offering a set of functionalities during execution time.

The framework is named PN-RTE (Petri net Robot Task Execution) and it can be seen as a software framework which is divided in three different packages: **Petri Net Execution, Primitive Action Manager, Predicate Manager**. Each package concerns a different scope of the execution of a robot task plan and is connected to the others in order to achieve a complete and integrated solution.

- **Petri Net Execution**, responsible for parsing, storing and executing the Marked Petri nets,
- **Primitive Action Manager**, responsible for controlling the execution of primitive actions;
- **Predicate Manager**, responsible for storing, handling and managing the logical predicates.

Since there was already a solution available for the Predicate Manager [27, 23], available as a ROS package repository, that suited completely the requirements, it was decided to integrate that package instead of implementing a new one.

It is important to notice that besides the current implementation and integration between this packages, the idea behind the development was also to achieve a solution where each package could be reused or integrated in a different framework, or even replaced by a user defined package. For instance a user could already have or create a Primitive Action Manager and use it along with the rest of the framework, as long as the implementation assures a correct communication between the packages using ROS topics.

3.1.1 Requirements

In order to assure a framework that contemplates the already stated definitions, a set of requirements that the framework must fulfill were defined:

- **Modularity** - in order to promote a modular architecture, the framework should be separated in components, each implementing a specific functionality,
- **Flexibility** - each component must be implemented having in mind it could be replaced, redefined, or even used in a different scope,
- **Realtime execution** - to allow the use of the framework for real-case scenarios/applications,
- **Extensibility** - the components developed should not only be open to further modifications and integrations, but also have concise definitions in order to ease the process of integration and implementation of new functionalities.

3.1.2 Architecture

The packages were developed, and integrated, as C++ programming libraries for the ROS Hydro Medusa version [28], using the ROS environment and the BOOST 1.48 C++ Libraries [29, 30].

The decision of using C++ as the programming language for the development was due to the native ROS bindings and to keep the same language on all packages. In terms of the code style the decision was to follow the ROS C++ style guide that is public available in [31].

As stated, there was clear concern of having separated packages according to each package functionality, allowing to minimize the complexity of each package and to maintain each package internal details as a black box to the others.

In terms of communication between the packages, the solution was achieved using ROS topics [32]. The strategy about the topics and the respective implementation are described in Subsection 3.1.3.

A scheme of the framework showing where each package stands in terms of a robotic component is depicted in Figure 3.1.

3.1.3 Framework Communication

In the development of this framework the decision was to implement the communication between packages using ROS Topics.

ROS topics are a message transport layer abstraction that uses the Transmission Control Protocol, providing a simple and reliable communication stream [33].

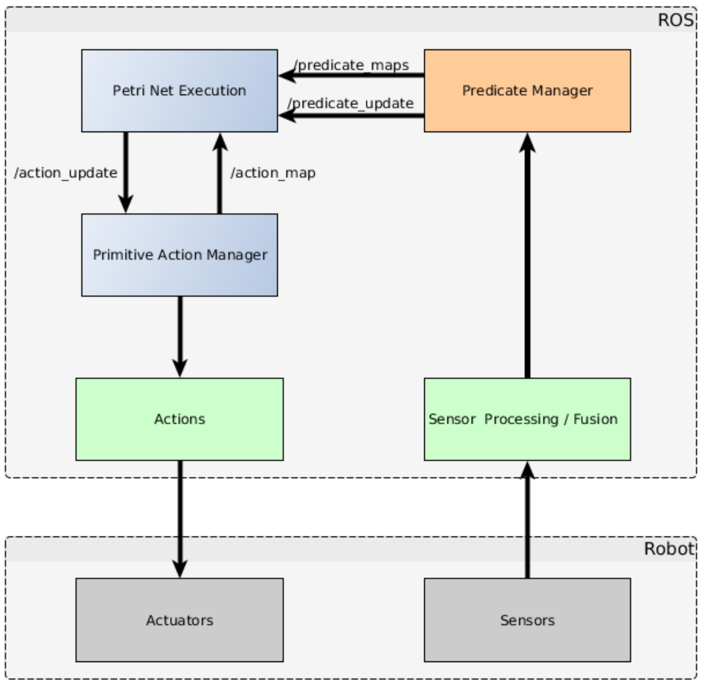


Figure 3.1: Framework Architecture

The achieved solution for the communication between packages comprises six different message types and four ROS topics and the implementation follows the messages and topics already implemented in the Predicate Manager package, which allows to minimize the size of the messages in order to promote efficiency and avoid network congestion.

The messages types are divided in two groups, the Primitive Action Manager and the Predicate Manager messages, and are explained in detail in Sections 3.3 and 3.4.

Additionally, the defined messages are used to define and exchange information using four different ROS topics:

Two topics published by the Predicate Manager package to communicate the predicates and their value to the Petri Net Execution package:

- predicate_maps, topic used to communicate the set of registered predicates, the topic message type is PredicateInfoMap. Usually, a single message is published in this topic to communicate the list of all predicates.
- predicate_updates, topic used to communicate the value of the registered predicates along the execution, the topic message type is PredicateUpdate. Each time a predicate value changes, a new message is published.

One topic published by the Primitive Action Manager to communicate the actions and one published by the Petri Net Execution package to inform the set of actions to be executed:

- action_map, topic used to communicate the set of registered actions, the topic message type is PAInfoMap. As before, usually, a single message is published in this topic to communicate the list of all actions.

    - action_update, topic used to communicate the set of actions to be executed, the topic message type is PAUpdate. Each time the set of actions to be executed change a new message is published.

Lastly, each package implements methods to handle possible failures, alerting the user whenever an irrecoverable failure occurs.

Figure 3.2 shows the ROS connection between the three packages nodes and the topics each node is subscribing and publishing, where rectangles represent topics, ovals represent packages and the direct arcs represent the publish/subscribe connection.

3.2 Petri Net Execution

The Petri Net Execution Package is the core package of the PN-RTE framework since it is the one responsible for parsing, validating and executing all the Marked Petri nets models according to the values of predicates received and responsible for broadcasting which actions should start or stop in each moment. The package follows the requirements defined for the framework and is composed by four data structures:

- **Parser**, responsible for parsing a Petri net file;
- **PetriNetStructure**, used to store a Marked Petri net immutable data;
- **PetriNetExecutor**, responsible for the execution of a Marked Petri net;
- **PetriNetManager**, responsible for managing and storing one or more Marked Petri nets, and to communicate with the PrimitiveActionManager and the PredicateManager.

The division among the four data structures was based on keeping the implementation of the package, and the respective functionalities, more clear and modular to further modifications or upgrades. While the first three data structures regard a single Marked Petri net, the last data structure regards the manager which creates, stores and calls the execution of each Marked Petri net when needed. The flowchart of Figure 3.3 displays the hierarchy between each data structure and the most important incoming and outgoing connections between the packages or data files.

In the following Subsections each data structures and the respective key features will be presented.

3.2.1 Parser

The first implemented feature and the lowest level unit of the Petri Net executor was the Parser. The Parser is a struct responsible for the parsing of the Petri net data from a file to a raw data structure. It is composed by several methods that allow a valid transcription of the data to a set of local data structures. It was coded almost entirely using the boost property tree library [34], where upon using the constructor of the struct properly and ensuring the file is correct from the syntax point of view, every member and data structure gets created and/or filled.

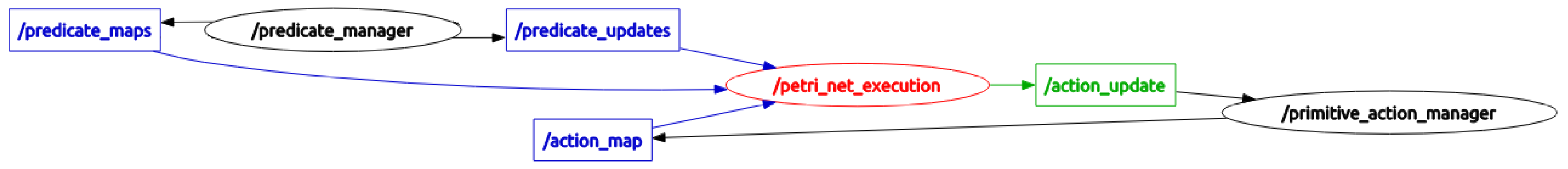


Figure 3.2: ROS Topics connections between packages

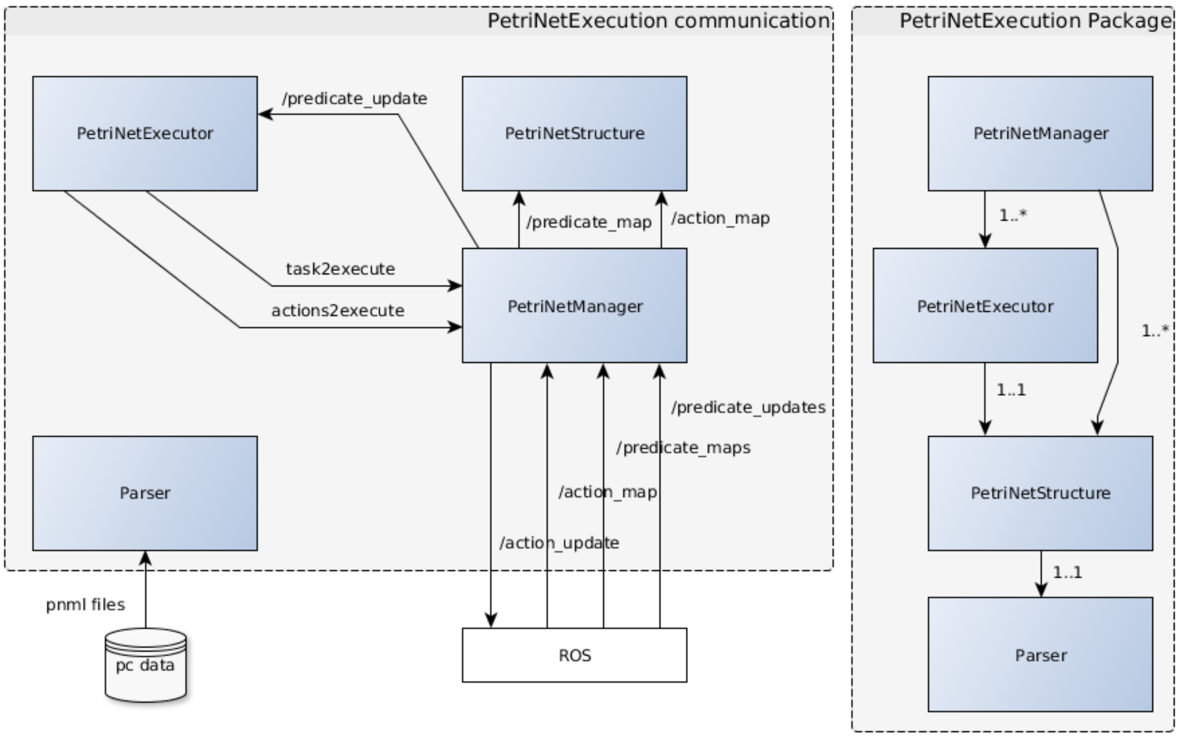


Figure 3.3: PetriNetExecution Package structure and communication

Although the Petri net markup language, pnml, is the standard interchange format for the Petri net models, several software programs that allow to design a Marked Petri net model with the features needed by the framework defined in 2.1.2 fail to use it, having instead their own files and syntax. With that in mind the implementation focus around the files produced by two tools, PIPE2, Platform Independent Petri net Editor [35], and PNLab [36].

The Parser unit not only is a concise and easier data structure for the execution module, but also simplifies and allows to have different methods to parse the files. Meaning that one could easily create, modify or extend the methods in order to parse files with a different syntax to the raw data structures ensuring nothing in the other classes from the package need to be modified and that everything is still functional.

Notice that, pnml files use a syntax where each Petri net arc between a place and transition, or vice-versa, is represented individually using a notation of pair <source, target> to represent the directional connection. Although transforming this notation to the matrices as it was defined in Definition 1 is needed, it will only be done in the PetriNetStructure class.

Furthermore, and regarding places and their respective initial marking, the decision was to not load initial marking values for the predicate places, since a predicate place represents a logical condition and its value must be set by the Predicate Manager.

The following items are the most important data structures and methods present on the Parser:

Data Structures:

1. **Node**, base struct for each parsed element of the Petri net model. Contains a string id and

two arrays of string ids to store the sources and targets nodes;

2. **Place**, struct that extends the Node struct with a capacity, a name and a type;
3. **Transition**, struct of the Node type;
4. **Arc**, struct that extends the Node struct with the numerical cost of the arc;
5. **arcs**, a map[1] from Arc Id to Arc, matches from arc id to arc;
6. **places**, a map from Place Id to Place;
7. **transitions**, a map from Transition Id to Transition;
8. **initial_marking**, a map from Place Id to an integer value that matches the number of tokens of the initial marking for that place.

Methods:

1. **Parser**, constructor of the struct, receives the filename of the file that is going to be parsed, calls the remainder methods in order to parser every element and populate the data structures,
2. **extractArc**, used to parse an arc element from the file
3. **extractPlace**, used to parse a place element from the file
4. **extractTransition**, used to parse a transition element from the file
5. **solveCapacities**, implemented to solve the capacities of the places in order to achieve a Petri net having only places with infinite capacity, according to the Definition 6,
6. **fillSourceTarget**, used to fill the source and target information for every place and transition according to the parsed arcs.

To have a better understanding of the actual process of the parsing of a file, the flowchart depicted in Figure 3.4 displays the order and in which conditions the methods are called upon a constructor call for a new object of the Parse struct, as well as where verifications of terminating conditions that result from errors found while parsing are made.

The possible errors that are identified while instantiating a parser object and that make the process terminate are due to errors detected by the parsing library, parsing elements that do not contain every needed element or have incorrect attributes that make the execution of the methods invalid.

3.2.2 Petri Net Structure

The PetriNetStructure is the class which implements Definition 2, where the referred immutable data corresponds to the set of places, the set of transitions, the pre and post matrices and the initial marking of the Marked Petri net.

As declared before, the decision of not to use the parsed data directly was because having a different representation of the data was a more consistent, less resourceful and easier way of implementing the algorithms needed for the Petri net execution.

[1] A map is a container that consists in a pair between a key and a value, where each key is unique.

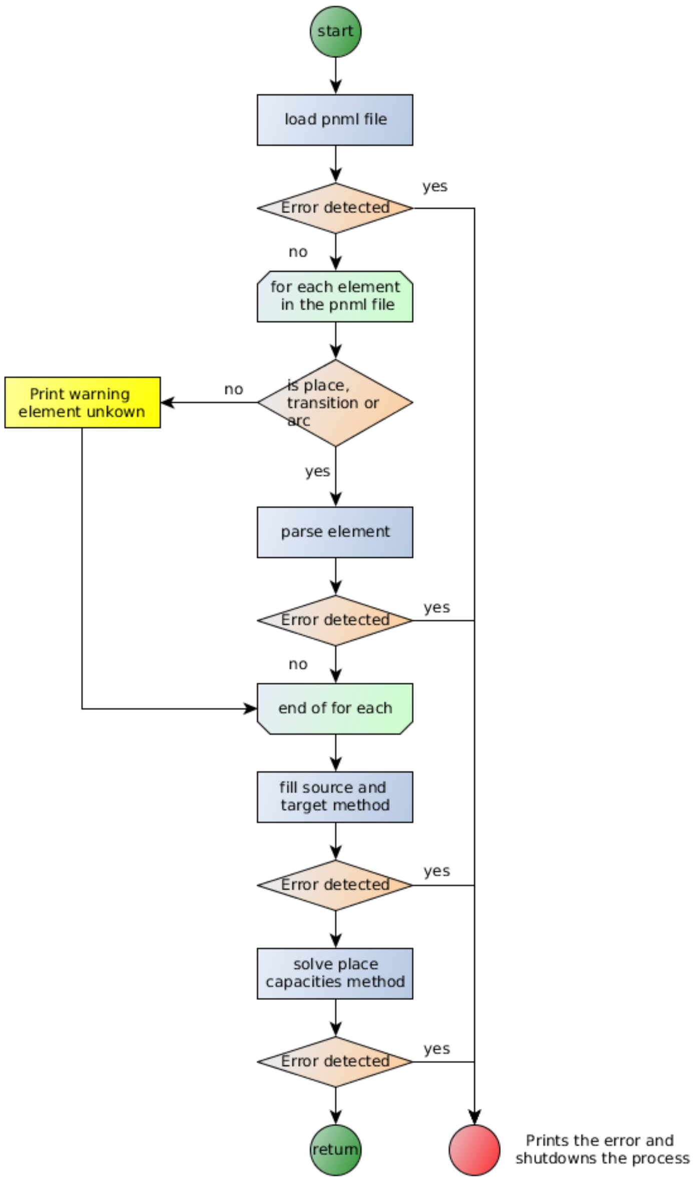

start
load pnml file
Error detected
yes
no
for each element in the pnml file
is place, transition or arc
no
Print warning element unkown
yes
parse element
Error detected
yes
no
end of for each
fill source and target method
Error detected
yes
solve place capacities method
Error detected
yes
return
Prints the error and shutdowns the process

Figure 3.4: Parser Constructor flowchart.

The previous parsed structs were relaxed, unique numerical ids that match the string ids of the parsed nodes were created and used to implement, represent, search and iterate over the defined data containers since they are faster and more efficient. Additionally, data containers that match the new ids to the previously parsed data structures ids were implemented.

Recalling that each parsed arc was represented as a pair $<$source, target$>$ and in order to implement them in the matrices notation, several possibilities were studied using distinct standard data structures containers. Taking in consideration that typically the pre and post matrices of a Marked Petri net model are sparse, transforming the arcs from the parsed struct to the typical two-dimensional array representation would result in a waste of memory. To overcome this issue, the decision was to implement the matrices using a combination of two dictionary of keys. Each matrix is represented using a combination of two maps and given their properties the implementation allows the ij element of a matrix to be accessed directly using two bracket operators in the similar fashion as if a two-dimensional array was used.

The class also ensures, upon the connection between the Petri Net Manager with the Predicate Manager and the Primitive Action Manager, that all the predicates and primitive actions needed by the Petri net model were implemented, alerting the user for each specific predicate or primitive that was not implemented on the respective manager.

The following items are the most important data structures and methods present on the Petri Net Structure:

Data Structures:

1. **filename**, the filename of the PetriNetModel this class describes;
2. **source**, the respective Parser struct;
3. **Place** and **Transition**, Place and Transition struct, contain only a numerical Id;
4. **pa_id2pn_id** and **pn_id2pa_id**, multimap[2] from a Primitive Action Id to Place Id and a map from a Place Id to a Primitive Action Id, matches between the Primitive Action Manager ids and to the Petri Net Execution ids;
5. **pm_id2pn_id** and **pn_id2pm_id**, multimap from a Predicate Id to Place Id and a map from a Place Id to a Predicate Id, matches between the Predicate Manager ids and the Pretri Net Execution ids;
6. **actions**, **predicates**, **tasks** and **regular**, four sets of places relative to actions, predicates, tasks and regular places;
7. **backwardsMatrix** and **forwardsMatrix**, map from *Transition Id* to a map from *Place Id* to the an *integer*;
8. **initial_marking**, a map from Place Id to the number of tokens of that place.

Methods:

1. **PetriNetStructure**, constructor of the class, receives the filename of the pnml file, calls the constructor of the Parser struct with that filename and afterwards calls the extract methods in order to populate the data structures;
2. **pmMapCallback**, receives a map that matches predicate names to ids (predicate ids of the predicate manager). Iterates over the received data, populating the containers that create the bridge between the ids of the two packages, and verifies if all needed predicates are implemented in the Predicate Manager side;
3. **paMapCallback**, similar to the pmMapCallback but relative to the primitive actions;
4. **extractPlaces**, used to extract and transform all Parsed Places elements to the new notation of Places;

[2]similar to a map, a multimap is a container that consist of a (key, value) pair, however in a multimap the same key could be specified for multiple values

5. **extractTransition**, similar to the extractPlaces but relative to transitions;
6. **extractPrePosMatrix**, method used to obtain the transposed of the Pre and Post Matrices;
7. **extractInitMarking**, method used to obtain the new notation of the initial marking of the Marked Petri Net from the Parser struct.

The flowchart depicted in Figure 3.5 depicts the order in which each method is called upon the construction of a PetriNetStructure object.

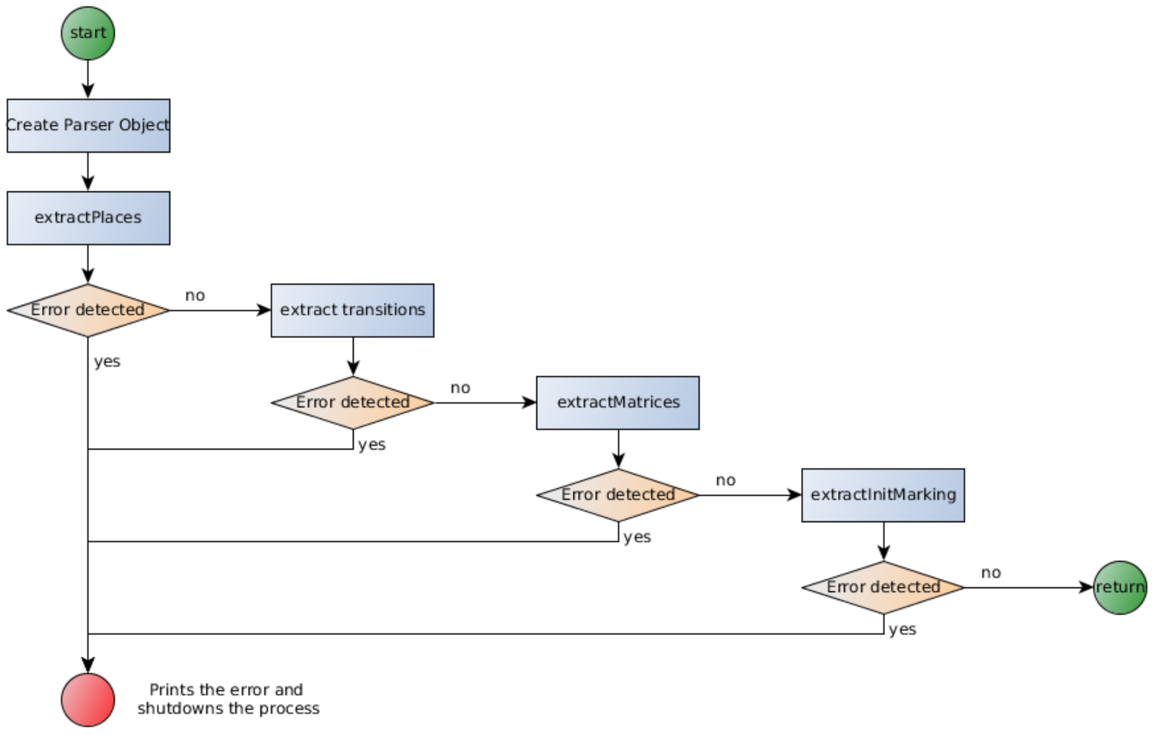


Figure 3.5: PetriNetStructure constructor flowchart.

Although the elements parsed validation in the Parser struct, additionally verifications were implemented to ensure that the Marked Petri net model was correctly loaded. Additionally, while extracting the matrices the transitions are verified, raising an alert in order to inform the user of a bad task design if a predicate place and its negated predicate place are connected to the same transition.

3.2.3 Petri Net Executor

Although the prior defined structures relate to parse, transform and store the data they do not define the methods for the execution of the Marked Petri net according to the definitions described in Subsection 2.1.1. Keeping the execution methods and variables separated from the immutable data allows not only to keep the software framework modular and flexible, but also to store only the execution representation of the Marked Petri nets that are actually running.

From both aspects, the PetriNetExecutor class arises, consisting in a collection of methods and variables that use the data structures from the PetriNetStructure in order to properly execute a Marked Petri net and perform useful verifications. The following items present the most relevant data structures and methods implemented in the PetriNetExecution class:

Data Structures:

1. **petrinet_structure**, a smart pointer to the respective PetriNetStructure class;
2. **curr_marking**, which is a map identical to the initial_marking from the PetriNetStructure class, but represents the current marking of the Petri net model while executing.

Methods:

1. **PetriNetExecution**, constructor of the class, receives a smart pointer of the respective object from the PetriNetStructure class and creates the current marking with the initial marking from the PetriNetStructure class;
2. **activeTransition**, used to obtain the set of transitions that are enabled according to the Marked Petri net and the current marking;
3. **chooseTransition**, returns the id of a randomly chosen transition from a set of active transitions;
4. **fireTransition**, ”fires” the previously chosen transition, which means update the current marking according to the definition 4;
5. **executeNet**, executes the Petri net while active transitions exist,
6. **updatePredicatesAndExecute**, receives an update message of the predicates values, and upon the verification if the maps of predicates and actions were already published and validated, changes the current marking according to the update received and calls the executeNet method;
7. **getActions2Execute**, returns the set of actions which should start or continue executing;
8. **getTasks2Execute**, similar to the getActions2Execute but relative to the tasks.

At this point it is important to fully understand how the executeNet method works, and in which conditions it stops. A Marked Petri net that represents a robot task change from one marking state to another by firing enabled transitions one at a time, until a state where there are none active transitions. This means that the execution method has to correct implement the Definitions 3 and 4, in order to evaluate the enabled transitions and fire them.

The Algorithm 1 and Figure 3.6 present the pseudo-code and a flow-chart of the executeNet method, respectively.

It is also relevant to review how the randomly chosen transition is obtained whenever more than one transition is enabled. The chooseTransition was implemented using the boost random libraries [37], using the Mersenne Twister pseudo-random number generator [38], with a uniform integer distribution of the enabled transitions, seeding the generator on the instantiation of the class.

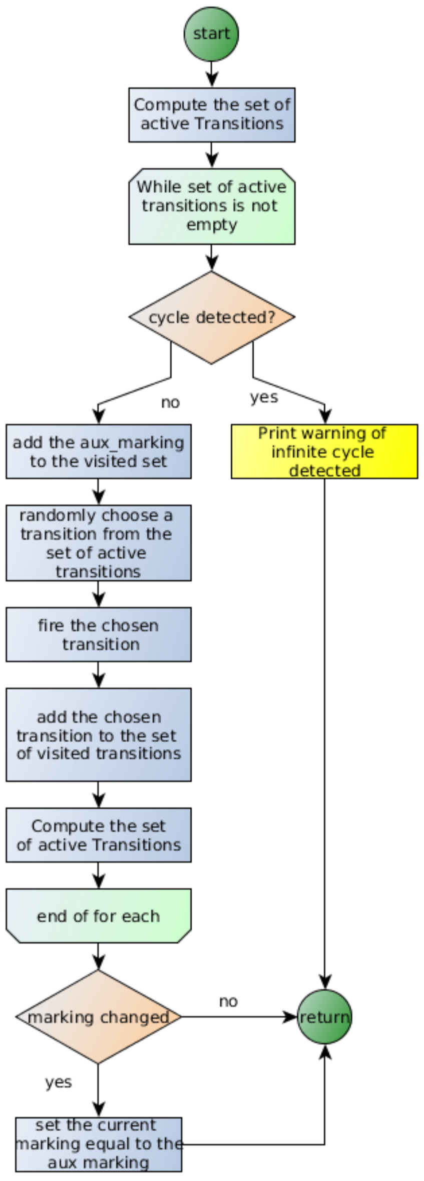

start
Compute the set of active Transitions
While set of active transitions is not empty
cycle detected?
no
yes
add the aux_marking to the visited set
Print warning of infinite cycle detected
randomly choose a transition from the set of active transitions
fire the chosen transition
add the chosen transition to the set of visited transitions
Compute the set of active Transitions
end of for each
marking changed
no
return
yes
set the current marking equal to the aux marking

Figure 3.6: Flowchart of the executNet method

```
Data: $visited \leftarrow$ auxiliary set to keep track of the different Petri net markings that were "visited"
      during this execution call
$visited\_transitions \leftarrow$ the set of visited transitions
$active\_transitions \leftarrow$ auxiliary set to keep track of the transitions that are active during each
execution step
$aux\_marking \leftarrow$ auxiliary map to be used in the execution steps instead of the curr_marking map
Result: a new Petri net marking.
$visited \leftarrow \{\}$;
$visited\_transitions \leftarrow \{\}$;
$aux\_marking \leftarrow curr\_marking$;
$active\_transitions \leftarrow$ computed using the $aux\_marking$ and Definition 3;
while $active\_transitions$ is not empty do
    if the Marked Petri net is in a loop situation then
        Publish a warning to the user; return
    end
    add $aux\_marking$ to $visited$;
    choose a transition from $active\_transitions$;
    fire the chosen transition according to Definition 4;
    add the chosen transition to $visited\_transitions$;
    $active\_transitions \leftarrow$ computed using the $aux\_marking$ and Definition 3;
end
if $curr\_marking \neq aux\_marking$ then
    $curr\_marking \leftarrow aux\_marking$;
end
```

Algorithm 1: Pseudo-code of the implementation of the executNet method

3.2.4 Petri Net Manager

The three data structures described until this moment are sufficient to represent and execute a single Marked Petri net of a robot task, even though a PetriNetManager class is defined in order to fully allow the development of a robot task plan using hierarchical Marked Petri nets.

This class is defined as a manager which not only stores and handles the execution of hierarchical Marked Petri nets but also ensures the communication between the Primitive Action Manager and the Predicate Manager packages. The following items describe the most relevant data structures and methods implemented:

Data Structures:

1. **immutable_petrinets**, a map from a string to a smart pointer of a PetriNetStructure object, match between a Marked Petri net name and its PetriNetStructure object;
2. **petrinets**, similar to the immutable_petrinets, but relative PetriNetExecutor objects;
3. **pn_name**, a string that stores the top level Marked Petri net name;
4. **pred_updates_sub**, **pred_map_sub** and **pa_map_sub**, ROS subscribers relative to the topics: /predicate_update, /predicate_maps and /actions_map;
5. **pa_update**, ROS publisher relative to the /action_update topic;
6. **actions**, a set of integers that represent the actions to be sent to the primitive action manager.

Methods:

1. **PetriNetManager**, constructor of the class, does not receive any argument, calls the addNet method;
2. **addNet**, recursive method used to create all PetriNetStructure objects and populate the immutable_petrinets map container;
3. **executeNet**, recursive method used to control the execution and existence of the PetriNetExecutor objects present on the petrinets map container. It broadcasts the predicate_update message to every object that is running and creates a set of *actions* according to the result of the execution of each net;
4. **paManagerCallback**, callback for the PrimitiveActionManager /action_map topic subscriber, verifies and transforms the received message and broadcasts it to every PetriNetStructure object present on the immutable_petrinets map;
5. **predicateMapCallback**, similar to the paManagerCallback but relative to the Predicate Manager /predicate_map topic;
6. **predicateUpdateCallback**, callback for the PredicateManager /predicate_update topic subscriber, verifies if the primitive action manager and the predicate manager were already published and loaded, and in an affirmative case calls the executeNet method;
7. **publishActionUpdate**, publisher of the /action_update topic, creates the appropriated topic message according to the *actions* set and publishes it.

The use of smart pointers provides an automatic efficient memory management. While the result from having two different classes, PetriNetStructure and PetriNetExecutor, to represent the immutable and the running state of a Marked Petri net, allows the PetriNetExecutor objects to be automatically destroyed from memory whenever the respective Marked Petri net is not being executed.

Concerning the *actions* set along the process life-time, the set is cleaned after every execution and then is again populated using all the results from the getActions2Execute method of the PetriNetExecutor objects that are running.

The approach used in the recursive methods addNet and executeNet to verify and travel along each container was based on the Depth-first search algorithm [39], where the methods are called starting always from the top level Marked Petri net. Additionally, a loop between hierarchical Petri nets verification was implemented which alerts the user whenever such happens.

3.2.5 How to use the package

As stated, the package works as a black box from a user development standpoint where a user only needs to specify two parameters on the launch file of the package, the directory, */pn_wd*, and the top level Petri net file, */pn_filename*. Launching the package will start the respective ROS node, which will load the Petri net model, or models, and wait until the primitive action manager and predicate manager maps get published before starting the execution.

As example, if one wants to execute the Petri Net Execution package for the task named "thesis.pnml", the parameters present in the pnexecution.launch should be modified to:

- /pn_filename = thesis
- /pn_wd = $(find petri_net_execution)/examples/$

3.3 Primitive Action Manager

The second package of the framework is the Primitive Action Manager, a library that implements a bridge between user defined primitive actions and the task plan execution.

3.3.1 Description

The Primitive Action Manager is a library for handling the execution of primitive actions, starting and stopping them according to the received information from the Petri Net Execution package. The diagram of the Primitive Action Manager depicted in Figure 3.7 illustrates what needs to be defined and registered by the user, the connection between the different elements of the package and the connection to ROS or other external packages. The three different arrow colors, black, orange and blue, represent communication/connection to ROS or external packages, registration of the user defined primitive actions by the user and the actions execution call from the manager, respectively.

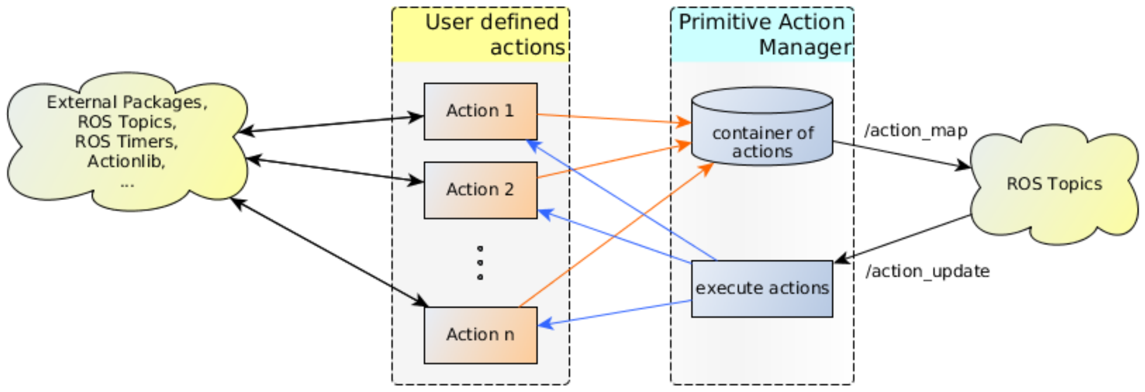


Figure 3.7: PrimitiveActionManager Package

User defined primitive actions are implemented through the use of C++ classes, each usually representing a single primitive action, existing a clear association between a class instantiation and the start of an action, or destruction and stop.

In fact, when a request to run a action is received by the manager, the respective action class is instantiated, being destroyed when it should stop. The user defined classes are registered in the manager by invoking a template method, associating them with the name. This name must be the same as the label of the action place this class is implementing[3].

[3]Recalling the label definitions present on Subsection 2.1.2, a label of an action place is composed by the prefix "a." or "action." plus the name of the action.

Internally, the classes are registered using a PrimitiveType instance, which replaces its template parameters with concrete types. The PrimitiveType class inherits from a PrimitiveTypeBase abstract class that does not have a template parameter, which is necessary to store and refer to this classes in C++ containers.

The manager class also establishes a communication protocol with the Petri Net Execution Package in order to notify which actions were registered and receive the set of actions that should be executed at each step. Although using strings would be sufficient to do that, the decision to implement unique numerical ids that matches the actions names was taken to simultaneous reduce the size of the communication messages, while allowing for a fastest execution of the implemented methods and operations over the containers. Additionally, three messages types were implemented in a similar fashion to the one present on the Predicate Manager:

- **PAInfo**, a message containing the information of a single action. It is composed by the action name and a unique identification number.
- **PAInfoMap**, a message containing information about multiple actions. It is composed by an array of PAInfo message types.
- **PAUpdate**, a message containing information about the actions to execute. It is composed by an array that contains the identification numbers of the actions to be executed.

After instantiation of the Manager, a PAInfoMap message is published, containing all registered primitives names and the respective ids. This is necessary in order for the Petri Net Execution to be able to store the matches between the internal ids and the ids from the primitive action manager, verify if all actions needed were implemented and registered, and to communicate back using a PAUpdate message to the Primitive Action Manager the actions to execute.

After a message is received, the manager computes the set of actions that should stop, which means the ones that were running and were not in the new message received, afterwards, stop each of those actions. Then the manager goes over the message and starts or continues the execution of each action.

All primitive actions run over the same ROS node. The simultaneous execution of the manager code and multiple primitive actions is achieved using common ROS mechanisms, as ROS Timers[4] or subscribed topic callbacks.

3.3.2 How to use the package

In order to use the primitive action manager package a user has to create a ROS node and implement primitive action classes. The node must instantiate a manager class and register the primitive action classes on it. Since the manager will publish the map of registered actions upon its instantiation, it should be declared as the last variable or using a smart pointer to it, so that one could initialize it only after everything is ready.

[4]A ROS Timer allow to schedule a callback to happen at a specific rate [40]

3.4 Predicate Manager

The Predicate Manager is the last package of the framework. As stated, the decision was to use a Predicate Manager package that is available in the ROS repositories and integrate it in the remainder framework. The decision was due to the requirements needed for the solution and the capabilities that are implemented in the Predicate Manager package. The package is part of the Markov Decision Making metapackage for ROS and was developed by João Messias during his Phd thesis [41]. Currently the package is publicly available in the ROS repository.

3.4.1 Description

The Predicate Manager is a library that allows a user to create and register predicates based on logical conditions, and to publish a set of updates whenever their logical values change. Similar to the Primitive Action Manager diagram depicted before, the diagram of the Predicate Manager depicted in Figure 3.8 illustrates what needs to be defined and registered by the user, the connection between the different elements of the package and the connection to ROS or other external packages. Once again, the different arrow colors, black and orange, represent communication/connection to ROS or external packages and the registration of the user defined predicates by the user, respectively.

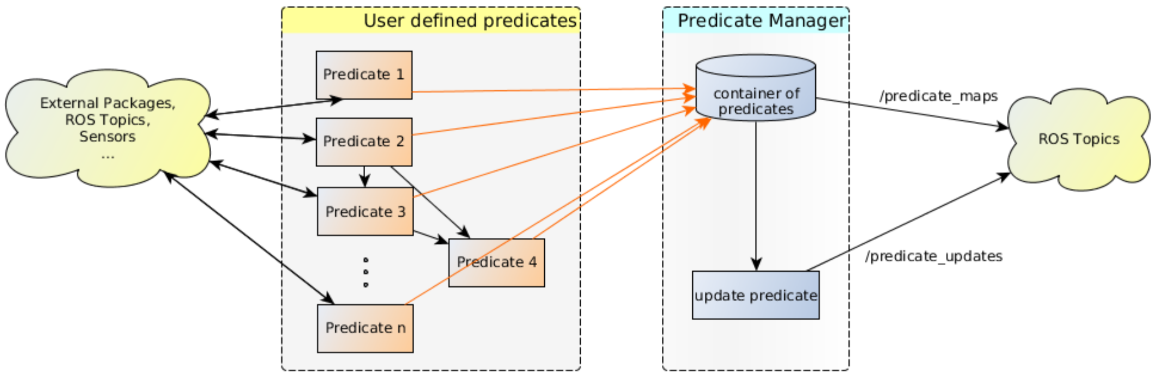


Figure 3.8: PredicateManager Package

A Predicate in the Predicate Manager library is characterized by its name or id and its current value. The library allows two types of predicates to be defined: generic abstract predicates and topological predicates. The former are implemented by the user using information from sensors. The latter use the definition of a topological predicates, which are based on the robot position relative to a certain labeled map.

Additionally, the Predicate Manager provides the definition of predicates using propositional calculus with other predicates, which are useful to create predicates that embody multiple conditions that were already addressed by other predicates.

In order to implement predicates of the generic abstract predicate type, a class has to be implemented and inherit the Predicate class available in the Predicate Manager library in order to inherit the class variables and methods. Addtionally, the Predicate class allows the definition of predicates dependencies that a user can specify for the case of a predicate value depends on the value of other predicates.

In the scope of this thesis and in terms of communication, the Predicate Manager publishes two topics: /predicate_maps and /predicate_update, and implements three messages types that are relevant:

- **PredicateInfo**, a message containing the information of a single predicate. It is composed by the predicate name and a unique identification number.
- **PredicateInfoMap**, a message containing information about multiple predicates. It is composed by a unique identification number that represents the Predicate Manager node that publishes the message and a array of PredicateInfo message types.
- **PredicateUpdate**, a message containing information about the value of the predicates. It is composed by a unique identification number that represents the Predicate Manager node that publishes the message, a numerical counter and three arrays of predicates identification numbers:
  - true_predicates, contains all the identification numbers of the predicates that are true.
  - rising_predicates, contains all the identification numbers of the predicates that had a false logical value in the last update message and are now true.
  - falling_predicates, contains all the identification numbers of the predicates that had a true logical value in the last update message and are now false.

3.4.2 How to use the package

Similar to the primitive action manager package, in order to use the predicate manager package a user has to create a ROS node, and implement predicate classes, that extends a Predicate class from the Predicate Manager library. The node created must declare a PredicateManager object and the predicates must be registered on that object.

Chapter 4

Experiments and Results

In order to test the implemented packages for the expected execution of the framework, each package and class was first analyzed individually to ensure they were correctly designed and implemented and then the framework was tested using several different case scenarios.

This chapter starts by presenting useful verifications and the solve places capacities functionality, giving information about them, and then goes over the execution of a set of case scenarios using the framework as a whole.

4.1 Functionality tests

In a project divided in several different components that are supposed to work integrated, it is critical to ensure that each component works as expected individually. Each class and method was manually tested in order to ensure it achieves the purpose of its implementation.

In fact, the implementation of the framework was made as an iterative process, testing and correcting each method every time it was needed, achieving a robust solution, which takes into account possible critical and non critical errors and informs the user according to those errors.

The following subsections will cover relevant functionalities of the implemented packages, explaining them using useful examples.

4.1.1 Useful verifications

Simple functionalities were implemented that trigger alerts to the user and continue or terminate the execution according to the severity of the error detected. The following subsections will cover implemented functionalities that are crucial for the execution.

File existence

Upon start the PetriNetExecution node a verification if the working directory, where all the needed pnml files are stored, and the top level Petri net filename are set in the node launch file. If any of them is not set, a FATAL error is triggered, publishing the respective message to the user and the execution is killed.

Additionally, after checking if the parameters were set, a verification for the existence of such working directory and for all pnml files needed is performed, starting from checking if the top level Petri net exists and then go from a top to bottom approach for each task place that is present in the Petri net model. Once again if the check for existence fails, a FATAL error is triggered, publishing a message to the user and killing the package node.

Registered Primitive Actions and Predicates

Other alerts happen when a user tries to execute a task which needs primitive actions that are not registered in the primitive action manager, the same applies relative to predicates and the predicate manager.

Using the task from Figure 2.12, and a primitive action and a predicate manager, where both the action *go2LRM* and the predicate *IsInLRM* are not registered, would result in a situation where none of the maps and respective contained ids were loaded and matched with the ids of the PetriNetExecutor, alerting the user for the specific primitive actions and predicates that are missing.

However, notice that these errors are not FATAL, meaning the PetriNetExecution package would still be running and if the correct maps were published after the alerts, it would start the execution of the Marked Petri net.

4.1.2 Extend Places Capacities

As stated, one of the functionalities implemented in the Petri Net Execution package is the capability to deal with a Marked Petri net where the number of tokens in the places is limited.

As an example the Marked Petri net depicted in Figure 4.1 will be used, where places *t.fetch_box*, *a.pick_box* and *p.box_detected*, have each a capacity of one token, and all the remainder places have all infinite capacity.

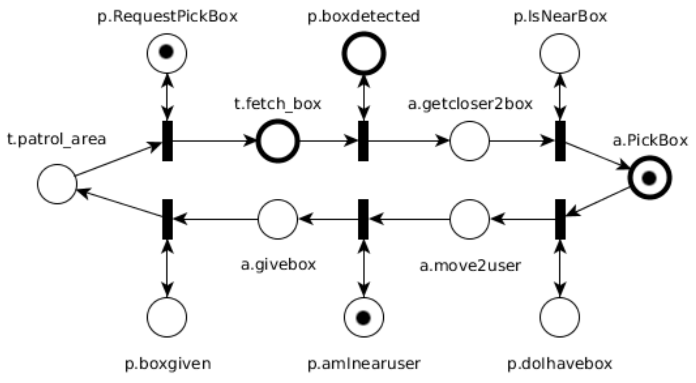


Figure 4.1: Task modeled by a Marked Petri net where three places have bounded capacities defined.

The Pre and Post matrices before using the solveCapacities method, implemented in order extend the capacity places of a Marked Petri net model, are present in Table 4.1.

	Pre Matrix						Post Matrix					
	T0	T1	T2	T3	T4	T5	T0	T1	T2	T3	T4	T5
a.pickbox	0	0	0	1	0	0	0	0	1	0	0	0
a.getcloser2box	0	0	1	0	0	0	0	1	0	0	0	0
a.givebox	0	0	0	0	0	1	0	0	0	0	1	0
a.move2user	0	0	0	0	1	0	0	0	0	1	0	0
p.DoIHaveBox	0	0	0	1	0	0	0	0	0	1	0	0
p.boxdetected	0	1	0	0	0	0	0	1	0	0	0	0
p.RequestPickBox	1	0	0	0	0	0	1	0	0	0	0	0
p.IsNearBox	0	0	1	0	0	0	0	0	1	0	0	0
p.box_given	0	0	0	0	0	1	0	0	0	0	0	1
t.fetch_box	0	1	0	0	0	0	1	0	0	0	0	0
t.patrol_area	1	0	0	0	0	0	0	0	0	0	0	1
p.amInearuser	0	0	0	0	1	0	0	0	0	0	1	0

Table 4.1: Pre and Post Matrices from the Marked Petri net present on Figure 4.1

Running the PetriNetExecution package on the Marked Petri net described above results in the new Pre and Post matrices present in Table 4.2. As one can see, the matrices have now two more rows that match the complement places and the respective opposite arcs of the *t.fetch_box* and *a.pick_box*. Figure 4.2 displays the Marked Petri net obtained from using the solve capacities method.

	Pre Matrix						Post Matrix					
	T0	T1	T2	T3	T4	T5	T0	T1	T2	T3	T4	T5
a.pickbox	0	0	0	1	0	0	0	0	1	0	0	0
a.getcloser2box	0	0	1	0	0	0	0	1	0	0	0	0
a.givebox	0	0	0	0	0	1	0	0	0	0	1	0
a.move2user	0	0	0	0	1	0	0	0	0	1	0	0
p.DoIHaveBox	0	0	0	1	0	0	0	0	0	1	0	0
p.boxdetected	0	1	0	0	0	0	0	1	0	0	0	0
p.RequestPickBox	1	0	0	0	0	0	1	0	0	0	0	0
p.IsNearBox	0	0	1	0	0	0	0	0	1	0	0	0
p.box_given	0	0	0	0	0	1	0	0	0	0	0	1
t.fetch_box	0	1	0	0	0	0	1	0	0	0	0	0
t.patrol_area	1	0	0	0	0	0	0	0	0	0	0	1
p.amInearuser	0	0	0	0	1	0	0	0	0	0	1	0
cap.pickbox	0	0	1	0	0	0	0	0	0	1	0	0
cap.fetch_box	1	0	0	0	0	0	0	1	0	0	0	0

Table 4.2: Pre and Post Matrices obtained after extending the capacity places of the Marked Petri net present on Figure 4.1

Additionally, and as expected, one can notice that although the graphical model of Marked Petri net had one token in the *p.RequestPickBox* and the *p.amInearuser*, the same were not loaded when executing the Petri Net Execution package.

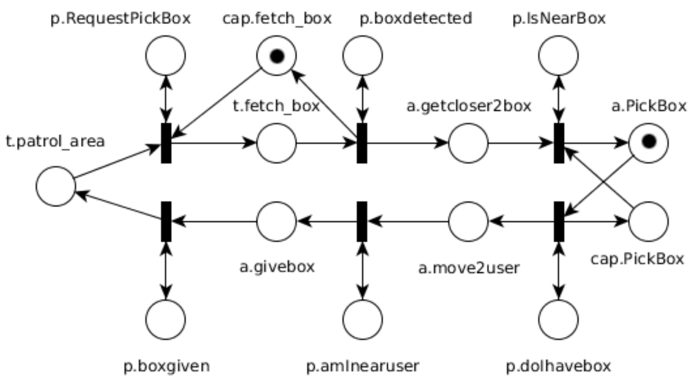


Figure 4.2: Marked Petri net obtained from using the solve capacities method on the Marked Petri net from Figure 4.1

4.2 Framework Proof of Concept

In order to demonstrate the proposed framework several Proof of Concept tests were developed and executed using a simulated and a real scenario.

Simulators and their respective properties are a useful tool when developing both complex and simple systems. In Robotics, simulators provide physics engines capable of computing the dynamics of both robot and world environment. Thus, they allow to create, evaluate and optimize applications without depending on having to deploy psychical robots, devices and environments.

In order to ease the development of the tests, and since the focus is in testing the framework integration and do not require a complex simulation environment, the decision was to take advantage of the mdm_example package present in MDM metapackage and modify it in order to implement the developed solution. The full scenario and environment were described in the following Subsection.

Additionally, the models used to perform the tests were designed in order to be easy to understand but powerful to exemplify different execution paradigms and how the framework behaves in each situation. Note that, although all models are based in patrol tasks, the proposed framework is not restricted to model this kind of problems.

The tasks models used to test and evaluate the proposed framework are:

1. Simple sequential patrol;
2. Patrol with two conflict situations;
3. Patrol with concurrent actions;
4. Hierarchical task;
5. Hierarchical patrol modeled using task places where a hierarchical loop is detected.

4.2.1 Simulated Scenario

The mdm_example package scenario is composed by a map based on the blueprint of the eight floor of the North Tower from the IST campus, and by a physical description of a nonholonomic robot, Piooner3-AT [42], with a laser range finder on the front of the robot. The simulation uses Stage [43] as the physical engine, which is a two-dimensional simulator that provides cheap and fast computation models. A graphical representation of the map and the robot as they appear in the Stage simulator is depicted in Figure 4.3.

Additionally, the package provides the configurations for the localization and guidance, through the amcl[1] and move_base[2] ROS packages, and an implementation of several nodes of the MDM_LIBRARY. Every node implementation that was not going to be used was removed, and the parameters and launch files modified in order to clean up the dependencies of those, ending with a package with only the

[1] http://wiki.ros.org/amcl
[2] http://wiki.ros.org/move_base

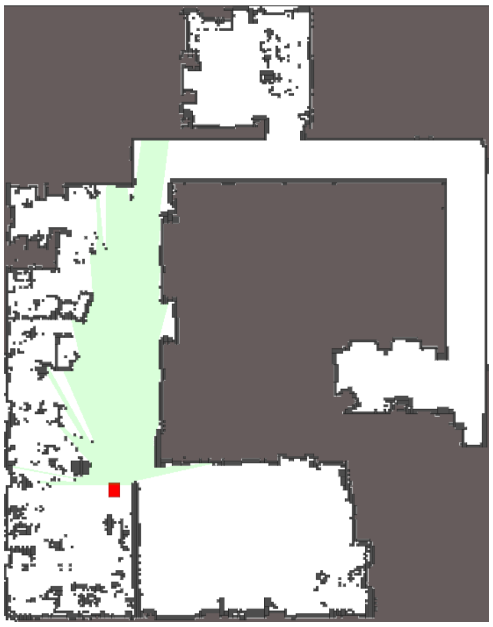


Figure 4.3: The Stage representation of the robot, red rectangle, and the map. The green area visible on the map is the laser range finder scan

configurations for the localization and guidance packages, and a implementation of a Predicate Manager node that was further modified to suit the needs.

Thereafter a set of tasks that are described in the following Subsections were designed, a node of the *sound play* package was added in order to provide audio output capabilities to the robot and, not only a Primitive Action Manager node was implemented, but also additionally needed predicates were added to the Predicate Manager node. The actions implemented are divided in two sets:

- **Navigation actions**, publish a message with a specific goal pose to the *move base* goal topic. The name of the navigation actions implemented start with ”go2”,
- **Text-to-Speech actions**, publish specific message with a text string to the *sound play*. The name of the text-to-speech actions implemented start with ”tts”.
- **Actions**:
  - **go2LRM**, move to a specific position inside the LRM office,
  - **go2Elevator**, move to a specific position near the elevator,
  - **go2CoffeeRoom**, move to a specific position inside the Coffee room, near to the coffee machine,
  - **go2SoccerField**, move to a specific position inside the Soccer field,
  - **ttsCurrPosition**, say the current position on the map,
  - **ttsLRM**, say ”I am over the predefined LRM position”,
  - **ttsElevator**,say ”I am near to the elevator”,

  - **ttsCoffee**,say ”I am near the coffee machine”,
  - **ttsMoving**, say ”Attention, I am moving to *position*”, where the position is given by the *move base* goal topic.

- **Predicates**:
  - **IsNearLRMPosition**, evaluation between the current position of the robot and an area defined in the LRM office, true if robot is inside that area and false otherwise
  - **IsNearElevator**, similar to IsNearLRMPosition but relative to the elevator position.
  - **IsNearCoffeeMachine**, similar to IsNearLRMPosition but relative to the coffee machine position.
  - **IsInSoccerField**, similar to the IsNearLRMPosition but relative to the Soccer Field,
  - **IsMoving**, check if the robot is currently moving,
  - **IsNearStairs**, check if the robot is currently near the stairs door.

Lastly, Figure 4.4 shows the navigation goal positions of each navigation action as filled circles and the areas that trigger a change in the localization predicates as squares. Please notice that the filled circles are centered inside the colored squares. The correspondence between the respective colors and the actions/predicates are:

- Yellow $\rightarrow$ go2LRM goal position and IsNearLRMPosition area;
- Green $\rightarrow$ go2Elevator goal position and IsNearElevator area;
- Blue $\rightarrow$ go2SoccerField goal position and IsInSoccerField area;
- Orange $\rightarrow$ go2CoffeeRoom goal position and IsNearCoffeeMachine area;
- Purple $\rightarrow$ IsNearStairs area;

4.2.2 Simple sequential patrol

For the first test of the framework, a simple sequential patrol task, depicted in Figure 4.5, was designed and used.

The patrol is composed by two actions that are executed individually and changes from one to the other as soon as the robot position is inside of the respective predicate area. The framework is capable of executing the task without problems as it can be seen in Figure 4.6, where it is displayed the robot path done while executing the task. Additionally a snippet of console output from the execution of the PetriNetExecution and the PrimitiveActionManager packages are depicted in Figure 4.7 and 4.8, where is also possible to identify the match between the firing of the transitions and the respective start and stop of each action using the timestamps of the info messages[3].

[3]The timestamps are in the form of [Unix/Epoch time, time since the beginning of execution].

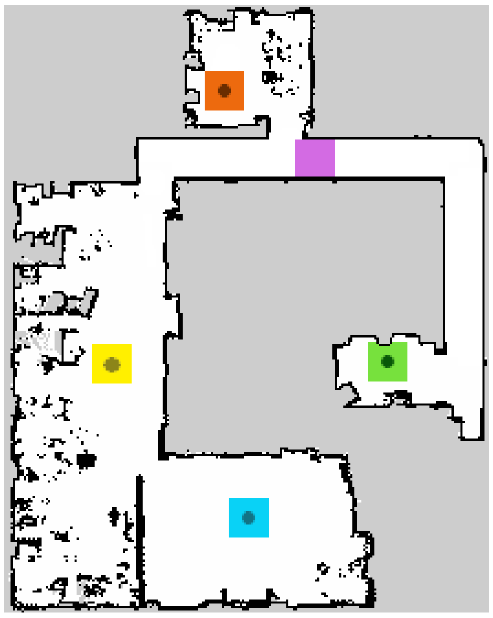


Figure 4.4: Map with navigation the goal positions and areas that trigger changes in the predicates.

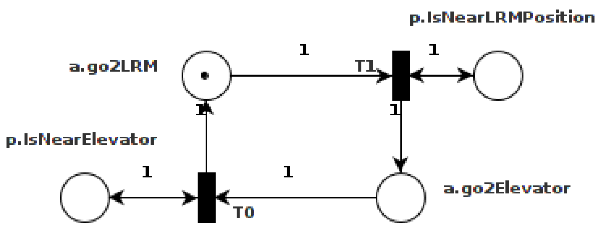


Figure 4.5: Simple sequential patrol task between two locations

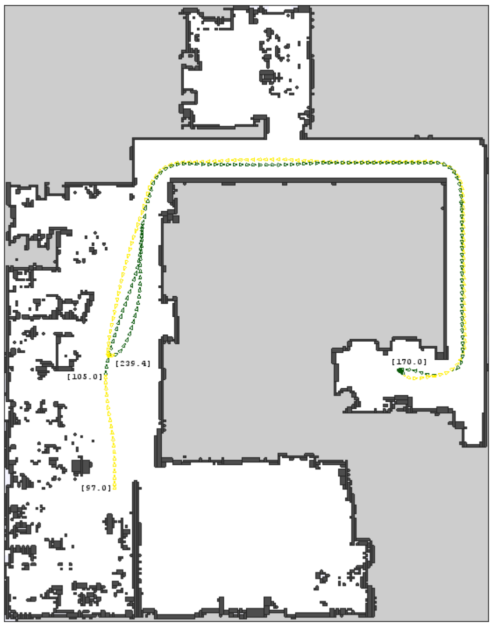
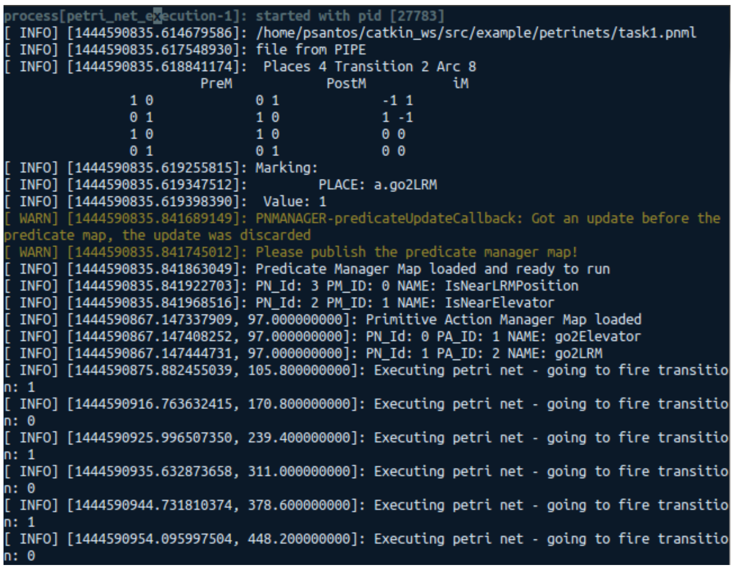
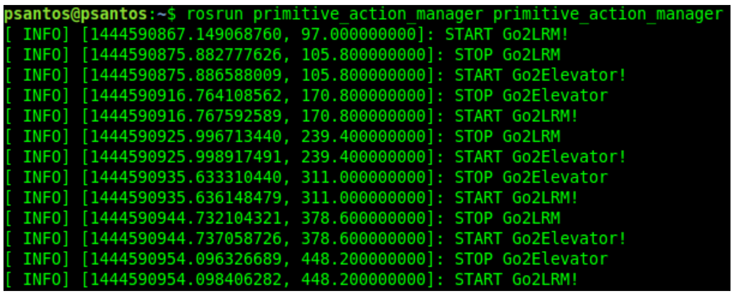


Figure 4.6: Representation of the robot path during the execution of the task from Figure 4.5. Yellow arrows represent the go2LRM action path, green arrows the go2Elevator action path and the text boxes are relative to the execution timestamps where the task and respective actions in execution changed.



```
process[petri_net_execution-1]: started with pid [27783]
[ INFO] [1444590835.614679586]: /home/psantos/catkin_ws/src/example/petrinets/task1.pnml
[ INFO] [1444590835.617548930]: file from PIPE
[ INFO] [1444590835.618841174]:  Places 4 Transition 2 Arc 8
                         PreM            PostM           iM
                1 0             0 1             -1 1
                0 1             1 0             1 -1
                1 0             1 0             0 0
                0 1             0 1             0 0
[ INFO] [1444590835.619255815]: Marking:
[ INFO] [1444590835.619347512]:         PLACE: a.go2LRM
[ INFO] [1444590835.619398390]:  Value: 1
[ WARN] [1444590835.841689149]: PNMANAGER-predicateUpdateCallback: Got an update before the
predicate map, the update was discarded
[ WARN] [1444590835.841745012]: Please publish the predicate manager map!
[ INFO] [1444590835.841863049]: Predicate Manager Map loaded and ready to run
[ INFO] [1444590835.841922703]: PN_Id: 3 PM_ID: 0 NAME: IsNearLRMPosition
[ INFO] [1444590835.841968516]: PN_Id: 2 PM_ID: 1 NAME: IsNearElevator
[ INFO] [1444590867.147337909, 97.000000000]: Primitive Action Manager Map loaded
[ INFO] [1444590867.147408252, 97.000000000]: PN_Id: 0 PA_ID: 1 NAME: go2Elevator
[ INFO] [1444590867.147444731, 97.000000000]: PN_Id: 1 PA_ID: 2 NAME: go2LRM
[ INFO] [1444590875.882455039, 105.800000000]: Executing petri net - going to fire transitio
n: 1
[ INFO] [1444590916.763632415, 170.800000000]: Executing petri net - going to fire transitio
n: 0
[ INFO] [1444590925.996507350, 239.400000000]: Executing petri net - going to fire transitio
n: 1
[ INFO] [1444590935.632873658, 311.000000000]: Executing petri net - going to fire transitio
n: 0
[ INFO] [1444590944.731810374, 378.600000000]: Executing petri net - going to fire transitio
n: 1
[ INFO] [1444590954.095997504, 448.200000000]: Executing petri net - going to fire transitio
n: 0
```

Figure 4.7: PetriNetExecution package console output result from the execution of the task from Figure 4.5.



```
psantos@psantos:~$ rosrun primitive_action_manager primitive_action_manager
[ INFO] [1444590867.149068760, 97.000000000]: START Go2LRM!
[ INFO] [1444590875.882777626, 105.800000000]: STOP Go2LRM
[ INFO] [1444590875.886588009, 105.800000000]: START Go2Elevator!
[ INFO] [1444590916.764108562, 170.800000000]: STOP Go2Elevator
[ INFO] [1444590916.767592589, 170.800000000]: START Go2LRM!
[ INFO] [1444590925.996713440, 239.400000000]: STOP Go2LRM
[ INFO] [1444590925.998917491, 239.400000000]: START Go2Elevator!
[ INFO] [1444590935.633310440, 311.000000000]: STOP Go2Elevator
[ INFO] [1444590935.636148479, 311.000000000]: START Go2LRM!
[ INFO] [1444590944.732104321, 378.600000000]: STOP Go2LRM
[ INFO] [1444590944.737058726, 378.600000000]: START Go2Elevator!
[ INFO] [1444590954.096326689, 448.200000000]: STOP Go2Elevator
[ INFO] [1444590954.098406282, 448.200000000]: START Go2LRM!
```

Figure 4.8: PrimitiveActionManager package console output result from the execution of the task from Figure 4.5.

4.2.3 Patrol with two conflict situations

The second test was to verify the execution of a more complicated patrol task, which includes adding conflict situations into the Marked Petri net model and verify the correct execution of the task. The task is composed by four different actions that are executed individually, and as before, the actions which the robot should perform change according to the predicates that evaluate if the robot arrived to the goal position of the respective action. The graphical representation of Marked Petri net model of the this patrol task is depicted in Figure 4.9.

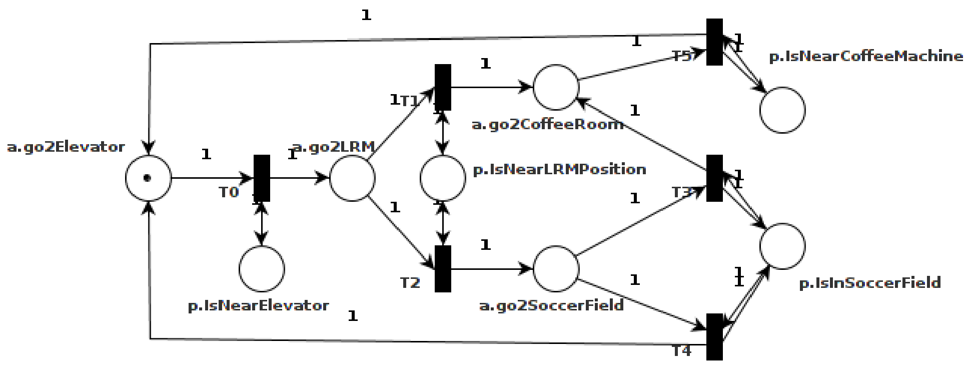


Figure 4.9: Non-deterministic patrol task between four positions.

When executing the framework with the given task, it was possible to confirm that the robot does not execute the exact same sequence of actions every time. As before, Figure 4.10 displays the robot paths done while executing the task.

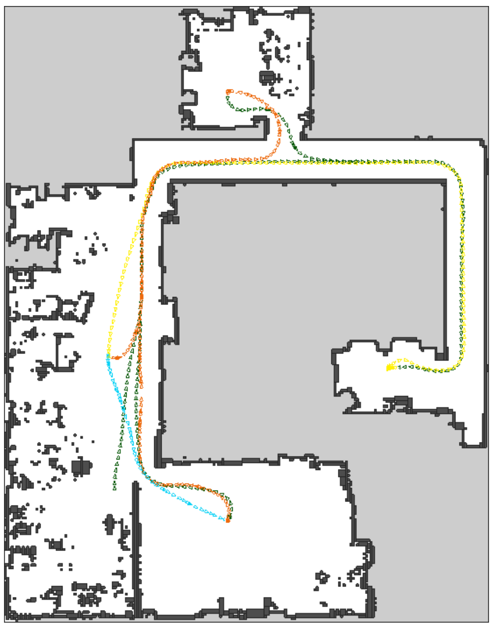


Figure 4.10: Representation of the robot path during the execution of the task from Figure 4.9. As before Yellow arrows represent the go2LRM action path, green arrows the go2Elevator action path, blue arrows the go2SoccerField action path and orange arrows the go2CoffeeRoom action path.

In order to be easier to verify the randomness of the method that is used to choose a transition when multiple transitions are active at the same time, a new task was designed using only a dummy regular place connected to all possible navigation actions. The graphical representation of this task is depicted in Figure 4.11, and a snippet of console output from the execution is displayed in Figure 4.12.

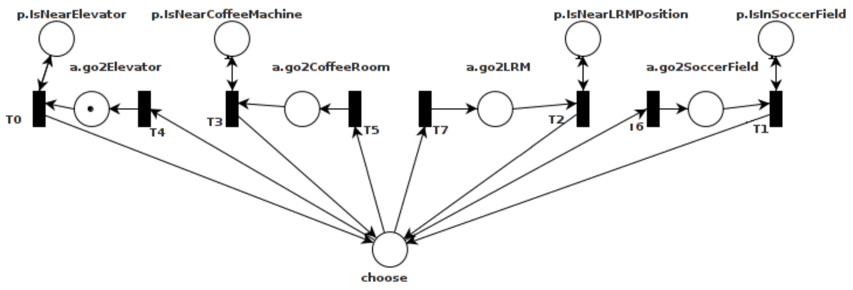
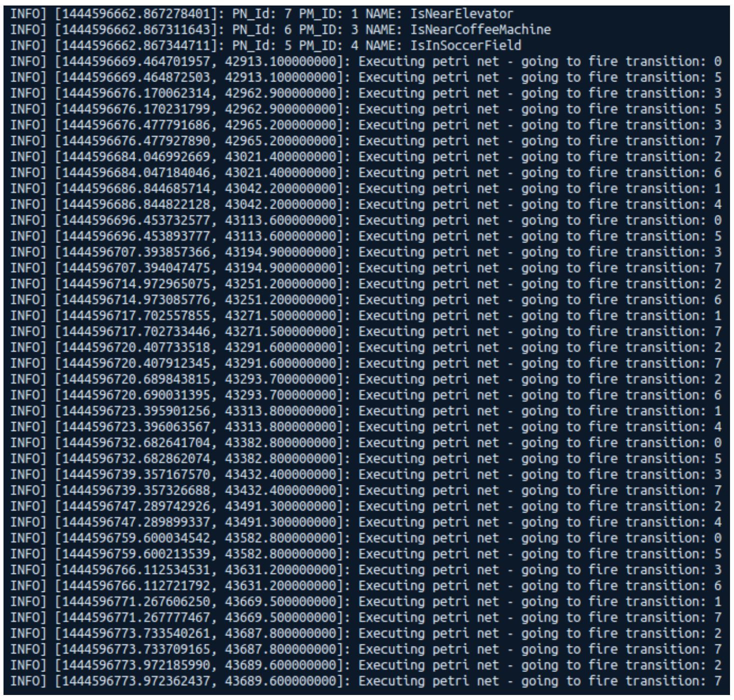


Figure 4.11: Non-deterministic task.



```
INFO] [1444596662.867278401]: PN_Id: 7 PM_ID: 1 NAME: IsNearElevator
INFO] [1444596662.867311643]: PN_Id: 6 PM_ID: 3 NAME: IsNearCoffeeMachine
INFO] [1444596662.867344711]: PN_Id: 5 PM_ID: 4 NAME: IsInSoccerField
INFO] [1444596669.464701957, 42913.100000000]: Executing petri net - going to fire transition: 0
INFO] [1444596669.464872503, 42913.100000000]: Executing petri net - going to fire transition: 5
INFO] [1444596676.170062314, 42962.900000000]: Executing petri net - going to fire transition: 3
INFO] [1444596676.170231799, 42962.900000000]: Executing petri net - going to fire transition: 5
INFO] [1444596676.477791686, 42965.200000000]: Executing petri net - going to fire transition: 3
INFO] [1444596676.477927890, 42965.200000000]: Executing petri net - going to fire transition: 7
INFO] [1444596684.046992669, 43021.400000000]: Executing petri net - going to fire transition: 2
INFO] [1444596684.047184046, 43021.400000000]: Executing petri net - going to fire transition: 6
INFO] [1444596686.844685714, 43042.200000000]: Executing petri net - going to fire transition: 1
INFO] [1444596686.844822128, 43042.200000000]: Executing petri net - going to fire transition: 4
INFO] [1444596696.453732577, 43113.600000000]: Executing petri net - going to fire transition: 0
INFO] [1444596696.453893777, 43113.600000000]: Executing petri net - going to fire transition: 5
INFO] [1444596707.393857366, 43194.900000000]: Executing petri net - going to fire transition: 3
INFO] [1444596707.394047475, 43194.900000000]: Executing petri net - going to fire transition: 7
INFO] [1444596714.972965075, 43251.200000000]: Executing petri net - going to fire transition: 2
INFO] [1444596714.973085776, 43251.200000000]: Executing petri net - going to fire transition: 6
INFO] [1444596717.702557855, 43271.500000000]: Executing petri net - going to fire transition: 1
INFO] [1444596717.702733446, 43271.500000000]: Executing petri net - going to fire transition: 7
INFO] [1444596720.407733518, 43291.600000000]: Executing petri net - going to fire transition: 2
INFO] [1444596720.407912345, 43291.600000000]: Executing petri net - going to fire transition: 7
INFO] [1444596720.689843815, 43293.700000000]: Executing petri net - going to fire transition: 2
INFO] [1444596720.690031395, 43293.700000000]: Executing petri net - going to fire transition: 6
INFO] [1444596723.395901256, 43313.800000000]: Executing petri net - going to fire transition: 1
INFO] [1444596723.396063567, 43313.800000000]: Executing petri net - going to fire transition: 4
INFO] [1444596732.682641704, 43382.800000000]: Executing petri net - going to fire transition: 0
INFO] [1444596732.682862074, 43382.800000000]: Executing petri net - going to fire transition: 5
INFO] [1444596739.357167570, 43432.400000000]: Executing petri net - going to fire transition: 3
INFO] [1444596739.357326688, 43432.400000000]: Executing petri net - going to fire transition: 7
INFO] [1444596747.289742926, 43491.300000000]: Executing petri net - going to fire transition: 2
INFO] [1444596747.289899337, 43491.300000000]: Executing petri net - going to fire transition: 4
INFO] [1444596759.600034542, 43582.800000000]: Executing petri net - going to fire transition: 0
INFO] [1444596759.600213539, 43582.800000000]: Executing petri net - going to fire transition: 5
INFO] [1444596766.112534531, 43631.200000000]: Executing petri net - going to fire transition: 3
INFO] [1444596766.112721792, 43631.200000000]: Executing petri net - going to fire transition: 6
INFO] [1444596771.267606250, 43669.500000000]: Executing petri net - going to fire transition: 1
INFO] [1444596771.267777467, 43669.500000000]: Executing petri net - going to fire transition: 7
INFO] [1444596773.733540261, 43687.800000000]: Executing petri net - going to fire transition: 2
INFO] [1444596773.733709165, 43687.800000000]: Executing petri net - going to fire transition: 7
INFO] [1444596773.972185990, 43689.600000000]: Executing petri net - going to fire transition: 2
INFO] [1444596773.972362437, 43689.600000000]: Executing petri net - going to fire transition: 7
```

Figure 4.12: PetriNetExecution package console output result from the execution of the conflict-transition task from Figure 4.11.

4.2.4 Patrol with concurrent actions

Additionally the framework was tested using a patrol task that has concurrent actions. For this case, a new task containing text-to-speech and navigation actions was created, with action *ttsCurrPosition* having a capacity of one defined, and where both the predicate *IsNearStairs* and is opposite *NOT_IsNearStairs* are used in order to avoid a model that would generate infinite tokens. The graphical representation of the achieved patrol task is depicted in Figure 4.13.

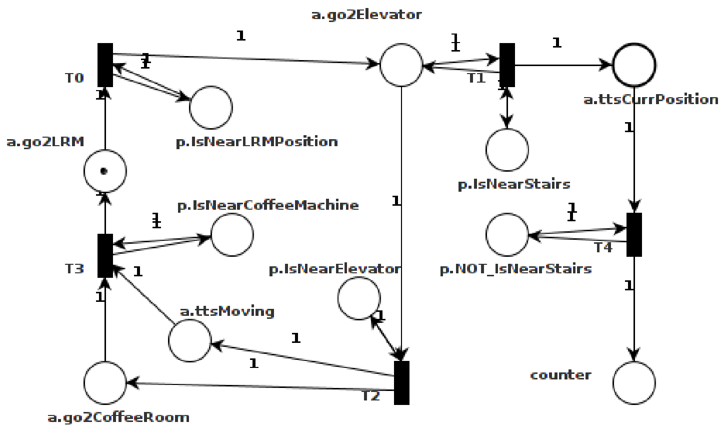


Figure 4.13: Patrol task that contains concurrent actions.

Once more, Figure 4.14 displays the path followed by the robot while executing the task and Figures 4.15 and 4.16 display a snippet of console output. Notice that, while the action *go2Elevator* was already being executed when *ttsCurrPosition* starts, the actions *go2CoffeeRoom* and *ttsMoving* start at the same time.

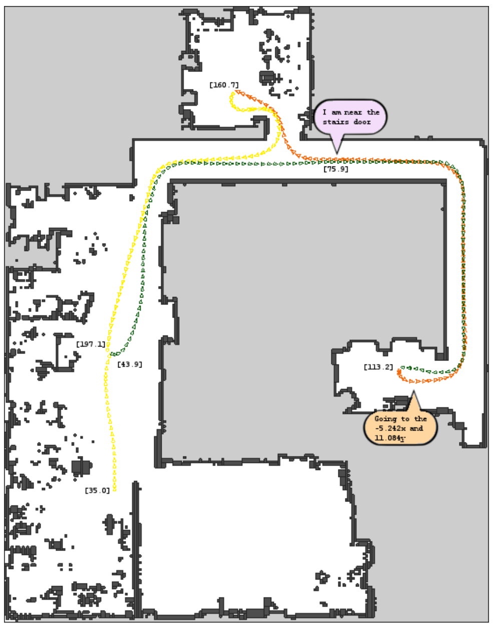
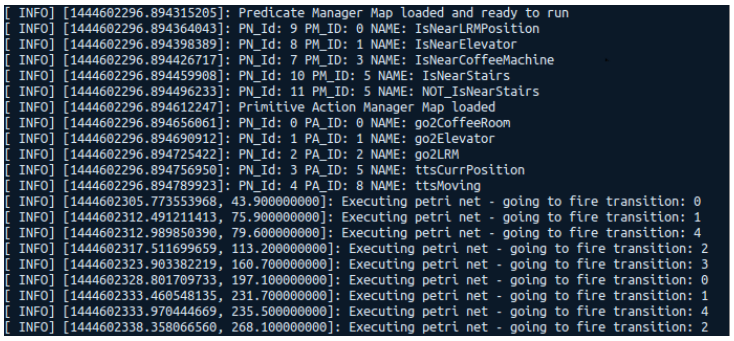


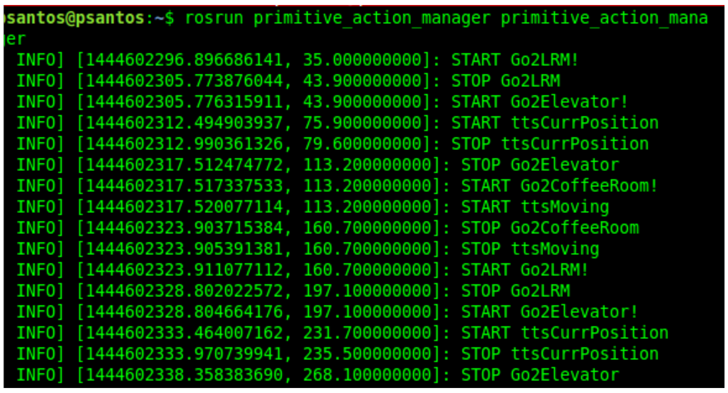
Figure 4.14: Representation of the robot path during the execution of the task from Figure 4.13. While the arrows correspond to the different navigation actions paths, the text balloons are used to display the places where text-to-speech actions occurred.

Additionally, the task was changed by modifying the place *ttsMoving* to *go2SoccerField*, which intro-



```
[ INFO] [1444602296.894315205]: Predicate Manager Map loaded and ready to run
[ INFO] [1444602296.894364043]: PN_Id: 9 PM_ID: 0 NAME: IsNearLRMPosition
[ INFO] [1444602296.894398389]: PN_Id: 8 PM_ID: 1 NAME: IsNearElevator
[ INFO] [1444602296.894426717]: PN_Id: 7 PM_ID: 3 NAME: IsNearCoffeeMachine
[ INFO] [1444602296.894459908]: PN_Id: 10 PM_ID: 5 NAME: IsNearStairs
[ INFO] [1444602296.894496233]: PN_Id: 11 PM_ID: 5 NAME: NOT_IsNearStairs
[ INFO] [1444602296.894612247]: Primitive Action Manager Map loaded
[ INFO] [1444602296.894656061]: PN_Id: 0 PA_ID: 0 NAME: go2CoffeeRoom
[ INFO] [1444602296.894690912]: PN_Id: 1 PA_ID: 1 NAME: go2Elevator
[ INFO] [1444602296.894725422]: PN_Id: 2 PA_ID: 2 NAME: go2LRM
[ INFO] [1444602296.894756950]: PN_Id: 3 PA_ID: 5 NAME: ttsCurrPosition
[ INFO] [1444602296.894789923]: PN_Id: 4 PA_ID: 8 NAME: ttsMoving
[ INFO] [1444602305.773553968, 43.900000000]: Executing petri net - going to fire transition: 0
[ INFO] [1444602312.491211413, 75.900000000]: Executing petri net - going to fire transition: 1
[ INFO] [1444602312.989850390, 79.600000000]: Executing petri net - going to fire transition: 4
[ INFO] [1444602317.511699659, 113.200000000]: Executing petri net - going to fire transition: 2
[ INFO] [1444602323.903382219, 160.700000000]: Executing petri net - going to fire transition: 3
[ INFO] [1444602328.801709733, 197.100000000]: Executing petri net - going to fire transition: 0
[ INFO] [1444602333.460548135, 231.700000000]: Executing petri net - going to fire transition: 1
[ INFO] [1444602333.970444669, 235.500000000]: Executing petri net - going to fire transition: 4
[ INFO] [1444602338.358066560, 268.100000000]: Executing petri net - going to fire transition: 2
```

Figure 4.15: PetriNetExecution package console output result from the execution of the task with concurrent actions.



```
santos@psantos:~$ rosrun primitive_action_manager primitive_action_mana
er
 INFO] [1444602296.896686141, 35.000000000]: START Go2LRM!
 INFO] [1444602305.773876044, 43.900000000]: STOP Go2LRM
 INFO] [1444602305.776315911, 43.900000000]: START Go2Elevator!
 INFO] [1444602312.494903937, 75.900000000]: START ttsCurrPosition
 INFO] [1444602312.990361326, 79.600000000]: STOP ttsCurrPosition
 INFO] [1444602317.512474772, 113.200000000]: STOP Go2Elevator
 INFO] [1444602317.517337533, 113.200000000]: START Go2CoffeeRoom!
 INFO] [1444602317.520077114, 113.200000000]: START ttsMoving
 INFO] [1444602323.903715384, 160.700000000]: STOP Go2CoffeeRoom
 INFO] [1444602323.905391381, 160.700000000]: STOP ttsMoving
 INFO] [1444602323.911077112, 160.700000000]: START Go2LRM!
 INFO] [1444602328.802022572, 197.100000000]: STOP Go2LRM
 INFO] [1444602328.804664176, 197.100000000]: START Go2Elevator!
 INFO] [1444602333.464007162, 231.700000000]: START ttsCurrPosition
 INFO] [1444602333.970739941, 235.500000000]: STOP ttsCurrPosition
 INFO] [1444602338.358383690, 268.100000000]: STOP Go2Elevator
```

Figure 4.16: PrimitiveActionManager package console output result from the execution of the task with concurrent actions.

duces a mistake in the model where two navigation actions are supposed to be executed at the same time, Figure 4.17. Afterwards the framework was again executed and the obtained result is displayed in Figures 4.18, 4.19 and 4.20.

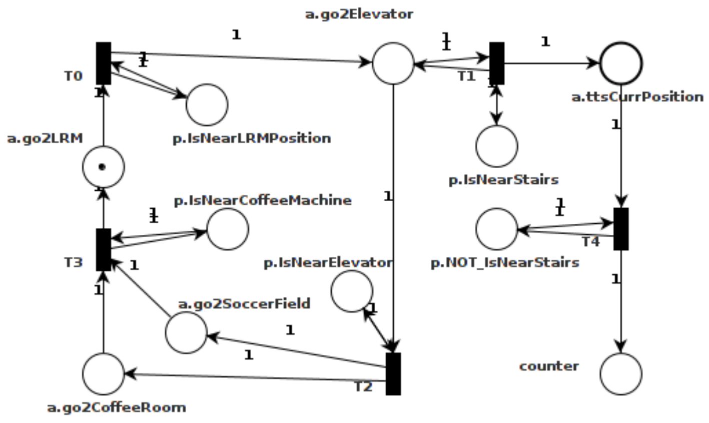


Figure 4.17: New Patrol task with concurrent navigation actions

As expected, and since the actions use the same physical actuator and are publishing contradictory informations, t he task is performed in a erroneous way. Currently, the problem of having multiple actions running that are trying to use the same resource is only possible to be identified beforehand by the designer. However, a possible solution based on classes of actions is proposed in the future work Section 5.2.

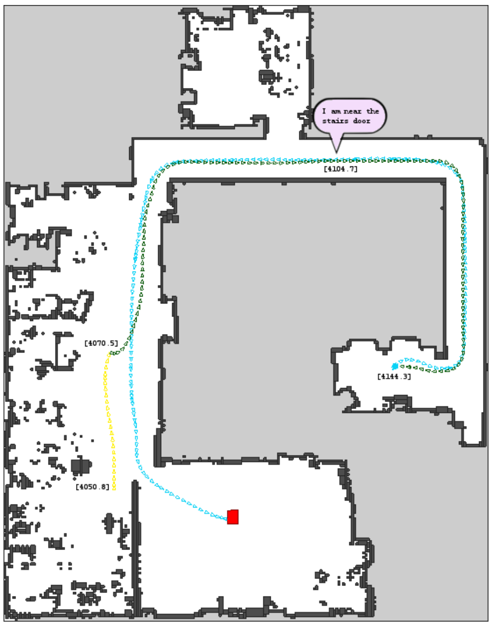
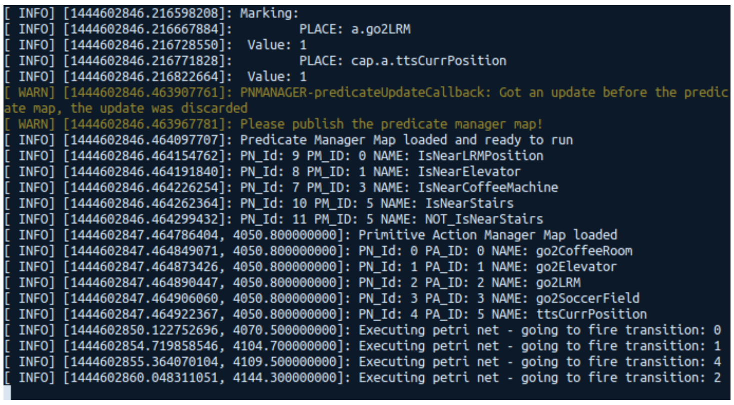
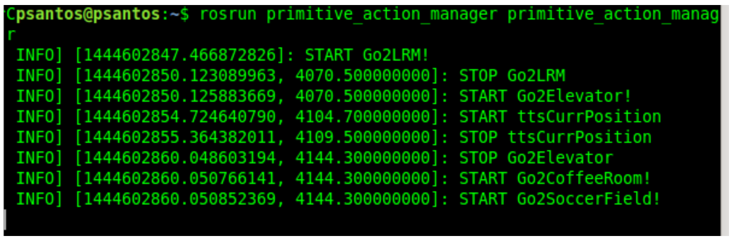


Figure 4.18: Representation of the robot path and the final position during the execution of the task with the concurrent navigation actions.



```
[ INFO] [1444602846.216598208]: Marking:
[ INFO] [1444602846.216667884]:         PLACE: a.go2LRM
[ INFO] [1444602846.216728550]:  Value: 1
[ INFO] [1444602846.216771828]:         PLACE: cap.a.ttsCurrPosition
[ INFO] [1444602846.216822664]:  Value: 1
[ WARN] [1444602846.463907761]: PNMANAGER-predicateUpdateCallback: Got an update before the predic
ate map, the update was discarded
[ WARN] [1444602846.463967781]: Please publish the predicate manager map!
[ INFO] [1444602846.464097707]: Predicate Manager Map loaded and ready to run
[ INFO] [1444602846.464154762]: PN_Id: 9 PM_ID: 0 NAME: IsNearLRMPosition
[ INFO] [1444602846.464191840]: PN_Id: 8 PM_ID: 1 NAME: IsNearElevator
[ INFO] [1444602846.464226254]: PN_Id: 7 PM_ID: 3 NAME: IsNearCoffeeMachine
[ INFO] [1444602846.464262364]: PN_Id: 10 PM_ID: 5 NAME: IsNearStairs
[ INFO] [1444602846.464299432]: PN_Id: 11 PM_ID: 5 NAME: NOT_IsNearStairs
[ INFO] [1444602847.464786404, 4050.800000000]: Primitive Action Manager Map loaded
[ INFO] [1444602847.464849071, 4050.800000000]: PN_Id: 0 PA_ID: 0 NAME: go2CoffeeRoom
[ INFO] [1444602847.464873426, 4050.800000000]: PN_Id: 1 PA_ID: 1 NAME: go2Elevator
[ INFO] [1444602847.464890447, 4050.800000000]: PN_Id: 2 PA_ID: 2 NAME: go2LRM
[ INFO] [1444602847.464906060, 4050.800000000]: PN_Id: 3 PA_ID: 3 NAME: go2SoccerField
[ INFO] [1444602847.464922367, 4050.800000000]: PN_Id: 4 PA_ID: 5 NAME: ttsCurrPosition
[ INFO] [1444602850.122752696, 4070.500000000]: Executing petri net - going to fire transition: 0
[ INFO] [1444602854.719858546, 4104.700000000]: Executing petri net - going to fire transition: 1
[ INFO] [1444602855.364070104, 4109.500000000]: Executing petri net - going to fire transition: 4
[ INFO] [1444602860.048311051, 4144.300000000]: Executing petri net - going to fire transition: 2
```

Figure 4.19: Stalled PetriNetExecution package, result from the execution of 4.17



```
Cpsantos@psantos:~$ rosrun primitive_action_manager primitive_action_manag
r
 INFO] [1444602847.466872826]: START Go2LRM!
 INFO] [1444602850.123089963, 4070.500000000]: STOP Go2LRM
 INFO] [1444602850.125883669, 4070.500000000]: START Go2Elevator!
 INFO] [1444602854.724640790, 4104.700000000]: START ttsCurrPosition
 INFO] [1444602855.364382011, 4109.500000000]: STOP ttsCurrPosition
 INFO] [1444602860.048603194, 4144.300000000]: STOP Go2Elevator
 INFO] [1444602860.050766141, 4144.300000000]: START Go2CoffeeRoom!
 INFO] [1444602860.050852369, 4144.300000000]: START Go2SoccerField!
```

Figure 4.20: Stalled PrimitiveActionManager package, result from the execution of 4.17

4.2.5 Hierarchical Task

After the evaluation of the framework with different task models that only used actions, predicates and regular places, a simple hierarchical patrol task was created, where top level evaluates if an user requests the presence of the robot in the soccer field or if the robot should perform the simple sequential patrol task from Figure 4.5 using it as a task place, as it is depicted in Figure 4.21.

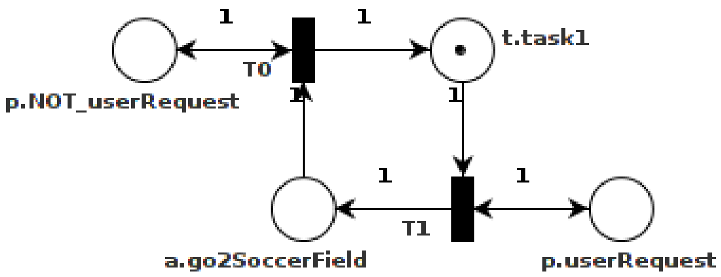


Figure 4.21: Hierarchical Task, where the robot executes the task1, simple sequential patrol from Figure 4.5, or, if requested, goes to the SoccerField.

A new predicate *userRequest* was added to the PredicateManager implementation to check if an user has requested the presence of the robot in the soccer field area, being true if a request has arrived and false otherwise. The implementation of the predicate was done using a subscriber, which receives a message that triggers the predicate value every time an *YES* or *NO* message is published. The execution of the task is displayed in Figure 4.22, while the respective console outputs are depicted in Figure 4.23.

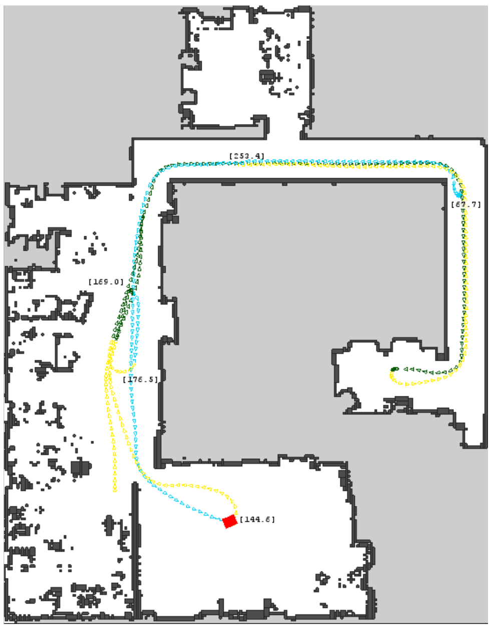
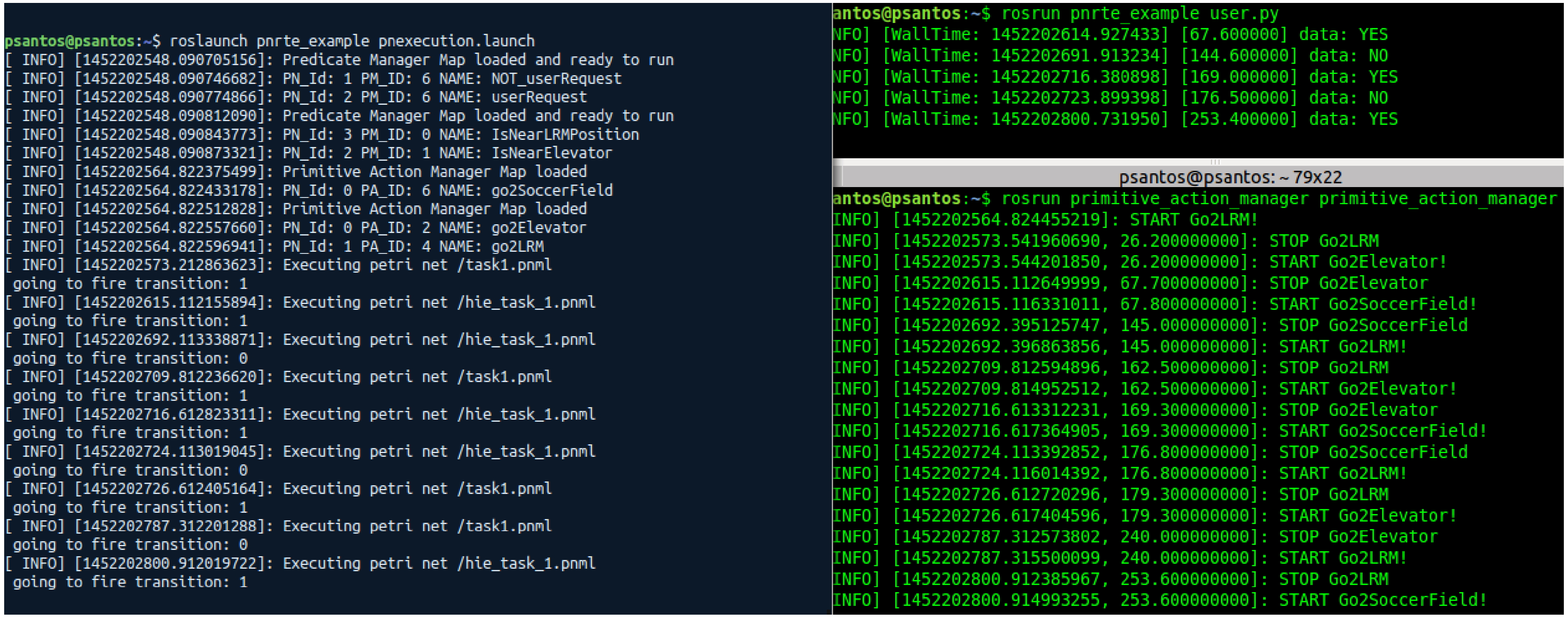


Figure 4.22: Representation of the robot path while executing the hierarchical task



```
psantos@psantos:~$ roslaunch pnrte_example pnexecution.launch
[ INFO] [1452202548.090705156]: Predicate Manager Map loaded and ready to run
[ INFO] [1452202548.090746682]: PN_Id: 1 PM_ID: 6 NAME: NOT_userRequest
[ INFO] [1452202548.090774866]: PN_Id: 2 PM_ID: 6 NAME: userRequest
[ INFO] [1452202548.090812090]: Predicate Manager Map loaded and ready to run
[ INFO] [1452202548.090843773]: PN_Id: 3 PM_ID: 0 NAME: IsNearLRMPosition
[ INFO] [1452202548.090873321]: PN_Id: 2 PM_ID: 1 NAME: IsNearElevator
[ INFO] [1452202564.822375499]: Primitive Action Manager Map loaded
[ INFO] [1452202564.822433178]: PN_Id: 0 PA_ID: 6 NAME: go2SoccerField
[ INFO] [1452202564.822512828]: Primitive Action Manager Map loaded
[ INFO] [1452202564.822557660]: PN_Id: 0 PA_ID: 2 NAME: go2Elevator
[ INFO] [1452202564.822596941]: PN_Id: 1 PA_ID: 4 NAME: go2LRM
[ INFO] [1452202573.212863623]: Executing petri net /task1.pnml
 going to fire transition: 1
[ INFO] [1452202615.112155894]: Executing petri net /hie_task_1.pnml
 going to fire transition: 1
[ INFO] [1452202692.113338871]: Executing petri net /hie_task_1.pnml
 going to fire transition: 0
[ INFO] [1452202709.812236620]: Executing petri net /task1.pnml
 going to fire transition: 1
[ INFO] [1452202716.612823311]: Executing petri net /hie_task_1.pnml
 going to fire transition: 1
[ INFO] [1452202724.113019045]: Executing petri net /hie_task_1.pnml
 going to fire transition: 0
[ INFO] [1452202726.612405164]: Executing petri net /task1.pnml
 going to fire transition: 1
[ INFO] [1452202787.312201288]: Executing petri net /task1.pnml
 going to fire transition: 0
[ INFO] [1452202800.912019722]: Executing petri net /hie_task_1.pnml
 going to fire transition: 1
```

```
antos@psantos:~$ rosrun pnrte_example user.py
NFO] [WallTime: 1452202614.927433] [67.600000] data: YES
NFO] [WallTime: 1452202691.913234] [144.600000] data: NO
NFO] [WallTime: 1452202716.380898] [169.000000] data: YES
NFO] [WallTime: 1452202723.899398] [176.500000] data: NO
NFO] [WallTime: 1452202800.731950] [253.400000] data: YES
```

psantos@psantos: ~ 79x22

```
antos@psantos:~$ rosrun primitive_action_manager primitive_action_manager
INFO] [1452202564.824455219]: START Go2LRM!
INFO] [1452202573.541960690, 26.200000000]: STOP Go2LRM
INFO] [1452202573.544201850, 26.200000000]: START Go2Elevator!
INFO] [1452202615.112649999, 67.700000000]: STOP Go2Elevator
INFO] [1452202615.116331011, 67.800000000]: START Go2SoccerField!
INFO] [1452202692.395125747, 145.000000000]: STOP Go2SoccerField
INFO] [1452202692.396863856, 145.000000000]: START Go2LRM!
INFO] [1452202709.812594896, 162.500000000]: STOP Go2LRM
INFO] [1452202709.814952512, 162.500000000]: START Go2Elevator!
INFO] [1452202716.613312231, 169.300000000]: STOP Go2Elevator
INFO] [1452202716.617364905, 169.300000000]: START Go2SoccerField!
INFO] [1452202724.113392852, 176.800000000]: STOP Go2SoccerField
INFO] [1452202724.116014392, 176.800000000]: START Go2LRM!
INFO] [1452202726.612720296, 179.300000000]: STOP Go2LRM
INFO] [1452202726.617404596, 179.300000000]: START Go2Elevator!
INFO] [1452202787.312573802, 240.000000000]: STOP Go2Elevator
INFO] [1452202787.315500099, 240.000000000]: START Go2LRM!
INFO] [1452202800.912385967, 253.600000000]: STOP Go2LRM
INFO] [1452202800.914993255, 253.600000000]: START Go2SoccerField!
```

Figure 4.23: Console output of both PetriNetExecution and PrimitiveActionManager packages for the execution of the hierarchical task 4.21.

4.2.6 Hierarchical Patrol where a loop situation is detected

A hierarchical patrol task was designed, based on a simplified version of the task depicted in Figure 4.9. The hierarchical patrol is composed by the four tasks depicted in Figures 4.24, 4.25, 4.26 and 4.27.

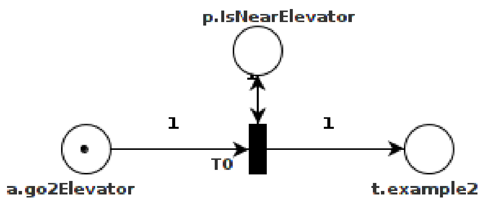


Figure 4.24: The top level task from the hierarchical patrol, task - example1

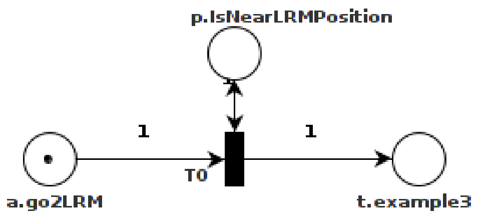


Figure 4.25: The second level task from the hierarchical patrol, task - example2

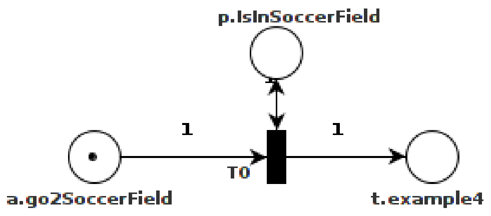


Figure 4.26: The third level task from the hierarchical patrol, task - example3

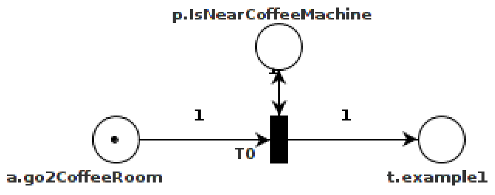
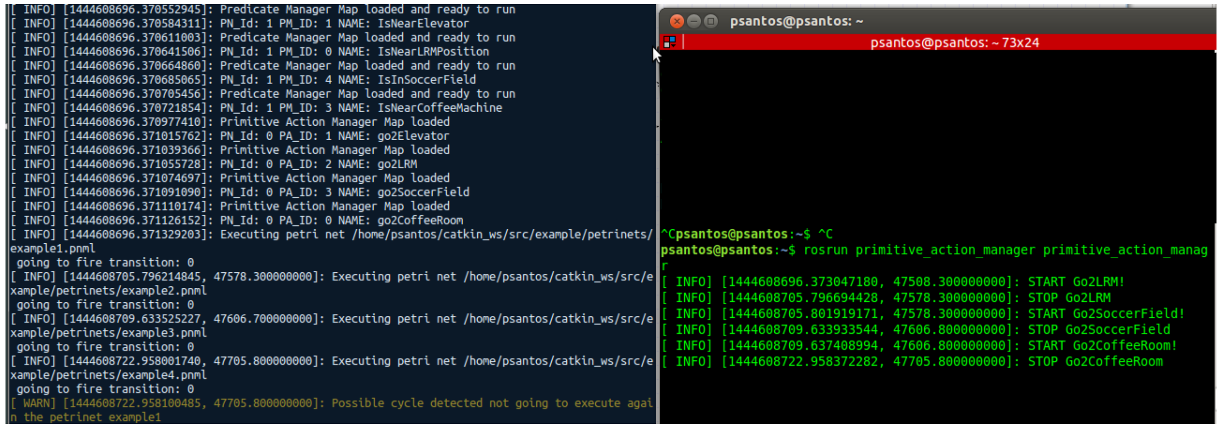


Figure 4.27: The last level task from the hierarchical patrol, task - example4

It is important to notice that this design is a bad implementation of a hierarchical task, not only it gets stalled at the end of the deepest task, depicted in Figure 4.28, but it also does not make much sense since every Marked Petri net model from each level will only evolve once, being stalled from that moment on. However, the design is valid from a syntax point of view and is useful to identify how cycles in execution of a hierarchical net are detected. As expected, the framework is able to execute each level of the hierarchical task patrol, and since tasks that were already called for execution and were still running were tracked, an alert is printed in order to let the user know that a task that was already being executed was called. For the given example this means that when the execution reaches the last level task, Figure 4.27, and as soon as the transition $T0$ fires and, the new marking is reached, where the place *t.example1* has a token, a loop in hierarchical Petri nets occurs.



```
[ INFO] [1444608696.370552945]: Predicate Manager Map loaded and ready to run
[ INFO] [1444608696.370584311]: PN_Id: 1 PM_ID: 1 NAME: IsNearElevator
[ INFO] [1444608696.370611003]: Predicate Manager Map loaded and ready to run
[ INFO] [1444608696.370641506]: PN_Id: 1 PM_ID: 0 NAME: IsNearLRMPosition
[ INFO] [1444608696.370664860]: Predicate Manager Map loaded and ready to run
[ INFO] [1444608696.370685065]: PN_Id: 1 PM_ID: 4 NAME: IsInSoccerField
[ INFO] [1444608696.370705456]: Predicate Manager Map loaded and ready to run
[ INFO] [1444608696.370721854]: PN_Id: 1 PM_ID: 3 NAME: IsNearCoffeeMachine
[ INFO] [1444608696.370977410]: Primitive Action Manager Map loaded
[ INFO] [1444608696.371015762]: PN_Id: 0 PA_ID: 1 NAME: go2Elevator
[ INFO] [1444608696.371039366]: Primitive Action Manager Map loaded
[ INFO] [1444608696.371055728]: PN_Id: 0 PA_ID: 2 NAME: go2LRM
[ INFO] [1444608696.371074697]: Primitive Action Manager Map loaded
[ INFO] [1444608696.371091090]: PN_Id: 0 PA_ID: 3 NAME: go2SoccerField
[ INFO] [1444608696.371110174]: Primitive Action Manager Map loaded
[ INFO] [1444608696.371126152]: PN_Id: 0 PA_ID: 0 NAME: go2CoffeeRoom
[ INFO] [1444608696.371329203]: Executing petri net /home/psantos/catkin_ws/src/example/petrinets/
example1.pnml
 going to fire transition: 0
[ INFO] [1444608705.796214845, 47578.300000000]: Executing petri net /home/psantos/catkin_ws/src/e
xample/petrinets/example2.pnml
 going to fire transition: 0
[ INFO] [1444608709.633525227, 47606.700000000]: Executing petri net /home/psantos/catkin_ws/src/e
xample/petrinets/example3.pnml
 going to fire transition: 0
[ INFO] [1444608722.958001740, 47705.800000000]: Executing petri net /home/psantos/catkin_ws/src/e
xample/petrinets/example4.pnml
 going to fire transition: 0
[ WARN] [1444608722.958100485, 47705.800000000]: Possible cycle detected not going to execute agai
n the petrinet example1
```

```
^Cpsantos@psantos:~$ ^C
psantos@psantos:~$ rosrun primitive_action_manager primitive_action_manag
r
[ INFO] [1444608696.373047180, 47508.300000000]: START Go2LRM!
[ INFO] [1444608705.796694428, 47578.300000000]: STOP Go2LRM
[ INFO] [1444608705.801919171, 47578.300000000]: START Go2SoccerField!
[ INFO] [1444608709.633933544, 47606.800000000]: STOP Go2SoccerField
[ INFO] [1444608709.637408994, 47606.800000000]: START Go2CoffeeRoom!
[ INFO] [1444608722.958372282, 47705.800000000]: STOP Go2CoffeeRoom
```

Figure 4.28: Console output for the hierarchical patrol task where a loop was found

4.2.7 Real robot scenario

After the validation of the implemented framework using simulated examples, the framework was tested using a real robot in the eight floor of the North Tower. The robot used was a MBOT robot from the MOnarCH project [3], a four wheel omni-directional drive robot, equal to the one displayed in Figure 4.29. An updated version of the map used in simulation was used, in order to cope with the more recent layout of the space, as depicted in Figure 4.30.

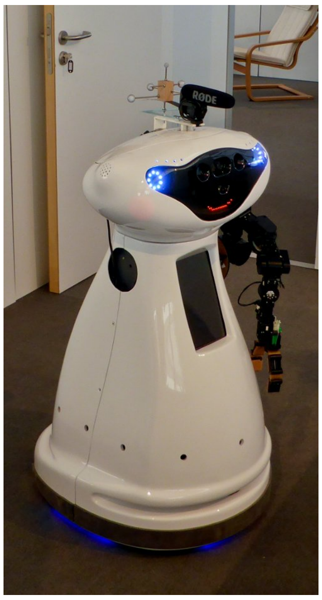


Figure 4.29: The MBOT robot from the MOnarCH project used for the tests.

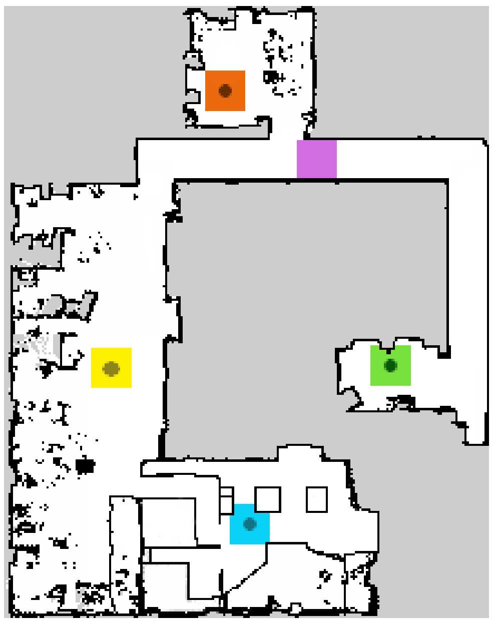


Figure 4.30: The updated map used for the real robot tests.

Additionally and to properly run the tests, the navigation computer containing ROS packages for the navigation and localization purposes of the robot was used, as also as an external computer connected through wireless to the robot in order to visualize, debug data or run specific components of the framework when required.

The framework was installed in the robot and the previous described scenarios were used. The robot was able to perform the tests correctly from the framework standpoint, obtaining differences in the randomness choice of actions for some tasks, the execution time of the navigation actions and the trajectory of the robot. Such differences were expected since different robots were considered between the real and the simulated tests, and the random nature of the firing rule when multiple transitions are enabled.

Lastly, a new task was designed and tested. The task consisted in a patrol between four positions, but instead of the framework being executed completely in the robot, it was executed from an external computer and the robot was only used to perform the actions and sense the environment. The Marked Petri Net of the task is displayed in Figure 4.31, while the actions and predicates were implemented in a similar fashion to the ones described for the simulated scenario and are relative to positions of a smaller area of the previous depicted map. The "new" map and positions are depicted in Figure 4.32.

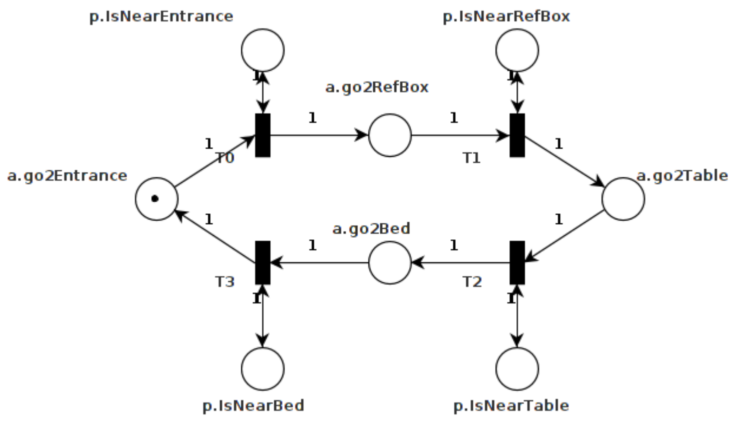


Figure 4.31: Marked Petri net model of the new Patrol task.

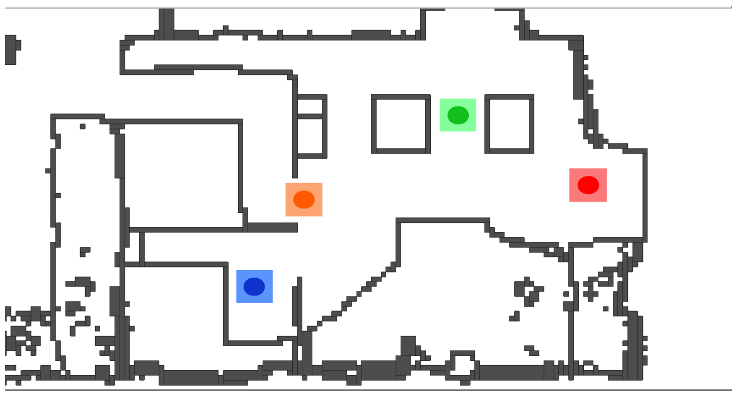


Figure 4.32: Zoomed area of the map with the new navigation goal positions and areas that trigger changes in the predicates for the new task.

- Orange → Go2Entrance goal position and IsNearEntrance;
- Red → Go2RefBox goal position and IsNearRefBox

- Green → Go2Table goal position and IsNearTable
- Blue → Go2Bed goal position and IsNearBed

As expected the framework was executed properly and the robot path for this task is represented in Figure 4.33.

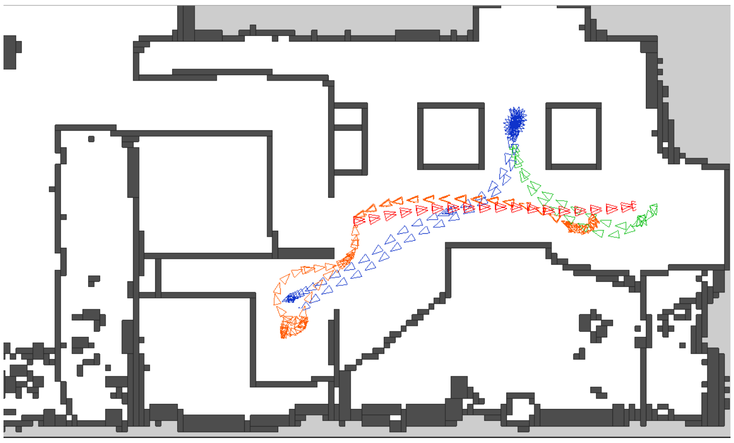


Figure 4.33: Representation of the robot path while executing the new patrol task.

Chapter 5

Conclusions

Concluding this thesis, a critical review of the work developed is presented, as well as the achievements and contributions. Finally, future features to implement in the framework and future work directives are identified.

5.1 Achievements

This thesis proposes a framework for the execution of robot task described by a Marked Petri net [22] for the Robot Operating System, called Petri Net Robotic Task Execution, PN-RTE. The framework is composed by three packages, PetriNetExecution, PrimitiveActionManager and PredicateManager, where the two first packages were developed and implemented from scratch and the last one was integrated.

There are several nuances of the PN-RTE framework relative to the SMACH. PN-RTE allows to execute a robot task represented by Marked Petri nets which are richer and have an inherent larger modeling power. Also, and as SMACH, PN-RTE allows to execute hierarchical tasks and concurrent actions, however those are extracted directly from the Marked Petri net model instead of having to be manually implemented. While SMACH provides generic classes in order for the user to implement their states with defined outcomes, PN-RTE allows user to create generic actions and predicates that can be used to design a task in any way the user wants, allowing the actions and predicates to be easily reused.

Additionally, PN-RTE provides a modular framework approach where the task executor is clearly detached from the primitive action and the predicate managers, allowing to, not only, separate packages between devices if needed, communicating only the minimal data required to function properly, but also to correct implementation errors in the actions or predicates without being needed to terminate the other packages. Lastly, PN-RTE allows each package to be replaced by user-defined packages, or used in different scopes.

On other hand, relative to PNP-ROS, the PN-RTE framework was completely developed inside the ROS environment, taking advantage of the ROS concepts and ensuring the framework can be compiled and executed for a given system, as long as ROS is compatible as well. Additionally, not only, the formalism defined by [22] to design the task models is less restrictive, but also the fact the formalism is not based on event constrained transitions, which allows to use the Marked Petri net definitions directly for the execution. Furthermore, in both frameworks actions are used-defined, PN-RTE allows for generic actions, which means they can implement actionlib actions as well as the PNP-ROS actions.

The solution achieved fulfills the set of requirements identified and ensures the correct execution of a task as long as the predicates and primitives are correctly implemented and registered.

Finally, the framework was tested in simulated and real scenarios. First by modifying the example scenario from the MDM_Library and using it with several Proof of Concept robot tasks, proving the framework correct execution and demonstrating the principal functionalities and the results from the execution, and lastly using a real robot to perform the previously defined Proof of Concept robot tasks.

5.2 Future Work

Although pnml is the standard for the Petri net description files, several graphical user interfaces tools that are used to create the Petri nets use other syntaxes, develop new parser methods for those is something that should be addressed. Additionally, it would also be useful to visualize graphically the tasks during execution and it would be interesting to develop new features as automatic correction of typing errors on the names of the actions and predicates that prevent a correct match between the packages.

With the current formalism and framework implementation there is no verification on the type of actions that are executing concurrently, which means more than one action could be actuating over the same physical component at the same time leading to execution errors or generating unexpected results, an example of this problem was addressed in Subsection 4.2.4. A possible solution is using classes, or types, of actions using a similar approach to the one present in the framework [22] for differentiate places, using for instance a suffix to represent each action type. Additionally one could implement a method to verify the task for the concurrent action classes before starting executing the task, alerting the user to every possible situation that would generate concurrent actions of the same class.

The last subject is related to the problem of the multi-robot representation. While the software framework proposed can be used for single or multi-robot execution, since one could model communication and synchronism procedures using action and predicates places and running the framework of each robot, the formalism presented in the framework [22] does not allow to represent multiple robots in the same Marked Petri net, which would allow to easily model communication and synchronism for the multi-robot tasks. This problem could possible be solved applying other extensions of Petri nets formalism as, for instance, colored Petri nets.

Bibliography

[1] A. Pennisi, F. Previtali, C. Gennari, D. Bloisi, L. Iocchi, F. Ficarola, A. Vitaletti, and D. Nardi, “Multi-robot surveillance through a distributed sensor network,” in *Cooperative Robots and Sensor Networks 2015* (A. Koubâa and J. M. de Dios, eds.), vol. 604 of *Studies in Computational Intelligence*, pp. 77–98, Springer International Publishing, 2015.

[2] H. Kitano, M. Asada, Y. Kuniyoshi, I. Noda, and E. Osawa, “Robocup: The robot world cup initiative,” 1995.

[3] J. Messias, R. Ventura, P. Lima, J. Sequeira, P. Alvito, C. Marques, and P. Carricço, “A robotic platform for edutainment activities in a pediatric hospital,” in *2014 IEEE International Conference on Autonomous Robot Systems and Competitions (ICARSC)*, 2014.

[4] RoCKIn, “Robot competitions kick innovation in cognitive systems and robotics.” `http://www.rockinrobotchallenge.eu/` (accessed April 14th 2016), 2013.

[5] M. Robinson, C. Collins, P. Leger, W. Kim, J. Carsten, V. Tompkins, A. Trebi-Ollennu, and B. Florow, “Test and validation of the mars science laboratory robotic arm,” in *System of Systems Engineering (SoSE), 2013 8th International Conference on*, pp. 184–189, June 2013.

[6] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. B. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, “ROS: an open-source robot operating system,” in *ICRA Workshop on Open Source Software*, 2009.

[7] P. Santos, “pn-rte.” `https://bitbucket.org/pedropsantos/pn-rte/overview` (accessed April 14th 2016), 2016.

[8] A. Robotics, “Pepper.” `www.aldebaran.com/en/a-robots/who-is-pepper` (accessed April 14th 2016), 2015.

[9] B. F. Robotics, “Buddy.” `http://www.bluefrogrobotics.com/` (accessed April 14th 2016), 2015.

[10] C. G. Cassandras and S. Lafortune, *Introduction to Discrete Event Systems*. Springer Publishing Company, Incorporated, 2nd ed., 2010.

[11] C. A. Petri, “Fundamentals of a theory of asynchronous information flow,” in *IFIP Congress*, pp. 386–390, 1962.

[12] L. Mudrova, M. Chiou, M. Becerra, S. Bastable, J. Smith, E. Bacci, T. Daskova, and A. Hristova, “Birmingham autonomous robotic club (barc) - team description paper - rockin 2015.” `http://lenka-robotics.eu/index.php/robotic-club` (accessed April 14th 2016).

[13] Bio-Robotics, “Pumas@home 2015 team description paper - rockin 2015.” `http://biorobotica.fi-p.unam.mx/` (accessed April 14th 2016).

[14] T. Murata, “Petri nets: Properties, analysis and applications,” *Proceedings of the IEEE*, vol. 77, pp. 541–580, Apr 1989.

[15] H. Costelha, *Robotic Tasks Modelling and Analysis Based on Discrete Event Systems*. PhD thesis, Instituto Superior Técnico, Universidade Técnica de Lisboa, February 2010.

[16] Ziparo, L. Iocchi, , P. L. Pedro, Nardi, and Palamara, “Petri net plans,” *Autonomous Agents and Multi-Agent Systems*, vol. 23, no. 3, pp. 344–383, 2011.

[17] G. Gemignani, F. Riccio, L. Iocchi, and D. Nardi, “Spqr robocup 2014 standard platform league team description paper,” 2014.

[18] J. Messias, A. Ahmad, J. Reis, M. Serafim, and P. Lima, “Socrob 2013 team description paper,” 2013.

[19] J. Bohren and S. Cousins, “The smach high-level executive [ros news],” *Robotics Automation Magazine, IEEE*, vol. 17, pp. 18–20, Dec 2010.

[20] L. Iocchi, “Pnp-ros.” `http://www.dis.uniroma1.it/~iocchi/Didattica/PNP-ROS.pdf` (accessed April 14th 2016), 2013.

[21] V. A. Ziparo, L. Iocchi, D. Nardi, P. F. Palamara, and H. Costelha, “Pnp: A formal model for representation and execution of multi-robot plans,” in *Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems-Volume 1*, 2008.

[22] H. Costelha and P. Lima, “Robot task plan representation by petri nets: modelling,identification, analysis and execution,” *Autonomous Robots*, vol. ”33”, no. ”4”, pp. ”337–360”, 2012.

[23] J. Messias and J. Reis, “Predicate manager.” `http://wiki.ros.org/predicate_manager` (accessed April 14th 2016), 2014.

[24] P. Lima, H. Gracio, V. Veiga, and A. Karlsson, “Petri nets for modeling and coordination of robotic tasks,” in *Systems, Man, and Cybernetics, 1998. 1998 IEEE International Conference on (Volume:1)*, 1998.

[25] M. Diaz, *Petri Nets: Fundamental Models, Verification and Applications*. Wiley-IEEE Press, 2009.

[26] E. Marder-Eppstein and V. Pradeep, “Actionlib ros.” `http://wiki.ros.org/actionlib` (accessed April 14th 2016).

[27] J. Messias, "Markov decision making metapackage." https://www.github.com/larsys/markov_decision_making (accessed April 14th 2016), 2014.

[28] "Ros hydro medusa." http://wiki.ros.org/hydro (accessed April 14th 2016).

[29] B. Schling, *The Boost C++ Libraries*. XML Press, 2011.

[30] "Boost 1.48.0 library." http://www.boost.org/doc/libs/1_48_0/ (accessed April 14th 2016).

[31] ROS, "Ros c++ style." http://wiki.ros.org/CppStyleGuide (accessed April 14th 2016), 2014.

[32] S. Cousins, "Welcome to ros topics [ros topics]," *Robotics Automation Magazine, IEEE*, vol. 17, pp. 13–14, March 2010.

[33] "Ros technical overview." http://wiki.ros.org/ROS/Technical%20Overview (accessed April 14th 2016).

[34] M. Kalicinski and S. Redl, "Boost property tree." http://www.boost.org/doc/libs/1_48_0/doc/html/property_tree.html (accessed April 14th 2016).

[35] N. Akharware, "Pipe2: Platform independent petri net editor," Master's thesis, Imperial College of Science, September 2005.

[36] A. Cuauhtli, "Pnlab." https://github.com/arcra/PNLab (accessed April 14th 2016), 2014.

[37] J. Maurer, "Boost random." http://www.boost.org/doc/libs/1_48_0/doc/html/boost_random.html (accessed April 14th 2016).

[38] M. Matsumoto and T. Nishimura, "Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator," *ACM Trans. Model. Comput. Simul.*, vol. 8, pp. 3–30, Jan. 1998.

[39] S. G. Williamson, "Depth-first search and kuratowski subgraphs," *J. ACM*, vol. 31, pp. 681–693, Sept. 1984.

[40] "Ros timers." http://wiki.ros.org/roscpp/Overview/Timers (accessed April 14th 2016).

[41] J. Messias, *Decision-Making under Uncertainty for Real Robot Teams*. PhD thesis, Instituto Superior Técnico, Universidade de Lisboa, 2014.

[42] A. MobileRobots, "Pioneer3at." http://www.mobilerobots.com/Libraries/Downloads/Pioneer3AT-P3AT-RevA.sflb.ashx (accessed April 14th 2016).

[43] R. Vaughan, "Massively multi-robot simulation in stage," *Swarm Intelligence*, vol. 2, no. 2-4, pp. 189–208, 2008.