INSTITUTO SUPERIOR TÉCNICO
Universidade Técnica de Lisboa

Robotic Tasks Modelling and Analysis Based on Discrete Event Systems

Hugo Filipe Costelha de Castro

Dissertation to obtain the Doctorate Degree in
Electrical and Computer Engineering

February, 2010

Resumo

Esta tese introduz um método de modelação de tarefas robóticas baseada em redes de Petri. Nestes modelos, os lugares das redes de Petri representam acções, tarefas, e predicados estabelecidos pela leitura de sensores e mensagens transmitidas, enquanto os eventos são representados por transições. O método proposto segue um modelo hierárquico que vai desde os modelos de componentes do ambiente ao modelo do plano da tarefa. Estes modelos podem ser usados quer para execução, quer para análise. O modelo do plano da tarefa pode ser executado directamente nos robôs. Para análise, todos os modelos são compostos num único modelo, do qual se extraem propriedades lógicas e de desempenho. São obtidas ambas as propriedades estacionárias e transitórias. Os modelos do ambiente incluem eventos não controláveis, os quais modelam o impacto de fenómenos físicos ou de outros agentes. O uso de modelos de observação permite estudar o impacto de falhas sensoriais no desempenho da execução da tarefa. A introdução de modelos de comunicação, quer para comunicação explícita (e.g., usando a rede sem fios), quer para comunicação implícita (e.g., observando os outros robôs usando visão), permite modelar e analisar tarefas envolvendo a coordenação de dois ou mais robôs. Propõe-se um método de identificação, que permite criar os modelos das acções e do ambiente a partir de dados reais. São apresentados resultados, usando um simulador realista, que demonstram a aplicabilidade do método de modelação proposto.

Palavras-chave

Tarefas Robóticas, Redes de Petri, Modelação, Análise, Identificação, Execução.

Abstract

This thesis introduces a Petri net (PN) based robot task modelling framework. In the models, PN places represent actions, tasks, and predicates set by sensor readings and communicated messages, while transitions represent events. The proposed framework follows a structured hierarchical approach ranging from the environment models to the task plan models. These models can be used both for task execution and analysis. The PN based task plan can be executed directly in the robots. For analysis, all the models are composed into a single PN which is analysed both for logical and (probabilistic) performance properties. Both stationary and transient properties are analysed. Environment models include uncontrollable events which model the world physics and/or other agents impact on the world. Observation models allow determining the impact of observation failures in the task performance. The introduction of communication models, either using explicit (e.g., wireless) or implicit (e.g., vision-based observation of teammates), allows modelling and analysis of multi-robot tasks involving the coordination of two or more robots. An identification method is proposed, which allows creating environment and action models from real data. Results illustrating the methodology are presented for a robotic soccer scenario using a realistic simulator.

Keywords

Robot Tasks, Petri Nets, Modelling, Analysis, Identification, Execution.

Acknowledgements

As this hard journey comes to an end, I must first say “Thank you Fernanda”. All my working nights and weekends did not let you enjoy life as you deserve. It is my long made promise to make up for all that you had to put up with, and is something I look further to do.

To my family, sorry for not being there as much as you deserve, and thank you for the support you gave through these years. To my parents, if it were not for you, for the education you gave me, I would not be where I am today. To my brother, who has been a father recently, I hope we can resume our sport evenings.

To Gonçalo, my long companion in this journey: we are almost there, and we are almost done. It was a pleasure working with you all these years, and I hope we can work together again in the future.

To Pedro Lima, my supervisor, with whom I started working since the graduation time, thanks for giving me your support through all these years. The logistics and conditions provided at the Institute for Systems and Robotics, made this thesis possible. Furthermore, I hope that the end of this stage marks the beginning of a new one, where both can work together to make the robotics field in Portugal an even greater field.

To all the lab people, specially the SocRob guys, Vasco Pires, Nelson Ramos, Marco Barbosa, João Santos and João Estilita, you were all sources of inspiration, and this thesis could not have been done without your work on the robots. All the long nights working in the robots, all the competitions we held together, they provided a living experience I will never forget.

To a great friend, Rui Sequeira, a.k.a “Tom Swayer”, let us continue embarking on those radical adventures.

Finally, to my colleagues and staff at the Polytechnic Institute of Leiria, thanks for keeping asking me how my thesis was going, thank you for the support, and thank you for the opportunity.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Motivation and Introduction

1.1 Motivation

The importance of robot tasks in everyday life has been increasing over the past years, with robots expected to perform even more tasks and with higher complexity in the future. Fig. 1.1 shows the cover of the 2005 International Conference on Robotics and Automation (ICRA), which illustrates that direction, showing robots performing several different everyday tasks, traditionally carried out by humans.

Figure 1.1: ICRA 2005 proceedings cover.

Traditionally a mobile robot task is programmed in a more or less ad-hoc fashion, without using formal approaches, but tailored to the task at hand. This approach usually leads to task plans with few actions and without any a priori knowledge of the expected task performance. The need for methodologies enabling design and analysis for mobile robot tasks is the main motivation for this work [Akin et al., 2008]. The availability of design and analysis tools, together with systematic and consistent modelling methods, will lead to richer task plans that, even though harder to understand intuitively, can be formally analysed and checked for performance quality.

1.2 Approach

The goal in this work is to develop a rich framework which allows the modelling, execution and analysis of robotic tasks using formal methods, by combining concepts from Computer Science, Decision and Control theory, and Manufacturing Systems to deal with the problem.

The concepts surrounding robot tasks are not uniform across the literature. As such, consider the following definitions, to be used throughout this thesis.

The main concept is the **robot task**. In this thesis, a robot task is a task to be executed by a robot, including a set of goals and/or a task plan, and a given environment in which that task plan is to be executed, and/or goals are to be achieved.

A **robot task plan** describes what actions and how a robot, or set of robots, need to perform in other to complete the desired task. It consists on a formalism which describes which actions or tasks should the robot run for any given world state. They can be specified, for instance, through a logic based approach or through a graph, such as a Finite State Automaton or a Petri net.

An **action** is the most primitive element in a robot task plan, representing an interaction of the robot with the environment and/or other robots.

The term **Robot Task Language** is often used to describe the formalism behind a robot task plan considering its execution by a robot. It is usually implemented as a type of programming language geared towards robot applications, be it single or multi-robot. In some cases it includes low-level specifications, such as hardware device management, and is indeed a full robot programming language.

To cope with the complexity and dynamic nature of mobile robot tasks, the problem is discretised using logic predicates. Then, Petri nets are used as the mathematical formalism for specification, analysis and execution of robot tasks. Further details are given later regarding this choice, but the main motivation comes from its modelling power, compactness (a Petri net with a finite structure can model an infinite state space) and available formal methods for Petri net analysis. As will be shown later, their modelling and analysis power applied to single or multi-robot tasks can prove to be quite effective.

Besides modelling the decision process, the robot impact on the environment and the actual environment are also modelled using Petri nets. This environment model includes transitions performed by other agents and/or the physics of the world. By composing all these models, a closed-loop Petri net model can be obtained which truly models (with a given approximation) the overall task. This task can then be analysed both for qualitative/logical and quantitative/performance properties.

Observation and communication models are also included allowing to analyse the impact on single and multi-robot tasks of observation and communication failures. An identification algorithm is provided which enables models to be created from real world data, allowing a decreased-error approximation for real world scenarios. Several theoretic and simulation (using a realistic robotics simulator) examples are provided showing the applicability of the developed framework.

Although the examples used throughout this thesis are based on a robotic soccer scenario, the framework is completely generic. However, given the discretisation using logic predicates, the framework is only adequate when the mobile robot navigation space and world state can be discretised.

1.3 Related Work

Having a robot planning and performing a task involves contributions from many different fields, including electronics, computer vision, logic, control theory, etc.. Traditionally three main communities have studied robot tasks: the AI community, mainly focused on planning and learning of task plans [LaValle, 2006], often for disembodied agents, but increasingly for robots as well; the manufacturing systems community, which studies performance properties for automated manufacturing lines [Viswanadham and Narahari, 1992]; and finally the Robotics community which joins people from several areas to study robots under an integrated perspective [Brooks, 1985].

Some components of the robot task execution have a continuous nature (e.g., motion control), while others have a discrete nature (e.g., decision making). Two different base approaches exist to model robot tasks: the Hybrid Systems [van der Schaft and Schumacher, 2000] approach and a purely discrete, denoted Discrete Event Systems (DES) [Ramadge and Wonham, 1989] approach. Hybrid systems are usually used to include a continuous model of the robot motion control, while having a decision or parameters change based on discrete events [Košecká et al., 1997; Egerstedt, 2000; Yu Sun, 2003]. However, the higher complexity of the models limits the size of the robot task plans which can be analysed and the properties that can be studied. As such, it is much more common to find implementations using pure Discrete Event System models.

In order to use a purely discrete state model, one needs to include some abstraction which provides an approximated state of the world. Typically this discretisation is obtained through the use of logic predicates and by using a topological map [Neto et al., 2003] for the localisation and navigation of the robot.

Some approaches provide a specific robot programming language, which can include analysis mechanisms by translating the programmed task to a known logic or temporal based language. Orccad [Kapellos et al., 1999] provides such programming environment for robot task missions, enabling extraction of both logic and temporal based properties. Other robot task programming languages exist which provide similar capabilities, such as CHARON (*Coordinated Control, Hierarchical Design, Analysis, and Run-Time Monitoring of Hybrid Systems*) [Alur et al., 2002] or PLEXIL (*Plan Execution Interchange Language For Executable Plans and Command Sequences*) [Verma et al., 2005]. However, none of these frameworks include models of the actual actions and the other agents (or physics) impact on the world state evolution.

Although specific robot programming languages are common, most of the work found in the literature is based on DES, using either Finite State Automata (FSA) [Cassandras and Lafortune, 2008] or Petri nets [Petri, 1966; Murata, 1989]. FSA-based models are mainly used to design and execute robot tasks, although they can also be used to perform quantitative and qualitative analysis. In [Košecká et al., 1997] an FSA based approach is provided, where composition operators are specified together with low-level continuous time control strategies, allowing task properties to be analysed. In [Damas and Lima, 2004] a modular FSA-based approach is used to model multi-robot systems showing how some transition parameters affect the task properties. Petri nets have been widely used to model and control Flexible Manufacturing Systems (FMS) [Viswanadham and Narahari, 1992; Flochova, 2003]. However, FMS consist mainly on systems with clearly defined inputs and outputs between the various processes, with uncontrolled events mainly associated with machine failures, as described for instance in [Castelnuovo et al., 2007] and in [Qin and Xu, 2009]. Mobile robot tasks, on the other hand, provide a much more dynamic environment where the points of interaction between the agents are not so clearly specified.

In [King et al., 2003] Marked Ordinary Petri Nets are used to model and synthesise deadlock

free plans for a multi-agent environment. However, the authors do not model failures, and uncontrolled events are modelled using a simplistic approach, by considering that the probability of resources availability change does not depend on the current state. Furthermore time is not taken into account, as all transitions are considered to be instantaneous.

In [Herrero-Perez and Martinez-Barbera, 2008] Petri nets are used to model and control a FMS with Autonomous Guided Vehicles (AGV). A collision free motion of the AGV is ensured by using a topological map and preventing more that one AGV in each node. Albeit using Marked Ordinary Petri Nets with no time associated, they consider that each node travelled in the topological map has an associated time, and minimise the number of nodes travelled to compute the task plan which minimises task time. In this system AGVs do not interact with each other and failures are not modelled. In [Ziparo and Iocchi, 2006] Petri nets are used to design robot tasks plans, providing qualitative (logical) but not quantitative (performance) analysis.

The closest work to ours is [Kim and Chung, 2007], where the authors use Generalised Stochastic Petri nets to model and analyse (both qualitatively and quantitatively) a robot task for a tour-guide robot. However, the authors approach is very application-oriented and has not provided a structured framework for modelling and analysing generic robot tasks. Furthermore, there is no clear distinction between the selection mechanisms and uncontrollable events induced by the environment, leading to a less modular design.

The developed framework described here can be seen as an extension of the ideas introduced in [Milutinovic and Lima, 2002] and [Damas and Lima, 2004], as well of the pioneer work of Wang and Saridis [Wang et al., 1991]. In [Milutinovic and Lima, 2002] the authors propose a framework for qualitative and quantitative performance analysis of robot tasks using Petri nets.

The work developed in this thesis aims at providing a framework to model, analyse and execute mobile multi-robot tasks. The framework includes environment models where uncontrollable stochastic timed transitions, due to other robots or the world physics, are modelled, thus providing a closed-loop model to analyse the robot task. Each action model is based on Generalised Stochastic Petri Nets, allowing for stochastic timed properties to be modelled and analysed. None of the above mentioned work provides all these capabilities. Furthermore, the framework was developed to be modular and to provide an hierarchical design.

Although planning was not a subject of research in this work there are several works [LaValle, 2006], including on synthesis and supervision of DES, both using FSA [Lacerda and Lima, 2008] and Petri nets Flochova [2003], and even on bringing together concepts from the AI community and the Petri nets community [Hickmott et al., 2007]. For instance, in [Rosell et al., 2003] robot task sequences of an assembly system are planned using Petri nets. It is part of our future work to include planning in this framework, as detailed in Chapter 8.

1.4 Original Contributions

The following list describes the main contributions of this work.

Unified Framework for Modelling, Analysis and Execution of Robot Tasks The main contribution of this work is the sum of its parts, that is, the complete developed framework. Having a clearly defined unified Petri net-based framework which allows for both modelling, analysis and execution of robot tasks, that does not rely solely on the task plan, is a key contribution of this work. This work includes not only the theoretical framework but also the implementation of the various algorithms within the MeRMaID middleware [Lima et al., 2007];

Environment Models The introduction of the environment models, containing uncontrollable events associated to other robots and/or the world physics, is an important contribution of the framework. Only through the use of these models can one analyse the true outcome of the task, as opposed to the typical analysis of the system which is purely based on a task plan or on an indistinguishable mixture of the two. The environment models also include observation models which allow studying the impact of observation failures in the task outcome;

Models Identification Although creating Markov chain models from real observed data is not new, using these models to create Petri net models of the actions and environment enables analysis of the overall task, and is an original contribution of this work.

1.4.1 Published Work

The following published papers contain part of the work described in this thesis:

Hugo Costelha and Pedro Lima. Petri Net Robotic Task Plan Representation: Modelling, Analysis and Execution. *Autonomous Agents, IN-TECH*, ISBN 978-953-307-089-6, pp. 65 – 89, June, 2010;

Hugo Costelha and Pedro Lima. Modelling, Analysis and Execution of Multi-Robot Tasks using Petri Nets. *Proc. of AAMAS 2008 - 7th International Joint Conference on Autonomous Agents and Multi-Agent Systems*, Estoril, Portugal;

Hugo Costelha and Pedro Lima. Modelling, Analysis and Execution of Robotic Tasks using Petri Nets. *Proc. of IROS 2007 - IEEE International Conference on Intelligent Robots and Systems*, San Diego, CA, USA, 2007.

The following paper was submitted but is yet waiting for a decision from the journal editorial board:

Hugo Costelha and Pedro Lima. Petri Net-based Robot Task Plan Representation: Modelling, Identification, Analysis and Execution. *IEEE Transactions on Systems, Man, and Cybernetics*.

1.5 Where Does This Work Apply?

Areas of possible application of the proposed framework are listed below:

Robotics The main area of application is mobile robotics, which is focused throughout the entire thesis. Particular areas of interest include robotic applications in space, rescue robots, or hazardous scenarios, in which cases manual intervention is very limited and costly, leading to the need of task execution guarantees before deployment;

Gaming Games are receiving increasing work on artificial intelligence, physics models, agent cooperation/competition, and are becoming very complex. Most games include autonomous characters which play with, or against, the human player(s) and interact with other computer players. Whether computer players are competing with, or against, the human player, they have goals to fulfil. With this complexity increase of the agents, easy workable models with enough modelling complexity must be used to improve and accelerate the design process. The work presented here could help in the definition of the players characters, the players action and task models, providing requisites for desirable performance levels;

Home appliances With increasing complexity and autonomy of home appliances comes the need to coordinate the everyday execution of their tasks. The proposed framework could be use to model and execute these tasks by autonomous home appliances, particularly if mobile robots are part of those appliances.

1.6 Thesis Outline

The second chapter of the thesis provides an overview on related work and how the developed framework fits in. It further provides an overview of robot tasks and the basic notions around their definition. Chapter 2 provides the formal definition of the types of Petri nets used in this work together with an overview on how to analyse Petri net models.

The framework for modelling single-robot tasks is detailed in Chapter 3. Here all the layers and basic building blocks are introduced, ranging from the environment models to the task plan models.

Chapter 4 details additional extensions to allow modelling of multi-robot tasks, by introducing communication models.

Chapter 5 details how Petri net analysis methods can be applied within the developed framework, explaining how single and multi-robot tasks can be analysed for both qualitative and quantitative properties.

Chapter 6 includes a series of application examples to robotic soccer scenarios, which illustrate the applicability of the framework.

Chapter 7 describes how action and environment models can be created from real data, allowing tasks to be analysed based on real data, as illustrated with a robotic soccer scenario using a realistic robotics simulator.

Finally, Chapter 8 includes the conclusions, discusses the developed work and provides directions for the future.

Appendix A and Appendix B are provided for completeness of the thesis, and describe the various methods used to perform both qualitative and quantitative analysis of Petri nets.

Chapter 2

Petri Nets

This chapter introduces and extends basic notions of modelling with Petri Nets. Throughout this work Petri Nets are the foundation of most models and ideas, thus it is important to understand these models.

One seeks for a modular design of complex robot tasks with some *a priori* knowledge of quantitative and qualitative specifications of the designed task. Petri Nets come up as an appropriate modelling and analysis tool to accomplish that task. Their modelling and analysis power applied to a single or multi-robot task definition can prove to be quite effective. The fact that Petri Nets and computer programs that run in robot platforms are very tightly coupled, makes the implementation of Petri net based task plans execution and their monitoring easier.

Petri nets [Petri, 1966] are a widely used formalism for modelling DES. They allow modelling important aspects such as synchronisation, resources availability, concurrency, parallelism and decision making, providing at the same time a high degree of modularity, making them suitable to model robot tasks. Petri nets provide not just a graphical representation of a system, but also a strong formal mathematical representation.

Petri nets are preferred to FSA due to their larger modelling power and because one can model the same state space with a smaller graph. Moreover, although composition of Petri nets usually leads to an exponential growth in the state space (as for FSA), structurally the growth is linear in the size of the composed graphs given that the state is distributed. This makes the design process simpler for the task designer, and helps managing the display of the tasks both for monitoring and designing purposes. Moreover, one uses Marked Ordinary Petri Nets and Generalised Stochastic Petri Nets [Murata, 1989], allowing the retrieval of logical and (probabilistic) performance properties. Both types of analysis with Petri nets already exist in the literature [Murata, 1989] and have been studied for some time, with a number of tools publicly available.

Modularity in Petri nets is achieved since each resource can be modelled separately and then composed with others. Although composition operators exist for FSA, Petri nets can model subsystems with input and output places, so that they can be connected as in a circuit.

2.1 Marked Ordinary Petri Nets

The simplest models we use are Marked Ordinary Petri Nets (MOPN) [Murata, 1989]:

Definition 2.1.1. *A MOPN is a five-tuple* $PN = \langle P, T, I, O, \mathcal{M}_0 \rangle$*, where:*

- $P = \{p_1, p_2, \ldots, p_n\}$ *is a finite, not empty, set of places;*
- $T = \{t_1, t_2, \ldots, t_m\}$ *is a finite set of transitions;*

- *$I = P \times T$ represents the arc connections from places to transitions, such that $i_{lj} = 1$ if, and only if, there is an arc from p_l to t_j, and $i_{lj} = 0$ otherwise;*
- *$O = T \times P$ represent the arc connections from transition to places, such that $o_{lj} = 1$ if, and only if, there is an arc from t_l to p_j, and $o_{lj} = 0$ otherwise;*
- *$\mathcal{M}(j) = [m_1(j), \ldots, m_n(j)]$ is the state of the net, and represents the marking of the net at time j, where $m_n(j) = q$ means there are q tokens in place p_n at time instant j. $\mathcal{M}(0)$ is the initial marking of the net.*

A simple MOPN is depicted in Fig. 2.1. It has two types of nodes: places, represented by circles, and transitions, represented by filled rectangles. The places can contain any number of tokens, represented by the number of dots (or a number) inside the place. For instance, in the Petri net shown in Fig. 2.1, place p_1 and place p_3 both have one token, while place p_2 has zero tokens.

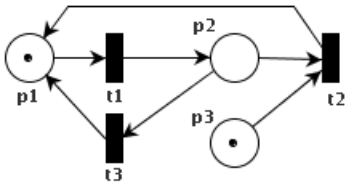


Figure 2.1: A simple Petri Net.

The state of the net is given by the marking of the net, which in turn is given by the number of tokens in each place. The net evolves through different states (or markings) through the firing of enabled transitions. A transition is enabled if all its input places have at least one token. Going back to the example on Fig. 2.1, the set of enabled transitions in the initial marking, $\mathcal{M}(0) = [1, 0, 1]$, is $\mathcal{T} = \{t_1\}$. When an enabled transition fires, all its input places loose one token and all its output places gain one token. Firing transition t_1 in this case would result in a new marking, $\mathcal{M}(1) = [0, 1, 1]$, as shown in Fig. 2.2.

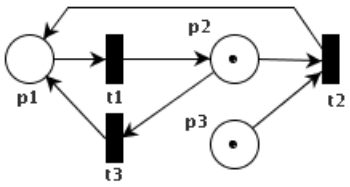


Figure 2.2: A simple Petri Net (after firing transition t_1).

In this class of Petri nets, all the transitions are *immediate* (have zero firing time), i.e., once they are enabled, they are fired and the new marking is instantly reached. Furthermore, when referring to input or output nodes of a particular node, one is referring to the nodes connected to or from that node. For instance, transition t_3 has places p_2 and p_3 as its input places, while it has only one output place, p_1.

2.2 Generalised Stochastic Petri Nets

While MOPNs are important for qualitative analysis, they are not suitable for performance analysis. For this purpose, we can use Generalised Stochastic Petri Nets (GSPN) [Murata, 1989; Viswanadham and Narahari, 1992].

Definition 2.2.1. *A standard GSPN is an eight-tuple $PN = \langle P, T, I, O, \mathcal{M}_0, R, S \rangle$, where:*

- *$P, T, I, O, \mathcal{M}_0$ are as defined in 2.1.1;*
- *T is partitioned in two sets: T_I, of immediate transitions, and T_E, of exponential transitions;*
- *R is a function from the set of transitions T_E to the set of real numbers, $R\left(t_{E_j}\right) = \lambda_j$, where λ_j is called the firing rate of t_{E_j};*
- *S is a set of* random switches*, which associate probability distributions to subsets of conflicting immediate transitions.*

Stochastic (exponential) transitions, once enabled, fire only when an exponentially distributed time d_j has elapsed. When two or more stochastic transitions are enabled, the firing probability depends on the enabled transitions rate. If $\mathcal{T} = \{t_{E_1}, \ldots, t_{E_n}\}$ is the set of enabled stochastic transitions for a given marking, the firing probability of each transition is given by

$$P\left(t_{E_j}\right) = \frac{\lambda_j}{\sum_{i=1}^{n} \lambda_i}$$

This definition of GSPNs includes also the possibility of associating a probability distribution to conflicting immediate transitions, by the use of the *random switches*. These random switches can be static (invariant to the marking of the net) or dynamic (dependent on the marking of the net) Viswanadham and Narahari [1992]; Bause and Kritzinger [2002]. We use the later case by associating weights to the immediate transitions, as described in Definition 2.2.2.

Definition 2.2.2. *A GSPN is an eight-tuple $PN = \langle P, T, I, O, \mathcal{M}_0, R, W \rangle$, where:*

- *$P, T, I, O, \mathcal{M}_0, R$ are as defined in 2.2.1;*
- *W is a function from the immediate transitions set T_I to a set of real numbers, $W\left(t_{I_j}\right) = w_j$, where w_j is the weight associated with immediate transition t_{I_j};*
- *For any given marking, the probability of firing an enabled transition t_i is equal to $w_i/\mathcal{W}$, where $\mathcal{W}$ is the sum of the weights of all enabled transitions for the given marking.*

Consider the GSPN model depicted in Fig. 2.3 (the double end arc is a shorthand for two arcs in opposite directions and is used only to visually reduce the clutter). In this example, t_{E_1} and t_{E_2} are exponentially timed transitions (drawn as an unfilled rectangle), while t_{I_1}, t_{I_2} and t_{I_3} are immediate transitions with associated weights. Initially t_{E_1} and t_{E_2} are enabled, since p_1 has a token, and either one will fire after an exponentially distributed time with rate λ_1 and λ_2 has elapsed, respectively. The firing probability of each transition is given by

$$P\left(t_{E_1}\right) = \frac{\lambda_1}{\lambda_1 + \lambda_2} \qquad P\left(t_{E_2}\right) = \frac{\lambda_2}{\lambda_1 + \lambda_2}$$

If t_{E_1} fires, the token flows from p_1 to p_2 and, since t_{I_1} is an immediate transition, it will immediately fire and the token will flow from p_2 to p_3, reaching marking $\mathcal{M}_3 = [0, 0, 1]$. In this

marking t_{I_2} and t_{I_3} form a set of conflicting transitions, whereas only one will fire, according to the following probabilities:

$$P\left(t_{I_2}\right) = \frac{w_2}{w_2 + w_3} \qquad P_f\left(t_{I_3}\right) = \frac{w_3}{w_2 + w_3}$$

If t_{I_3} is fired, the marking remains the same, if t_{I_2} is fired, the net returns to the initial marking.

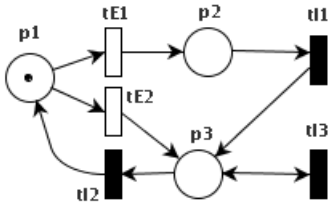


Figure 2.3: Generalised stochastic Petri net.

The GSPN marking is a semi-Markov process with a discrete state space given by the reachability graph of the net for an initial marking [Murata, 1989; Viswanadham and Narahari, 1992]. A Continuous Time Markov Chain (CTMC) and/or the corresponding Embedded Markov Chain (EMC) can be obtained from the marking process, and both the transition rate matrix (CTMC) and the transition probability matrix (EMC) can be computed by using the firing rates of the exponential timed transitions and the probabilities associated with random switches. With these one can perform transient and stationary analysis of the chain, thus obtaining performance properties for the corresponding Petri net model. As such, one has two different types of analysis that can be performed with GSPNs: conservation properties (based on T-invariants and P-invariants) of the associated MOPN and performance evaluation (based on the continuous time, discrete state space equivalent Markov process). See Appendix A and Appendix B for more details.

2.3 Petri Nets Composition

One of the advantages of Petri nets is its modularity. The various models that compose a full model can be designed separately and combined only when needed, such as when performing analysis of the full model. The key issue of that process is the composition of the models.

The composition method will depend on the type of places available, and will be thoroughly explained later in Chapter 3.

2.4 Additional Specifications

In the developed framework, the Petri net models are embodied with some additional building blocks and place labels are used to distinguish between different types of places, such as: *predicate places*, *action places*, *communication action places*, *counter places*, *task places* and *regular* (or *memory*) *places*. These different types of places do not introduce any change regarding the definition of the Petri nets, but are crucial in the analysis process explained later, particularly in the composition of the models.

Regular (or memory) places are normal Petri net places, without any special properties. The remaining types of places are introduced in the following sections.

2.4.1 Predicate Places

Predicate places are used to represent logic predicates, having always one or zero tokens. Although Predicate Petri nets exist in the literature [Röck and Kresman, 2006], the tools available to work with this type of Petri nets are very scarce. As such, predicates are represented by regular places, as explained next.

Definition 2.4.1. *A predicate place p_n is a place associated with the predicate $\mathcal{P}$, described by $p_n \models \mathcal{P}$, such that:*

- $\forall_j, \mathcal{P}(j) = true \Leftrightarrow m_n(j) = 1$
- $\forall_j, \mathcal{P}(j) = false \Leftrightarrow m_n(j) = 0$,

where $\mathcal{P}(j)$ is the predicate $\mathcal{P}$ at time step j.

Basically, a place representing a predicate has one token if that predicate is true and zero tokens otherwise.

Definition 2.4.2. *A Petri net model of a predicate is a MOPN where:*

- $P = \{\neg p, p\}$, *where and $\neg p$ and p are predicate places associated with predicates $\neg\mathcal{P}$ and $\mathcal{P}$ respectively;*
- $I = \emptyset$;
- $O = \emptyset$;
- $\forall_j, \mathcal{M}_j = [0, 1] \vee [1, 0]$.

Although one could achieve the same results by using just one place to represent a predicate, that would lead to the use of inhibitor arcs. Once again the choice was to maintain the use of the base Petri nets with minimal extensions added, so as to be able to use a larger set of available Petri net tools. Furthermore, although it increases the number of places, it does not increase the state space, and provides a cleaner interface to the user.

Figure 2.4: Representation of predicate by a set of places.

As an example, a Petri net model representing the predicate `SeeBall` is depicted in Fig. 2.4. Note the usage of the "*predicate.*" (or, alternatively, "*p.*") prefix to denote that the place is a predicate place, and the "*NOT_*" prefix to denote the negated predicate.

2.4.2 Task Places

Task places are macro places [Bernardinello and Cindio, 1992] which, albeit not always using the same definition, are used to create hierarchical Petri nets, leading to a higher degree of modularity. A task place represents a Petri net, allowing to draw entire Petri net models from lower layers as single places in higher layers, providing for cleaner and reusable models.

Places associated with tasks have their labels prefixed with "*task.*" (or, alternatively, "*t.*"). These task places will have an important role in the analysis part, particularly in what we denote as the *Expansion Phase*, as described later in Section 5.1. Task places will be further detailed in Section 3.4.

2.4.3 Action Places

Action places are also a type of macro places. The main difference towards task places, is that the Petri net models associated with action places cannot include action or task places.

Action places have their labels prefixed with "*action.*" (or, alternatively, "*a.*"). Like task places, action places will also have an important role in the analysis part. More details about action places will be given in Section 3.4, while details on expanding actions places will be given in Section 5.1.

2.4.4 Communication Action Places

Communication action places are similar to action places, but are associated with the transmission of information through communication.

Communication action places have their labels prefixed with "*message.*" (or, alternatively, "*m.*"). Like task and action places, communication action places will also have an important role in the analysis part. Communication action places will be detailed in Section 4.3.1, while details on expanding communication action places will be given in Section 5.1.

2.4.5 Counter Places

Counter places are part of the communication actions, and will be used to determine the number of communication actions, associated with the same message, running simultaneously. Counter places cannot be used in models other then communication actions.

Counter places have their labels prefixed with "*counter.*" (or, alternatively, "*c.*"). These places are also relevant for the analysis part, particularly when analysing tasks that involve communication. The use of counter places will be described in more detail in Section 4.3.1.

2.5 Petri Nets Analysis

Petri net properties can be divided in two main sets: qualitative and quantitative. The logical (qualitative) properties are based on the structure of the Petri net, while performance (quantitative) properties take time into consideration. In the qualitative case, Petri nets can be analysed for deadlocks, conservation properties and other qualitative properties. In the quantitative case, stochastic time is used in the analysis, providing results regarding for both stationary and transient analysis. Note that a MOPN can only be analysed for logical properties, since time is not included.

Several logical and performance properties can be analysed, using different analysis techniques, as described thoroughly in the literature [Murata, 1989; Viswanadham and Narahari, 1992; Lindemann, 1998; Girault and Valk, 2003; David and Alla, 2005; Cassandras and Lafortune, 2008]. The reminder of this section briefly describes some of these properties and analysis techniques, with more details being given in Appendix B.

Regarding logical properties, one can analyse properties such as: *Reachability and Coverability*, which allows determining if a given state is reachable or covered; *Boundedness*, which determines if all places in all reachable markings have a finite number of tokens; *Safety*, where having a k-safe Petri net means all places have at most k tokens for all reachable markings; or *Liveness and Deadlock*, which is associated with the firing possibility of each transition for all reachable states. Regarding performance properties, one can study both *Steady-state* and *Transient* properties. In the steady-state case, one can obtain, for instance, information regarding the mean time to reach a given state, the mean time spent on each state, the average number of tokens in each place, or the average throughput of each transition. In the transient case

one can study, for instance, how the probability distribution of the number of tokens per place evolves until the steady-state is reached. Furthermore, one can analyse how changes in the Petri net affect these properties, be it structural changes, initial marking changes or transition rate changes, by comparing the results for the different test cases.

These various properties can be analysed using different methods, as summarised in the following list, for each type of analysis (for more details see the literature referred above):

- **Logical**

  Algebraic - using the incidence matrix, which describes algebraically the transitions between all places, and the marking of the net, one can obtain a state equation which describes the Petri net dynamics and compute T-invariants and P-invariants;

  Reachability graph and coverability tree - by enumerating the set of (possibly) reachable states and performing the analysis of the graph or tree;

  Transformation - by transforming or reducing the Petri net such that it belongs to a class where enumerating all possible states is no longer needed;

  Simulation - by analysing, for instance, the result of a Monte Carlo simulation of the Petri net;

- **Performance**

  - **Transient**

    Numerical Ordinary Differential Equations (ODE) - by explicitly solving the ODE describing the CTMC using numerical approximation techniques;

    Uniformisation - applying uniformisation to the associated CMTC one can obtain a DTMC with identical mean sojourn time across states, allowing to use a simple series to compute the probability distribution evolution over the states of the CTMC;

    Laplace Transform - by using symbolical Laplace transforms, for small models, or analytical Laplace transforms, for large models;

    Direct evaluation of the matrix exponential series - by directly evaluating the expression which describes the probability distribution over time using a matrix exponential.

  - **Steady-state**

    Algebraic - Using the direct approach to solve a set of linear equations;

    Iterative - Using iterative methods to converge to the solution of the set of linear equations.

Algebraic techniques enable the analysis of logical properties without having to enumerate all states, but are not always applicable. For instance, these can be use to determine the boundedness of the net, but generally only provide necessary conditions for reachability properties. On the other hand, the reachability graph implies enumerating all reachable states but enables determining reachability properties. However, the reachability graph can only be used with bounded Petri nets. With unbounded Petri nets, instead of the reachability graph, one can use the coverability tree, in which case reachability properties are not always guaranteed, depending on the Petri net. Using transformation techniques which maintain the net properties, often in the form of reduction, leads to less complex Petri nets, easing the analysis process. For instance, by greatly reducing the state space one can obtain a smaller reachability graph.

Regarding performance analysis, particularly transient analysis, the use of direct evaluation of the matrix exponential series can lead to numerical instabilities due to rounding errors, while using analytic Laplace transforms requires high numerical precision. The most used techniques are based on iterative numerical methods to compute the ODE solution, and the uniformisation approach, which might yield less computational problems. For the steady-state analysis, the use of a direct approach leads to a result in a fixed number of arithmetic operations, but might prove problematic due to rounding errors. In this cases an iterative approach can lead to more accurate results, although it might take a large number of iteration to converge to the solution Knottenbelt [1996].

The methods implemented for this work were based on the Reachability Graph and algebraic analysis which, for completeness, are detailed in Appendix B.

Chapter 3

Modelling Individual Robot Tasks

This chapter details the modelling of single-robot tasks using Petri Nets. Several building blocks will be defined and then used to model, analyse and execute robot tasks.

Performing a robot task involves different levels of abstraction and different layers of actuation. One would like to give a set of goals to a robot and let it plan how to achieve those goals under a given set of conditions, and subsequently execute its plan. In this sense, a robot task consists of modelling, planning and execution.

In this work one does not want to separate completely task planning from task execution, but rather be able to define tasks and their building blocks under a set of constraints and to be able to analyse *a priori* (and *a posteriori*) the possible outcomes of the task execution. This means modelling the agent, the actions and the environment, and use a planning algorithm on top of it to achieve the goals. The algorithm however, needs to be tightly coupled with the models in order to extract the necessary information to handle the problem constraints, typical in a robot task problem.

The following points summarise the goals of the developed framework:

Modularity fostering the reuse of developed components;

Design providing an intuitive, and possibly graphical, task design solution;

Analysis providing means to analyse a robot task both before and after its execution;

Execution keeping the models suitable for execution, so that these closely follow the framework theoretic foundations;

Planning allow synthesis of task plans.

To achieve these goals, a Petri net based solution was developed, using various layers. Throughout this chapter the models at each layer and their relation are detailed. This chapter starts with the definition of the different layers, representing different abstractions (Section 3.1). Then each layer is detailed from the lowest to the highest-level layer, starting with Environment related models (Section 3.2), followed by the Action Executor layer (Section 3.3), the Action Coordinator layer (Section 3.4), and ending with the Organisation layer (Section 3.4). Note that automatic synthesis of Petri net based task plans (planning) was not yet addressed, being part of the future work.

3.1 Design Methodology

Using different layers provides a modular model of the task. Fig. 3.1 gives an overview of the used layers.

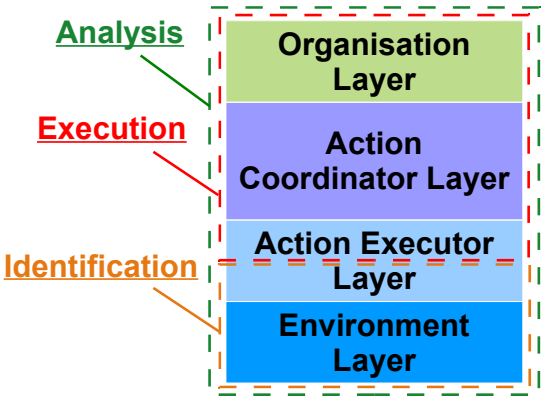


Figure 3.1: Models Hierarchy, pointing out the 4 layers and those that intervene in the analysis, execution and identification.

Each layer is composed as a set of Petri net models which represent different resolution levels: the Environment layer the bottom, and the Organisation layer at the top. The meaning of each layer is as follows:

Environment layer Petri net models in this layer represent changes made by other agents (such as other robots) or physics (such as the motion of a free rolling ball);

Action Executor layer In this layer one finds Petri net models of the actions, representing the changes performed in the environment by these actions, and the conditions under which these changes can occur;

Action Coordinator layer Here lies the Petri net based task plan models, which basically consist of compositions of actions and/or tasks;

Organisation layer This layer is where higher decision models appear, such as goal selection, thus consisting of compositions of Action Coordinator layer models.

As can be seen in Fig. 3.1, all models are used in the analysis process, but only the two higher layers and, partially, the Action Executor layer models will be used for execution. Both the Environment layer and part of the Action Executor layer models can be parametrised through an identification process on real data, as explained in Chapter 7.

Each layer includes several models (e.g., in the Action Executor there is one model per action), and each model can be designed separately, thus simplifying the design process.

3.2 The Environment Layer

The Environment layer includes models of the environment dynamics, caused by the controlled robot, other agents, or simply by the laws of physics. A discrete set of relevant states is abstracted from the actual environment. Naturally these models will not fully model the environment, but an abstraction of it.

The environment models will prove to be important for a priori verification and validation of tasks. It further allows to determine undesired behaviours and possible unexpected occurrences, providing means for simulation and decision making based on prediction.

The environment abstraction is achieved by discretising the world using logic predicates. As such, environment models consist on GSPNs with predicate places, as introduced in Chapter 2. The environment models definition is as follows:

Definition 3.2.1. *An environment model is a GSPN where all places $p_j \in P$ are either predicate or memory places.*

To better understand how the environment models are designed, consider a free rolling ball. In this case, due to friction on the floor, it is expected that the ball will stop after some time. To model this process using a GSPN model under the given framework, one must first discretise it, such that we can describe it through the use of logic predicates. In this example, one could consider that the ball could be moving fast, slowly or be stopped, and that the ball will, with time, pass from the fastest movement to the stopped state. With this discretisation, one can model the free ball movement with the Petri net model depicted in Fig. 3.2.

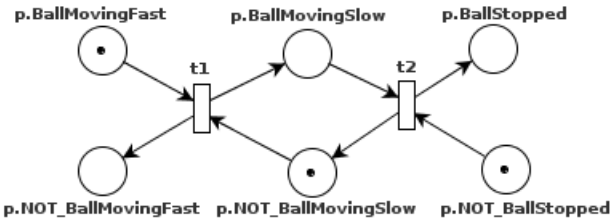


Figure 3.2: Petri net model of a free rolling ball.

Since predicate places must follow Definition 2.4.2, there is no need to always draw both the positive and negative forms of the predicate. In practice, the models can be drawn in a simpler form, such that the missing nodes are added during the analysis part (explained in Chapter 5). For instance, the model depicted in Fig. 3.2 can be draw as shown in Fig. 3.3.

Figure 3.3: Compact Petri net model version of a free rolling ball.

If, for instance, one also wanted to model the fact that some other agent could increase the ball speed, one could add transitions in the opposite direction, albeit with different associated rates, considering the probability of that occurrence. Furthermore, it is also possible to include several transitions with different rates, associated with the same state change, as in the example depicted in Fig. 3.4. In this example, the rate at which the ball slows down depends on the weather conditions.

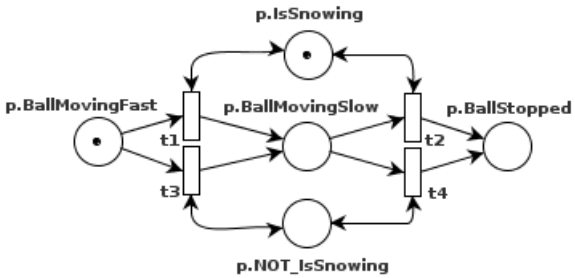


Figure 3.4: Petri net model of a free rolling ball considering the weather conditions.

3.2.1 Observation Models

The environment models model the world state. If full observability with no observation failures or delays is considered, the environment model predicates are in fact the predicates which

describe the world state seen by the robot. However, it is possible to include models which reflect observation failures or delays.

Consider the model shown in Fig. 3.3. In a full observability scenario the predicates used in that model would also be the predicates used by the robot. Consider now the same scenario, but where the robot perceives that the ball stopped with an exponentially distributed delay. To model this scenario, one would need to introduce a predicate, here denoted `R1BallStopped`, and include the model shown in Fig. 3.5, describing the ball stopped situation as seen by the robot. The model in Fig. 3.3 combined with this model would achieve the desired result.

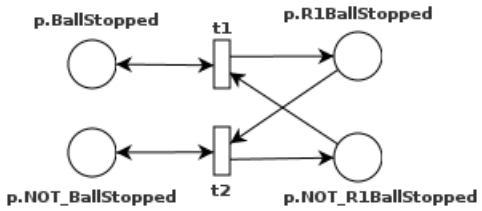


Figure 3.5: Petri net model of the observation with delays.

Additionally to modelling observation delays, it is also possible to include observation errors. Considering again the same model of the moving ball, modelling observation errors is achieved by including the model depicted in Fig. 3.6

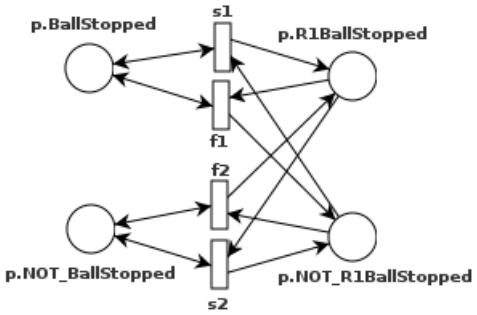


Figure 3.6: Petri net model of the observation with delays and failures.

Adding observation delay or error models naturally as a cost of increasing the state space.

3.3 The Action Executor Layer

The Environment layer represents the environment and it has no direct association with what runs on the robot, i.e., it models what is expected to happen in the environment, not how the robot performs (although its information will be used to make decisions). The other three layers represent what is actually performed by the robot and how it is suppose to act. The first, and lowest layer of this group, is the *Action Executor* layer, which holds the *action models*.

An action is mainly described by the effects it causes on the environment and the conditions that need be met for the effects to take place. In logical terms, the action properties can be partitioned in the following sets:

- **Running-conditions**: Conditions that need be met for the action to be able to produce changes in the world;
- **Effects**: Composed of *Success Effects* and *Failure Effects*, reflect the impact an action has on the world:

- **Success-effects**: These are the effects associated with the success of the action. These include the *desired-effects* of the action plus additionally intermediate effects that might occur in order to achieve success;
- **Failure-effects**: These effects are the undesired ones, which might happen as a direct result of running the given action.

In this framework, one is additionally interested in modelling the probabilistic nature of the action, including time properties. For instance, and as explained for the environment models case, the rate at which the action effects occur will depend on the world state. Such a model can be achieved using a GSPN, consisting on a set of transitions representing the environment changes, associated with either the success or failure of the action, following the rules described in Definition 3.3.1. The general model of an action is depicted in Fig. 3.7. Although immediate transitions were not included in this model, transitions can be either stochastic timed or immediate. Only stochastic timed transitions were included in the figure, since generally an action does not lead to immediate world changes, but takes time.

Definition 3.3.1. *A Petri net model of an action is a GSPN, where:*

1. $P = P_E \cup P_R$ *contains only predicate places, where*

   P_E *is the* effects *place set;*

   P_R *is the* running-conditions *place set;*

2. *All places in* P_R *have "*`r.`*" after the "*`predicate.`*" prefix;*

3. $P_E = P_{E_S} \cup P_{E_F}$*, where* P_{E_S} *and* P_{E_F} *are designated respectively* success places set *and* failure places set*.*

4. $P_{E_S} = P_{E_{S_I}} \cup P_{E_{S_D}}$*, where* $P_{E_{S_I}}$ *and* $P_{E_{S_D}}$ *are designated as* intermediate effects place set *and* desired-effects place set*, respectively.*

5. *All places in* $P_{E_{S_D}}$ *have "*`e.`*" after the "*`predicate.`*" prefix;*

6. $T = T_S \cup T_F$ *with* $T_S \cap T_F = \emptyset$*, where:*

   T_S *is the set of transitions associated with successful impact of the action;*

   T_F *is the set of transitions associated with failure impact of the action;*

7. *If there is an arc from place* p_n*, associated to predicate* $\mathcal{P}$*, to transition* t_j*, then there is an arc from* t_j *to place* p_m*, associated to predicate* $\neg\mathcal{P}$*, or an arc back to* p_n*;*

8. *All transitions have one input arc from each* running-condition*;*

9. *All transitions* t_j *in* T_S *have the label* $\texttt{success}_j$ *or* $\texttt{s}_j$*;*

10. *All transitions* t_j *in* T_F *have the label* $\texttt{failure}_j$ *or* $\texttt{f}_j$*;*

The fundamental changes from traditional models, is that one does not model only the *desired-effects* of the action, but also intermediary and failure effects, all of them within a probabilistic framework through the use of stochastic transitions. Note that enabling an action does not necessarily imply that any of its transitions will fire, due once again to their stochastic nature. Given that the action model will be composed with other models, enabling the action simply implies that the state changes associated to its enabled transitions will have an increased probability of occurring.

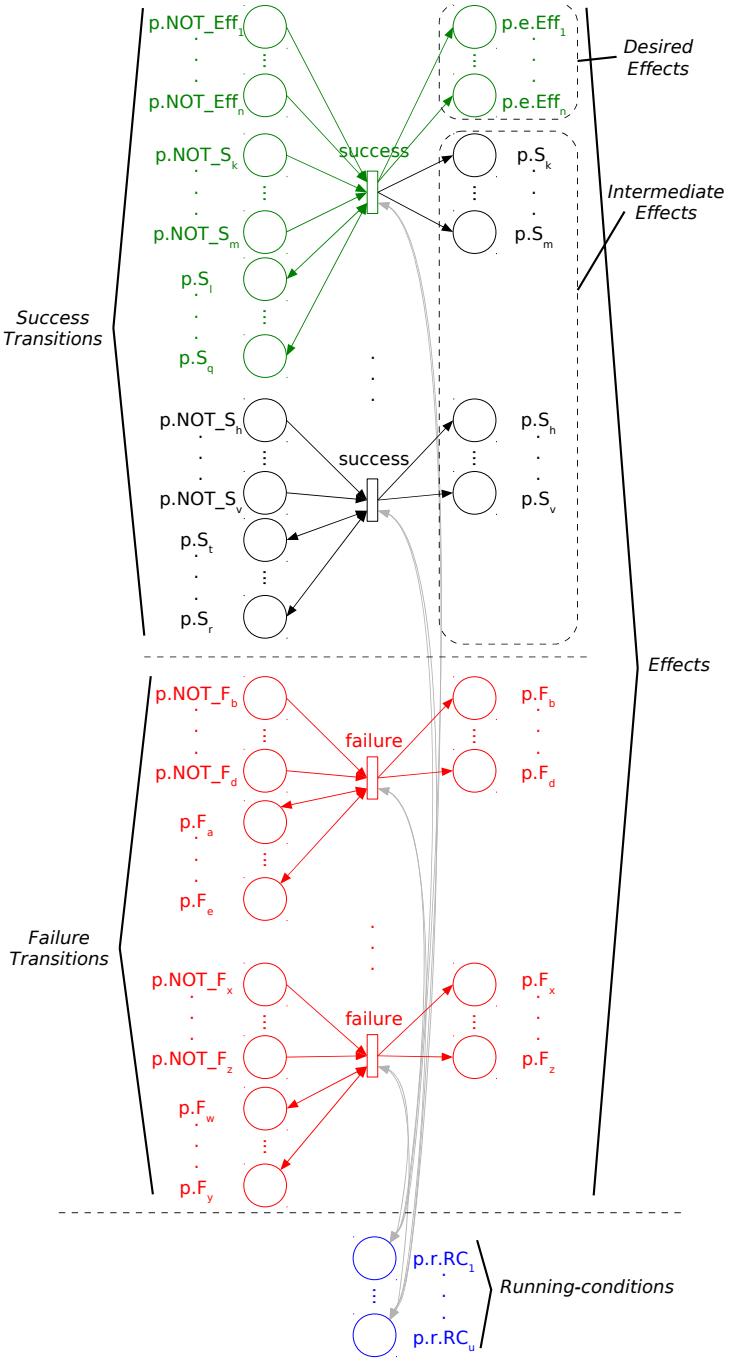


Figure 3.7: General action model.

The *running-conditions* are input places of all transitions to model the fact that the action can only cause any impact on the environment if these conditions are met. Given that all places are predicate places, rule 7 implies that the action model maintains the predicates according to Definition 2.4.2, resulting in a safe Petri net (since there is at most one token per place for all markings).

As an example, consider an action named `CatchBall`, where the goal of the robot is to catch a ball. It is expected that the robot can catch the ball only when near the ball and if it sees the ball, meaning its *running-conditions* should be `IsBehindBall` and `SeeBall`. Furthermore, the *desired-effect* of this action is catching the ball, i.e., getting the predicate `HasBall` to true. The resulting Petri net model is shown in Fig. 3.8.

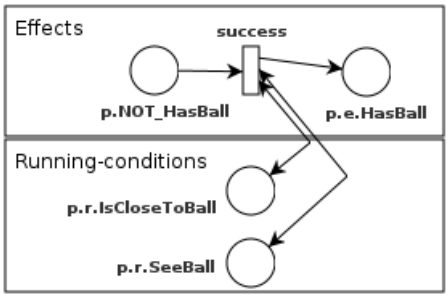


Figure 3.8: Petri net model of action `CatchBall`.

Failures were not explicitly included in `CatchBall` action model. Although including them is possible, and even desired in many situations, these can be already implicitly present, given that this model will be composed with the environment model, which models external changes. Only world changes which have a relevant probability increase as a direct result of the action execution should be included.

For execution purposes the Action Executor models are currently used partially, by taking into consideration the *running-conditions* to prevent enabling each action outside of their scope. If the *running-conditions* of a given an action are not enabled that action will not be executed, even if selected by an higher layer.

It is important to note that the *running-conditions* might be an empty set, meaning that the respective action can be executed with success at any time. Concerning the *desired-effects*, it does not make sense to have an empty set, given that every action should have a purpose.

3.4 The Action Coordinator Layer

The Action Coordinator layer contains Petri net models of robot task plans. A Petri net model of a task plan consists of a MOPN which models predicate based decisions.

Definition 3.4.1. *A task plan model is a MOPN where*

1. *All places $p_j \in P$ are either* predicate places, memory places, action places *or* task places.
2. *If there is an arc from place p_n, associated to predicate $\mathcal{P}$, to transition t_j, then there is an arc from t_j back to place p_n;*
3. *The desired output state of the MOPN, in terms of action and task places, is specified by including ".o" in the place labels. These places are denoted as* task output places.

Item 2 in this Definition is of particular importance, since environment and action models are the only ones which model changes in the world state, while task plans model action/task selection decisions based on those predicates. Given that a task plan can contain task places, which also correspond to task plans, in practice one has an hierarchical Petri net task plan model without a predefined depth limit.

Task macro places and actions macro places are used to represent a Petri net based task plan and action, within another Petri net based task plan, respectively. They are referred to as task and actions places, as the macro condition is implicit.

As an example, consider a soccer-playing robot with the task `Get_Ball`, depicted in Fig. 3.9, which is used for the robot to capture the ball. This plan corresponds to a loop between actions `MoveBehindBall` and `CatchBall`.

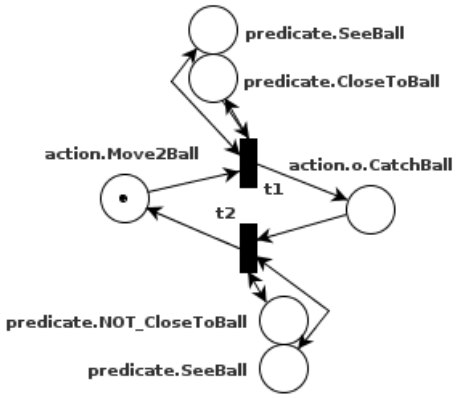


Figure 3.9: Petri net model of the `Get_Ball` task plan.

The initial marking in the task plan models is very important, since it determines which actions, or included tasks, should run when the task is started. The usage of "`.o.`" in the action and/or place labels indicates which are the *task output places*, i.e., which actions or tasks should be running in the desired final marking of the task plan Petri net in case of success. Although this knowledge is not used yet, we expect it to allow us to determine a task *desired-effects* in the future, which will be important for planning purposes.

To better understand the concept of tasks within tasks, consider now a task plan for a full soccer playing robot, depicted in Fig. 3.10. Here, besides action `Dribble2Goal` and task `Shoot_For_Goal`, one also uses the `Get_Ball` task plan described previously.

Petri net based task plans can run directly in a robot if a Petri net execution module is used, preventing the user from having to do any additional code. Alas, regarding execution, in the worst case scenario the designer only has to specify the task plan.

3.5 Organisation Layer

The Organisation layer is a conceptual layer, used to perform higher-level decisions. Its definition is very similar with the Action Coordinator layer, but only tasks are used.

Definition 3.5.1. *A task plan model is a MOPN where*

- *All places $p_j \in P$ are either* predicate places*,* memory places *or* task places*.*
- *If there is an arc from place p_n, associated to predicate $\mathcal{P}$, to transition t_j, then there is an arc from t_j back to place p_n;*

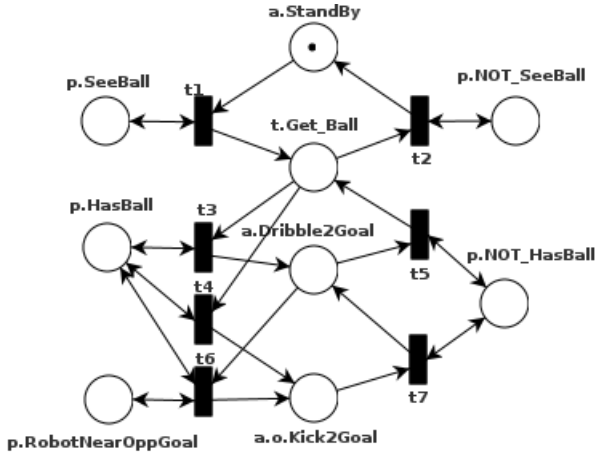


Figure 3.10: Petri net model of the `Score_Goal` task plan.

- *The desired output state of the MOPN, in terms of action and task places, is specified by including ".o" in the place labels. The places are denoted as* output places.

As an example, consider again a soccer playing robot. Usually robots have different roles in a soccer game, such as defender, attacker or goalie, which are dynamically assigned during the game. In this scenario, the role selection would be modelled by a Petri net following Definition 3.5.1 in the Organisation layer. Although the same could be achieved by including additional models in the Action Coordinator layer, by providing different layers, the models can be designed and used in a more modular, and thus cleaner, approach.

Given the definition of the Organisation layer, it is clear that one can define additional higher layers as needed, depending on the application.

Chapter 4

Modelling Multi-Robot Tasks

This chapter details and extends the framework introduced in the previous chapter to multi-robot tasks.

There is often the need to coordinate a group of robots in order to achieve a common goal or improve the probability of success of a given task. This coordination is achieved by synchronising two or more robots based on heuristics applied to the world state, where each robot would take the same collective decision given the same view of the world, and/or through message passing among the robots. In the later case, a robot uses a specific mean of communication to send information (e.g., using wireless LAN), while in the first case the robot simply perceives information regarding the world, including other robots, without explicitly exchanging messages (e.g., using vision).

Modelling the implicit communication case can already be achieved by using the observation models introduced in Section 3.2.1, as exemplified in Section 4.1. For the explicit case, one needs to introduce *communication models*, as detailed in Section 4.3. Section 4.2 introduces *model templates*, used to simplify the design of multi-robot task related models.

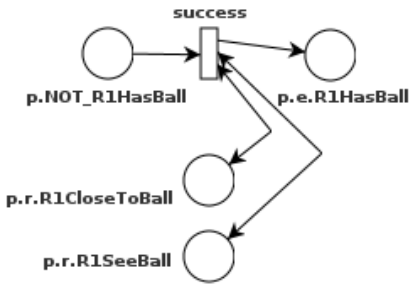
4.1 Multi-Robot Task Models Without Explicit Communication

In the developed framework knowledge is represented through logic predicates, which appear as predicate places in the Petri net models. The multi-robot case is no different, albeit the need to distinguish between the predicates of the different robots. To do so, each robot is tagged with a specific name, Rx, where x is the robot number. This tag is then used to prefix each place label to distinguish to which robot the place belongs.

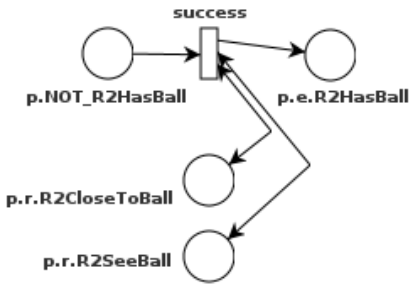
As an example consider the action `CatchBall` model shown in Fig. 3.8. If this action were to be used in a multi-robot task with two robots, then there would be two models, one for each robot, as depicted in Fig. 4.1.

If full observability, without any errors or delays, is to be considered, then each robot task plan can use directly the other robots and non-robot predicates, as was detailed for the single-robot case. For instance, consider a two-robot task plan (denoted `Role_Supporter`) where the goal is to have just one robot trying to grab the ball, which is achieved using the models depicted in Fig. 4.2. In this task, if one robot is close to the ball, the other robot will not try to go for it.

The observation models introduced in Section 3.2.1 can be used in the multi-robot case to model delays and errors in the observations of other robots information, as it is done for the predicates of non-robot objects in the environment. In this case, each robot will have its own

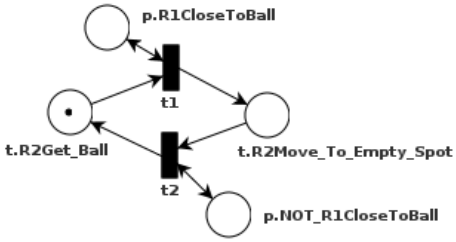


(a) Action `CatchBall` model for robot $R1$.

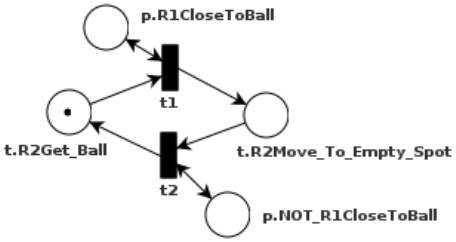


(b) Action `CatchBall` model for robot $R2$.

Figure 4.1: Petri net models of action `CatchBall` for the multi-robot case.



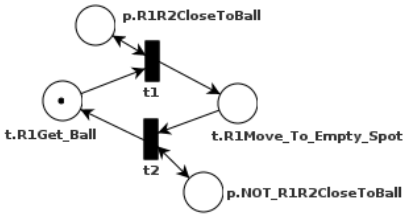
(a) Task `R1Role_Supporter` model (robot $R1$).



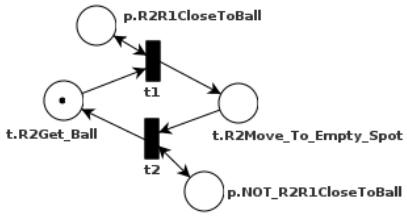
(b) Task `R2Role_Supporter` model (robot $R2$).

Figure 4.2: Petri net models of task `Role_Supporter` for the multi-robot case.

predicate associated with the observation of the other robot predicate state. Considering the example given above, if errors and/or delays were to be considered, the task plan models would be the ones depicted in Fig. 4.3. Here predicate `R1R2CloseToBall` denotes the observation of predicate `R2CloseToBall` made by robot $R1$, while predicate `R2R1CloseToBall` denotes the observation of predicate `R1CloseToBall` made by robot $R2$.



(a) Task `R1Role_Supporter` model with observation error/delay (robot $R1$).

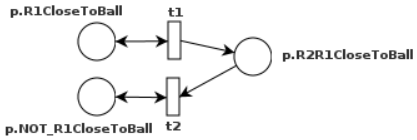


(b) Task `R2Role_Supporter` model with observation error/delay (robot $R2$).

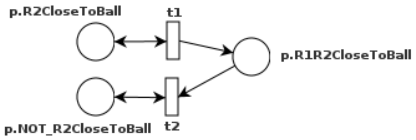
Figure 4.3: Petri net models of task `Role_Supporter` for the multi-robot case.

Having for each robot a predicate that reflects another robot predicate, one can now introduce the observation models in the environment layer to model those predicates updates. Following the given example one would need to introduce the models depicted in Fig. 4.4 to model observation delays among robot predicates.

To conclude, Fig. 4.5 shows the models that would be used when modelling observation



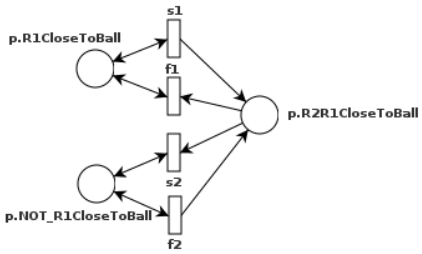
(a) Model of predicate propagation with delays for robot $R1$ predicate.



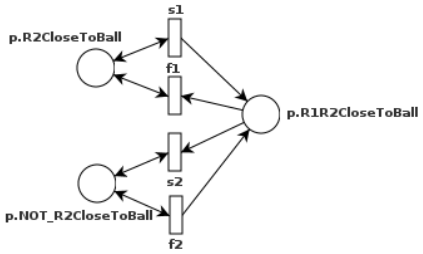
(b) Model of predicate propagation with delays for robot $R2$ predicate.

Figure 4.4: Petri net models of observation update with delay for the multi-robot case.

errors with delays for the given example.



(a) Model of predicate propagation with delayed errors for robot $R1$ predicate.



(b) Model of predicate propagation with delayed errors for robot $R2$ predicate

Figure 4.5: Petri net models of observation update with delay for the multi-robot case.

The concepts introduced in the single-robot case together with the robot tags are enough to model tasks in the implicit communication case. Nevertheless, if the number of robots increases, creating these models for each robot becomes too cumbersome. To solve this issue one uses model templates, as described in the next section.

4.2 Multi-Robot Model Templates

In the multi-robot case, each robot model follows a very similar structure, distinguished by the robot tag. It is thus possible to create a template for each model and generate each robot model from that template.

Definition 4.2.1. *A* model template *is a model that uses specific* place tags*, which can be instantiated to generate models usable in multi-robot scenarios.*

Definition 4.2.2. *A* model template place tag *specifies how the tagged places in a model template are processed when instantiating the template into an usable Petri net model. The tag must be one of "(&all)", "(&!Rx)", "(|all)", "(|!Rx)" or "Rx".*

Remark 4.2.1. *At most one of the tags "(&all)", "(&!Rx)", "(|all)" or "(|!Rx)" can be included in each place label.*

Remark 4.2.2. *The template model name can itself be a template by using the tag "Rx" or both the "Rx" and "!Rx." tags.*

Remark 4.2.3. *If a template model name includes both the "Rx" and "!Rx." tags, then that models is denoted an* extended template model*.*

Remark 4.2.4. *A model template can only include the place tags "(&!Rx)", "(|!Rx)" or "Rx" if the model template name is itself a template.*

Each tag as a specific set of rules which states clear how the actual Petri net model is obtained from the template, with each template model generating one Petri net model. The introduction of templates in the template model name allows generating more than one Petri net model from each template model. Extended template models will be useful when modelling explicit communication, with more details given in Section 4.3. Algorithm 4.2.1 specifies how the models are obtained from the templates.

To illustrate the algorithm application and the idea behind model templates, let us rewrite the models used above using model templates. The action `CatchBall` models shown in Fig. 4.1 can be written using a simple extended template model, as depicted in Fig. 4.6. The number of action models will depend on the number of robots included in the multi-robot task plan, having one action `CatchBall` model per robot.

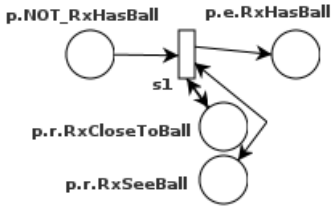


Figure 4.6: Petri net model template for action `CatchBall`, denoted `RxCatchBall`.

To create the template model for the task `Role_Supporter` shown in Fig. 4.2 one needs to use additional tags. In this task a robot should start task `Get_Ball` only if no other robot is close to the ball. As such, it must switch from task `Move_To_Empty_Spot` to task `Get_Ball` when there is no other robot close to the ball, and must switch back when there is at least one robot close to the ball. Such behaviour is achieved using the model template `RxRole_Supporter` depicted in Fig. 4.7.

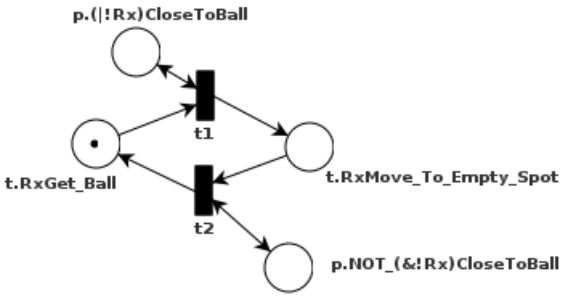


Figure 4.7: Petri net model template for task `Role_Supporter`, denoted `RxRole_Supporter`.

If the task `Role_Supporter` is to be conducted using two robots, instantiating task template `RxRole_Supporter` leads to the two models depicted in Fig. 4.2. However, the given task template is not limited just to two robots tasks, but can be used with any number of robots. For instance, if one were to consider a three robots task, then the instantiation of the template model `RxRole_Suporter` would lead to three tasks, one for each robot. As an example, task `R1Role_Supporter` Petri net model is depicted in Fig. 4.8 for the three robot case.

Finally, as another example, the templates to generate the models depicted in Fig. 4.4 and Fig. 4.5 are shown in Fig. 4.9 and Fig. 4.10, respectively.

Algorithm 4.2.1: Petri net template transformation.

Input: Petri net (extended) template model, PT, and the number of robots, r.
Output: Set of Petri net models, $\mathbb{P} = \{PN(1), \ldots, PN(n)\}$.

```
1  begin
2      all ← {R1, ..., Rr}; // tags for each robot
3      for i = 1 to r do
4          for j = 1 to r − 1 do
5              PN′ = PN((i − 1) ∗ r + j) ← PT;
6              notRx ← all \ Ri;
7              foreach place p_k in PN′ do
8                  if p_k has tag "(&all)" or "(&!Rx)" then
9                      foreach tag r_m in notRx do
10                         Create a new place p′ with the same input and output transitions
                           as p_l;
11                         Label p′ using the label of p_l with the tag replaced by r_m;
12                     end
13                 else // p_k has tag ''(|all)'' or ''(|!Rx)''
14                     foreach tag r_m in notRx do
15                         Create a new place p′;
16                         foreach transition t_l connected to/from p_k do
17                             Create a new transition t′ with the same input and output
                               places as t_l;
18                             Replace the connections from/to t′ to/from p_k, to/from p′;
19                         end
20                         Label p′ using the label of p_k with the tag replaced by r_m;
21                     end
22                 end
23                 if p_j has tag "(&all)" or "(|all)" then
24                     Replace the tag in p_k label with Ri;
25                 else
26                     Remove p_k;
27                 end
28             end
29             if PT is not an extended template model then
30                 Break;
31             else
32                 Label PN′ using the label of PT with tag "!Rx" replaced by notRx(j);
33             end
34         end
35         if PN′ label includes the tag "Rx" then
36             Label PN′ using the label of PT with tag "Rx" replaced by Ri;
37         else
38             Break;
39         end
40     end
41 end
```

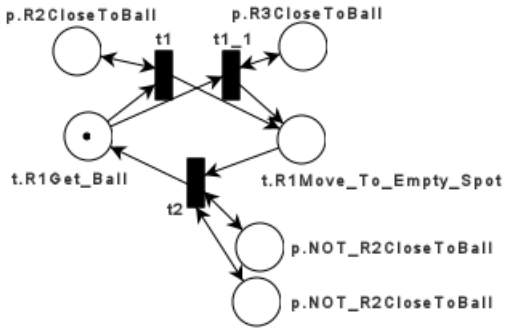


Figure 4.8: Task `R1Role_Supporter` Petri net model template for a three robot case.

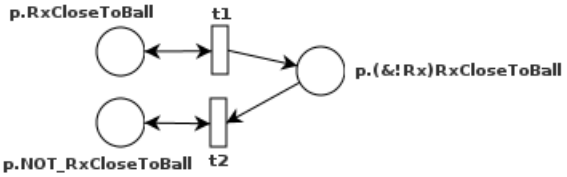


Figure 4.9: Petri net model template for predicate `CloseToBall` propagation with delays in the multi-robot case.

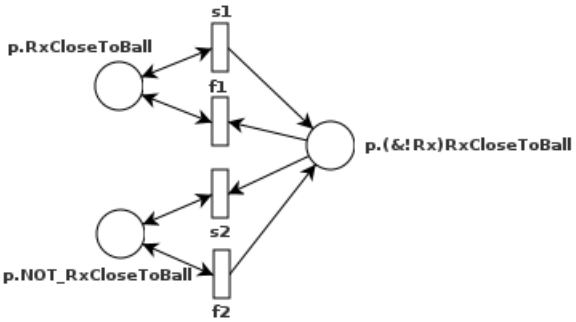
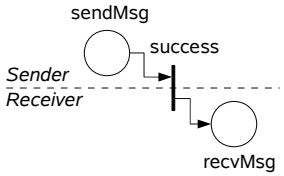


Figure 4.10: Petri net model template for predicate `CloseToBall` propagation with delayed errors in the multi-robot case.

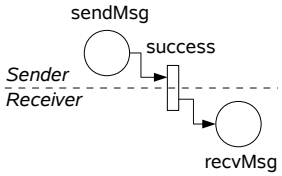
4.3 Modelling Explicit Communication

The major problem when using communication is the time information takes to go from the sender to the receiver, which, theoretically, can go from zero time to infinite time (communication failure). To model communication, three different communication models were considered, which cover this time range. The base concept in these models is that a robot has a predicate place at a given value and wishes to transmit that information to a teammate. The teammate, upon receiving the information, gets its predicate updated to the same value as its teammate, in case of successful communication.

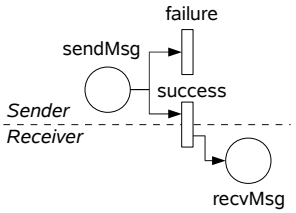
The simplest communication model is presented in Fig. 4.11a. Here the communication is considered instantaneous and always successful. Increasing the model complexity by adding a probabilistic arrival time for the communication, results in the model depicted in Fig. 4.11b. In this case, communications are still considered always successful, but the amount of time it takes varies according to an exponential distribution. The full communication model, which adds the failure possibility to the previous model, is presented in Fig. 4.11c. Here, besides including a varying time delay, there is also the possibility that the message does not reach its destination, thus modelling failures.



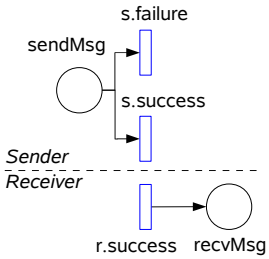
(a) Deterministic communication model without failures.



(b) Communication model with probabilistic time and without failures.



(c) Full communication model with probabilistic time and failures.



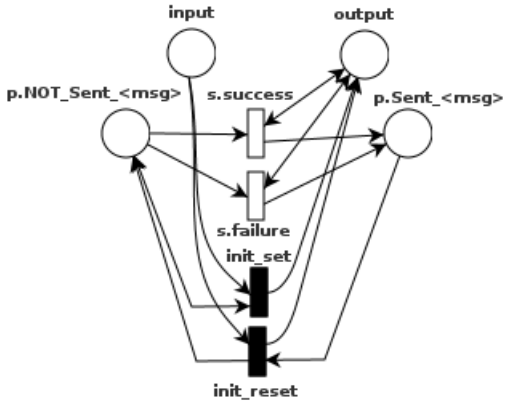
(d) Separate view of the full communication model.

Figure 4.11: Explicit communication models.

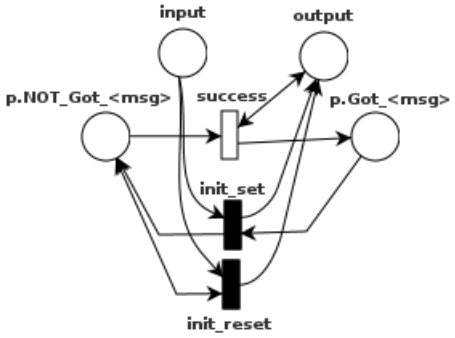
Given the various communication models, one can choose which one to use, according to the context where the model is being applied and the property sought to analyse. When using the explicit communication models, they will be seen in a distributed way to simplify the graphical view, given that there will be a sender model and a receiver model, as depicted in Fig. 4.11d.

4.3.1 Communication Actions

In order to use the communication models to model explicit communication between robots in a multi-robot task plan, a new type of actions is introduced, denoted *Communication Actions*. The structure of this action was designed such that a message can only be successful received if both the sender and the receiver run the sending and receiving action simultaneously, respectively. Fig. 4.12 shows two communication action models using the communication model presented in Fig. 4.11d.



(a) General send communication action.



(b) General receiving communication action.

Figure 4.12: General communication action models.

The models in Fig. 4.12 are not yet suitable, given that the message transition does not occur simultaneously both for the sender and for the receiver. To achieve that behaviour, one needs the models depicted in Fig. 4.13. Here a counter place is used to establish that a robot is running a receiving action. By using that counter as an enabling place of the successful communication transitions in the sender communication action, one guarantees that the communication is successful only if both the sender and receiver are running the corresponding communication actions. Note that this place is called a counter since its number of tokens represents the number of receiving actions running. Nevertheless, since these communication action models are used to model explicit robot to robot communication, it is expected that the designed tasks guarantee that only one receiving communication action of a kind is running at any given time, which can be checked during the analysis phase.

Definition 4.3.1. *A full sending communication action is a GSPN with the structure shown in Fig. 4.13a, where* <msg> *represents the exchanged information, Rs denotes the sending robot, and Rr denotes the receiving robot.*

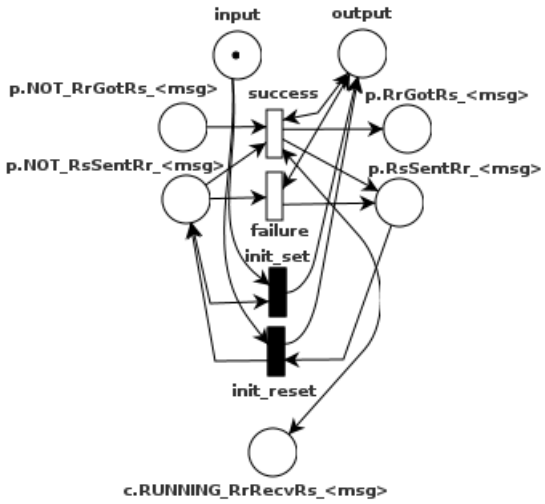
Definition 4.3.2. *A full receiving communication action is a GSPN with the structure shown in Fig. 4.13b, where* <msg> *represents the exchanged information, Rs denotes the sending robot, and Rr denotes the receiving robot.*

Although the shown actions include the full models, i.e., include both exponential distributed delays and failures, it should be clear that any of the described communication models can be used.

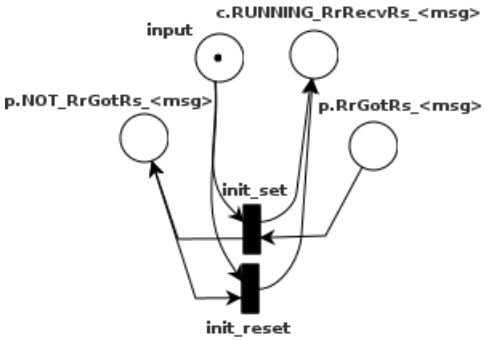
Communication action macro places represent communication actions in the Petri net based task plans. They might be referred to as communications action places, since the macro property is implicit.

As an example, consider a robot $R1$ which needs to communicate the value of its predicate `CloseToBall` to another robot, $R2$, this time using explicit communication. The needed communication action models for this task are depicted in Fig. 4.14.

As a final remark, recall that these models are used for explicit robot to robot communication, hence the need for two action models for each pair of sending and receiving robots. However, one does not need to create all the robot to robot communications actions. By using template models, one just needs to create two models for each type of message, one for the sender, and one for the receiver. For instance, the template models for the models shown in Fig. 4.14 are the ones depicted in Fig. 4.15.

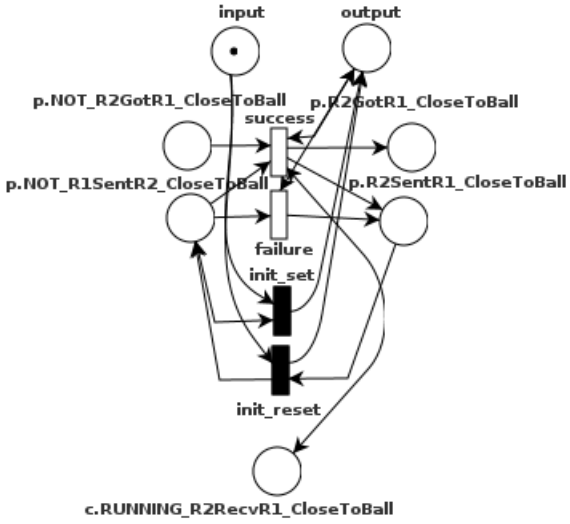


(a) General final send communication action.

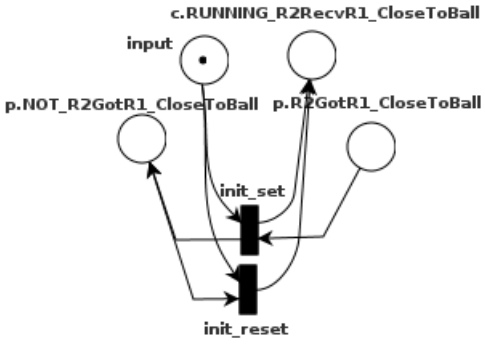


(b) General final receiving communication action.

Figure 4.13: General final communication action models.



(a) `R1SendR2_CloseToBall` communication action.



(b) `R2RecvR1_CloseToBall` communication action.

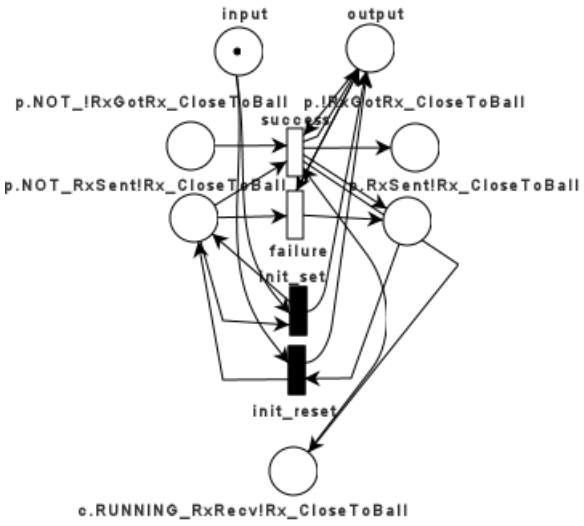
Figure 4.14: Communication actions example.

4.4 Task Plans

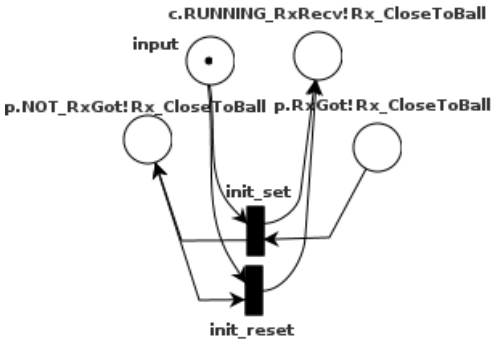
With the introduction of the communication models for both the explicit case and implicit case, specifying a multi-robot task is similar to the individual robot task case. The only difference lies in the inclusion of communication actions and the usage of the tags to denote to which robot the places belong.

As an example, consider a multi-robot task where the goal is to have n robots, with one robot going for the ball and the $n-1$ remaining robots going for an empty spot. The Petri plan for this task using implicit communication is obtained by using the task template model shown in Fig. 4.7 and specifying n as the number of robots.

Although specifying multi-robot tasks can be achieved using the same guidelines as in single-robot robots tasks it is important to include selection mechanisms. These mechanisms prevent having the robot sending direct messages to all robots at all times, but communicate instead



(a) `RxSend!Rx_CloseToBall` communication action template.



(b) `RxRecv!Rx_CloseToBall` communication action template.

Figure 4.15: Communication actions template example.

with a robot, or robots, which have an higher probability of making a commitment (and keep that commitment) in a multi-robot task, by using for instance the state of the world, or implicit communication. It should be possible to manage these commitment mechanism using, for instance, predicates in the proposed framework, however that subject was not yet researched in detail, remaining part of the future work (see Section 8.2 for further details).

Chapter 5

Analysis of Robot Tasks

This chapter details how robot task plans can be analysed through the analysis of the involved Petri net models.

Section 2.5 introduced some properties of interest and analysis methods, with Appendix B having a more thorough explanation on the subject. In order to use those analysis methods, one needs to create a Petri net model of the overall task [1]. Given that all layers are modelled using Petri nets, these can all be composed together into that single Petri net model. This single Petri net model represents the overall task, which can be analysed a priori. The analysis can be both for logical (e.g. deadlocks) and probabilistic (e.g. probability of reaching a given state) performance properties.

Furthermore, there are a number of properties that must be met during design time, which allow for some error detection at an early stage of development. As an example consider the boundedness of the net. Given that we are using predicate places, these can have only one or zero tokens. If one detects more than one token in a predicate place at design time, or that the sum of tokens in the two places associated with a predicate is not always one, it means that there is an error in the models. In the predicate places case, this translates to a simple design rule which states that if a given predicate p is an input place of a transition t, then one, and only one, of predicate places NOT_p or p must be an output place of transition t. If additionally one requires action and task places to have at most one token, it results in a safe net requirement. If the total number of tokens in the two places associated with a predicate is constant (equal to one in this case) they form a place invariant [Murata, 1989], which can also be determined from a priori analysis.

Having the modelling and analysis processes integrated under the same framework allows for a design process based on a continuous loop of design-analysis-design. This loop guides the development of the tasks in a structured way, leading to improved task plans even before gathering results from the execution process.

Furthermore, data can also be extracted from the execution process in order to analyse the task a posteriori, and to further improve the models.

5.1 Expansion Process

The *Expansion Process* enables us to obtain the single Petri net for analysis by merging all the environment, action and task Petri net models. The place labels play an important role in this process, since these allow us to distinguish between the different types of places. The current expansion algorithm was designed to work with Petri nets where all non counter places are safe.

[1] It is possible to obtain some properties without having to generate the full Petri net.

Given that all predicate places are safe by construction, it is thus important that each task Petri net model is safe, i.e., that each memory, task or action place be safe.

When composing the various Petri nets, there are four basic rules that must always be followed:

- Predicate places with the same label are considered the same place;
- Counter places with the same label are considered the same place;
- Action, task and memory places are always different places, regardless of their label;
- All transitions are different, regardless of their label.

Given these rules, whenever two Petri nets are composed, if there are several predicate places (or counter places) associated with the same label, then these are duplicate places and can be merged. Merging these places consists on keeping just one of the places while maintaining all the connections of the removed duplicate places. Similar techniques exist in the literature, such as the *composition by place fusion* in [Girault and Valk, 2003].

5.1.1 Petri Net Complement

Recall that when designing the Petri net models one is not forced to include the two predicate places associated with each predicate. As such, those missing places must be added during the expansion process, through the use of Algorithm 5.1.1, denoted *Complement Algorithm*. The Petri net model which results from complementing Petri net PN will be denoted as $\widehat{PN}$.

Algorithm 5.1.1: Petri net complement algorithm.

Input: Petri net model, PN.
Output: Complemented Petri net model, $\widehat{PN}$.

```
1  begin
2      Merge all duplicate places in PN;
3      foreach place p_j in PN do
4          if p_j is a predicate place then
5              If ¬p_j does not exist yet, add it;
6          else
7              Create a complementary place of p_j, denoted ¬p_j, with marking
               #(¬p_j) = 1 − #(p_j);
8          end
9          foreach input transition t_i of p_j that is not an output of p_j or ¬p_j do
10             Add an arc from ¬p_j to t_i;
11         end
12         foreach output transition t_i of p_j that is not an input of p_j or ¬p_j do
13             Add an arc from t_i to ¬p_j;
14         end
15     end
16 end
```

The complement algorithm is not applicable to communication actions. Since these follow a rigid structure, there is no need to complement them.

As an example, applying the complement algorithm to the Petri net model depicted in Fig. 3.3 results in the model shown in Fig. 3.2 except, purposely, for the place tokens (see predicate places `p.NOT_BallMovingSlow` and `p.NOT_BallStopped`).

Note that the complement algorithm also introduces complementary places to non-predicate places. The importance of these complementary places will be more clear later.

5.1.2 Extended Reachability Graph

Before detailing the actual expansion algorithm, one needs to introduce another auxiliary algorithm, used to compute the *Extended Reachability Graph*. The reachability graph [Murata, 1989] of a Petri net with initial marking $\mathcal{M}_0$, denoted $\mathcal{G}(\mathcal{M}_0)$, allows determining the reachability set, denoted $\mathcal{R}(\mathcal{M}_0)$, i.e., the set of all reachable markings from the given initial marking (see Appendix B for more details on computing the reachability graph).

Definition 5.1.1. *Given a task plan Petri net model PN, the* Extended Petri net model *of that task plan, denoted $\triangle PN$, is obtained by adding transitions to PN such that predicates can switch values in any state. Considering an initial marking $\mathcal{M}_0$, the reachability graph of $\triangle PN$ is the* Extended Reachability Graph *of PN, and is denoted $\triangle\mathcal{G}(\mathcal{M}_0)$. The reachability set of $\triangle PN$ is the* Extended Reachability Set *of PN, and is denoted as $\triangle\mathcal{R}(\mathcal{M}_0)$.*

Algorithm 5.1.2 describes how to create the extended Petri net models that can be used to compute the extended reachability graph.

Algorithm 5.1.2: Petri net extension algorithm.

Input: Petri net model, PN.
Output: Extended Petri net model, $\triangle PN$.

```
1  begin
2      Merge all duplicate places in PN;
3      foreach place p_j in PN do
4          if p_j is a predicate place then
5              If ¬p_j does not exist yet, add it;
6              Add a new transition t' with an arc from p_j to t' and an arc from t' to ¬p_j;
7              Add a new transition t'' with an arc from ¬p_j to t'' and an arc from t'' to p_j;
8          end
9      end
10     Set all positive and negative predicate place markings to 0 and 1 respectively;
11 end
```

As an example, the extended Petri net model of task `Get_Ball` (shown in Fig. 3.9) is depicted in Fig. 5.1.

5.1.3 Reduced Reachability Set

Definition 5.1.2. *Given a reachability set $\mathcal{R}(\mathcal{M}_0)$ of task plan Petri net model PN, for an initial marking $\mathcal{M}_0$, the* Reduced Reachability Set *of PN for the given initial marking, denoted $\triangledown\mathcal{R}(\mathcal{M}_0)$, is obtained by removing all predicate places from the markings. The marking obtained by removing the predicate places is called* reduced marking *and is denoted as $\triangledown\mathcal{M}$.*

Note that no definition was given for the *Reduced Reachability Graph*, as it is not applicable. As such, although it makes sense to compute $\triangledown(\triangle\mathcal{R}(\mathcal{M}_0))$, computing $\triangle(\triangledown\mathcal{R}(\mathcal{M}_0))$ is undefined.

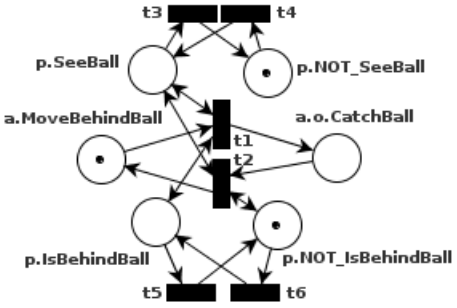


Figure 5.1: Extended Petri net model of the `Get_Ball` task plan.

Definition 5.1.3. *An* Active Reachability Set *is defined as* $\mathfrak{R}(\mathcal{M}_0) \equiv \triangledown(\triangle\mathcal{R}(\mathcal{M}_0))$ *with* $\mathfrak{M}_j$*, the markings of* $\mathfrak{R}(\mathcal{M}_0)$*, denoted as* Active Markings*.*

As an example, consider again the `Get_Ball` task plan. Computing $\mathfrak{R}(\mathcal{M}_0)$ of this task plan, implies that one needs to determine the extended reachability graph of the Petri net model depicted in Fig. 3.9, which implies computing the reachability graph of the Petri net model shown in Fig. 5.1, resulting in the graph presented in Fig. 5.2 (considering that the marking place labels are given by {`MoveBehindBall`, `CatchBall`, `SeeBall`, `NOT_SeeBall`, `IsBehindBall`, `NOT_IsBehindBall`}).

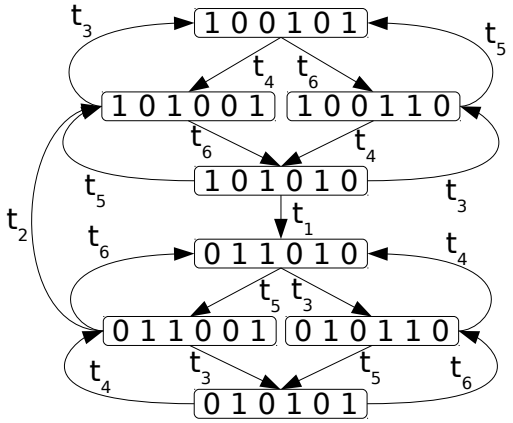


Figure 5.2: Extended reachability graph for the `Get_Ball` task plan Petri net model.

The extended reachability set of the `Get_Ball` task plan Petri net model is given by all the markings shown in Fig. 5.2. Computing the reduced reachability set of the extended `Get_Ball` task plan Petri net model, where the marking is given by {`MoveBehindBall`, `CatchBall`}, yields:

$$\mathfrak{M}_0 = \{1, 0\},\ \mathfrak{M}_1 = \{0, 1\}$$

Having obtained all possible states for a Petri net, independently of the predicate states, will allow any task plan Petri net model to interrupt an included task Petri net model at any marking, by considering all these states, as explained in the next section.

5.1.4 Petri Net Expansion

The actual expansion process is performed using Algorithm 5.1.3. This algorithm was written considering that the resulting Petri net model must follow these guiding principles:

1. A task should always be started in its initial reduced marking;
2. When a task is not enabled, the associated Petri net model reduced marking must only contain zeros;
3. All tasks and actions should be interruptible in zero time;

The last item of the guiding principles enforces that, having decided to terminate a task or an action, no exponential transition should have to occur for that task or action to terminate.

Algorithm 5.1.3: Single Petri net generation algorithm.

Input: Base task (b_{Net}) model and all environment (e_{Net_i}), action and task models
Output: Single, expanded Petri net

```
1  begin
2      Create an empty Petri Net, denoted f_Net;
3      p ← 0; // Base priority
4      foreach environment model, e_Net_i, do
5          Add ê_Net_i to f_Net with priority p, and merge duplicate places;
6      end
7      Add b̂_Net to f_Net with priority p − 1;
8      foreach task place, then each action or communication action place, in f_Net do
9          p ← priority of parent model − 1;
10         if the expanding place is a communication action place then
11             Add the corresponding communication action Petri net model using
               Algorithm 5.1.4;
12         else
13             Compute the complemented Petri net model associated with the place being
               expanded, hereby denoted m̂_Net;
14             Prefix m̂_Net transition labels with m_Net name;
15             if the expanding place is an action place then
16                 Add m̂_Net to f_Net with priority p;
17             else
18                 Add m̂_Net using Algorithm 5.1.5;
19             end
20             Add an arc from the expanding place to all transitions in m̂_Net;
21             Add an arc from all transitions in m̂_Net to the macro place;
22         end
23         Merge all duplicate predicate and counter places;
24         Prefix the macro place label with an "e" to denote that it has been expanded;
25     end
26     if b_Net is a template model then apply Algorithm 4.2.1;
27     Remove the tokens from all predicate places;
28 end
```

The phrase *add Petri net with priority* p means that all immediate transitions of the added Petri net have priority p assigned. The Environment model reflects the world state which is not controlled by the robot, as such, it makes sense that the environment state be update before decisions are made when there are conflicts. For the remaining models, priorities are assigned such that child models immediate transitions have a lower priority. The idea is to model the fact that upper level decisions take precedence over lower level decisions.

Algorithm 5.1.4: Communication actions Petri net model insertion algorithm.

Input: Communication action Petri net model (m_{Net}) and associated macro place, parent model (f_{Net}) and priority ($\mathfrak{p}$)

Output: Parent Petri net model f_{Net} with communication action model inserted

begin
  Add m_{Net} to f_{Net} with priority $\mathfrak{p}$;
  Move all the output connections of the expanding macro place to the *output* place of m_{Net};
  Merge the *input* place of m_{Net} with the expanding macro place, maintaining the marking of the expanding macro place;
end

Algorithm 5.1.5: Task Petri net model insertion algorithm.

Input: Complemented Petri net task model ($\widehat{m_{Net}}$) and associated task place, parent model (f_{Net}) and priority ($\mathfrak{p}$)

Output: Parent Petri net model f_{Net} with task model inserted

begin
  Compute $\mathfrak{R}(\mathcal{M}_0)$ of $\widehat{m_{Net}}$;
  Add $\widehat{m_{Net}}$ to f_{Net} with priority $\mathfrak{p}$;
  foreach *input transition* t_i *of the expanding place in* f_{Net} **do**
    Add an arc from transition t_i to all places containing one token in $\mathfrak{M}_0$;
    Add an arc from all non-predicate NOT_p_j places to transition t_i;
  end
  foreach *output transition* t_n *of the expanding place in* f_{Net} **do**
    foreach $\mathfrak{M}_j$, *with* $j > 0$ **do**
      Add a new immediate transition t'_n with the same input places, output places and priority as t_n;
      Add an arc from each place with one token in $\mathfrak{M}_j$ to transition t'_n;
      Add an arc from transition t'_n to all non-predicate NOT_p_j places;
    end
    Add an arc from each place containing one token in $\mathfrak{M}_0$ to transition t_n;
    Add an arc from transition t_n to all non-predicate NOT_p_j places;
  end
  if *the expanding place has zero tokens* **then**
    Remove the tokens from all places with a token in $\mathfrak{M}_0$ and add a token to all non-predicate NOT_p_j places;
  end
end

The action places function as enabling places of all transitions on the associated models, i.e., if there is a token in the action place, then the transitions of the associated Petri net model are enabled (as long as the *running-conditions* and remaining input predicate places are also true). As such, expanding an action place consists of adding arcs from the action place to all action model transitions and back.

For the communication actions the addition is performed such that one of the reset mechanism transitions is always fired. Having one of those transitions fired, the transitions associated with the actual communication will fire as long as the communication action is enabled.

Task places, besides working also as enabling places for the associated Petri net model, must guarantee that the task is always interruptible and that no actions or lower level tasks keep running when the task is disabled. This is the reason one needs to create complementary places for non-predicate places, and why we need output transitions for every possible reduced marking of the associated Petri net model.

Prefixing the transitions with the model names during the expansion algorithm allows determining to which model they belong when performing the analysis of the final model. Applying the template rules allows the usage of the expansion algorithm for multi-robot tasks.

After having obtained the single Petri net, one needs to choose an initial state for the task by setting the number of tokens in the predicate places. This Petri net can then be analysed using well known techniques, as described in Section 2.5 and Appendix B. For instance, the probability of running a given action is the the sum of the probability of having at least one token in all places associated with that action.

For certain Petri net models some properties of the final Petri net can be obtained from the composed Petri nets.

Proposition 5.1.1. *A composed Petri net model is k-safe if it includes at most k communication actions of a given kind*[2] *and the included tasks are l_i-safe (considering the active reachability set), with $l_i \leq k$ for each task i.*

Proposition 5.1.1 can be verified by going through the various types of models and the expansion process. Consider first an expansion process with no communication actions. Recall Definition 2.4.1 and Definition 2.4.2, which imply that any models using predicate places must guarantee by design that the positive and negative form of the predicate places work as a one token buffer, i.e., with the sum of the tokens of the two places associated with a given predicate is always one. Since during the expansion process all predicate places with the same label are considered the same place, the rule is maintained during the expansion process. Furthermore, given that both the environment and action models, as of Definition 3.2.1 and Definition 3.3.1 respectively, only contain predicate places, these are safe by design.

Looking at the expansion process, particularly when adding environment models, the only rule used is that duplicate predicate places must be merged. Since merging duplicate places means maintaining all the input and output transitions, the safeness will me maintained by this operations, i.e., Definition 2.4.1 will still be valid for all predicate places.

Regarding the expansion of non communication actions, besides merging duplicate predicate places, one needs to add the enabling arcs to all transitions in the action model. Adding these enabling arcs means adding an arc from the action place to each transition in the action model, and another arc from each transition back to the action place. This addition does not change the possible number of tokens in the action place and does not invalidate Definition 2.4.1 for the expanded model, even if considering the active reachability set. As such, expanding a non

[2]By "kind of communication action" one refers to a particular instantiation of a communication action. For instance, `R1RecvR2_<msg>` is one kind of communication action, while `R2RecvR1_<msg>` is another.

communication action model still keeps the expanded model safe. As an example, if a task was formed only by the `CatchBall`, expanding that task would yield the Petri net show in Fig. 5.3.

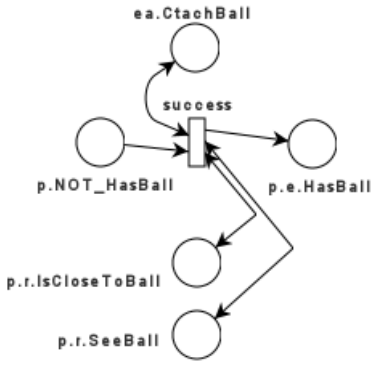


Figure 5.3: Safeness property after expanding a simple task model.

Consider now a task without communication actions. If this task does not include other tasks, but only actions, and the task is safe (considering its active reachability set), then we can conclude that the overall task, associated with the expanded Petri net, is safe since, as seen previously, it does not depend on the non-communication action or environment models. Recall that the expansion process guarantees that each included task is always started in its initial reduced marking and, naturally, cannot have different active markings other than the ones included in its active reachability set before expansion.

If besides including non communication actions a task includes other tasks, in the worst case the overall active reachability set is the active reachability set of the top task times the active reachability set of each included task. This implies that, considering the active reachability set, the maximum number of tokens in each included place is maintained by the expansion process, and thus safeness is also maintained. The same reasoning applies if there is a k-safe task model, with other task models being l_i-safe, for all tasks i, in which case the composed task model is k-safe. Note that this k is a supreme value, as the composed model might be safer due to some active markings not being reachable, as exemplified below.

When communication actions are included, particularly receiving communication actions, a new type of place is included: the counter place. Like predicate places, counter places with the same label are considered the same during the expansion process, but these are not restricted in their marking like predicate places are. In any given task, if k receiving communication actions of a given kind are enabled in any possible active marking, then the corresponding counter place will have k tokens. In this situation, and as long as there is no included task with a higher safe value, the composed task will be k-safe.

Although Proposition 5.1.1 states a sufficient condition, it is not a necessary one, as including an unsafe task model might still yield a safe final Petri net model after the expansion process. As an example consider the task model depicted in Fig. 5.4. The active reachability set of this model would be unbounded, with action `StandBy` always having one token and action `CatchBall` having 0 or more tokens. However, if the environment or action models do not include transitions to get predicate `SeeBall` to get true, transition t_1 will never fire in the final composed model. In this case the composed model would be safe. Nevertheless, given the type of tasks involved, Proposition 5.1.1 provides a sufficient and important result.

Proposition 5.1.2. *If all included tasks are bounded then the composed Petri net model is bounded.*

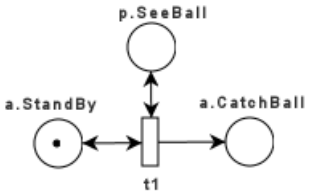


Figure 5.4: Example of an unsafe task yielding a final safe composed model.

Proposition 5.1.2 can be explained using the same reasoning as for Proposition 5.1.1, thus it will not be detailed further.

Chapter 6

Results of the Application to Robotic Soccer

This chapter provides results of the developed framework to some examples in a theoretical robotic soccer scenario, while results using a realistic robotics simulator are provided in Chapter 7. Although all the examples provided are under this scenario, the framework is generic, and not only applicable to these cases. Since the knowledge is represented through predicates, the framework is more adequate to be used in robot tasks where the robot location can be represented using a discrete representation.

RoboCup [Kitano et al., 1997], particularly the robot soccer competition, provides an excellent case test for research in multi-robot systems, where the developed framework can be tested. The aim of this competition is to forest research on artificial intelligence and robotics, with the goal of having a team of human-like robots defeating the FIFA world cup champion by 2050.

In the Middle-Size League (MSL) of RoboCup, the robotic soccer scenario is composed with two teams of robots with 5 to 7 players each, playing on a field with size about 12m by 18m. The ball is a regular orange FIFA site 5 approved ball. Each team has a predefined colour marker, which can be magenta or cyan. Over the years, restrictions are lifted in order to approach the regular human soccer field and rules. As an example, the goals have just recently been changed into regular goals, removing the blue and yellow colours that were previously used to distinguish them. Fig. 6.1 shows a top view of a robotic soccer kick-off.

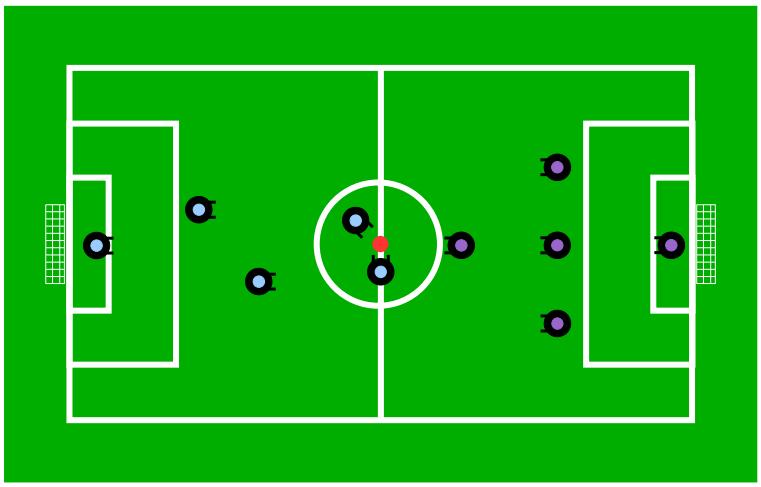


Figure 6.1: Robotic soccer overview.

The Institute for Systems and Robotics at Instituto Superior Técnico has developed a team of robots to participate in RoboCup [Kitano et al., 1997] through the SocRob project [Lima and Custódio, 2005]. Several problems surfaced when developing the team of robots to play robotic soccer, which led in part to the development of this work. As such, the robotic scenario will be used to test and obtain results regarding the developed framework.

In order to be able to execute the task plans developed within the framework, one needs to have a Petri net execution framework. In this work case such framework has been implemented in the decision layer of the MeRMaID middleware [Barbosa et al., 2007]. Furthermore, the implementation allows using the same code both in the real robots and in the simulator. The Petri net execution framework was initially developed by Nelson Ramos, but then extensively extended and improved by the author of this thesis.

In MeRMaID, the sensorial part of the implementation keeps the predicates up to date (at least all the predicates that are relevant at any given state). Given a Petri net based task plan model, the *Petri net Executor* checks which transitions are enabled, considering the current selected actions and enabled predicates, and fires them accordingly. All actions that have tokens at any given moment are the actions that will be enabled. We have also taken advantage of part of the information provided in the Action Executor layer, namely the *running-conditions*, so as to prevent running an action at the lower level when these are not satisfied.

The execution of the tasks can be monitored in order to assert and compare experimental results with the theoretical ones, allowing to check the models for errors or needed improvements.

This chapter provides three different examples: a single-robot theoretical example in Section 6.1 and two multi-robot theoretical examples, one using explicit communications and the other using implicit communication, in Section 6.2. Results using a realistic simulator are provided later in Chapter 7.

6.1 Single-Robot Theoretical Scenario

This section presents results using a theoretical robotic soccer scenario. Several setups are included, showing the impact on using different tasks, different environment models and different observation models.

6.1.1 Base Setup

The base setup includes the models used to study a single-robot soccer playing task where no observation errors or delays are considered. Furthermore, the ball position across the field is considered to change only as a direct effect of the robot actions.

6.1.1.1 Predicates

The following list contains the predicates used for this example:

Ball position: `BallOwnGoal`, `BallNearOwnGoal`, `BallMidField`, `BallNearOppGoal` and `BallOppGoal`;

Robot position: `RobotNearOwnGoal`, `RobotMidField` and `RobotNearOppGoal`;

Other: `SeeBall`, `HasBall` and `CloseToBall`.

The ball position predicates are used to describe the ball position, expecting that only one ball position predicate be true at any given time. As the names suggest, when the ball is inside the robot own goal, `BallOwnGoal` is true, while if the ball is in the opponent goal, then predicate

`BallOppGoal` is true. The other predicates reflect the ball position in the area between the goals, i.e., in the actual field. The same holds for the robot position predicates, except that the robot cannot go inside the goals.

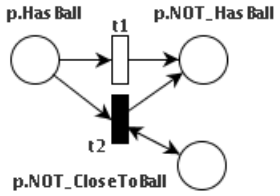
Predicates `SeeBall`, `HasBall` and `CloseToBall` describe, respectively, the robot seeing the ball, having possession of the ball, and being in close proximity of the ball.

6.1.1.2 The Environment Models

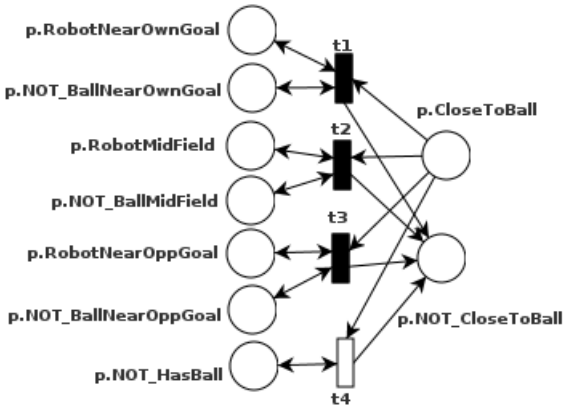
The environment layer include the ball and robot position models, plus additional environment information through the modelling of the predicate status, as shown in Fig. 6.2.

(a) Ball position model.

(b) Robot Position model.



(c) `HasBall` model



(d) `CloseToBall` model

Figure 6.2: Environment models for the base setup.

Has can be seen, the robot and ball position do not change as a result of some other robot actions (or other phenomenon). However, it was considered that whenever the robot has the ball possession or is close to the ball, there is a probability of loosing the ball or its proximity, respectively. The `CloseToBall` model also includes transitions to maintain the expected logic properties of predicate `CloseToBall`, i.e., if the robot and ball are not in the same area of the field, then it is not possible for the robot to be close to the ball. Similarly, the `HasBall` model includes a transition to maintain the correct logic relation with predicate `CloseToBall`, as it is not possible for the robot to have the ball possession if it is not close to it.

The rates used in the exponential transitions for the various models are 1/10 for all transitions, except for the `HasBall` model, where 1/5 was used. Note that this setup uses theoretical models and that the rate values were not computed from any real data. Nevertheless, time was

Action	Running-conditions	Desired-effects
StandBy	-	-
Move2Ball	SeeBall	CloseToBall
CatchBall	SeeBall, CloseToBall	HasBall
Dribble2Goal	HasBall	RobotNearOppGoal, BallNearOppGoal
Kick2Goal	HasBall	BallOppGoal

Table 6.1: Action properties.

considered to be measured in seconds, meaning that an exponential transition with a rate of 1/5 will fire in average 1/5 times per *second* when enabled or, if referring in terms of delay, will have an average firing delay of 5 seconds once enabled. For instance, for the `CatchBall` action, this means that catching the ball takes an average of 5 seconds to be achieved.

6.1.1.3 Action Executor Models

Five different actions were used for this setup:

`StandBy` The robot does nothing, i.e., does not perform any change on the environment;

`Move2Ball` Get close to the ball;

`CatchBall` Grab the ball, if close to it;

`Dribble2Goal` Take the ball to near the opponent goal while avoiding obstacles;

`Kick2Goal` Kick towards the goal.

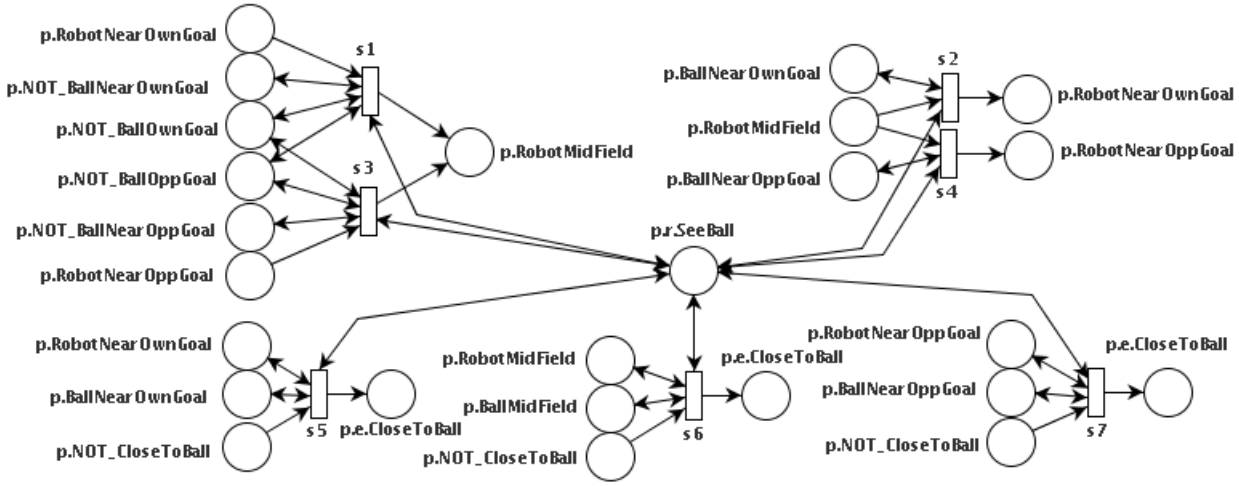
Table 6.1 gives a summary of the actions *running-conditions* and *desired-effects*. While Fig. 6.3 contains the actual models. Recall that the *running-conditions* and *desired-effects* information is available in the predicate labels of the action models, as explained in Section 3.3, and as can be seen from Fig. 6.3. The `StandBy` action model is not shown because it is an empty model, i.e, since it does not perform changes in the environment, it does not contain any transition. The `CatchBall` action model is the one depicted in Fig. 3.8. Note the use of label s_n for success transitions, and f_n for failure transitions.

The `Move2Ball` action is used by the robot to get near the ball. The model basically makes the robot position predicates change towards the ball position predicate that is true, as long as the robot sees the ball. It includes additional tests to avoid the robot moving to the ball when it is inside a goal.

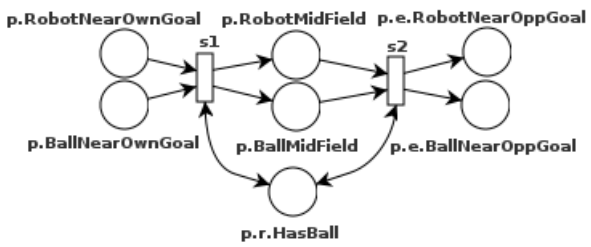
The `CatchBall` action purpose is to grab the ball when the robot is close to the ball. As such, it makes the predicate `HasBall` become true, as long as the robot sees the ball and is close to the ball.

The `Dribble2Goal` action is used by the robot to take the ball from its current position to near the opponent goal, thus changing the robot and ball position predicates until `BallNearOppGoal` and `RobotNearOppGoal` predicates become true, as long as the robot has the ball.

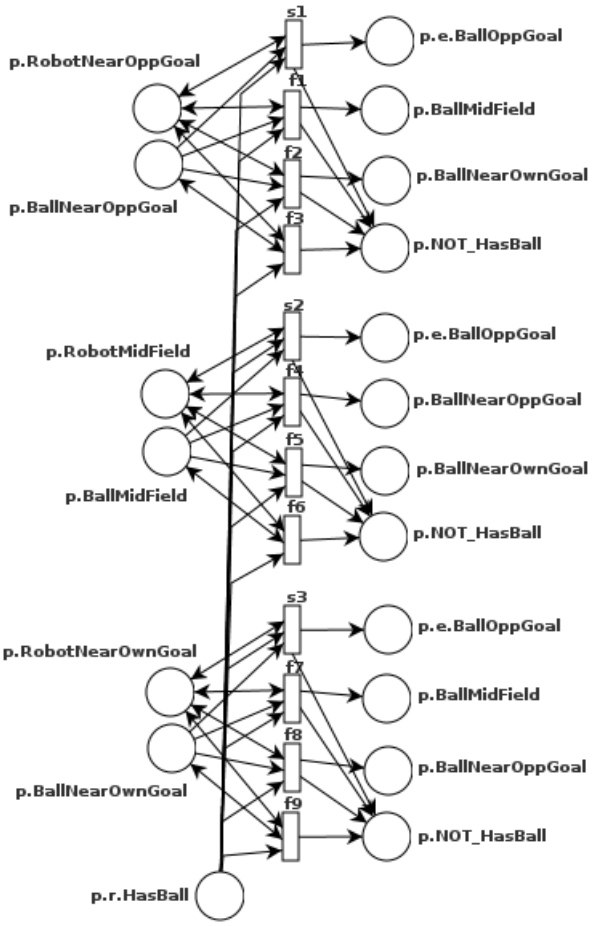
Action `Kick2Goal` purpose is to score a goal, making the predicate `BallOppGoal` become true, as long as the robot has the ball. While actions `StandBy`, `Move2Ball`, `CatchBall` and `Dribble2Goal` do not explicitly include failures, the `Kick2Goal` action models does so. Action `Kick2Goal` explicitly models the fact that the robot can shoot towards the goal from any place of the field, but the ball can end in any place of the field. Transitions s_i correspond to success



(a) `Move2Ball` action model.



(b) `Dribble2Goal` action model.



(c) `Kick2Goal` action model.

Figure 6.3: Action models

transitions, while transitions f_i correspond to failures. Setting an higher rate to transition s_1 then s_2 and s_3, models an higher scoring probably when closer to the opponent goal.

Although the other actions do not explicitly include failures, they can still fail through the environment model transitions. For instance, the predicate `HasBall` can become false at any time (see Figure 6.2c), leading to a failure of actions such as `CatchBall` and `Dribble2Goal`.

The rates used in the exponential transitions for the actions models are as follows: 1.0 for all success transitions, except for transitions s_2 and s_3 in action `Kick2Goal`, where 1/4 and 1/8 was used respectively; 1/4 for the failure transitions.

6.1.1.4 The Task Plans

The goal in a robotic soccer scenario is to score goals in the opponent goal, and not let the opponent score the robot own goal. Currently the opponent is not directly modelled, but only considered as part of the stochastic environment. As such, the consideration for now was only to build task plans in order to score in the opponent's goal.

The base task plan for this example is the `Score_Goal` task plan depicted in Fig. 3.10, which used the task plan `Get_Ball` depicted in Fig. 3.9 and the actions described in the precious section.

Task `Score_Goal` includes a random switch, formed by transitions t_3 and t_4, when the robot is running the `Get_Ball` task and gains possession of the ball. In this situation the robot can switch to action `Dribble2Goal` or `Kick2Goal`, with the probability depending on the weight of the two transitions. As such, three different cases were studied and compared, by using different weights for these two transitions, which represent three different tasks:

Shoot_First: by assigning weight 0 to transition t_3 and weight 1 to t_4, t_3 will never fire, meaning the robot goes from action `CatchBall` to action `Kick2Goal` without going through action `Dribble2Goal`, thus kicking to the goal as soon as it grabs the ball;

Shoot_50_50: by assigning weigh 1 to transitions t_3 and t_4, the robot chooses one of `Dribble2Goal` and `Kick2Goal` with probability 0.5, as soon as it grabs the ball while running action `CatchBall`;

Shoot_Later: by assigning weight 1 to transition t_3 and weight 0 to t_4, t_4 never fires, meaning the robot runs action `Kick2Goal` after having run action `Dribble2Goal` successfully. The robot will only kick the ball when it has possession of the ball and it is near the opponent goal;

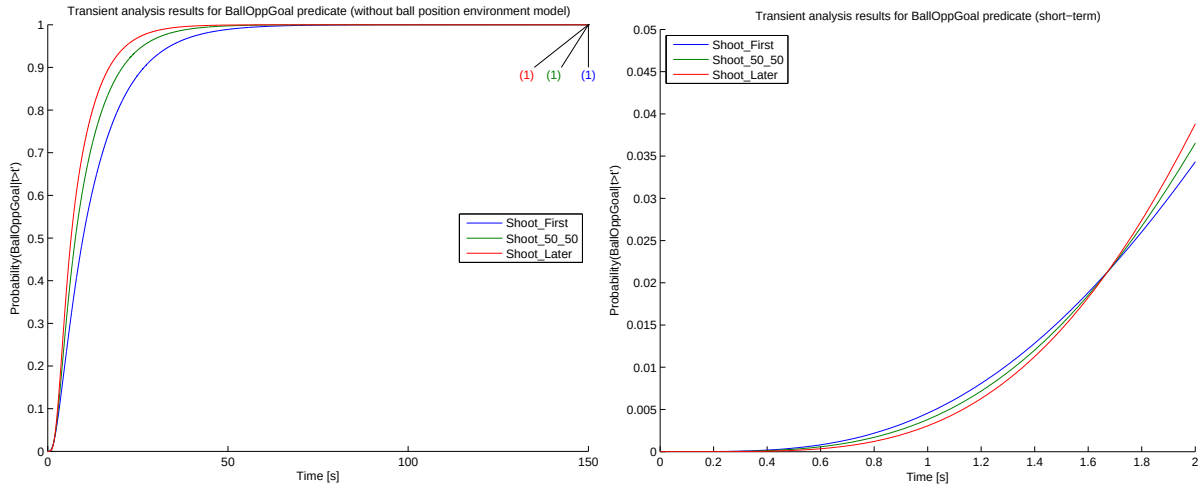
6.1.1.5 Results

For this base setup, it was considered that the robot could always see the ball, and that the robot and ball were placed initially near its goal and the field center respectively, resulting in the following initial predicate state: `NOT_BallOwnGoal`, `NOT_BallNearOwnGoal`, `NOT_BallMidField`, `BallMidField`, `NOT_BallNearOppGoal`, `NOT_BallOppGoal`, `RobotNearOwnGoal`, `NOT_RobotMidField`, `NOT_RobotNearOppGoal`, `SeeBall`, `NOT_HasBall` and `NOT_CloseToBall`. All the following setups for the theoretical single-robot scenario follow these considerations.

Since none of the actions performs changes on the environment when the ball is inside a goal, one can expect the task to include deadlocks, corresponding to scored goals. Furthermore, given that no transition to the `BallOwnGoal` was included, there can only be goals in the opponent goal. Qualitative analysis of the full task model confirmed that expectation, resulting in deadlock states where a goal was always scored in the opponent goal. Each task was determined to be safe, having at most on token per place. Each pair of predicate places formed an invariant

as expected, and all five actions also formed an invariant, given that one, and only one action, runs at any given time.

Given that both three tasks only include deadlocks with a goal being scored in the opponent goal, the long term probability of having one token in predicate places `BallOppGoal` and `BallOwnGoal` is 1 and 0 respectively. This means that comparing the steady state analysis for these two predicates for the three tasks is not much useful. In such cases the transient analysis proves to be more interesting.



(a) Probability of scoring in the opponent goal.

(b) Probability of scoring in our goal.

Figure 6.4: Score goal probability evolution.

As Fig. 6.4a shows, although the stationary probability of scoring in the opponent goal is one for all tasks, it takes longer to reach that probability when shooting first. On the other hand, shooting later means reaching the stationary state faster. This is an expected result, since the `Kick2Goal` action has a lower probability of scoring when kicking far from the opponent goal.

Investigating what happens in the begging of time, the results are the opposite. As Fig. 6.4b shows, the short-term probability of scoring in the opponent goal is higher when the robot runs task `Shoot_First`, which is also an expected result given that the ball travels faster than the robot. If the robot and ball are far from the opponent goal, the ball will reach the opponent goal faster if the robot kicks it, instead of dribbling the ball and shooting later.

This is one example of interesting a priori results one can obtain using this framework. This knowledge can then be used in runtime, for instance, to change the weights of transitions t_3 and t_4 according to the score status and the game time left.

6.1.2 Base Setup with Uncontrolled Ball Position

This second example is identical to the previous one, but considers that the ball position can change without being a direct result of the robot actions. Such behaviour is achieved by introducing the environment model depicted in Fig. 6.5. In this model, the ball position can change at any time as long as the robot does not have possession of the ball.

The rate used in the exponential transitions for the the ball position models is 1/10.

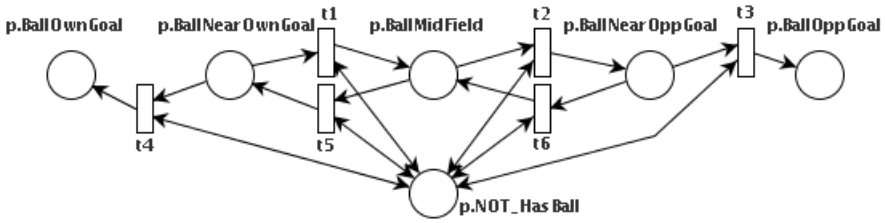
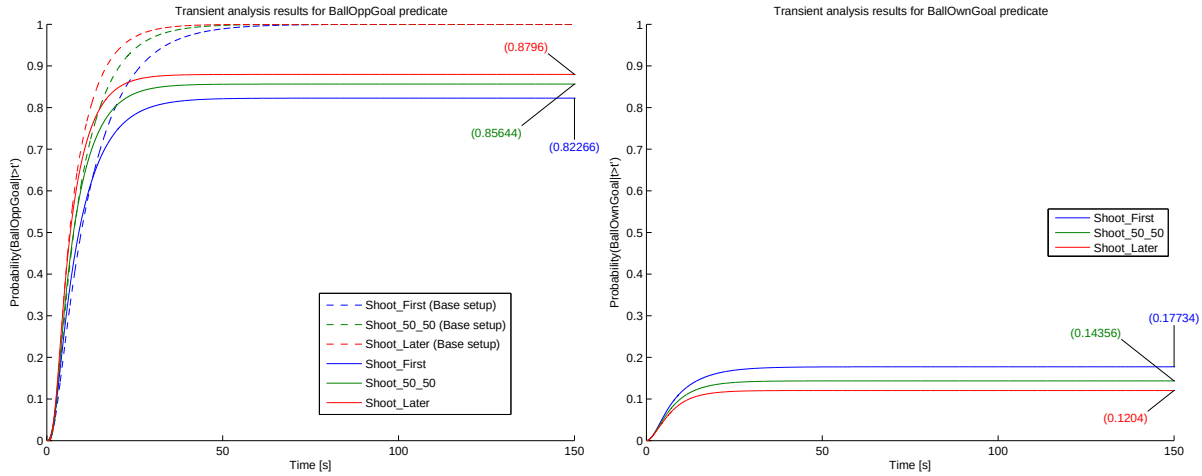


Figure 6.5: Ball position model.

6.1.2.1 Results

Performing the transient analysis as in the previous case yields the results shown in Fig. 6.6. Given that there are uncontrollable transitions which might lead to a goal being scored in the robot own goal, the probability of scoring in the opponent goal is no longer one. Given that these uncontrollable ball position transitions can only occur if the robot is not in possession of the ball, dribbling the ball closer to the goal yields less risks to the robot, leading to a higher stationary probability of scoring in the opponent goal when shooting later, as detailed in Fig. 6.6a.



(a) Probability of scoring in the opponent goal.

(b) Probability of scoring in our goal.

Figure 6.6: Score goal probability evolution with uncontrolled ball position.

The evolution of having a goal scored in the robot own goal is depicted in Fig. 6.6b. As expected, shooting first leads to an higher probability of having a goal scored in the robot goal at all times.

The short-term results are presented in Fig. 6.7. The relation between the outcome of the three different tasks with the uncontrolled ball position model is identical to the one in the previous example. Comparing each task outcome with the previous example results that the task with the uncontrolled ball position model leads to higher scoring probabilities in the short-term. This might be explained by the fact that the model in Fig. 6.5 includes both a transition to the robot goal and the opponent goal, associated to the fact that the uncontrollable transitions are enabled only when the robot does not have possession of the ball.

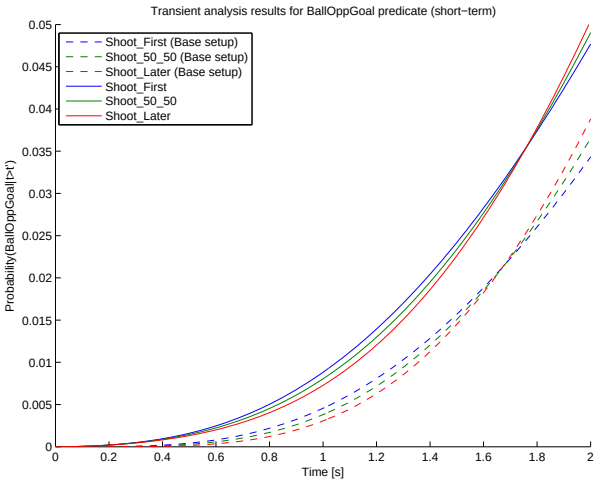


Figure 6.7: Probability of scoring in the opponent goal (initial time instants).

6.1.3 Base Setup with Uncontrolled Ball Position and Observability Failures

This last case studies a single-robot scenario involving observability problems, by using the observation models detailed in Section 3.2.1 for the predicate `CloseToBall`.

In order to analyse the task using observation models for predicate `CloseToBall` one must first create an additional predicate which reflects the robot observation of the actual predicate, here denoted `R1CloseToBall`. The robot decisions based on the `CloseToBall` predicate must also be updated to use the `R1CloseToBall` instead, which in the present case involves updating the `Get_Ball` task to the model presented in Fig. 6.8. Note that the action models remain unchanged, since the conditions needed for these to have an impact on the world do not depend on the robots observations but on the true state of the world.

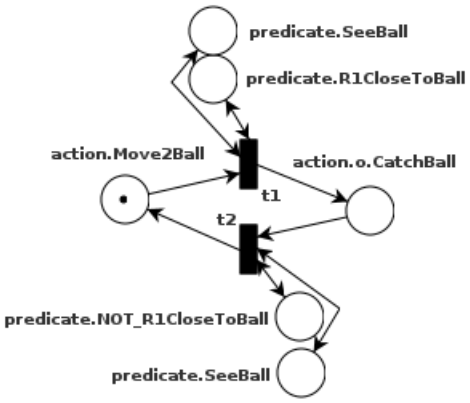


Figure 6.8: Petri net model of the `Get_Ball` task plan for the observability failure setup.

Four different setups were considered by using the two different observation models depicted in Fig. 6.9 with different rates, together with the `Shoot_Later` task plan model:

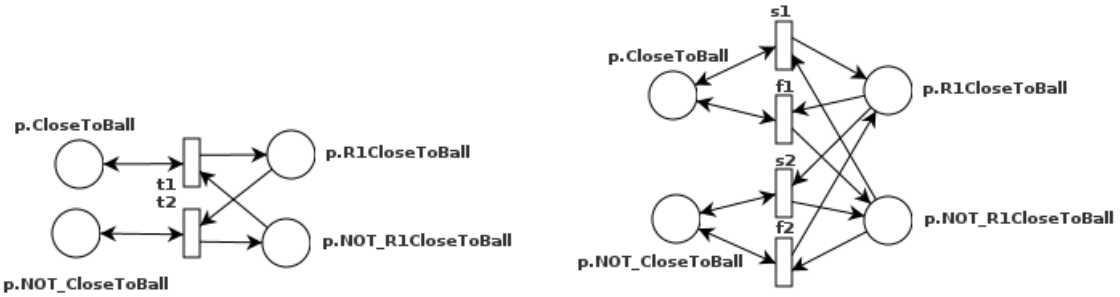
No Observation Errors or Delays The previous setup with the ball position model including the uncontrollable transitions;

Observation delay The same setup as in the previous item but with the `Get_Ball` task shown in Fig. 6.8 and the observation model shown in Fig. 6.9a. The rate used for transitions t_1 and t_2 of the observation model is 10;

Larger observation delays The same setup as in the previous item but with the observation model transition rates decreased to 1;

Observation errors and delays The same setup as in the previous item, but with the observation model replaced by the one depicted in Fig. 6.9b. The transitions in the observation model have rates 1.0 for s_1 and s_2, and 0.1 for f_1 and f_2;

Larger observation errors and delays The same setup as in the previous item, but with the rates of transitions f_1 and f_2 increased to 0.5;



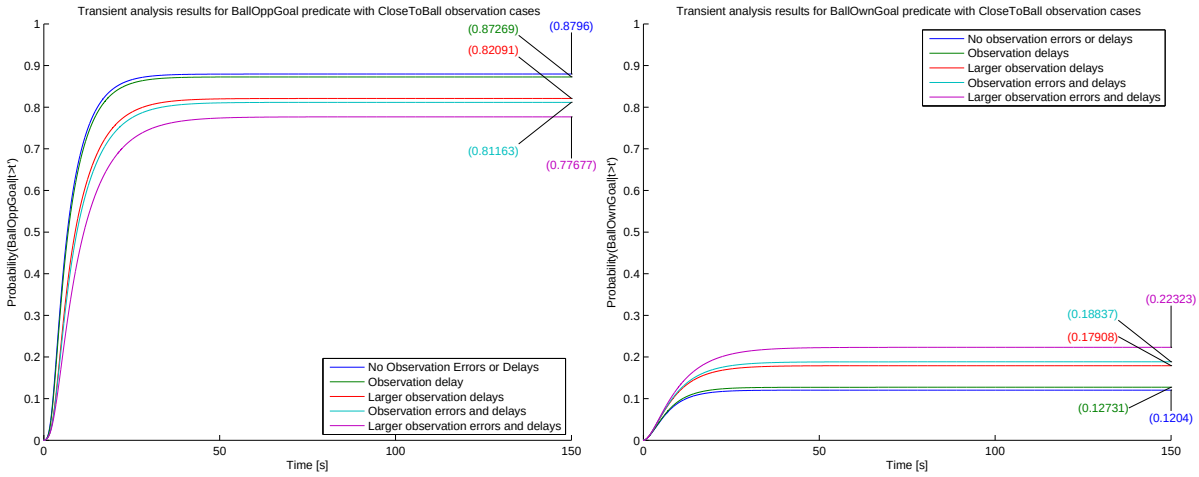
(a) Observation model with delays for the `CloseToBall` predicate.

(b) Observation model with delays and errors for the `CloseToBall` predicate.

Figure 6.9: Observation models for the `CloseToBall` predicate.

6.1.3.1 Results

Performing again a transient analysis for each of these setups yields the results shown in Fig. 6.10.



(a) Probability of scoring in the opponent goal.

(b) Probability of scoring in our goal.

Figure 6.10: Score goal probability evolution with different observation models.

The results in Fig. 6.10 show that if the observation delay is very small then the impact on the task outcome is almost negligible. Increasing the observation delay to a value comparable with the values used in the actions transitions leads to a higher decrease on the probability of

scoring in the opponent goal. Introducing low rate observation errors on top of the delays also does not affect much the outcome of the task. Only when the failure rates are increased to a value in the order of magnitude of the exponential transition rates, does the outcome of the task show a relevant decrease on the probability of scoring in the opponent goal.

The lessons to be learned here is that observation models should only be used when their delays and/or failure rates are significant. Since introducing these models leads to a larger state space, they should only be used when there is an expected relevant impact on the task performance and/or logical properties.

6.2 Multi-Robot Theoretical Scenario

Nothing prevents using explicit and implicit communication models simultaneously, however, to better illustrate the application of each case, two different setups are exemplified: one where explicit communication is used (Section 6.2.1) and one using implicit communication (Section 6.2.2).

6.2.1 Multi-Robot Example Using Explicit Communication

For the explicit communication application example consider a pass between two robots, the kicker and the receiver.

Given two tasks, `coordinatedKick`, for the kicker, and `coordinatedReceive`, for the receiver, a two-robot PASS task plan corresponds to a single `coordinatedPass` relational task, which consists of running both individual tasks in parallel, one in each robot. The key here is to make sure that both individual tasks run synchronously, either by implicit or explicit communication.

In this example it is assumed that the decision to commit the two robots in the coordinated pass was already made, and focus on the task execution analysis, keeping the critical sections synchronised.

This example uses the same list of predicates used in the single-robot example (see Section 6.1.1.1), plus the predicates associated with the communication actions. The environment layer models used are the same ones as used in the single-robot base setup as depicted in Fig. 6.2. However, given that this is a multi-robot task, there will be a `CloseToBall` and `HasBall` for each robot.

The multi-robot PASS task plan uses actions `StandBy`, `Move2Ball` and `CatchBall`, used previously in the single robot example base setup (see Fig. 6.3), but adapted to the multi-robot case. Additionally, the following actions were used:

`Dribble2KickerPosture:` The robot dribbles to the kicker posture to be ready to pass the ball (Fig. 6.11b), which is always considered to be near its own goal;

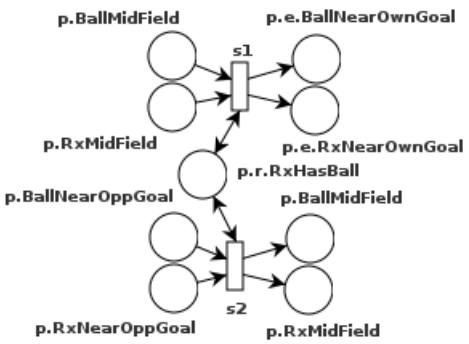
`Send_Ready2ReceiveBall:` The pass receiver acknowledges that it is ready to receive the ball;

`Recv_Ready2ReceiveBall:` Waits for a communication from the receiver to know it is ready to receive the ball;

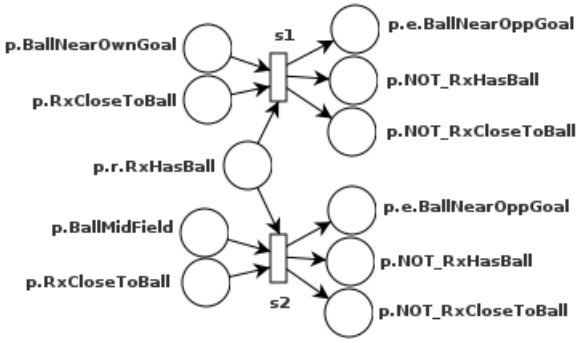
`PassBall:` Passes the ball to another robot. It was considered that passes are only done from near the own goal or the midfield to near the opponent goal (Figure 6.11c);

`Go2ReceiverPosture:` The robot moves to a destination posture, which is good for receiving the ball. The receiving posture was considered to be always near the opponent goal (Figure 6.11a).

(a) `RxGo2ReceiverPosture` action template model.



(b) `RxDribble2KickerPosture` action template model.

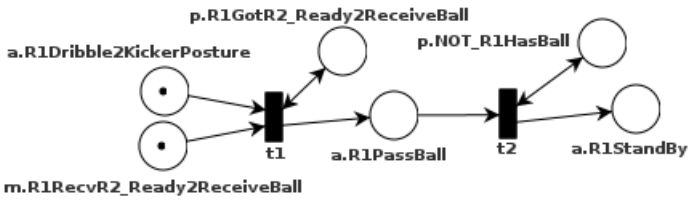


(c) `RxPassBall` action template model.

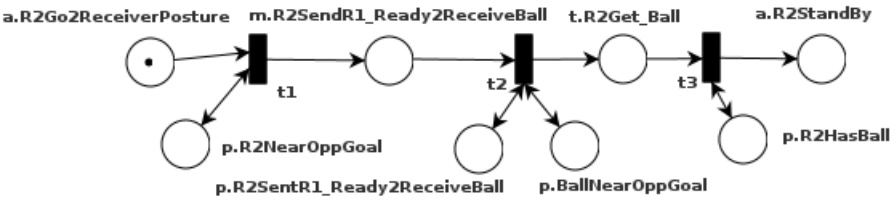
Figure 6.11: Action template models used in the multi-robot example.

Note that Fig. 6.11 shows the actions templates, used to generate the action themselves.

Task `Coordinated_Kick` is obtained by running actions `Dribble2KickerPosture` and `Recv_Ready2Receive` in parallel, followed by action `PassBall` upon getting predicate `Got_Ready2ReceiveBall` to true. Task `Coordinated_Receive` is formed by a sequence of actions, starting with `Go2ReceiverPosture`, followed by `Send_Ready2Receive` when predicate `NearOppGoal` gets true, and ending with action `CatchBall` when `Sent_Ready2ReceiveBall` gets true. Regarding communication, the relevant actions for the `coordinatedPass` multi-robot task are `Recv_Ready2ReceiveBall` and `Send_Ready2ReceiveBall`. These two communication actions follow exactly the structure shown in Fig. 4.13, but with "<msg>" replaced by *Ready2ReceiveBall*. The Petri net models of both tasks are depicted in Fig. 6.12a and Fig. 6.12b (recall that $R1$ is the kicker and $R2$ the receiver).



(a) Task `Coordinated_Kick` model.



(b) Task `Coordinated_Receive` model

Figure 6.12: Task models used in the multi-robot example.

Note that for depicting the Petri net models of tasks `Coordinated_Kick` and `Coordinated_-`

`Receive` no templates were used. Although templates are useful to describe actions and communication actions for multi-robot scenarios, they are not always suitable for designing multi-robot tasks with explicit communication. Something more is needed, which relates to the commitment mechanisms, as discussed later in Chapter 8.

Given that the focus here is on the analysis of the execution of the multi-robot task, without using yet selection or commitment mechanisms, both robots were considered to be already set up for the execution of the pass. As such, the pass between the two robots can be obtained through the PASS task plan depicted in Figure 6.13.

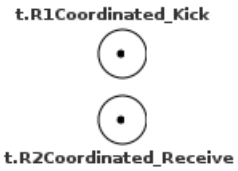


Figure 6.13: Multi-robot PASS task plan.

6.2.1.1 Results

The setup used for the results consisted on placing both robots in the midfield area, with robot R1 holding the ball, resulting in the following initial predicate state: `NOT_BallOwnGoal`, `NOT_BallNearOwnGoal`, `BallMidField`, `NOT_BallNearOppGoal`, `NOT_BallOppGoal`, `NOT_R1NearOwnGoal`, `R1MidField`, `NOT_R1NearOppGoal`, `R1SeeBall`, `R1HasBall`, `R1CloseToBall`, `NOT_R1GotR2_Ready2ReceiveBall`, `NOT_R2NearOwnGoal`, `R2MidField`, `NOT_R2NearOppGoal`, `R2SeeBall`, `NOT_R2HasBall`, `NOT_R2CloseToBall`, and `NOT_R2SentR1_Ready2ReceiveBall`.

The PASS task plan success probability was analysed by monitoring the number of tokens in place `action.R2StandBy`. Since robot R2 only reaches action `StandBy` if it was able to successfully receive the ball, reaching this action means the PASS task plan was successful.

The first results were conducted considering a deterministic environment (by removing all the stochastic transitions from the environment models). Given that no failures were explicitly included in the action models, the only failure in this case is the communication failure. As such, the plan success probability should depend only on the relation between the communication failure and success rates, yielding:

$$P_{Plan\ success} = \frac{\lambda_{comm\ success}}{\lambda_{comm\ success} + \lambda_{comm\ failure}}$$

Exp. #	Action success rates	Comm. success rates	Comm. failure rates	Plan success probability
1	1	1	1	0.50
2	1	1	10	0.09
3	1	10	1	0.91
4	1	10	10	0.50
5	10	10	10	0.50

Table 6.2: Plan success probability vs transition rates with deterministic environment.

Table 6.2 shows the results obtained with different transitions rates for this setup, confirming the above statement. The graph showing the expected number of tokens in place

`action.R2_StandBy` over time is shown in Fig. 6.14 for experiments 1, 4 and 5. This graph shows that, although the stationary plan success probability only depends on the communication rates, increasing the success transition rates leads to a performance improvement in the short term.

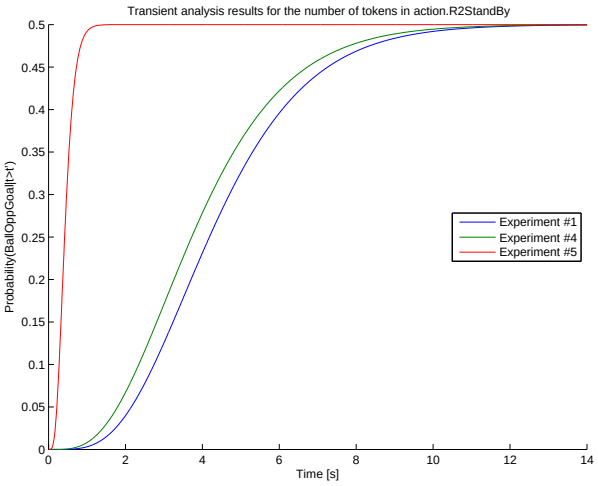


Figure 6.14: PASS task plan success probability over time for different transition rates.

The following experiments include additional failures by using the full `HasBall` and `CloseToBall` environment models for each robot, as shown in Fig. 6.3. The ball position model was kept deterministic, without stochastic timed transitions. This setup was tested with different transition rates, obtaining the results shown in Table 6.3.

Exp.	**Env. rates**		**Action**	**Comm.**	**Comm.**	**Plan success**
#	**HasBall**	**CloseToBall**	**suc. rates**	**suc. rates**	**fail rates**	**probability**
1	0.2	0.1	1	1	1	0.32
2	0.2	0.1	1	1	10	0.06
3	0.2	0.1	1	10	1	0.62
4	0.2	0.1	1	10	10	0.34
5	0.2	0.1	1	100	0.1	0.69
6	0.2	0.1	1	10000	0.0001	0.69
7	0.2	0.1	10	10000	0.0001	0.96

Table 6.3: Plan success probability vs transition rates with probabilistic environment.

In this case, increasing the communication success also increases the plan success probability as expected, but only to a certain point, as experiments 5 and 6 show. Only by increasing the remaining action transitions success rate can the plan success probability increase further. In experiment 7, the success rates are much higher than the failure rates, leading to an almost 100% success probability.

Qualitatively, and like in the single-robot example, the task was determined to safe, and the predicate places form place invariants. Furthermore, as expected, both setups end always in deadlock, given that the tasks are sequential.

6.2.2 Multi-Robot Example Using Implicit Communication

This setup shows an example of a multi-robot task using implicit communication and its results. It consists on a multi-robot task with three robots where the goal is to have, at any given time,

one robot trying to score a goal, while the other two robots have a supporting role. The supporting robots try to move to an empty spot in the field.

The list of predicates used for this setup was the same as for the single robot case (see Section 6.1.1.1), but adapted for each robot. The environment models used were the ones shown in Fig. 6.2, adapted for the multi-robot case and with the stochastic transitions removed, as to not include failures in the environment models. An additional environment model, denoted `BallReset` was added to enable the robots to play continuously. The template for this model is depicted in Fig. 6.15.

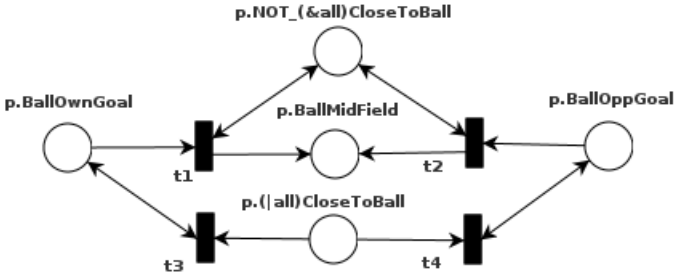


Figure 6.15: `BallReset` environment template model.

The multi-robot task plan is achieved by running the `Play_Coordinated_Soccer` task in each robot, whose template model is depicted in Fig. 6.16.

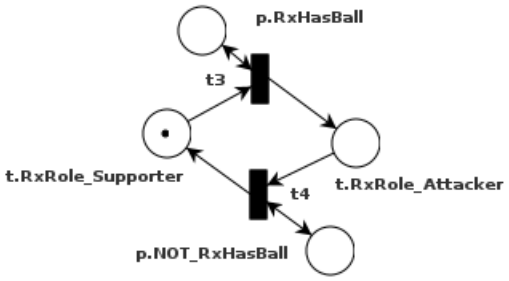


Figure 6.16: `RxCoordinated_Soccer` task template model.

The `Role_Supporter` task template model is the one depicted in Fig. 4.7, while the `Role_Attacker` task template model is depicted in Fig. 6.18a. The `Move_To_Empty_Spot` task template model is shown in Fig. 6.18b.

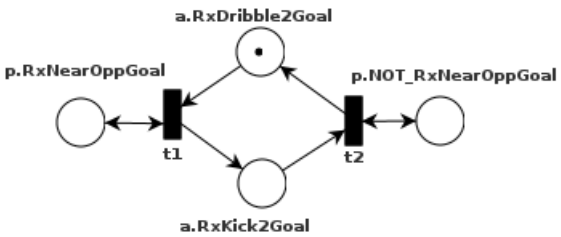
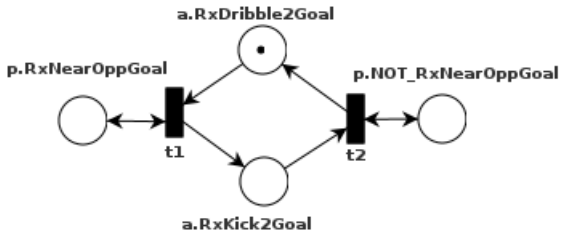
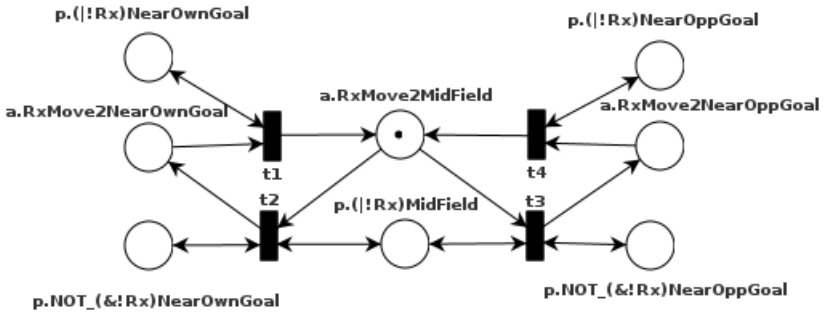


Figure 6.17

The actions used in this setup are `Move2Ball`, `CatchBall`, `Dribble2Goal` and `Kick2Goal`, which were already used in previous examples, with their models shown in Fig. 6.3, plus actions `Move2NearOwnGoal`, `Move2MidField` and `Move2NearOppGoal`, with their template models depicted in Fig. 6.19.



(a) `RxRole_Attacker` task template model.



(b) `Move_To_Empty_Spot` task template model

Figure 6.18: `RxRole_Supporter` and `Move_To_Empty_Spot` task template models.

The rates used in these new action models stochastic transitions were the same as in the previous models, i.e., 1.0.

Three different case were analysed as to assert the implication of observation failures in a multi-robot scenario:

A Considers no observation failures;

B Considers observation delays for predicate `CloseToBall` with rate 10;

C Considers observation delays and errors for predicate `CloseToBall`, with rate 10 for transitions s_1 and s_2 and rate 0.5 for transitions f_1 and f_2.

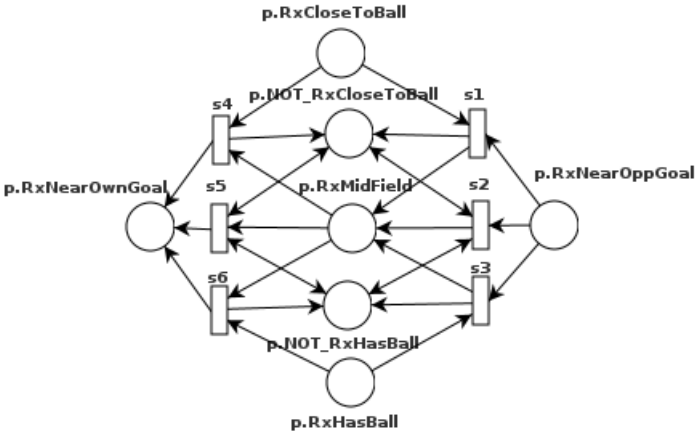
As it was done in Section 6.1.3, in order to consider observation failures in test cases B and C, additional predicates were introduced to reflect the observation of each robot proximity with the ball by other robots. Test cases B and C include an environment model per robot, following the template model depicted in Fig. 4.9 and Fig. 4.10, respectively.

6.2.2.1 Results

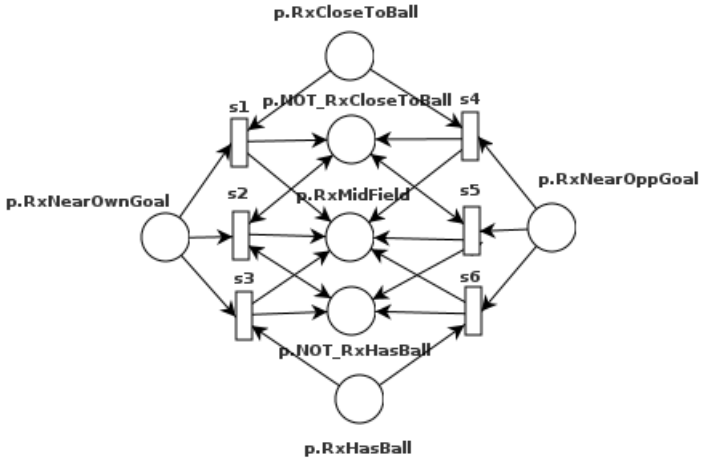
The multi-robot task plan of this example was designed such that only one robot should be trying to catch the ball at any given time. This property can be verified by analysing the number of tokens in places `action.R1CatchBall`, `action.R2CatchBall` and `action.R3CatchBall`.

Exp.	0 robots	1 robot	2 robots	3 robots
A	0.73	0.27	0	0
B	0.74	0.26	9.28E-4	1.47E-5
C	0.75	0.25	1.11E-3	1.69E-5

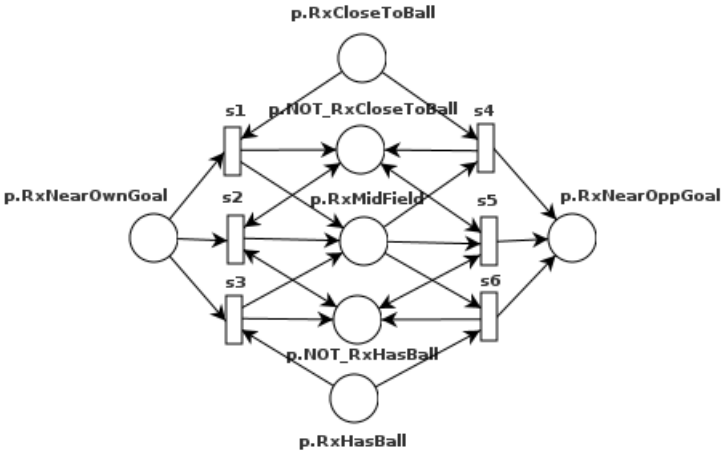
Table 6.4: Probability of a given number of robots trying to catch the ball, per experiment.



(a) `RxMove2NearOwnGoal` action template model.



(b) `RxMove2MidField` action template model.



(c) `RxMove2NearOwnGoal` action template model.

Figure 6.19: Additional action template models.

Table 6.4 shows the probability of having 0, 1, 2 or 3 robots trying to catch the ball simultaneously. As expected, when no observation failures exist (case A), only one robot at most tries to catch the ball. By including observation delays having two, or even all the robots trying to catch the ball, no longer has zero probability. By further introducing observation errors this unwanted situations have a probability increase. Nevertheless, even in the worst case, C, the probability of having more than one robot going for the ball is still very low.

Given that a ball reset was included, there are no deadlocks in any of the three cases. As such, the values in Table 6.4 result from a stationary analysis and represent steady-state values.

Chapter 7

Model Identification

Chapter 5 detailed how to design task, action and environment models, and use them to perform robot task analysis, with results provided in Chapter 6. Although the environment and action models can be created manually, this might prove to be a cumbersome experience in many situations, especially if the number of actions and/or predicates increases. This chapter details a proposed method for creating the action and environment models automatically from real data, based on the following steps:

1. Specify a list of actions with their *running-conditions* and *desired-effects*;
2. Specify, if needed, episode *start state* and *end state*;
3. Run data collection experiment;
4. Estimate the action and environment models;

By estimating these models from real data one improves the models approximation, leading to improved models.

This chapter starts by detailing how data is collected (Section 7.1) and then describes how this data is used to generate the Petri net models (Section 7.2). Finally, Section 7.3 provides results with a robotic scenario using a realistic robotics simulator.

7.1 Data Collection Experiment

The data collection experiment is used to gather information about the actions impact in the world in order to estimate the action and environment models. The experiment consists on running each action based on the actions *running-conditions* and *desired-effects*. The experiment flowchart is depicted in Fig. 7.1.

The current implementation includes a counter per action to hold the number of times each action was selected during all data collection experiment episodes. These counters are used both to determine when the experiment should end and to increase the selection probability of less selected actions. Additionally, *start* and *end* predicate states can be specified. The *start* state is used to guarantee that each episode starts with the given *start* state predicates true. If an *end* state is specified, whenever that state is reached during an experiment, the current data collection experiment episode is ended and a new one is started.

During the data collection experiment only one action is selected at any given time, and each action is selected only if its *running-conditions* are true. Note that when randomly selecting a new action, due to the current action *desired-effects* becoming true, nothing prevents selecting the same action.

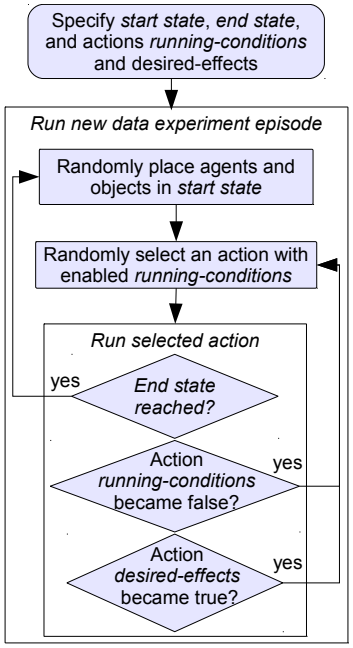


Figure 7.1: Models identification data collection experiment.

7.2 Model Estimation

Once the data collection is accomplished, the goal is to obtain the Petri net model of each action and of the environment from the experiment data, such that the analysis can be performed using automatically obtained realistic models. For each episode ran during the data collection experiment all predicate values and changes are stored, plus the selected action over time. Each time one or more predicates changes, there is a state change. Since one action is enabled at a time, each state change can be attributed either to the enabled action or the environment (recall that environment transitions occur in parallel with the action transitions). Given enough time, we will eventually capture all possible state changes. However, given that the a priori knowledge consists only of the available predicates list, and the actions *running-conditions* and *desired-effects*, we cannot know if a state change is caused by the action or the environment, or both. As such, and since during the data collection experiment we are associating each state change with an action, each identified action model will in fact (partially) contain the environment model, i.e., part (or all) of the environment model will be embedded in each action model. Details on how to overcome this limitation is part of the future work, as discussed in Chapter 8.

Having processed all data, one obtains a Markov chain model for each action (plus environment) where each event is defined by the set of predicates that were changed and the set of predicates that were maintained, i.e., the event is fully defined by the states it connects. A GSPN model can be created containing only stochastic transitions, whose marking process is equivalent to the obtained Markov chain.

As an example, consider an hypothetical result for the `CatchBall` action used in previous examples, depicted in Fig. 7.2. As can be seen, all identified transitions have as inputs the predicate places associated with all predicates that were true in the starting state, and have as outputs predicate places associated with all predicates which were true in the reached state.

For the GSPN model to be fully specified, one needs to compute the transition rates. From the experiment data, it results for each event, for each action, a set of elapsed times corre-

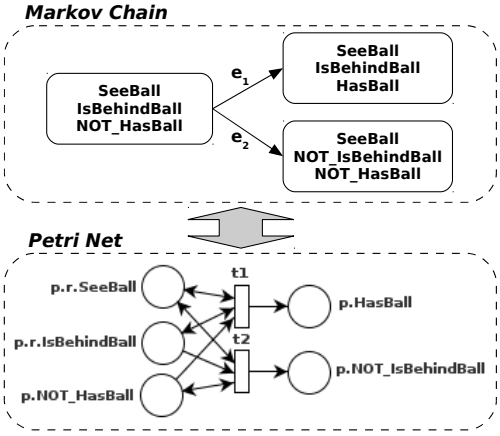


Figure 7.2: Hypothetical identified Petri net model.

sponding to the time taken from the start state to the reached state on every occurrence of the event. With this information one can compute the transition rates. Given the consideration of all transitions following an exponential distribution, the probability of the event e_i occurring in a given state s, for a given action a, is given by [Murata, 1989]:

$$ {}^{a}_{s}P(e_i) = \frac{{}^{a}_{s}\lambda_i}{\sum_{j=1}^{{}^{a}_{s}N} {}^{a}_{s}\lambda_j}, $$

where ${}^{a}_{s}N$ is the number of events that can occur in state s, in action a.

For each state, for each action, the probability of an event occurring can be estimated based on its frequency, considering all occurrences of all events that can occur in that state:

$$ {}^{a}_{s}P(e_i) = \frac{\#^{a}_{s}\Delta t_{e_i}}{\sum_{j=1}^{{}^{a}_{s}N} \#^{a}_{s}\Delta t_{e_j}}, $$

where ${}^{a}_{s}\Delta t_{e_i}$ is a vector with the elapsed time obtained for each occurrence of event e_i in state s, in action a, and # denotes vector length.

Since the time for each event to occur follows an exponential distribution, and all events are considered independent, the rate at which any event occurs in a given state, in a given action, is the sum of the rates of all events occurring in that state. Furthermore, the maximum likelihood estimator of the rate parameter for an exponential distribution is given by the mean of the elapsed times, resulting in

$$ {}^{a}_{s}\lambda = \sum_{j=1}^{{}^{a}_{s}N} {}^{a}_{s}\lambda_j = \left(\frac{1}{\sum_{i=1}^{{}^{a}_{s}N} \#^{a}_{s}\Delta t_{e_i}} \sum_{l=1}^{{}^{a}_{s}N} {}^{a}_{s}\Delta t_{e_l} \right)^{-1} $$

It results that one can estimate the rate of each event using:

$$ \begin{aligned} {}^{a}_{s}\lambda_i &= {}^{a}_{s}P(e_i) * \sum_{j=1}^{{}^{a}_{s}N} {}^{a}_{s}\lambda_j = \\ &= \frac{\#^{a}_{s}\Delta t_{e_i}}{\sum_{j=1}^{{}^{a}_{s}N} \#^{a}_{s}\Delta t_{e_j}} * \left(\frac{1}{\sum_{i=1}^{{}^{a}_{s}N} \#^{a}_{s}\Delta t_{e_i}} \sum_{l=1}^{{}^{a}_{s}N} {}^{a}_{s}\Delta t_{e_l} \right)^{-1} \\ &= \frac{\#^{a}_{s}\Delta t_{e_i}}{\sum_{l=1}^{{}^{a}_{s}N} {}^{a}_{s}\Delta t_{e_l}} \end{aligned} $$

The rate associated with the event is the rate associated with the corresponding transition.

In practice one will not have an infinite number of episodes, which implies that one might not capture all existing transitions and/or states. Since higher probability transitions are captured with higher probability, the impact on the performance properties should be negligible, given an enough number of episodes. However, there can be a higher impact on logical properties, particularly concerning reachable states. Nevertheless, most of the logical properties are still valid. For instance, both the invariance over predicate places property and the safeness of the net must hold (for the states/transitions found), even if not all transitions were identified.

Since one cannot know if all possible transitions were detected, currently the number of episodes is selected empirically. Nevertheless is part of the future work studying measures that might give a runtime indication of how well the approximation is doing, enabling to automatically determine when an experiment should end.

7.3 Single-Robot Example with Identification

In order to test the framework with real data, several experiments were performed using WeBots [Michel, 1998], a realistic simulator. Fig. 7.3 shows an overview of the simulation environment.

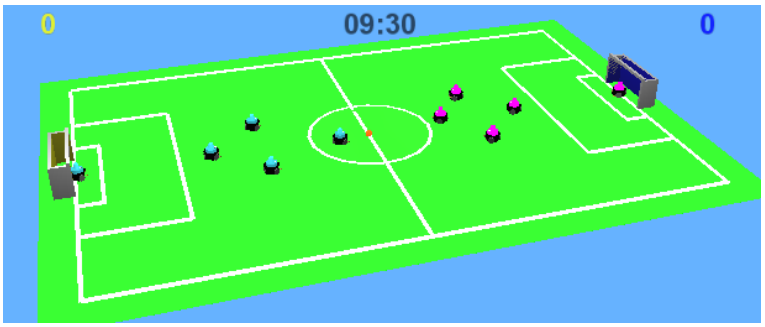


Figure 7.3: Simulation environment.

Since the code that runs in the real robots is the code that runs in the simulator, all the experiments in this setup were performed using the actual robot code implementation.

Given that the proposed identification algorithm does not yet enable separating the environment model form the actions models, one can only analyse tasks with real data when there is at most one running action at any given time. As such, it is not yet possible to apply the framework with real data on multi-robot scenarios.

Most of the models and predicates used for the single-robot scenario in the identification experiment were similar to the ones used in the theoretical cases. However, some improvements add to be made to accommodate this more realistic scenario, such as the inclusion of ball movement predicates.

During the entire experiment, only one robot was running the task under analysis, while all other 9 robot were running the `Move2StartPosition` action. The idea here is that each "non-experimenting" robot tries to maintain its posture during the entire experiment.

7.3.1 Predicates

The following list contains the predicates used for this example:

Ball position: `BallOwnGoal`, `BallNearOwnGoal`, `BallMidField`, `BallNearOppGoal` and `BallOppGoal`;

Action	Running-conditions	Desired-effects
Stop	-	RobotStopped
MoveBehindBall	SeeBall	IsBehindBall
CatchBall	IsBehindBall, SeeBall	HasBall
Dribble2Goal	HasBall	NearOppGoal, Aimable2Score BallNearOppGoal
Aim2Score	HasBall, Aimable2Score	GoalOpportunity
Kick2Goal	HasBall, GoalOpportunity	BallKicked

Table 7.1: Action properties for the identification experiment.

Ball movement: BallMovingToOwnGoal, BallMovingToOppGoal, BallMovingFast and BallStopped;

Robot position: NearOwnGoal, MidField and NearOppGoal;

Robot movement: RobotStopped;

Other: HasBall, IsBehindBall, NearBall, SeeBall, BallKicked, GoalOpportunity and Aimable2Score.

Most of the predicates were already used and explained previously. The ball movement predicates are used to indicate when the ball is moving and where to. The IsBehindBall is true when the ball is between the robot and the goal with the robot facing the ball. Predicate GoalOpportunity is true when there is a scoring probability, while predicate Aimable2Score is true when the robot is one rotation away from being in a good position to score a goal.

7.3.2 Actions

The actions included in this experiment are Stop, MoveBehindBall, CatchBall, Dribble2Goal, Aim2Score and Kick2Goal, with the properties indicated in Table 7.1. Recall that these properties are needed to run the identification algorithm.

For the data collection experiment the *start state* was defined as having predicates BallOwnGoal and BallOppGoal false, while the *end state* was defined as having any of these two predicates true. Furthermore, whenever the ball enters the goal, it stays in the goal until a new episode is started. As such, no identified model will include transitions having simultaneously input predicate BallOwnGoal and output predicate NOT_BallOwnGoal, or input predicate BallOppGoal and output predicate NOT_BallOppGoal, i.e., there will be no identified transition modelling the removal of the ball from the goal.

7.3.3 Identification Results

When comparing the theoretical results with the experimental ones, four different data collection experiments will be considered, A, B, C and D, with each experiment including an enough number of episodes such that each action was selected at least 10, 100, 1000 and 10000 times, respectively. Note that experiment A data is contained in experiment B data, which is contained in C, which is contained in D. The actual number of action selection times per experiment is shown in Table 7.2 (the action names are compressed to save space).

Fig. 7.4 shows the same data as table 7.2 using a logarithmic scale. The graph shows that the average rate selection of each action does not change with the number of episodes. Including the weights in the action selection during the data collection experiment is not enough to lead

Exp.	# Action Selection					
	A2S	**CB**	**D2G**	**K2G**	**MBB**	**Stop**
A	21	42	23	11	48	40
B	131	262	137	100	331	274
C	1011	1940	1012	1000	2468	1922
D	10013	18209	10025	10001	23405	18136

Table 7.2: Number of times each action was selected per experiment

to a more uniform action selection, but at least prevents form having a larger difference. The larger number of transitions fired for some actions is not only a result of that action including an higher number of transitions, but also due to the conditions associated with each action. For instance, for the given test scenario, action `MoveBehindBall` *running-conditions* only includes `SeeBall`, meaning that it could always be selected during the data collection experiment (since predicate `SeeBall` was considered to be always true) while, for instance, `Dribble2Goal` has *running-conditions* predicates `SeeBall` and `HasBall`. As such, even with the weights included to increase the selection probability of less selected actions, some actions can only be selected in a much smaller state space, leading to a smaller number of selection times over time.

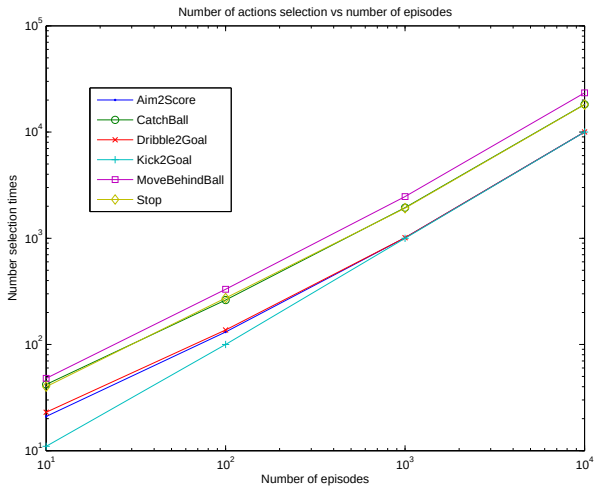
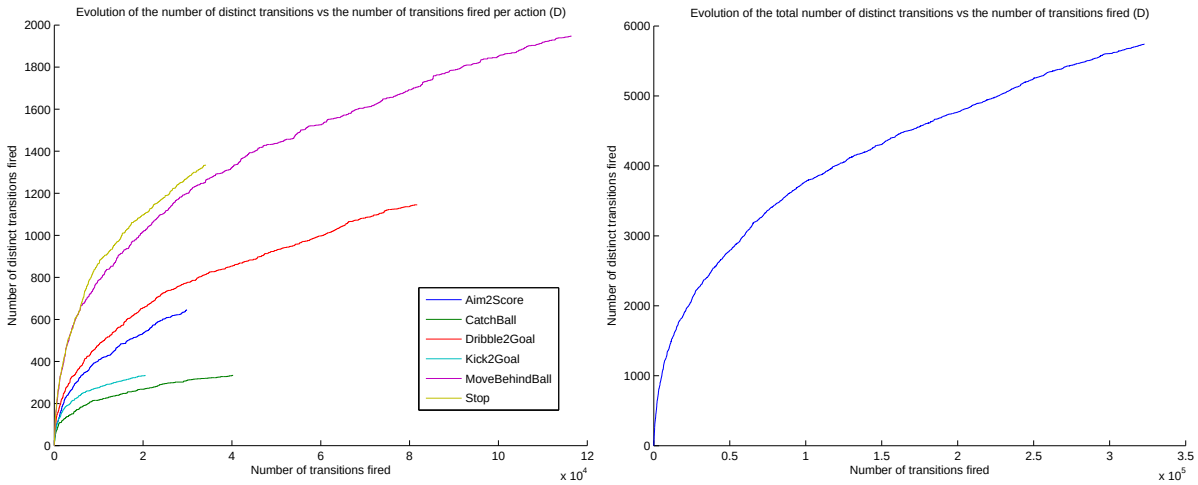


Figure 7.4: Number of action selection times versus number of episodes.

Fig. 7.5a and Fig. 7.5b depict the number of distinct transitions fired versus the total number of transitions fired obtained while running experiment D. Although the ratio between the number of distinct transitions fired and the total number of fired transitions is decreasing with the number of episodes, the number of new transitions found is still increasing by a reasonable rate. Fig. 7.6 provides a different view by showing the number of distinct transitions fired for each action versus the number of times that action was selected. The figure shows that the rate at which the number of distinct transitions grows is indeed decreasing. This means that the system contains a very large number of transitions and that, even when having run each action 10000 times, there are still transitions and states that were never identified. We will see later that this has a limited impact on the task analysis results.

The information in Fig. 7.6 might be used in runtime to determine when an experiment should end, although there is still no stoppage condition defined for the data collection experiment. For instance, the figure shows that the `MoveBehindBall` action still has a new transition



(a) Number of distinct transitions fired vs total number of fired transitions per action.

(b) Total number of distinct transitions fired vs total number of fired transitions

Figure 7.5: Number of distinct transitions fired vs number of total fired transitions.

firing rate higher than the remaining actions, in spite of being the most selected action. This means that indeed this action is the most complex action in the identification process, prone to include a higher number of transitions.

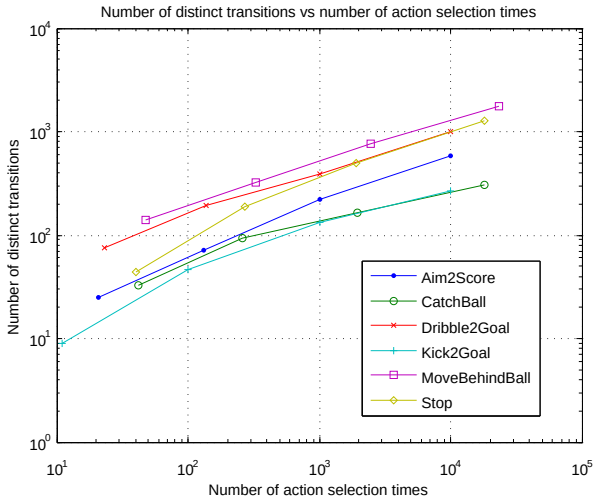


Figure 7.6: Number of distinct transition versus the number of action selection times.

7.3.4 Analysis Results

In order to compare the theoretical results with the experimental ones, three different tasks were devised, similar to the ones used in Section 6.1.1.4:

`Score_Goal_Shoot_First` Whenever the robot captures the ball (predicate `HasBall` is true), if the direct path to the opponent goal is free (predicate `Aimable2Score` is true), the robot tries to score immediately;

`Score_Goal_Shoot_50_50` This task represents a midterm between the other two actions. Whenever the robots captures the ball, it will randomly decide (with 0.5 probability) if it should try to score or dribble the ball closer to the goal.

`Score_Goal_Shoot_Later` The robot only tries to score if, besides having a direct path to the goal free, it is near the opponent goal (predicate `NearOppGoal` true);

The task plan models are depicted in Fig. 7.7. The `Get_Ball` task plan is identical to the one shown in Fig. 3.9, but with action `MoveBehindBall` instead of action `Move2Ball`.

For each task, the theoretical models are obtained by generating the single Petri net model, using the expansion process described earlier, with the action models obtained through the identification process. Since the ball is not removed from the goals during each episode, and given that the purpose of these tasks is to score a goal, it is expected that the obtained Petri net model of the overall task will contain deadlocks, corresponding to a goal scored in one of the goals. As such, and similarly with the single-robot theoretical example, a transient analysis was performed in order to perform a comparison with the transient results of the previous experiments. In these tests the experimenting robot and ball were initially positioned near the robot own goal and middle field, respectively.

7.3.4.1 Transient Analysis

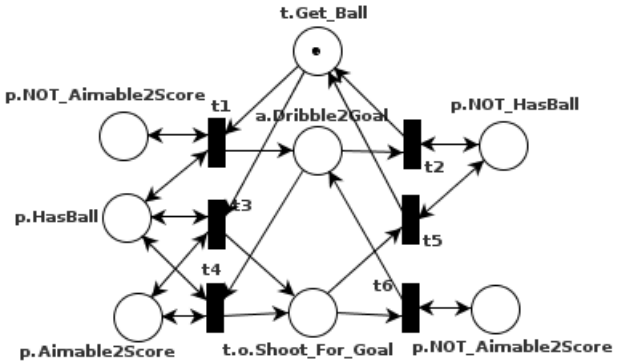
Conducting the transient analysis and measuring th probability of scoring a goal in the opponent goal for the Petri net generated from `Score_Goal_Shoot_Later`, for each data collection experiment set, lead to the results depicted in Fig. 7.8. As can be seen from the figure, the result obtained with experiment C data is very similar to the one obtained with experiment D data, in spite the transition increase rate shown in Fig. 7.5, i.e., in spite of the number of new transitions still being found.

For the experimental results, each task was ran 10000 episodes, i.e., the experiment ran until the robot scored 10000 goals (including goals scored both in its goal and the opponent goal). Each episode consisted in placing the robot in the same initial conditions as the theoretical case and running the task until a goal was scored (the same end conditions as the theoretical transient analysis).

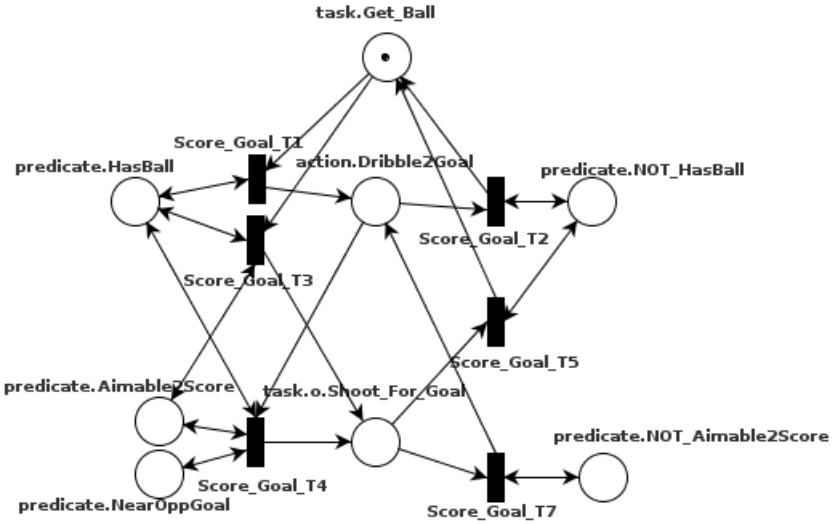
Fig. 7.9a shows the theoretical transient analysis results, obtained using the D data set, versus the experimental results. In spite of all the approximations done, the results still yield a low error. Furthermore, the difference between the three tasks for the experimental case is similar to the difference for the theoretical case. This is of the utmost importance, since one of the goals of the framework is to be able to give insight when comparing different tasks in terms of performance, i.e., allowing to evaluate the performance impact on changing some parameters of the task plan.

Analysing each task during the initial 14s (see Fig. 7.9b), one can see that shooting earlier leads to higher probability of scoring in the short term, but in the long term it leads to a lower scoring probability. This is expected as when kicking immediately the robot can score sooner, but with a high failure probability, as opposed to kicking only after getting close to the goal, which takes longer time but has higher probability of scoring. Furthermore, it is interesting to compare this result with the one obtained in the Section 6.1.1. In spite of having used a fewer number of predicate and simpler action models, the information provided is not much difference, specially in what concerns comparing the three devised tasks with each other. This strengthens the idea that for many situations there is no need to create complex models or even creating the models from real data, as simpler models would provide similar information.

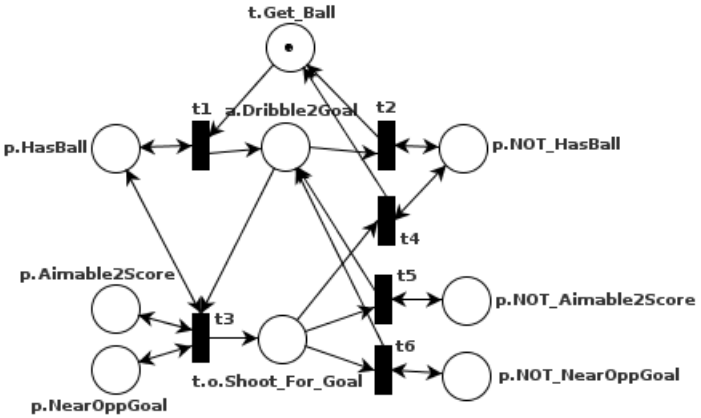
There is a reasonable time difference between the increase of scoring probability between the theoretical and experimental result, which is explained by the fact that we are considering that



(a) `Score_Goal_Shoot_First` task plan model.



(b) `Score_Goal_Shoot_50_50` task plan model.



(c) `Score_Goal_Shoot_Later` task plan model.

Figure 7.7: Task plans used in the identification experiment.

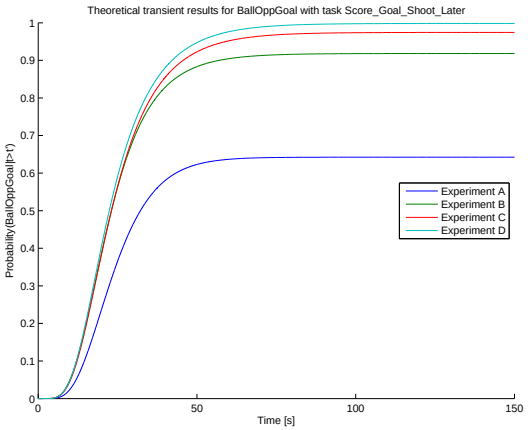
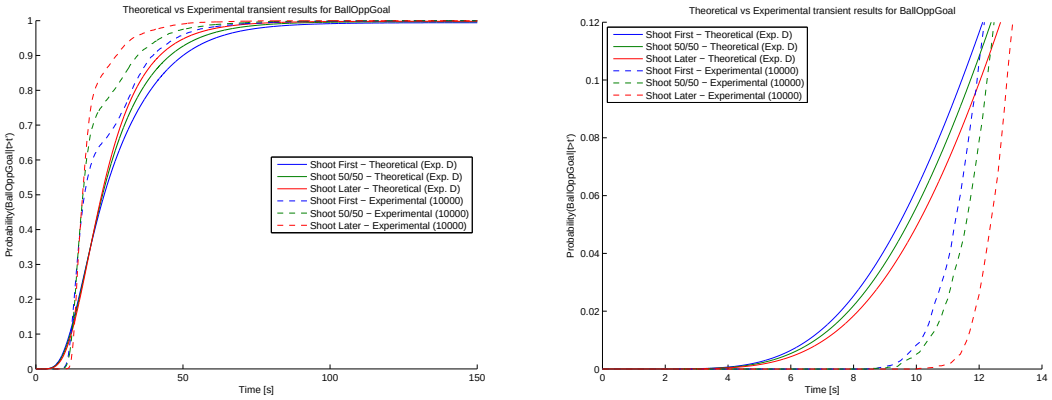


Figure 7.8: Theoretical analysis evolution with the number of episodes.



(a) Long-term. (b) Short-term.

Figure 7.9: Theoretical vs experimental transient analysis for predicate BallOppGoal.

Task	Probability		
	BallOwnGoal	**BallOppGoal**	**"No goal"**
Shoot_First	3.09E-04	9.94E-01	5.5E-03
Shoot_50_50	2.64E-04	9.974E-01	2.82E-03
ShootLater	2.36E-04	9.98E-01	1.15E-03

Table 7.3: Theoretical steady-state probability of scoring goals (from transient analysis).

all stochastic timed transitions follow an exponential distribution. In reality, some transitions always have a minimum time which must elapse before the transition can fire, which is not captured at the moment (this issue is part of future work).

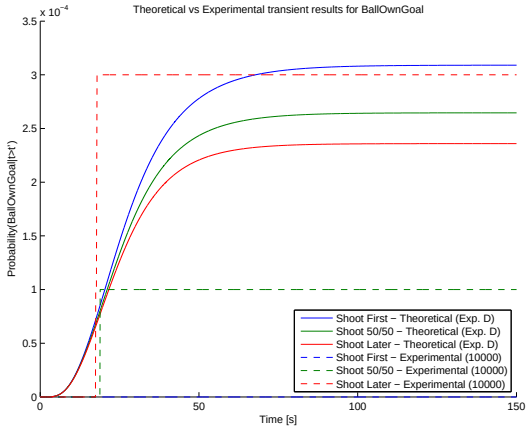


Figure 7.10: Theoretical vs experimental transient analysis for predicate BallOwnGoal.

The comparison of the probability of scoring in our own goal for the theoretical prediction versus the experimental results is depicted in Fig. 7.10. The probability of scoring in the robot own goal is very low, having little statistical significance for the number of episodes performed. Nevertheless, the maximum probability error is less than 3.2E-04 (as obtained from the plotted values).

An important aspect of these results, is that while the sum of the probabilities of predicates `BallOwnGoal` and `BallOppGoal` for the experimental case when a steady-state is reached is 1, the same does not happen for the theoretical case, as shown in Table 7.3. This result is due to the fact that the generated single Petri net model of the overall task includes deadlocks which do not correspond to a goal scored, either due to not having performed enough data collection episodes for the action and environment models, or due to the fact that the task plan was actually poorly designed. Knowing which of the two answers is the right one can only be achieved by analysing the found deadlocks, the path that led to those deadlocks, and the designed task. For the given test cases, it results that these unexpected deadlocks were due to not having performed enough data collection episodes. This conclusion is further backed up by the results shown in Fig. 7.5 and by the fact that the experimental results did not include any deadlock other than a goal scored.

Finding deadlocks when performing the theoretical analysis, which are due to having not performed enough data collection episodes, mainly has implications in the qualitative (logical) analysis. In the quantitative (performance) analysis, as it was shown, the impact on the determined values is very low, given that higher probability transitions will be properly identified and modelled. Concerning the qualitative analysis, the main impact is on the analysis of the set

of reachable states, since some states might not be included. Nevertheless, we can still perform all the qualitative analysis, knowing that the results are valid with a given probability that increases with the number of data collection episodes. For instance, in our test scenarios, we were able to conclude that, for all states found, the task is safe. We also determined that both sets of predicates formed by `BallOwnGoal`, `BallNearOwnGoal`, `BallMidField`, `BallNearOppGoal` and `BallOppGoal`, and formed by `NearOwnGoal`, `MidField` and `NearOppGoal` form place invariants. This is an expected and mandatory result, since both the ball and the robot can only be in one part of the field at any given time. If these invariants were not found, then it would indicate that the software running in the robots, responsible for computing the predicates during execution, would contain an error.

The fact that the net is safe and all pairs of predicate places form an invariant, makes the average number of tokens per place represent the average time spent in any action, for action places, and the average time a predicate was in a given state, for predicate places (for additional performance measures computation check Appendix B).

7.3.4.2 Steady-State Analysis

Based on the above results, and in order to further study the impact of the unexpected deadlocks described earlier, a steady-state analysis was performed. With the three task plans shown, and by repositioning the ball in the centre of the field whenever a goal is scored, the robot can play indefinitely. To model and analyse this setup under our framework, all that is needed is to add an environment model (manually designed), which models the repositioning of the ball when a goal is scored. For the action models we will use exactly the same data as used in the results above, but will only consider the D case. The initial state consisted on placing the robot and ball in the field centre. The environment model used to reposition the ball is depicted in Fig. 7.11.

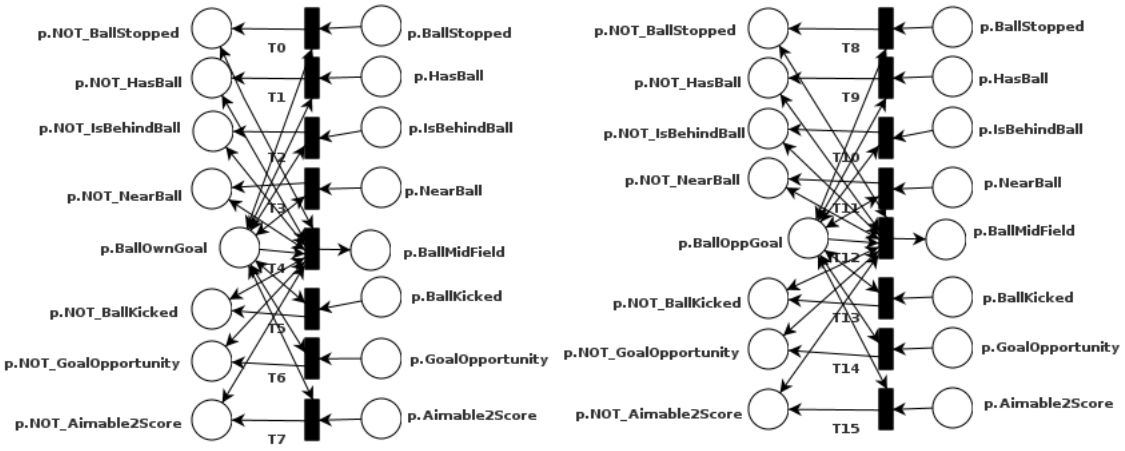


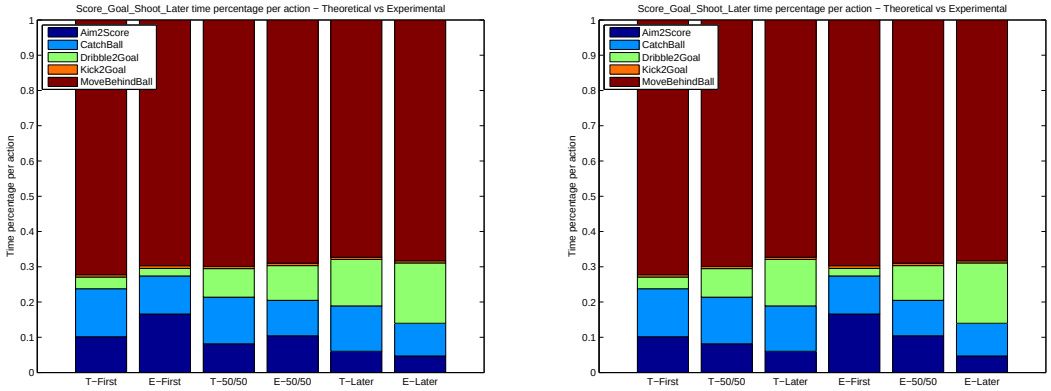
Figure 7.11: `BallReset` environment model.

With this new setup one would expect that the resulting generated Petri net would yield no deadlocks. However, and as explained earlier, since there were not enough data collection episodes, that is not the case. Nevertheless, by performing an analysis of the communication classes found (see Appendix A for details on communication classes), it was determined that for each task plan there was one large transient class containing around 90% of the tangible states, and all the remaining classes contained only one absorbing state. As such, it was decided to perform the steady-state analysis of the task excluding all absorbing states, i.e., considering only the large communication class.

Tasks	Action Places	Predicate Places
Shoot_First	2.61E-02	8.73E-02
Shoot_50_50	1.62E-02	8.77E-02
Shoot_Later	1.20E-02	9.01E-02

Table 7.4: Average steady-state error of the number of tokens per place.

For the experimental results, the robot played until the computed average number of tokens per place stabilised, i.e., until an experimental steady-state was reached. For the steady-state analysis, comparing the average time spent by the robot in each action yields the results depicted in Fig. 7.12.



(a) Experimental versus theoretical.

(b) Experimental variation versus theoretical variation.

Figure 7.12: Theoretical vs experimental average time spent per action.

The steady state average error, measured as the average absolute difference between the experimental and predicted number of tokens for all action places and for all predicate places is shown in Table 7.4.

The steady state average error between the number of tokens for all action places and for all predicate places is shown in Table 7.4. In spite all the approximations the results are still very similar. This result further strengths the conclusions described earlier, allowing to use these techniques to perform stationary analysis whenever there are deadlocks due to insufficient data collection episodes.

Chapter 8

Conclusions and Future Work

The diversity and complexity of tasks using mobile robots is increasing. However, most robot tasks are still designed without any knowledge of their expected execution properties. There is a need to be able to determine and study the task properties to ensure that some task performance levels and logical properties are achieved.

Mobile robot tasks provide a greater challenge than manufacturing systems, where this type of properties have been studied for some time. Unlike mobile robots, manufacturing systems have a more controlled environment with clear interfaces between each device in the manufacturing line. On the other hand, mobile robots are bond to work on an highly dynamic environment where the interactions with both the environment and other robots are not so strict, increasing the complexity of the task. To model these interactions, besides having to introduce more complex action models, one needs to include models of the environment. These models account not only for expected outcomes, conditions when these outcomes are possible, but also uncontrolled changes either due to the robot executing the task, other robots, agents or physics.

Given this inherent complexity and dynamics of mobile robot tasks, only by performing some simplifications and approximations can one ensure that it is possible to analyse and determine those properties for a given mobile robot task.

8.1 Thesis Summary

This work uses Discrete Event Systems, particularly Petri nets, as a modelling and analysis framework for mobile robot tasks where the world state is approximated through the use of logic predicates.

The key contribution of this work is the introduction of a framework to address the problem of modelling, analysis and execution of robot tasks, providing a structured, hierarchical and modular approach to the design of robot tasks. By using Generalised Stochastic Petri Nets, stochastic time is associated with world changes (transitions), allowing to model and analyse the task for performance/quantitative properties. Both transient and steady-state analysis can be performed. By a suitable modification of the Petri net structure (e.g., for conflicting immediate and stochastic transitions) logical/qualitative analysis is also accomplished. Petri nets provide a suitable tool to model both concurrency, parallelism and synchronisation, which are essential aspects of robot tasks.

The action models are probabilistic, containing uncontrollable transitions, which mimic the uncontrollable characteristics of a mobile robot task. The inclusion of environment models provide the means to model environment changes which can be attributed either to the world physics or other robots impact. These environment models allow closing the loop, enabling a

Petri net model of the overall task to be created, from which the desired properties can be determined.

The introduction of observation models enables asserting the impact of observation failures on the task outcome. These observation models range from full observability without failures of any kind, to exponential-distributed time observation delays and/or errors. These models also prove useful to model and analyse multi-robot tasks where robot coordination is achieved through decision-making based on, possibly, collective perception of the world, hereby denoted as implicit communication. The introduction of communication models further extends the ability to model and analyse multi-robot tasks, by providing means of modelling robot-to-robot communication. Several communication models are presented which range from zero-time/zero-failure communication to exponentially distributed time delays and failures, allowing to determine the impact of communication problems in multi-robot task properties.

A priori analysis of the task properties allows for important information to be retrieved even before executing the task. The framework provides a design-analysis-design iterative approach which leads to improved task plans. Simulation results and runtime analysis of experimental data can be used to further reduce model incorrections and test the validity of the models. The models used for the actions applied over an environment can be created manually or obtained through an identification algorithm which builds the models from real data using a specific unmanned data collection experiment. By using these identified models one ensures that the task properties reflect the real task. Once these identified models are obtained, a task which uses those models can be analysed in a very short time span (depending on the state space of the task) using the provided framework, as opposed to the number of hours or days one would need to perform real experiments to collect task execution data. These models can be combined with manually parametrised models, preventing having to re-run the entire experiment if small changes are made to the environment. Although using real-data-based models provides for real-value-based properties, there is no need to always use real-data-based models, particularly if one is only interested in understanding the relative impact of changing some properties of the task under analysis. By manually changing transition parameters one can also study the sensitivity of the task to the variation of those parameters.

8.2 Discussion and Future Work

The main limitation of the developed framework is not being able yet to identify the environment models separately from the action models. Although this limitation does not affect manually-built models, it prevents real data based models from being used in the analysis of tasks where more than one task runs simultaneously, which is the case for all multi-robot tasks. This is an important issue to be solved in the future, either by performing additional analysis on the collected data, or by providing more a priori information to allow distinguishing an action transition from an environment one. Including stopping and quality assessment methods for the identification algorithm is also part of the future work. Currently the algorithm runs "blindly" for a specific amount of time or number of episodes.

In order to model more complex multi-robot tasks, with multi-robot behaviours interleaved with individual behaviours in more than two robots, one needs to incorporate selection and commitment maintenance mechanisms. In the proposed framework dealing with the existence and removal of commitments can be viewed as any other predicate test. What still needs to be done, is how to commit, when to enter the commitment, and how to select which robots to commit with. Considerable work has been done regarding these aspects of relational behaviours [Cohen and Levesque, 1991; van der Vecht and Lima, 2005; Huang et al., 2005; Toktam Ebadi and Purvis, 2009]. We expect in the future to be able to integrate existing work into the

developed framework in order to solve these aspects.

The usage of template models allowed to ease the burden when creating multi-robot action and task models. Nevertheless, it might be useful to use, besides these templates, another Petri net formalism, denoted Coloured Petri nets [Jensen, 1989]. Coloured Petri nets are suitable to model distributed systems, and could provide a more compact design of the task models for the multi-robot case.

As future work one will investigate how the complexity of analysis algorithms can be decreased by allowing some of the properties to be derived, or to be ensured, during design time, without having to compose the full Petri net model. A possible approach is to create specific task plan composition operators as proposed in [Košecká et al., 1997; Huang et al., 2005; Ziparo and Iocchi, 2006] (not to be confused with the composition operators used in the expansion algorithm). By limiting the types of composition operators used to design Petri net based task plans, one might ensure that some logical properties are met at design time. The composition operators could also be used to obtain a set of *running-conditions* and *desired-effects* for a given task plan based on the Petri net model of that task plan and the *running-conditions* and *desired-effects* of the actions and tasks used in that task plan. The task's set of conditions could then be used to provide a more intuitive design of higher-level task plans from the lower-level ones.

Planning, or synthesis of plans, within the current framework is also an interesting subject for the future. Several work exists in the literature concerning the synthesis of Petri net based controllers from specifications [Hickmott et al., 2007; Dideban and Alla, 2008; Mauser and Lorenz, 2009]. The action models already include *running-conditions* and *desired-effects*, which provide the base logic knowledge for STRIPS [Fikes and Nilsson, 1971] base planning (for task models the composition operators mentioned in the previous paragraph could be used to provide the set of conditions). However, this type of planning does not take into consideration performance issues, as it is a pure logic-based planner. Probabilistic planning approaches exist in the AI community, including planners using high performance methods based on Stochastic Satisfiability (SSAT) [Majercik and Boots, 2005]. Recent work aims at bringing together approaches from the AI planning and the Petri nets community [Hickmott et al., 2007]. In this sense, Markov Decision Processes (MDP) [Puterman, 1994] are also tightly coupled with the developed framework. Recalling the `Score_Goal` Petri net task plan depicted in Fig. 3.10, there was a random switch formed by transitions t_3 and t_4, whose weights changed according to the desired task. These two transitions and their associated weights can be thought to provide an action selection decision process, which allow viewing the system as an MDP. Another approach would be to synthesise a Petri net task plan based on the available models with random switches for actions and task selection, and then use a learning algorithm to improve the task plan outcome by changing the weights of the transitions involved in the random switches. A similar approach is proposed in [Neto, 2008] with FSA.

All the analysis results presented were obtained using an analysis tool (implementing algorithms publicly available) developed by the author. The monitoring tools plus the execution Petri net framework in MeRMaID are also mainly a work from the author. However these tools are implemented across different technologies, using Matlab®, C++ and Java®. It is the intention of the author to provide an unified tool which allows both for the design, analysis and monitoring of robot tasks in the future. Such a tool will greatly improve the usability of the developed framework, making it more easy to be used with real robots in real scenarios.

Bibliography

H. Levent Akin, Andreas Birk, Andrea Bonarini, Gerhard Kraetzschmar, Pedro Lima, Daniele Nardi, Enrico Pagello, Monica Reggiani, Alessandro Saffiotti, Alberto Sanfeliu, and Matthijs Spaan. White paper on network robot systems, and formal models and methods for cooperation, 2008.

Rajeev Alur, Aveek Das, Joel Esposito, Rafael Fierro, Greg Grudic, Yerang Hur, R. Vijay Kumar, Insup Lee, James P. Ostrowski, George J. Pappas, Ben Southall, John Spletzer, and Camillo J. Taylor. A Framework and Architecture for Multirobot Coordination. *International Journal of Robotics Research*, 21(10-11):977–995, October-November 2002.

Marco Barbosa, Nelson Ramos, and Pedro Lima. MeRMaID - multiple-robot middleware for intelligent decision-making. In *IAV2007 - 6th IFAC Symposium on Intelligent Autonomous Vehicles*. Elsevier, 2007.

Falko Bause and Pieter S. Kritzinger. *Stochastic Petri Nets: An Introduction to the Theory*. Vieweg Verlag, 2nd edition, 2002.

L. Bernardinello and F. De Cindio. A survey of basic net models and modular net classes. In *Advances in Petri Nets 1992, The DEMON Project*, pages 304–351. Springer, 1992.

Rodney A. Brooks. A Robust Layered Control System for a Mobile Robot. Technical report, Massachusetts Institute of Technology, 1985.

Christos G. Cassandras and Stéphane Lafortune. *Introduction to Discrete Event Systems*. Springer US, 2nd edition, 2008. ISBN 978-0-387-33332-8.

A. Castelnuovo, L. Ferrarini, and L. Piroddi. An incremental petri net-based approach to the modeling of production sequences in manufacturing systems. *IEEE Transactions on Automation Science and Engineering*, 4(3):424–434, July 2007.

Philip R. Cohen and Hector J. Levesque. Teamwork. *Noûs*, 25(4):487–512, 1991.

Bruno D. Damas and Pedro U. Lima. Stochastic Discrete Event Model of a Multi-Robot Team Playing an Adversarial Game. In *Proceedings of the 5th IFAC/EURON Symposium on Intelligent Autonomous Vehicles*. Elsevier, July 2004.

René David and Hassane Alla. *Discrete, Continuous and Hybrid Petri Nets*. Springer, 2005. ISBN 3-540-22480-7.

Abbas Dideban and Hassane Alla. Reduction of constraints for controller synthesis based on safe Petri Nets. *Automatica*, 7(7):1697–1706, 2008.

Magnus Egerstedt. Behavior Based Robotics Using Hybrid Automata. In *HSCC*, pages 103–116, 2000.

Richard Fikes and Nils J. Nilsson. STRIPS: A New Approach to the Application of Theorem Proving. *Artificial Intelligence*, 2(3):189–208, 1971.

J. Flochova. A Petri Net Based Supervisory Control Implementation. In *Proc. of the IEEE Int. Conf. on Systems, Man and Cybernetics*, volume 2, pages 1039–1044. IEEE, October 2003.

Claude Girault and Rüdiger Valk. *Petri Nets for Systems Engineering*. Springer, 2003. ISBN 3-540-41217-4.

D. Herrero-Perez and H. Martinez-Barbera. Petri Nets based coordination of flexible autonomous guided vehicles in flexible manufacturing systems. In *ETFA 2008, IEEE Int. Conf. on Emerging Technologies and Factory Automation*, pages 508–515, 2008.

Sarah Hickmott, Jussi Rintanen, Sylvie Thiébaux, and Lang White. Planning via petri net unfolding. In *IJCAI'07: Proc. of the 20th int. joint conf. on Artifical intelligence*, pages 1904–1911. Morgan Kaufmann Publishers Inc., 2007.

He-Jiao Huang, Xuan Wang, Qing-Cai Chen, and Xiao-Long Wang. Specification and verification of multi-agent systems with a porperty-preserving component-based methodology. In *Fourth International Conference on Machine Learning and Cybernetics*. IEEE, August 2005.

Dean L. Issacson and Richard W. Madsen. *Markov Chains, Theory and Applications*. Wiley, 1976.

Kurt Jensen. Coloured Petri nets: A high level language for system design and analysis. In *Applications and Theory of Petri Nets*, pages 342–416, 1989.

Konstantin Kapellos, Daniel Simon, Muriel Jourdan, and Bernard Espiau. Task Level Specification and Formal Verification of Robotics Control Systems: State of the Art and Case Study. *Int. Journal of Systems Science*, 30(11):1227–1245, November 1999.

Gunhee Kim and Woojin Chung. Navigation Behavior Selection Using Generalized Stochastic Petri Nets for a Service Robot. *IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews*, 37(4):494–503, July 2007.

Jamie King, R.K. Pretty, and R.G. Gosine. Coordinated execution of tasks in a multiagent environment. *IEEE Transactions on Systems, Man and Cybernetics, Part A: Systems and Humans*, pages 615–619, 2003.

Hiroaki Kitano, Minoru Asada, Yasuo Kuniyoshi, Itsuki Noda, Eiichi Osawa, and Hitoshi Matsubara. RoboCup: A Challenge Problem for AI and Robotics. In *RoboCup-97: Robot Soccer World Cup I*, pages 1–19. Springer-Verlag, July 1997. ISBN 3-540-64473-3.

William J. Knottenbelt. *Generalised Markovian Analysis of Timed Transition Systems*. PhD thesis, Department of Computer Science, University of Cape Town, June 1996.

Jana Košecká, Henrik I. Christensen, and Ruzena Bajcsy. Experiments in Behaviour Composition. *Robotics and Autonomous Systems*, 19(3-4):287–298, March 1997.

Bruno Lacerda and Pedro Lima. Linear-Time Temporal Logic Control of Discrete Event Models of Cooperative Robots. *Journal of Physical Agents*, 2(1), 2008.

Steven M. LaValle. *Planning Algorithms*. Cambridge University Press, 2006. ISBN 5-21862051-3.

Pedro U. Lima and Luís M. Custódio. The Socrob Project: Soccer Robots or Society of Robots. In *Cutting Edge Robotics*, chapter 5, pages 417–432. pIV, July 2005. ISBN 3-86611-038-3.

Pedro U. Lima, Nelson Ramos, Marco Barbosa, and Hugo F. Costelha. MeRMaID - Multiple-Robot Middleware for Intelligent Decision-making. Technical Report RT-701-07, January 2007. ISR-IST Internal Report.

Christoph Lindemann. *Performance Modelling with Deterministic and Stochastic Petri Nets.* John Wiley & Sons Inc, bk&cd-rom edition, 1998. ISBN 4-71976466-5.

Stephen M. Majercik and Byron Boots. DC-SSAT: A Divide-and-Conquer Approach to Solving Stochastic Satisfiability Problems Efficiently. In *Proceedings of the 20th National Conference on Artificial Intelligence (AAAI-05).* AAAI Press / MIT Press, July 2005.

S. Mauser and R. Lorenz. Variants of the Language Based Synthesis Problem for Petri Nets. In *Ninth Int. Conf. on Application of Concurrency to System Design*, pages 89–98, July 2009.

O. Michel. Webots - fast prototyping and simulation of mobile robots, 1998.

Dejan Milutinovic and Pedro U. Lima. Petri Net Models of Robotic Tasks. In *Proc. of the 2002 IEEE Int. Conf. on Robotics and Automation.* IEEE, May 2002.

Tadao Murata. Petri nets: Properties, analysis and applications. *Proc. IEEE*, 77(4):541–580, April 1989.

Gonçalo F. Neto. Combining Supervisory Control of Discrete Event Systems and Reinforcement Learning to Control Multi-Robot Systems. In *Workshop on Formal models and methods for multi-robot systems, AAMAS 2008 - the 7th International Conference on Autonomous Agents and Multiagent Systems*, 2008.

Gonçalo F. Neto, Hugo F. Costelha, and Pedro U. Lima. Topological Navigation in Configuration Space Applied to Soccer Robots. In Springer-Verlag, editor, *RoboCup 2003 Book, Padova, Italy.* RoboCup, July 2003.

Víctor A. Ocasio. Stability of Boolean Dynamical Systems and Graph Periodicity. Master's thesis, University of Puerto Rico, Mayagüez Campus, May 2009.

Carl Adam Petri. Kommunikation mit automaten. Technical report, Bonn: Institut für Instrumentelle Mathematik, 1966. English translation.

Martin L. Puterman. *Markov Decision Processes—Discrete Stochastic Dynamic Programming.* John Wiley & Sons, Inc., 1994.

Yongfa Qin and Rongyou Xu. GSPN-based modeling and analysis for robotized assembly system. In *IEEE Int. Conf. on Robotics and Biomimetics, 2008. ROBIO 2008*, pages 1070 –1075, 2009.

P. G. Ramadge and W.M. Wonham. The control of discrete event systems. 77(1):81–98, January 1989.

Andrea Röck and Ray Kresman. On Petri nets and predicate-transition nets. In *Proceedings of the International Conference on Software Engineering Research and Practice & Conference on Programming Languages and Compilers, SERP 2006*, pages 903–909, 2006.

J. Rosell, N. Munoz, and A. Gambin. Robot tasks sequence planning using Petri nets. In *Proc. of the IEEE Int. Symposium on Assembly and Task Planning*, pages 24–29, 2003.

Robert Tarjan. Depth-First Search and Linear Graph Algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

Maryam Purvis Toktam Ebadi and Martin Purvis. A framework for facilitating cooperation in multi-agent systems. *The Journal of Supercomputing*, December 2009.

Arjan J. van der Schaft and Hans Schumacher. *An Introduction to Hybrid Dynamical Systems*. Springer, 2000.

Bob van der Vecht and Pedro U. Lima. Formulation and Implementation of Relational Behaviours for Multi-robot Cooperative Systems. In *Proceedings of RoboCup-2004: Robot Soccer World Cup VIII*, pages 516–523. Springer-Verlag, 2005.

Vandi Verma, Tara Estlin, Ari Jónsson, Corina Pasareanu, Reid G. Simmons, and Kam Tso. Plan Execution Interchange Language (PLEXIL) for Executable Plans and Command Sequences. In *Proceedings of the 8th International Symposium on Artificial Intelligence, Robotics and Automation in Space*. ESA Publications Division, September 2005.

N. Viswanadham and Y. Narahari. *Performance Modeling of Automated Manufacturing Systems*. Prentice Hall, 1992. ISBN 0-13-658824-7.

F.Y. Wang, K.J. Kyriakopoulos, A. Tsolkas, and G.N. Saridis. A Petri-net Coordination Model for an Intelligent Mobile Robot. *Transactions of IEEE Systems, Man and Cybernetics Society*, 21(4), 1991.

Ning Xi Yu Sun, Jindong Tan. Hybrid system model for event-based planning and control of robot operations. In *Proc. 2003 IEEE International Symposium on Computational Intelligence in Robotics and Automation*, volume 2, pages 700–705, July 2003.

Vittorio A. Ziparo and Luca Iocchi. Petri net plans. In *Proc. of the Fourth Int. Workshop on Modelling of Objects, Components, and Agents (MOCA'06)*, pages 267–290. University of Hamburg, 2006.

Appendix A

Markov Chains

Markov chains are the underlying stochastic process of the marking graph of stochastic timed Petri nets. As such, this chapter introduces the concepts around Markov Chains and explores the various analysis techniques that where applied in this work.

A stochastic process, i.e., a collection of random variables over time, that satisfies the Markovian property and have a countable state space is a Markov Chain. If time is discrete, these are referred to as *Discrete Time Markov Chains* (DTMC), otherwise, if time is continuous, these are called *Continuous Time Markov Chains* (CTMC).

This appendix does not introduce anything new, but is a detailed summary concerning the concepts and analysis of both DTMCs and CTMCs for the sake of completeness of this document. More details can be found in the literature [Issacson and Madsen, 1976; Viswanadham and Narahari, 1992]. We will focus only on finite homogeneous Markov Chains.

A.1 Geometric Distribution

The probability mass function of a geometrically distributed random variable X which describes the number of independent and identical Bernoulli trials needed to obtain the first success is given by

$$P\{X = k\} = (1-p)^{k-1}, \quad k = 1, 2, \ldots \tag{A.1}$$

where p is the probability of success of each Bernoulli trial. The mean of X is given by

$$E[X] = \frac{1}{p} \tag{A.2}$$

The cumulative distribution function of X is

$$F_X(k) = Pr\{X < k\} = 1 - (1-p)^k \tag{A.3}$$

It can be shown that a geometrically distributed random variable X has the following property

$$P\{X = m + n \mid X > m\} = P\{X = n\} \tag{A.4}$$

Eq. (A.4) describes what is know as the memoryless, or Markov, property. The future only depends on the current state, and not on the history.

A.2 Exponential Distribution

The exponential distribution is the continuous analogue of the geometric distribution.

The probability density function of an exponentially distributed random variable X with parameter λ is given by

$$\begin{aligned} f_X(x) &= 0, & x \leq 0 \\ &= \lambda.e^{-\lambda x}, & x > 0, \end{aligned} \tag{A.5}$$

where λ is called the rate of X with $x \in \mathbb{R}$. The mean of X is given by:

$$E[X] = \frac{1}{\lambda} \tag{A.6}$$

The cumulative distribution function of X is given by

$$\begin{aligned} F_X(x) = P\{X < x\} &= 0, & x < 0 \\ &= 1 - e^{-\lambda x}, & 0 \leq x \leq \infty \end{aligned} \tag{A.7}$$

Given an exponentially distributed random variable X, the following properties hold

$$\begin{aligned} &P\{X > x\} = e^{-\lambda x} && x \geq 0 \\ &P\{x \leq X \leq y\} = e^{-\lambda x} - e^{-\lambda y} && 0 \leq x \leq y \end{aligned} \tag{A.8}$$

Furthermore, the following can be shown

$$P\{X > x + y | X > x\} = P\{X > y\}, \quad x, y \geq 0 \tag{A.9}$$

Like Eq. (A.4) demonstrated the Markov property for geometric distributions, Eq. (A.9) demonstrates it for exponentially distributions.

A.3 Discrete Time Markov Chains

A discrete time stochastic process that satisfies the Markov property and has a countable state space is a DTMC. The study of DTMCs is rather important since the study of CTMCs can be partially done recurring to DTMCs, as will be detailed later.

Definition A.3.1. *A* Discrete Time Markov Chain *is a discrete time stochastic process* $\{X_n, n \geq 0\}$ *that satisfies the Markov (or memoryless) property, i.e., given states* $i, j, k \in \mathcal{S}$*,*

$$P\{X_{m+1} = i | X_m = j, \ldots, X_1 = k\} = P\{X_{m+1} = i | X_m = j\} \tag{A.10}$$

The values of X_m *form a countable discrete state space.*

The *n-step transition probabilities*, i.e., the probability of going from state i at time step m to state j in n steps is given by

$$p_{ij}(m, m+n) = P\{X_{m+n} = j | X_m = i\}$$

If the above probability is independent of the initial time step m, the DTMC is homogeneous.

Definition A.3.2. *A DTMC is* homogeneous*, or has* stationary transition probabilities*, iff for all* n*,* $p_{ij}(n, m)$ *does not depend on* n *or* m*, but only on* $m - n$*.*

For homogeneous DTMCs the n-step transition probabilities can be written

$$p_{ij}(m, m+n) = p_{ij}(n)$$

The one step transition probability is obtained with $p_{ij}(1)$, and thus given by

$$p_{ij}(1) = P\{X_n = j | X_{n-1} = i\}$$

The 1-step transition probabilities of the entire chain can be written in a matrix form.

Definition A.3.3. *Given a DTMC, the* transition probability matrix *is a square matrix denoted by P and given by*

$$P = [p_{ij}] = \begin{bmatrix} p_{00} & p_{01} & \cdots \\ p_{10} & p_{11} & \cdots \\ \vdots & \vdots & \ddots \end{bmatrix}$$

The transition probability matrix P is a non-negative stochastic matrix, i.e., given states $i, j \in \mathcal{S}$

$$\sum_{j \in \mathcal{S}} p_{ij} = 1 \text{ and } p_{ij} \geq 0$$

The relevant Markov chains for this work are all homogeneous, therefore non-homogeneous Markov chains will not be subject of study here.

A.3.1 Sojourn Times in States

The *sojourn times* are important to compute several properties of Markov chains and could be used, for instance, to compute the time spent responding to failures in a real system.

Definition A.3.4. *Given a DTMC, the* sojourn time T_i *is a geometric random variable corresponding to the number of time steps spent in state i each time state i is visited.*

It can be shown that, given the Markovian nature of the DTMC, the sojourn time in state i is given by

$$T_i = \frac{1}{1 - p_{ii}}$$

A.3.2 Evolution of the Chain

The evolution of DTMC is described by the *Chapman-Kolmogorov equation*, which can be written as

$$p_{ij}(m+n) = \sum_{k \in \mathcal{S}} p_{ik}(m) p_{kj}(n) \tag{A.11}$$

With this equation, the n-step probabilities can be computed from the 1-step probabilities. Considering the transition probability matrix P, and noting that $P(0) = I$ (there is no transition if time does not elapse), Eq. (A.11) leads to

$$P(n) = P^n \tag{A.12}$$

Note that $P(n)$ is still a non-negative stochastic matrix.

Consider now the probability of being in state j at step n

$$\pi_j(n) = P\{X_n = j\}, \quad \forall n \geq 0, \ \forall j \in \mathcal{S}$$

By the total probability theorem we have

$$P\{X_n = j\} = \sum_{i \in \mathcal{S}} P\{X_n = j \mid X_0 = i\} P\{X_0 = i\},$$

which results in

$$\pi_j(n) = \sum_{i \in \mathcal{S}} \pi_i(0) p_{ij}(n) \tag{A.13}$$

Let $\Pi(n) = [\pi_0(n)\ \pi_1(n)\ \ldots]$ be the probability of reaching all the states after n time steps have elapsed, with $\Pi(0)$ being the probability mass function of the random variable X at the beginning of time. Then, Eq. (A.13) can be written in matrix form as

$$\Pi(n) = \Pi(0)P(n) = \Pi(0)P^n \tag{A.14}$$

Given the pmf of the initial random variable, X_0, and the transition probability matrix P, we can compute the pmf of the random variable for any given step n using Eq. (A.14).

A.4 Continuous Time Markov Chains

The previous section detailed Markov chains with discrete time. This section now continues the study but for the case when time is continuous.

Definition A.4.1. *A* CTMC *is a continuous time stochastic process* $\{X(t) : t \geq 0\}$*, with a countable discrete state space, that satisfies the Markov (or memoryless) property, i.e., let* $\mathcal{S}$ *be the countable state space, given times* $0 \leq t_1 < t_2 \leq t_3$ *and states* $i, j, x \in \mathcal{S}$*, then*

$$P\{X(t_3) = i | X(t_2) = j \wedge X(t_1) = x\} = P\{X(t_3) = i | X(t_2) = j\} \tag{A.15}$$

The transition probabilities are given by

$$p_{ij}(t_1, t_1 + t_2) = P\{X(t_1 + t_2) = j \mid X(t_1) = i\} \tag{A.16}$$

The homogeneous concept also applies to CTMCs, in which case the transition probabilities can be written as

$$p_{ij}(t_1, t_1 + t_2) = p_{ij}(t_2)$$

Like with DTMC, the probability transition matrix can also be written in matrix form.

Definition A.4.2. *Given a CTMC, the* transition probability matrix*, denoted by* $H(t)$ *is given by*

$$H(t) = [p_{ij}(t)] = \begin{bmatrix} p_{00}(t) & p_{01}(t) & \cdots \\ p_{10}(t) & p_{11}(t) & \cdots \\ \vdots & \vdots & \ddots \end{bmatrix}$$

Remark A.4.1. *By definition, given time index* $t \geq 0$*,* $p_{ij}(0)$ *is 1 for* $i = j$ *and 0 otherwise.*

Note that, like in the DTMCs case, the transition probability matrix is a non-negative stochastic matrix, so

$$\sum_{j \in \mathcal{S}} p_{ij}(t) = 1 \text{ and } p_{ij} \geq 0$$

In the DTMCs case the 1-step probability transition matrix was computed. In the CTMCs case, since they work over continuous time, the transition probability matrix only makes sense with a given time interval.

Like in the DTMCs case, only homogeneous CTMC will be referred in this appendix.

A.4.1 Sojourn Times

The meaning of sojourn times in CTMCs is exactly the same. The difference is while the sojourn time T_i in state i was geometrically distributed in DTMCs, in CTMCs it is exponentially distributed. Furthermore, the following definition is introduced

Definition A.4.3. *Given a CTMC, consider the sojourn time* T_i *of state* i *exponentially distributed with rate* λ_i*. If* $\lambda_i = 0$*, the state is denoted an* absorbing state*, if* $\lambda_i = \infty$*, the state is said to be* instantaneous*, and if* $0 < \lambda_i < \infty$*, the state is said to be* stable*.*

A.4.2 Evolution of The Chain

The evolution of CTMC can be also described by the *Chapman-Kolmogorov* equations (the continuous version) and the *Kolmogorov* differential equations.

Given a CTMC and times $\Delta t_1, \Delta t_2 \geq 0$, with $\Delta t_1 + \Delta t_2 = \Delta t$, the transition probabilities can be written as

$$p_{ij}(\Delta t) = \sum_{k \in S} p_{ik}(\Delta t_1) p_{kj}(\Delta t_2) \tag{A.17}$$

In matrix form, Eq. (A.17) can be written as

$$H(\Delta t) = H(\Delta t_1) H(\Delta t_2) \tag{A.18}$$

Eq. (A.18) represents the continuous time version of *Chapman-Kolmogorov* equations, which model the CTMC evolution.

Definition A.4.4. *Given a CTMC, $Q(t)$ is called the* infinitesimal generator*, or* transition rate matrix*, where $q_{ij}(t)$ is the rate of going from state i to state j at time t for $i \neq j$, and q_{ii} is the rate for leaving state i at time t.*

It can be shown that the transition rate matrix fulfils the following property

$$\sum_{j \in \mathcal{S}} q_{ij}(t) = 0, \quad \forall\, t$$

Since we are only studying homogeneous CTMCs, the transition rate matrix will be constant over time, thus, for all time $t \geq 0$,

$$q_{ij} = q_{ij}(t)$$
$$Q = Q(t) = [q_{ij}]$$

It can be shown that the forward and backward Kolmogorov equations, which describe the evolution of the chain forward and backward in time, are given respectively by

$$\frac{dH(t)}{dt} = H(t)Q, \quad H(0) = I \tag{A.19}$$
$$\frac{dH(t)}{dt} = QH(t), \quad H(0) = I \tag{A.20}$$

Extracting the solution from these equations gives the relation between the transition probability matrix and the transition rate matrix,

$$H(t) = e^{Qt}, \tag{A.21}$$

where expression e^{At} is defined for a square matrix as

$$e^{At} = \sum_{n=0}^{\infty} \frac{(At)^n}{n!}$$

Eq. (A.21) allows the computation of the probability of reaching a state given the transition rate matrix.

Consider now the probability of reaching a given state

$$\pi_j(t) = P\{X(t) = j\} \tag{A.22}$$

and let $\Pi(t)$ be the state probability for all states,

$$\Pi(t) = [\pi_0(t)\ \pi_1(t)\ \ldots]$$

By the total probability theorem, the transition probabilities are given by

$$\begin{aligned}\pi_j(t) &= \sum_i P\{X(0) = i\} P\{X(t) = j \mid X(0) = i\} \\ &= \sum_i \pi_i(0) p_{ij}(t)\end{aligned}$$

In matrix form results in

$$\Pi(t) = \Pi(0)H(t) = \Pi(0)e^{Qt} \tag{A.23}$$

A.5 Markov Chains Classification

Before getting into the analysis of Markov Chains, it is convenient to introduce some important terms, used for classifying Markov Chains. We will introduce these terms using the DTMC specification, but the same applies considering continuous time instead of discrete time steps.

A.5.1 Communication Classes and Irreducibility

A Markov chain can be partitioned and classified accordingly to the *accessibility* between its states.

Definition A.5.1. *Given states $i, j \in \mathcal{S}$, state j is said to be* accessible *from state i if $p_{ij}(n) > 0$ for some $n \geq 0$. States i and j are said to* communicate *if each is accessible from each other.*

From Definition A.5.1 the following holds true:

1. State i communicates with itself for all $i \in \mathcal{S}$;
2. If state i communicates with state j, then state j communicates with state i, for all $i, j \in \mathcal{S}$;
3. If state i communicates with state k, and state k communicates with state j, then state i communicates with state j, for all $i, j, k \in \mathcal{S}$.

Definition A.5.2. *In a DTMC, a* communication class *is defined as the class composed by states that can communicate with all states within that class.*

Using communication classes we can partition DTMCs into sets of states sharing the same class.

Definition A.5.3. *A DTMC is called* irreducible *if it contains only a single communication class, i.e., all states communicate with each other.*

Definition A.5.4. *A communication class C in a DTMC with state space $\mathcal{S}$ is said to be* closed *if, given two states, i and j, j is not accessible from i for all $i \in C$ and $j \in \mathcal{S} \backslash C$, otherwise, C is said to be* open.

A.5.2 Transience and Recurrence

Given a DTMC, let $f_{ii}(n)$ denote the probability of returning to state i for the first time in n steps, after leaving state i. The probability of ever returning to state i after leaving it, denoted by f_i, is given in this case by

$$f_i = \sum_{n=1}^{\infty} f_{ii}(n)$$

The expected number of time steps taken by the DTMC to return to state i after leaving it is

$$v_i = \sum_{n=1}^{\infty} n f_{ii}(n)$$

Definition A.5.5. *Given a DTMC, a state i is said to be* recurrent *if $f_i = 1$, and* transient *if $f_i < 1$. A recurrent state i is said to be* positive recurrent*, or* non-null recurrent*, if v_i is finite, and* null recurrent *if v_i is infinite.*

A.5.3 Periodicity and Ergodicity

Definition A.5.6. *Let d_i be the greatest common divisor of the number of steps such that $p_{ii}(n) > 0$. A recurrent state is called* periodic *with period d_i if the $d_i > 1$, and* aperiodic *if $d_i = 1$.*

Definition A.5.7. *A DTMC is called* ergodic *it is irreducible and all its states are aperiodic and positive recurrent.*

Remark A.5.1. *The periodicity of a chain does not make sense in the CTMCs case given time is continuous. It results that all CTMCs are aperiodic.*

Remark A.5.2. *A Markov chain is said to be positive recurrent, null recurrent, periodic, etc., if all its states are respectively positive recurrent, null recurrent, periodic, etc..*

A.5.4 State Classification

A Markov chain can be classified based on the periodicity and recurrence of its states. Moreover, these are in fact class properties, as detailed here. The following list details some properties regarding classes in Markov chains:

1. All states within the same class share the same properties regarding transience, recurrence, positive recurrence and periodicity, meaning that, if a state i holds one of these properties, and state i communicates with state j, then state j also holds the same property.
2. All states in an open communication class are transient;
3. All states in a finite closed communication class are positive recurrent;
4. If a communication class has null recurrent states then it is an infinite closed communication class;
5. In a finite chain, not all states are transient and all recurrent states are positive recurrent. Therefore, if a chain is irreducible and finite, then all its states are positive recurrent;
6. If a chain is irreducible, then all the states are transient or all the states are recurrent;
7. If a state is recurrent, the mean number of visits is infinite, while if the state is transient, the mean number of visits is finite.

A.6 Analysis of Markov Chains

There are two types of analysis we can perform with Markov chains: *transient* analysis and *steady-state* analysis. As the names indicate, the transient analysis devotes to the study of properties of the chain evolution until it reaches a steady-state (if it exists). Steady-state analysis concerns with the study of stationary probabilities as time goes to infinite. This section focus on the study of the analysis of both DTMCs and CTMCs, including how to benefit from the analysis of CTMCs from DTMCs through the use of *Embedded Markov Chains*.

The Markov chain analysis techniques described in this section were implemented using Matlab®.

A.6.1 Steady-State Analysis

As said previously, the steady state analysis searches for the existence of stationary state probabilities and its computation. We shall see how these are computed and under which conditions they exist for both the DTMCs and CTMCs cases.

A.6.1.1 DMTCs

Consider the state probability limits, denoted by $\mathbf{\Pi}$,

$$\mathbf{\Pi} = [\boldsymbol{\pi}_0\, \boldsymbol{\pi}_1\, \ldots] = \lim_{n\to\infty} \Pi(n)$$

Recalling Eq. (A.14), it results

$$\mathbf{\Pi} = \lim_{n\to\infty} \Pi(n) = \lim_{n\to\infty} \Pi(0)P^n = \Pi(0) \lim_{n\to\infty} P^n \tag{A.24}$$

If the DTMC is irreducible, aperiodic and positive recurrent, then

$$\exists_n p_{ij}(n) > 0, \tag{A.25}$$

i.e., eventually any state is reachable from any state.

Using the Perron-Fronebius theorem it results that, under the above conditions, $\mathbf{\Pi}$ exists, is unique, and independent of $\Pi(0)$. Moreover, it results that

$$\begin{aligned} \mathbf{\Pi} &= \mathbf{\Pi}P \\ \sum_{j\in\mathcal{S}} \boldsymbol{\pi}_j &= 1 \\ \boldsymbol{\pi}_j &\geq 0 \end{aligned} \tag{A.26}$$

and

$$\lim_{n\to\infty} P^n = \begin{bmatrix} \mathbf{\Pi} \\ \vdots \\ \mathbf{\Pi} \end{bmatrix} \tag{A.27}$$

Note that if the chain starts with probability distribution $\mathbf{\Pi}$, the probability distribution does not change regardless of the number of steps. Thus, $\mathbf{\Pi}$ is called the stationary probability vector, or the steady-state probability vector, of the DTMC.

Remark A.6.1. *An irreducible aperiodic DTMC is positive recurrent iff there is a unique stationary probability vector associated with the DTMC satisfying Eq. (A.27).*

Given an irreducible, aperiodic, positive recurrent DTMC, $\boldsymbol{\pi}_j$ denotes the fraction of time steps spent on state j in the long run. Given that v_j is the expected number of steps between successive visits to state j, it further results that

$$\boldsymbol{\pi}_j = \frac{1}{v_j} = \lim_{n\to\infty} p_{jj}(n)$$

Next, it will be shown how these results apply to the cases where the chain has different periodicity and recurrent properties.

A.6.1.2 CMTCs

In the CTMCs case, the stationary probability vector is defined by

$$\boldsymbol{\Pi} = [\boldsymbol{\pi}_0\, \boldsymbol{\pi}_1\, \ldots]$$

$$\boldsymbol{\pi}_j = \lim_{t\to\infty} p_j(t)$$

Like in the case of DTMCs, and given that all CTMCs are aperiodic by definition, when a CTMC is irreducible and positive recurrent the following holds true:

1. The stationary probability vector exists and is unique, regardless of the initial conditions;
2. The stationary probability vector constitutes a probability distribution, thus $\sum_{j\in\mathcal{S}} \boldsymbol{\pi}_j = 1$;

Given the continuity of time, the probability $\boldsymbol{\pi}_j$ can be interpreted as the time proportion spent in state j over all time.

By differentiating Eq. (A.23), we obtain the following differential equation for the state probability distribution for CTMCs:

$$\frac{d\Pi(t)}{dt} = \Pi(0)e^{Qt}Q = \Pi(t)Q \tag{A.28}$$

The steady-state is characterised by having a null variation of the state probability distribution. It results that, for CTMC, the steady state values can be summarised by the following equations:

$$\begin{aligned} \boldsymbol{\Pi}Q &= 0 \\ \sum_{j\in\mathcal{S}} \boldsymbol{\pi}_j &= 1 \\ \boldsymbol{\pi}_j &\geq 0 \end{aligned} \tag{A.29}$$

From the above equations, it results that the individual terms are given by

$$\boldsymbol{\pi}_j \left(\sum_{(k\neq j)\in\mathcal{S}} q_{jk} \right) = \sum_{(k\neq j)\in\mathcal{S}} q_{kj}\boldsymbol{\pi}_k \tag{A.30}$$

Eq. (A.30) is referred to as the *rate balance equation* for state j, given that it balances the inflow and outflow to state j.

A.6.1.3 Embedded Markov Chains

Consider the evolution of a CTMC $\{X(t) : t \geq 0\}$ only at the time instants when a state change occurs. The equivalent stochastic process $\{X_n : n \geq 0\}$, where X_0 is the initial state, and X_n is the state after the n^{th} change, is a DTMC.

Definition A.6.1. *Given a CTMC $\{X(t) : t > 0\}$, the discrete time process $\{X_n : n \geq 0\}$, where X_n denotes the state reached by the CTMC after n state transitions, is a DTMC called the* Embedded Markov Chain *(EMC) of the CTMC.*

Consider a CTMC with states $i, j, k \in \mathcal{S}$. The time to jump from a state i to state j is exponentially distributed and mutually independent. Plus, $p_{ii} = 0$ in a EMC, since we are looking only at state changes. It results that the 1-step transition probabilities of the EMC corresponding to the CTMC is given by

$$\begin{aligned} p_{ij} &= 0, && \text{if } i = j \\ p_{ij} &= \frac{q_{ij}}{\sum_{k \neq i} q_{ik}}, && \text{if } i \neq j \end{aligned} \tag{A.31}$$

The CTMC and its EMC share the same communication classes, in particular, the CMTC is irreducible and positive recurrent iff its EMC is irreducible and positive recurrent. As detailed in Sec. A.3, an irreducible and positive recurrent DTMC has an unique stationary probability vector $\mathbf{\Pi} = [\boldsymbol{\pi}_0 \ \boldsymbol{\pi}_1 \ \ldots]$, with

$$\mathbf{\Pi} P = \mathbf{\Pi}$$

$$\sum_{j \in \mathcal{S}} \boldsymbol{\pi}_j = 1$$

$$\boldsymbol{\pi}_j \geq 0$$

Now, we can compute the relation between the stationary probability vector of the CMTC, denoted here by $\mathbf{\Pi}^c = [\boldsymbol{\pi}_0^c \ \boldsymbol{\pi}_1^c \ \ldots]$, and the stationary probability vector of its EMC, denoted by $\mathbf{\Pi}^d = [\boldsymbol{\pi}_0^d \ \boldsymbol{\pi}_1^d \ \ldots]$, which can be shown to be given by

$$\boldsymbol{\pi}_i^c = \frac{\boldsymbol{\pi}_i^d m_i}{\sum_{j \in \mathcal{S}} \boldsymbol{\pi}_j^d m_j}, \tag{A.32}$$

where m_j is the sojourn time of the CTMC state j which, when exponentially distributed, is given by

$$m_j = \frac{1}{\sum_{k \neq j} q_{jk}} \tag{A.33}$$

While $\boldsymbol{\pi}_j^c$ is the long-run time proportion that the CMTC spends in state j, $\boldsymbol{\pi}_j^d$ is the equivalent for discrete time, i.e., the proportion in number of steps spent in state j.

Given the above results, it is clear that many properties of a CTMC can be computed from the analysis of its corresponding EMC, using the DTMCs analysis techniques.

A.6.1.4 Special Cases

Periodic Case Since a CTMC cannot be periodic by definition, we will focus only on DTMCs. When the DTMC is irreducible and positive recurrent, but not aperiodic, $\lim_{n \to \infty} p_{jj}(n)$ does not exist. However, given that the DTMC is irreducible and periodic, all states have the same period d. As such, the following limit exists

$$\boldsymbol{\pi}_j = \lim_{n \to \infty} (nd) = \frac{d}{v_j}$$

Transient Case Since a DTMC, or a CMTC, spends 0 proportion time in transient states, given a transient state j, its limiting probability $\boldsymbol{\pi}_j$ will be 0.

Null Recurrent Case Null recurrent states appear only in infinite closed communication classes. The expectation value of the number of steps take to return to transient state j after leaving, v_j, is infinite, and as such, its limiting probability, $\boldsymbol{\pi}_j$, is 0.

Non-Irreducible Case Given a non-irreducible MC, with two or more positive recurrent classes, each class will have an unique limiting stationary probability vector. The overall probability vector will also be stationary, although not unique. Nevertheless, given an initial probability distribution, the limiting stationary probability still exists and is unique, as long as the chain is aperiodic. We will show how to compute the stationary probability for these cases.

These cases are often seen as deadlock or live-lock analysis, since some states might never be reached, depending on the initial probability distribution. Furthermore, these often correspond to unrecoverable failures in the modelled system.

Consider a DTMC with n closed communication classes and m transient states. Let the set of states be partitioned into $n+1$ sets:

$$\mathcal{S} = \left\{ 1, 2, \ldots, m, m+1, \ldots, m+s_1+1, \ldots, m+\sum_{j=1}^{n} s_i \right\},$$

where s_i is the number of states in the closed communication class i.

If the states are ordered such that the transient states appear first, followed by the closed communication classes states in the order of the class number, the transition probability matrix can be written as:

$$P = \begin{bmatrix} T & C_1 & C_2 & \cdots & C_n \\ 0 & P_1 & 0 & \cdots & 0 \\ 0 & 0 & P_2 & & 0 \\ \vdots & \vdots & & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & P_n \end{bmatrix},$$

Matrix T represents the one-step transition probability between transient states, C_i represents the one-step transition probability between transient states and states belonging to the closed communication class i, and P_i represents the one-step transition probability between states of the closed communication class i. Note that matrices T and C_i are sub-stochastic matrices, while P_i are stochastic matrices.

Recall that we are interested in computing the stationary limiting probability for a given initial probability distribution. To do so, consider now a transition probability matrix P^*, obtained by considering that each closed communication classes is composed by a single state:

$$P^* = \begin{bmatrix} T & C_1 & C_2 & \cdots & C_n \\ 0 & I & 0 & \cdots & 0 \\ 0 & 0 & I & & 0 \\ \vdots & \vdots & & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & I \end{bmatrix},$$

Matrix P^* models the same system as P disregarding transitions between states of the same closed communication class. As such, P^* can be used to study the system behaviour until absorption by one or more closed communication classes, allowing us to compute the probability

of entering each closed communication class. Consider know the evolution of the chain for k steps:

$$P^*(k) = P^{*k} = \begin{bmatrix} T(k) & C_1(k) & C_2(k) & \cdots & C_m(k) \\ 0 & I & 0 & \cdots & 0 \\ 0 & 0 & I & & 0 \\ \vdots & \vdots & & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & I \end{bmatrix},$$

where

$$\begin{aligned} T(k) &= T^k \\ C_i(k) &= \sum_{j=0}^{k-1} T^j C_i = \left(\sum_{j=0}^{k-1} T^j \right) C_i \\ k &= 1, 2, \ldots \end{aligned} \tag{A.34}$$

It is clear that since T and C_i, for all i, are sub-stochastic matrices, all the elements of $T(k)$ and $C_i(k)$ tend to zero as k goes to infinity. Furthermore, it can be shown that

$$\boldsymbol{F} = \lim_{k\to\infty} \sum_{j=0}^{k-1} T^j = (I - T)^{-1} \tag{A.35}$$

$\boldsymbol{F} = \left[\boldsymbol{f}_{ij}\right]$ is called the *fundamental matrix*, and $\boldsymbol{f}_{ij}$ gives the expected number of times, when starting in state i, state j is visited before absorption. Considering m_j the mean sojourn time in state j, the mean time before absorption when starting in a state i, b_i, is given by

$$b_i = \sum_{j=1}^{m} \boldsymbol{f}_{ij} m_j$$

Considering the initial probability of being in state i, p_i, the mean time before absorption by one or more closed communication classes of the Markov Chain is given by

$$b = \sum_{j=1}^{m} \left(\sum_{i=1}^{m} p_i \boldsymbol{f}_{ij} \right) m_j$$

Note that if the initial distribution contains only states which belong to closed communication classes, the mean time before absorption will be zero.

From Eqs. (A.34) and (A.35) we obtain

$$\boldsymbol{C}_l = \left[\boldsymbol{c}_{\boldsymbol{l}_{\boldsymbol{ij}}}\right] = \lim_{k\to\infty} \left(\sum_{j=0}^{k-1} T^j \right) C_l = \boldsymbol{F} C_l, \tag{A.36}$$

where $\boldsymbol{c}_{l_{ij}}$ gives the probability of reaching state j of the closed communication class l when starting in transient state i (for the system modelled by P^*).

We can now compute the long term probability of the original system by considering the limiting stationary probability of each closed communication class, which exists and is unique if the corresponding class is aperiodic. We will denote $\boldsymbol{P}_i$ the stationary limiting probability matrix associated with the communication class i, formed by s_i row vector copies of the class limiting stationary probability (see Eq. (A.27)). The long term probability of reaching state j of the closed communication class l, when starting in a transient state i is given by $\boldsymbol{g}_{l_{ij}}$, where

$$\boldsymbol{G}_l = \left[\boldsymbol{g}_{l_{ij}}\right] = \boldsymbol{F}\boldsymbol{C}_l\boldsymbol{P}_l \tag{A.37}$$

If closed communication class i is periodic, $\boldsymbol{P}_i$ does not exist. However, if we are studying a CMTC based on its EMC, we can still complete the stationary analysis for the CMTC by using the limiting stationary probability of the corresponding closed communication class of the CTMC.

A.6.2 Transient Analysis

The transient analysis is performed by observing the evolution of the Chapman-Kolmogorov equations for both the DTMC and CTMC.

In the case of DTMCs it was shown that the state probability vector evolution is described by

$$\Pi(n) = \Pi(0)P^n,$$

while for CTMC, it is described by

$$\Pi(t) = \Pi(0)e^{Qt}$$

Appendix B

Petri Net Analysis

In this chapter we cover briefly the various analysis methods of Petri nets, based on the *Coverability Tree*, which were implemented to complete this work. For other structural analysis methods not based on the coverability tree, or for more details, see for instance [Murata, 1989; Girault and Valk, 2003; Cassandras and Lafortune, 2008]. The algorithms described in Sections B.1, B.2 and B.3 where implemented using C++, while the performance measures, described in Section B.4, where computed using Matlab®.

B.1 Coverability Tree

For the purposes of analysis, we will consider Generalised Stochastic Petri Nets, as defined in Section 2.2. Recall that the state of a Petri net is given by the marking of the net, i.e., by the number of tokens in each place. Given an initial state, one can obtain the coverability tree[1], by firing each enabled transition for all markings, as described in algorithm B.1.1 . In the coverability tree, each node represents a different state (or a set of states) with arcs connecting these nodes. Each arc represents a transition fired in the original Petri net. Recall also that a stochastic transition is enabled only if no immediate transition is enabled. In Algorithm B.1.1 the symbol ω is used to denote "infinity" for an unbounded place, in the sense that $\omega + k = \omega$.

If the number of markings in the Petri net is finite, the coverability tree is denoted as *Reachability Graph*. Many Petri net qualitative properties can be obtained through the analysis of the coverability tree, some of which are detailed in the following sections.

B.1.1 Qualitative Properties

The following sections briefly details some of the most important structural properties used in this work.

B.1.1.1 Boundedness

Definition B.1.1. *A Petri net place $p_i \in P$ is called* k-bounded*, or* k-safe*, if $m_i(j) \leq k$ for all j, i.e., for all reachable states. A Petri net is k-bounded, or k-safe, if all its places are k-bounded. One-safe places and Petri nets are denoted safe places and safe Petri nets, respectively.*

Determining the boundedness of a Petri net is straightforward once the coverability tree is obtained, since we have for each marking the number of tokens. Note that for unbounded places we just know that the number of tokens is unbounded.

[1]Some authors use different meanings for *coverability tree* and/or different expressions, such as *Reachability Tree*.

Algorithm B.1.1: Coverability tree generation algorithm.

Input: GSPN, PN.
Output: Coverability tree, $\mathcal{C}$.

```
1  begin
2      Add a node to C corresponding to the initial marking M(0), mark it as the root
       node, and push it to the stack;
3      while stack is not empty do
4          Pop a marking M from the stack and mark it as done;
5          if no transition is enabled in marking M then
6              Tag marking M as a deadlock and as tangible;
7          else
8              if marking M' has immediate transitions enabled then
9                  Tag marking M' as vanishing;
10             else
11                 Tag marking M' as tangible;
12             end
13             foreach enabled transition t_i in marking M do
14                 Obtain the new marking, M' by firing t_i;
15                 if marking M' was not already marked done then
16                     if On the path from the root node to M' there exists a marking M''
                       such that m'_j ≥ m''_j for all j, and M' ≠ M'' then
17                         Replace m'_j by ω for each j. In this case, M' is said to cover M'';
18                         Add an arc from M to M'', associated with t_i;
19                     end
20                     Add a new node to C, corresponding to M', and add an arc from M to
                       M', associated with t_i;
21                     Push marking M' into the stack;
22                 end
23             end
24         end
25     end
26 end
```

B.1.1.2 Liveness and Deadlock

Definition B.1.2. *Given a Petri net with initial state $\mathcal{M}_0$, a transition t_j is said to be live if, for all reachable states $\mathcal{M}_i$, there is a firing sequence starting in $\mathcal{M}_i$, such that t_j is fired. A Petri net is live if all its transitions are live.*

Definition B.1.3. *Given a Petri net, a deadlock state corresponds to a reachable state where none of the transitions are fireable. A Petri net is deadlock-free if it contains no reachable deadlock state.*

As detailed in Algorithm B.1.1, the deadlock states are marked as so during the algorithm execution.

B.2 Continuous Time Markov Chain

The main focus of this work on analysis is on bounded Petri nets. As such, the remainder of this chapter is based on the coverability tree for bounded Petri nets, i.e., the reachability graph.

While the coverability tree allowed us to obtain some qualitative properties of the Petri net, the equivalent CTMC will enable obtaining quantitative properties, in the sense of studying the Petri net behaviour over time. As such, the goal is to obtain the equivalent CMTC in order to use the analysis techniques described in Appendix A.

Obtaining the CTMC from the reachability graph consists mainly on removing all vanishing states, as detailed in Algorithm B.2.1. Recall that a vanishing state is one which has immediate transitions enabled, meaning that the CMTC will correspond to a graph where only tangible states exist. The only issue here is when removing one vanishing state with two or more immediate transitions, since the probability of each transition will depend on the weights and rates of the enabled immediate and stochastic timed transitions, respectively, as shown in the algorithm.

Having obtained the CTMC, one needs to compute the infinitesimal generator, Q, through Algorithm B.2.2.

Although not included in Algorithm B.2.1 (for clarity sake), each stored arc has a list of associated "original" Petri net transitions, which is important when computing the transitions throughput.

B.3 Communication Classes

Finally, in order to fully use the analysis techniques described in Appendix A, one needs to compute the classes of the CTMC and determine if they are recurrent and, for the corresponding EMC, the class period. The algorithm used to compute the communication classes is based on the *Strong Connected Components*, part of graph theory. A strong connected component in graph theory is equivalent to a communication class in Markov chain theory. Furthermore, if a class is recurrent, then the class is a closed communication class.

In order to compute both the communication classes and their periods simultaneously, one implemented the algorithm *SCC and periods* described in [Ocasio, 2009], which is a modified version of the original Tarjan algorithm [Tarjan, 1972] (modified to include the periods).

Note that the communication classes can be computed using as input the reachability graph or the Markov chain (CTMC or EMC). If used with the Markov chain, some information will not be available since only tangible states are considered. On the other hand, the period output only makes sense if the EMC is used as input. In this work, the EMC was used as the algorithm input.

Algorithm B.2.1: Continuous Time Markov Chain generation algorithm.

Input: Reachability graph, $\mathcal{R}$.
Output: Continuous Time Markov Chain.

begin
  Assign to all states an initial probability, p_i, equal to zero if state i is not the initial state, and equal to one otherwise;
  foreach *vanishing state* $\mathcal{S}$ *in* $\mathcal{R}$ **do**
    Let $T' = \{t'_1, \ldots, t'_n\}$ be the set of transitions associated with the output arcs of $\mathcal{S}$;
    foreach *output arc* i*, associated with transition* t'_i **do**
      Compute the transition firing probability, $p_i^t = \frac{w_i}{\sum_{j=1}^{n} w_j}$
    end
  end
  foreach *vanishing state* $\mathcal{S}$ *in* $\mathcal{R}$ **do**
    Let $T' = \{t'_1, \ldots, t'_n\}$ be the set of transitions associated with the output arcs of $\mathcal{S}$;
    if $\mathcal{S}$ *has no input arcs* **then**
      foreach *output arc* i*, associated with transition* t'_i **do**
        Let $\mathcal{S}'$ be the state reached through t'_i, p' its initial probability, and p the initial probability associated with state $\mathcal{S}$;
        $p' \leftarrow p' + p * p_i^t$;
      end
      Remove state $\mathcal{S}$ and all its output arcs;
    else
      foreach *output arc* i*, associated with transition* t'_i **do**
        Let $\mathcal{S}'$ be the state reached through t'_i, and $T'' = \{t''_1, \ldots, t''_n\}$ be the set of transitions associated with the input arcs of $\mathcal{S}$;
        foreach *input arc* j*, associated with transition* t''_j **do**
          Let $\mathcal{S}''$ be the state input arc j connects from;
          if t''_j *is immediate* **then**
            Create an arc from state $\mathcal{S}''$ to state $\mathcal{S}'$, associated with an immediate transition with firing probability $p_j^t * p_i^t$, equivalent to firing t''_j and t''_i;
          else
            Let λ_j the the rate associated with transition t''_j;
            Create an arc from state $\mathcal{S}''$ to state $\mathcal{S}'$, associated with a stochastic transition with rate $\lambda_j * p_i^t$, equivalent to firing t''_j and t''_i;
          end
          Remove input arc j;
        end
        Remove output arc i;
      end
      Remove state $\mathcal{S}$.
    end
  end
end

Algorithm B.2.2: Computing the CTMC infinitesimal generator, Q.

Input: Continuous Time Markov Chain.
Output: Infinitesimal generator, Q;

```
1  begin
2  |  Let S = {S_1, ..., S_n} be the set of states;
3  |  foreach state S_i do
4  |  |  Let 𝔖 be the set of m states connected through output arcs from state S, such
   |  |  that there are no duplicate states;
5  |  |  foreach state S_j ∈ 𝔖 \ S_i do
6  |  |  |  q_ij = Σ rates of all transitions associated to arcs connecting S_i to S_j;
7  |  |  end
8  |  |  q_ii = −Σ_{j≠i} q_ij;
9  |  end
10 end
```

Once the communication classes are obtained, determining if each class is open or closed is straightforward (recall one is considering only finite Markov chains). If all states in a given class only have transitions to states of the same class, then that communication class is closed, otherwise the communication class is open. All open communication classes are transient, while the closed communication classes are recurrent. However, if there is more than one closed communication class (reducible Markov chain), reaching that class depends on the initial state (see Section A.6.1.4 for more details).

B.4 Performance Measures

Having the CTMC, the EMC and the communications classes, one can use the techniques described in Appendix A to perform transient and/or stationary analysis of the chains. Those results can then be used to computer performance measures of the corresponding Petri net [Viswanadham and Narahari, 1992; Bause and Kritzinger, 2002]. The following sections detail some of the performance measures which can be obtained.

In the following sections consider a GSPN with an associated EMC with s reachable tangible states. As shown in Appendix A, π_j is the stationary probability of the tangible state j, with $\mathcal{M}_j$ corresponding to the Petri net marking associated with that state j.

B.4.1 Probability that a Condition Holds

The probability that a particular condition C holds is computed by considering the probability of being in any state where the conditions is satisfied:

$$Pr\left(C\right) = \sum_{j \in \mathcal{S}_1} \pi_j$$

where $\mathcal{S}_1 = \{j \in \{1, 2, \ldots, s\} : C \text{ is satisfied in } \mathcal{M}_j\}$.

B.4.2 Probability of Having a Number of Tokens in a Place

The probability of having exactly k tokens in a place p_i is computed by considering the probability of being in any state where place p_i has exactly k tokens:

$$Pr\left(\#\left(p_i\right) = k\right) = \sum_{j \in \mathcal{S}_2} \pi_j$$

where $\mathcal{S}_2 = \{j \in \{1, 2, \ldots, s\} : \#\left(p_i\right) = k \text{ in } \mathcal{M}_j\}$.

B.4.3 Expected Number of Tokens in a Place

The expected number off tokens in a place p_i is computed by using the probability of having k tokens in place p_i for all possible number of tokens in that place:

$$E\left[\#\left(p_i\right)\right] = \sum_{k=1}^{K} Pr\left(\#\left(p_i\right)\right)$$

where K is the maximum possible number of tokens in place p_i for all reachable tangible markings.

B.4.4 Transition Throughput Rate

The throughput of an exponential transition t_j is computed by considering its firing rate over the probability of all states where t_j is enabled:

$$Tr\left(t_j\right) = \sum_{i \in \mathcal{S}_3} \pi_i \lambda_j$$

where $\mathcal{S}_3 = \{i \in \{1, 2, \ldots, s\} : t_j \text{ is enabled in } \mathcal{M}_i\}$.

The throughput of an immediate transition t_j can be computed by considering the throughput of all exponential transitions which lead immediately to the firing of transition t_j, i.e., without crossing any tangible state, together with the probability of firing transition t_j among all the enabled immediate transitions.