TÉCNICO LISBOA

Multi-Robot Surveillance Task Planning Under Uncertainty

António Maria Albuquerque Matos de Paula

Thesis to obtain the Master of Science Degree in

Aerospace Engineering

Supervisor(s): Prof. Pedro Urbano Lima

Examination Committee

Chairperson: Prof. José Fernando Alves da Silva
Supervisor: Prof. Pedro Urbano Lima
Member of the Committee: Dr. Bruno Lacerda

December 2021

Declaration

I declare that this document is an original work of my own authorship and that it fulfills all the requirements of the Code of Conduct and Good Practices of the Universidade de Lisboa.

Acknowledgments

First and foremost, I want to thank Professor Pedro Lima for giving me the opportunity to conduct this research under his supervision. His humaneness when I was in a difficult situation proved to be a tipping point and this thesis would not have been finished without him. I would also like to show my appreciation for all the help that Carlos Azevedo provided me. His guidance was essential throughout every step of this research work, and his insights and expertise were invaluable in writing this thesis.

I want to acknowledge the funding received from Mathworks that specifically supported the work done in Chapter 5, through a joint Mathworks-Robocup Federation project, obtained via application.

Resumo

O uso de equipas multi-robô tem sido cada vez mais considerado como uma boa solução para, ao mesmo tempo, aumentar a confiabilidade e diminuir tempos de paragem de aplicações tanto industriais como científicas. Com aparecimento de equipas multi-robô, surgiram desafios em obter estratégias de coordenação eficientes que levem em conta incerteza quanto à duração das acções executadas pelos robôs. Modelos baseados nas *Generalized stochastic Petri nets with rewards* (GSPNRs) onde os *tokens* são interpretados como robôs proporcionam um formalismo capaz de modelar problemas multi-robô com um espaço de estados reduzido, e em simultâneo providenciam uma modelação estócastica explícita da incerteza quanto à duração das acções executadas. Esta tese estende vários estudos recentes baseados nestes modelos GSPNR para que possam modelar equipas robóticas heterogéneas constituídas por diferentes tipos de robôs e para abilitar a modelação de um único atributo arbitrário intrínsico a um tipo de robô: e.g. a bateria de cada robô. Além disto, a tese desenvolve uma ferramenta para MATLAB que fornece uma implementação das GSPNRs e métodos para calcular políticas óptimas sobre estes modelos. A ferramenta desenvolvida também inclui a possibilidade de executar estas políticas óptimas diretamente em robôs, comunicando com estes através do *Robotic Operating System* (ROS). Por último, um problema de planeamento de inspeções de painéis fotovoltaicos é resolvido, e a política óptima foi testada em robôs simulados no *Gazebo*.

Palavras-chave: Sistemas multi-robô, Coordenação multi-robô, Planemento sob incerteza, Redes estocásticas de Petri generalizadas com recompensas, Sistemas heterogéneos multi-robô

Abstract

The usage of multi-robot teams has increasingly been considered as a good solution for improving reliability and productivity in industrial and scientific applications. With the advent of the multi-robot team, challenges have emerged in coming up with efficient coordination strategies that take into account uncertainty regarding to action duration. Generalized stochastic Petri nets with rewards (GSPNRs) models that interpret tokens as robots provide a formalism capable of modeling these multi-robot scenarios with a compact state space, whilst also allowing an explicit stochastic modeling of action execution time. The present thesis extends recent approaches based on these GSPNRs in order to model heterogeneous teams made up of multiple types of robots and to allow the inclusion of a single robot-specific attribute such as battery. Additionally, a software toolbox for MATLAB is developed that provides an implementation of GSPNRs and methods to calculate optimal policies on these models. The software package also includes the possibility of executing these policies on real robots by interfacing with Robotic Operating System (ROS) middleware. Finally, an inspection task planning scenario is solved and results were obtained in realistically simulated robots in Gazebo.

Keywords: Multi-robot Systems, Multi-robot coordination, Planning under uncertainty, Generalized stochastic Petri nets with rewards, Heterogeneous multi-robot systems

Contents

Acknowledgments . . . iii

Resumo . . . iv

Abstract . . . v

List of Tables . . . viii

List of Figures . . . ix

1 Introduction **1**

1.1 Motivation . . . 1

1.2 Objective and Contributions . . . 2

1.3 Dissertation Outline . . . 3

2 Background **4**

2.1 Petri Nets . . . 4

2.1.1 Marked Ordinary Petri Nets . . . 4

2.1.2 Generalized Stochastic Petri Nets . . . 5

2.1.3 Generalized Stochastic Petri Nets with Rewards . . . 7

2.2 Markov Decision Processes . . . 7

2.3 Policies . . . 8

3 Related Work **9**

3.1 Petri Net Approaches . . . 9

3.2 Other Approaches . . . 11

3.3 Software for Modelling, Analysis and Execution of Multi-Robot Systems . . . 12

4 Modelling Multi-Robot Systems with GSPNRs **13**

4.1 Homogeneous Systems . . . 13

4.1.1 Decision-Action Graphs . . . 14

4.1.2 Building GSPNR Models from Decision-Action Graphs . . . 16

4.2 Heterogeneous Systems . . . 17

4.2.1 Algorithm to Build GSPNR Model . . . 21

4.3 Modelling Attributes . . . 23

4.3.1 System Attributes . . . 24

4.3.2 Robot Attributes 25

5 Multi-Robot GSPNR Toolbox 33

5.1 Overview and Architecture 33

5.2 Creating GSPNR Models 34

5.2.1 Using the API 34

5.2.2 Using a GUI 35

5.2.3 Using a Topological Map 36

5.3 Policy Synthesis and Evaluation 39

5.3.1 Conversion to MDP 40

5.3.2 Value Iteration 44

5.3.3 Policy Evaluation on GSPNRs 45

5.4 Execution Manager 46

5.4.1 Executing GSPNR Task Plan 46

5.4.2 Using the API 48

5.5 Computational Performance and Scalability 50

5.5.1 Increasing number of locations 52

5.5.2 Increasing number of robots 55

6 Experiments and Results 58

6.1 Home Vacuuming Scenario 58

6.1.1 Problem Description 58

6.1.2 GSPNR Model 60

6.1.3 Results 64

6.2 Solarfarm Inspection Scenario 66

6.2.1 Problem Description 66

6.2.2 GSPNR Model 69

6.2.3 Results 73

7 Conclusions 78

Bibliography 80

List of Tables

4.1 Examples of Full Transition Models for $Travel$ (left) and $Recharge$ (right) action 28

6.1 Rough approximation for discharging probabilities for *Navigate* and *Vacuum* actions for all locations and navigation edges used to obtain the policy with less rigorous parameters; . 65

6.2 Comparison between average time between charges $\Delta_{Charges}$, average time taken to vacuum the entire house $\Delta_{VacuumedAll}$, average reward $\bar{r}$, and vacuums per hour, for both policies executed in Gazebo 65

6.3 Comparison between simulating GSPNR plan without robots and executing it with simulated robots in Gazebo for "Charge" action (at *low* and *medium* battery level) and time taken to vacuum all locations; These results were obtained by executing GSPNR plan obtained with model with most accurate parameters; 66

6.4 Comparison between the GSPNR plans simulated in the model and executed in Gazebo. Three GSPNR plans are compared: a plan with the **random policy**, with the **handcrafted policy** and the **optimal policy**. $downtime$ is the proportion of time, where at least one robot was waiting to be charged, $\Delta_{InspectAll}$ is the average time between inspecting all 4 panels, and $\bar{r}$ is the average accumulated reward. 77

List of Figures

2.1 Before transition t_0 (in blue) fires; 5

2.2 After transition t_0 fires; 5

2.3 Under the urgency assumption, this GSPN has no conflict, t_0 will always fire; 6

2.4 After transition t_0 fires; 6

2.5 Small example of a MDP 8

4.1 Example of a Multi-robot system with 2 robots that can travel between solar panels and inspect one; 14

4.2 Decision-action graph for an inspection scenario; in green are decision nodes, and in red are action nodes; 15

4.3 An action model for arbitrary action A; the immediate transition t_D models the decision to execute A, and the exponential transition t_F models finishing the execution of A; 16

4.4 Model of two solar panels with inspection and travelling tasks, built with *DecAct-G* in Figure 4.2 17

4.5 *DecAct-G* of a heterogeneous system with a recharging task done in cooperation by the two robot-types 19

4.6 Separate and merged models for a synchronized cooperative action between two different robot types 20

4.7 Separate and merged models for an asynchronized action between two different robot types 21

4.8 Action models for every action node in *DecAct-G* depicted in Figure 4.2 and for the two types of robots involved. These action models were used to build full GSPNR model shown in Figure 4.9; 23

4.9 GSPNR model built with Algorithm 3 using the heterogeneous *DecAct-G* in Figure 4.5, and action models depicted in Figure 4.8; 24

4.10 Two examples of resource places usages based in the GSPNR model of Figure 4.4; . . . 25

4.11 Three GSPNR action models for $Travel$: template action model (above), action model for battery level B2 (left), action model for battery level B1 (right) 26

4.12 Three GSPNR action models for the small UGV $Recharge$ action: template action model (above), action model for battery level B1 (left), action model for battery level B0 (right) . . 27

4.13 Attribute models for transition models specified in Table 4.1; 28

4.14 *DecAct-G* representing system where small UGVs can travel between location ”Panel1” and ”Panel2”, and recharge in cooperation with UGVs at ”Panel1” 30
4.15 Full GSPNR model for *DecAct-G* found in Figure 4.14, with the smaller UGV’s battery discretized into three levels. 31

5.1 Overview of the toolbox’s architecture 34
5.2 GSPNR model built in code snippets 34
5.3 Example of GreatSPN ”.PNPRO” file 36
5.4 Visualization of input topological map; 39
5.5 Visualization of GSPNRs in input `models` 39
5.6 Outline of *Policy Synthesis* module 40
5.7 Partial GSPNR model with two immediate transitions that represent arbitrary choices; . . 40
5.8 Partial GSPNR model with two immediate transitions that represent a random switch; . . 40
5.9 Partial GSPNR model with both types of immediate transitions; 41
5.10 Example GSPNR model equivalent to MDP in Figures 5.11 and 5.12, without and with wait states, respectively; 43
5.11 States and Actions of Equivalent MDP of GSPNR found in Figure 5.10, without wait states; 43
5.12 States and Actions of Equivalent MDP of GSPNR found in Figure 5.10, but with wait states; 43
5.13 The same GSPNR Model of Figure 4.4, with the action places outlined in red and their exponential transitions outlined in green; 46
5.14 GSPNR model of a synchronized cooperative action where the transition outlined in green can only fire when both robots have finished their actions; 47
5.15 GSPNR model of an action with result, where according to the result of the action execution, one of the transitions outlined in green will fire; 47
5.16 Example YAML file that specifies input `YAML_filepath` for GSPNR model in figure 5.13, outlined in red are the action places; 50
5.17 Code example that defines input `action_map` for a single place of GSPNR model in Figure 5.13, ”Inspecting_panel1”; 50
5.18 Example of domestic multi-robot scenario, with three separate locations to clean and three robots; 50
5.19 Summary of measurements taken for each experiment; 51
5.20 Topological map of the domestic scenario with N locations 52
5.21 Computational performance when creating GSPNR model with an increasing number of locations and building equivalent MDP 52
5.22 Time taken to create GSPNR model of a scenario with an increasing number of locations, using a topological map; 53
5.23 Time taken to create equivalent MDP model for a multi-robot scenario with increasing amount of locations, MDP with wait states (in red) and without (in blue); 53

5.24 For an increasing amount of locations, the number of states in the equivalent MDP without wait states (blue), with wait states (red), and a comparison with the number of places squared (in black); . . . 54
5.25 Computational performance when executing GSPNR model with an increasing number of locations . . . 54
5.26 Topological map of the domestic scenario with 4 locations . . . 55
5.27 Computational performance when creating GSPNR models up to 6 robots; . . . 55
5.28 Time taken to create GSPNR with 1-20 robots (left) and time taken to create equivalent MDP model, up to 6 robots (right) . . . 56
5.29 For an increasing amount of robots, the number of states in the equivalent MDP without wait states (blue), with wait states (red), and a comparison with the number of places squared by the number of robots (in black) . . . 56
5.30 Computational performance when executing GSPNR model with an increasing number of robots . . . 57
5.31 Size in megabytes of the `GSPNR` object (in blue) and the equivalent `MDP` object (in red) for an increasing amount of reachable states/markings; . . . 57

6.1 Gazebo world used, with the topological map superimposed, and the corresponding vacuumed areas for each location in red; . . . 59
6.2 Turtlebot robot charging at location ”BS”; . . . 59
6.3 Topological map of the vacuuming scenario: 6 locations + a base station (BS) where robots can recharge; . . . 61
6.4 GSPNR Model for Vacuuming at *L2*; . . . 62
6.5 Global counter that restricts vacuums; . . . 62
6.6 GSPNR Model for Navigating from *L1* to *L2*; . . . 63
6.7 GSPNR Model for Charging at *BS*; . . . 64
6.8 Gazebo world used in solarfarm simulation, with the topological map overlapped and the areas transversed by robots carrying out inspections outlined in red; . . . 67
6.9 Clearpath Jackal with a standard LMS laser range finder, and a custom made frame with a charging plug; . . . 68
6.10 Clearpath Warthog with 2 LMS laser range finders, and custom made charging port; . . . 68
6.11 Jackal coupled to the Warthog to simulate recharging; . . . 68
6.12 Solarfarm topological map; . . . 69
6.13 GSPNR Model for Inspecting ”Panel2”; . . . 70
6.14 GSPNR Model for Inspecting ”Panel2”; . . . 71
6.15 GSPNR Model for Jackal Navigation between ”Panel2” and ”Panel1”; . . . 71
6.16 GSPNR Model for synchronized cooperative recharging action at ”Panel2”; . . . 72
6.17 GSPNR Model for Warthog navigation between ”Panel2” and ”Panel1”; . . . 73

6.18 Average accumulated reward as a function of time. Obtained in Gazebo, for three different policies: the **optimal policy** (in blue), the **handcrafted policy** (in green), and the **random policy** (in red). 74
6.19 Number of inspected panels on average, along one hour runs of Gazebo simulation, for the three different policies: the **optimal policy** (in blue), the **handcrafted policy** (in green), and the **random policy** (in red). 75
6.20 Number of inspected panels on average, along ten hour runs of model simulation, for the **optimal policy** (in blue), and the **handcrafted policy** (in green); 75
6.21 The lines depicted represent the number of inspections as a function of time and the shaded areas the periods when one of the UGVs is charging. Both graphs show the best run of the optimal policy (in blue) and the handcrafted policy (in green). 76

Chapter 1

Introduction

1.1 Motivation

Small, autonomous robots have become widely used in both scientific and industrial fields. The possibility of customizing their payload with economical and off-the-shelf sensors has broadened the range of uses for these kind of robots. From small Unmanned Aerial Vehicles (UAVs) with the ability to hover and traverse distances quickly, to ground-based platforms that can allow for the usage of robotic arms that can grip and manipulate objects, these systems can be an adequate solution in a wide spectrum of industries. Some recent applications can be found in an assortment of industries: precision agriculture in the agro-food industry [1], border surveillance in the security sector [2], pipeline inspection in the oil and gas industry [3].

However, while these small robots have become more economical and easier to deploy they generally suffer from a significant shortfall: they are solely battery powered, and limitations present on current battery technology account for agents with relative short autonomy. On the other hand, their low-cost allows for the deployment of multiple robots at once, and recent research has examined the use of multi-robot teams to mitigate these battery limitations.

The utilization of heterogeneous teams consists in using together different robot types with distinct capabilities. The main advantage of heterogeneous teams is to allow for solutions that exploit each robot-types' strengths more effectively. For example, using a UAV team coupled with a ground-based vehicle, where the UAVs exploit their aerial perspective and better mobility, and the ground-based vehicle can recharge the UAVs. In any long-running cyclic tasks, this approach becomes particularly powerful, as the overall multi-robot heterogeneous system is capable of covering bigger areas with fewer downtime due to the increased autonomy.

Multi-robot heterogeneous teams only provide such advantages when correctly and efficiently coordinated. Moreover, the use of heterogeneous teams originates the problem of enforcing cooperation while maintaining efficiency. Ideally, this efficiency should be defined against some determined criterion, e.g. maximizing visits to locations, minimizing number of charges, etc. A promising line of research advocates the use of formal models to capture the multi-robot problem and the use of dynamic program-

ming algorithms to solve these decision-making and planning problems. These approaches provide formal guarantees such as showing that the multi-robot system has no deadlocks, and at the same time allow synthesizing optimal policies that coordinate the multi-robot system. Nonetheless, there is a lack of models that can capture heterogeneous teams in a compact manner, while also being able to track robot-specific attributes such as battery and uncertainty in the duration of action execution.

There is also a lack of software tools that allow users to solve multi-robot problems in an comprehensive manner, where users can model the problem and then synthesize optimal policies with the same tool. Furthermore, to the best of our knowledge, there is no such tool that allows the execution of the obtained policies directly on real robots.

1.2 Objective and Contributions

The main objective of this thesis is to develop methods that can coordinate heterogeneous multi-robot systems by optimizing some defined performance criteria, while taking into account various constraints. To achieve this, this dissertation will use an extension of Generalized stochastic Petri nets - Generalized stochastic Petri nets **with rewards** (GSPNRs). This will allow to first model the multi-robot problem, and then convert it into the equivalent Markov Decision Process (MDP) model. Policy synthesis algorithms can be applied in this equivalent MDP that extract the optimal policy by optimizing the utility function while taking into account uncertainty in the duration of actions. This problem has been solved as reported in [4] and [5]. This work introduces three novel contributions, required to solve the motivating problem:

- Extend current formal models based in GSPNRs into a general framework that is capable of modelling a multi-robot task planning problem. This extension allows modelling of heterogeneous robot teams, and will also include the ability to model system wide resources, e.g. counters, or track robot specific attributes, e.g. battery.

- Build and test a software package in MATLAB environment capable of building GSPNR models, obtaining optimal policies for these models, and executing the obtained policies in real robots, by integrating with the Robotic Operating System (ROS).

- Solve a specific inspection task planning problem, described in Problem 1. This will use the modelling framework established, and utilize the software package developed to obtain and execute optimal policies in simulated robots and environment.

Problem Definition: Solar Farm Inspection Task Planning

Even though approaches developed for this thesis are relatively generic and can be applied to wide range of problems in diverse application areas, in this thesis we chose to solve a specific multi-robot problem of interest to the research group where it has been developed:

Problem 1: Consider an inspection scenario, where the photo-voltaic panels in different locations of a solar farm must be continuously inspected by several small unmanned ground vehicles (UGVs) with limited autonomy, and a single larger UGV is used to recharge the smaller ones and so increase the autonomy of the system. How should the robot team be coordinated in order to make this inspection process as efficient as possible? How can we take into account the battery constraints of the small UGVs and the uncertainty associated with the navigation, inspection and charging tasks?

1.3 Dissertation Outline

Chapter 2 provides some background on theoretical concepts used throughout the research. Chapter 3 outlines previous research relevant to this dissertation, and on which this work is based on. Chapter 4 establishes the extension of the modelling framework in order to allow for heterogeneous teams and the modelling of system-wide or robot-specific attributes. Chapter 5 provides an overview of the architecture and the functionality of the developed software package along with analyzing its performance and scalability. Chapter 6 focuses on two experiments done with multi-robot scenarios, including the photo-voltaic plant inspection problem. Chapter 7 summarizes conclusions reached along the research work, explains limitations found regarding the model-based approach used and provides possible avenues for future work.

Chapter 2

Background

The chapter that follows introduces the discrete-event dynamic system (DEDS) formalisms that will be used throughout the research work. Firstly, two kinds of Petri nets are presented, marked ordinary Petri nets (MOPNs) and generalized stochastic Petri nets (GSPNs). Secondly, Markov decision processes (MDPs) are formalized, and finally, policies are defined in the context of GSPNs and MDPs.

2.1 Petri Nets

2.1.1 Marked Ordinary Petri Nets

A marked ordinary Petri net [6] is formally defined by the following tuple:

$$\text{PN} = (P, T, I(.), O(.), \boldsymbol{m}_0)$$

$P = (p_1, p_2, \cdots, p_{|P|})$ is a finite set of places and $T = (t_1, t_2, \cdots, t_{|T|})$ is a finite set of transitions. $I(.) : P \times T \to \mathbb{N}$ is a function defined over the set of input arcs (i.e. arcs connecting places to transitions) specifying their multiplicity. The multiplicity of each input arc corresponds to the number of tokens consumed in the place connected to the transition that fires. In the same manner, $O(.) : T \times P \to \mathbb{N}$ is the output multiplicity function, defined over the set of output arcs (i.e. arcs connecting transitions to places). For output arcs, the multiplicity corresponds to the number of tokens created in the place connected to the transition that fires. Finally $\boldsymbol{m}_0 = (m_{01}, m_{02}, \cdots, m_{0P})$ is the initial marking. A marking m is an assignment of tokens to places, and is represented by a vector, where each *i-th* entry is the number of tokens assigned to place p_i.

For transition t, the set of its input arcs is denoted by $I(t)$ and the set of output arcs by $O(t)$. The *input places* of a transition t are the places connected to t by an input arc, while *output places* of t are places connected to t by output arcs.

The dynamics of a marked PN correspond to the firing of transitions, which consume and create tokens, and so provide a mechanism for change in state, as represented by a certain marking. There are two rules that govern the firing of transitions:

1. **Enabling Rule** - a transition t is *enabled* if and only if each input place is marked by at least the number of tokens specified by the arc multiplicity. The set of enabled transitions in a marking m is denoted by $E(m)$, and $t \in E(\boldsymbol{m}) \rightarrow \forall p_i \in P : m_i \geq I(p_i, t)$

2. **Firing Rule** - only an enabled transition may fire, and only one transition fires at a time. When a transition t fires, it removes the number of tokens specified by the input arc multiplicity from each input place, and creates the number of tokens specified by the output arc multiplicities in each of its output places. An arbitrary transition t, enabled in making m, when fired, produces a marking m' such that: $m' = m + O(t) - I(t)$

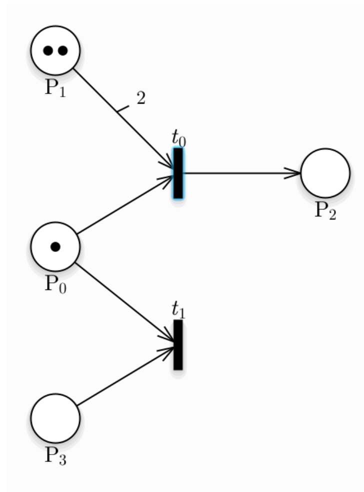


Figure 2.1: Before transition t_0 (in blue) fires;

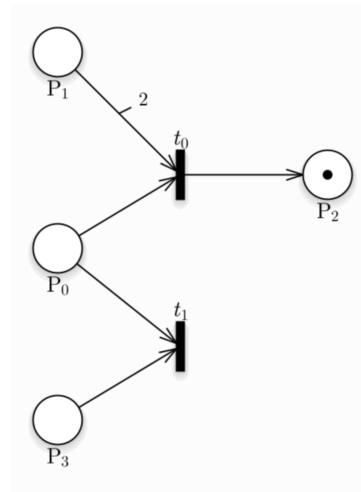


Figure 2.2: After transition t_0 fires;

An example of these firing dynamics can be seen in Figures 2.1 and 2.2. The same MOPN can be seen in both figures, and arc multiplicities are assumed to be one if they have no labels. The only arc to have a multiplicity different than one is the input arc between P_1 and t_0. In Figure 2.1, the only transition enabled is t_0, as t_1 requires place P_3 to have at least one token, which it does not have. As can be seen in Figure 2.2, the firing of t_1 consumes the two tokens in place P_1 - as the arc between P_1 and t_0 has a multiplicity of two - and one token in P_0 - because the corresponding arc has a multiplicity of one. Additionally, it produces only one token in place P_2, according to the multiplicity of the output arc.

2.1.2 Generalized Stochastic Petri Nets

Generalized stochastic Petri nets (GSPN) extend MOPNs with the concept of stochastically timed transitions. Formally, a generalized stochastic Petri net (GSPN) is a tuple:

$$\text{GSPN} = (P, T, I(.), O(.), W(.), \boldsymbol{m}_0)$$

The meaning of P, $I(.)$, $O(.)$ and $\boldsymbol{m}_0$ is as with MOPNs. GSPNs however divide the transition set into mutually exclusive subsets: $T = T_I \cup T_E$. T_I is the set of immediate transitions, and T_E the set of

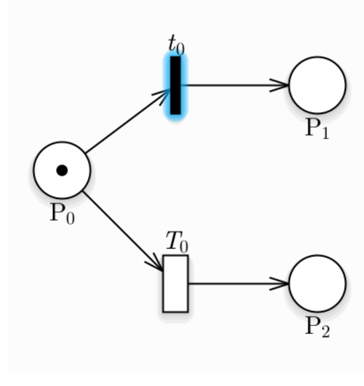


Figure 2.3: Under the urgency assumption, this GSPN has no conflict, t_0 will always fire;

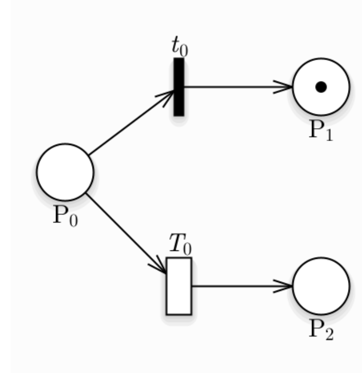


Figure 2.4: After transition t_0 fires;

exponential transitions. Immediate transitions fire instantaneously, and exponential transitions fire with a stochastic delay determined by an exponentially distributed random variable. The probability that an exponential transition will fire with no delay is zero, and as a consequence, immediate transitions have an inherent priority over exponential transitions. In any marking where both an immediate transition and exponential transitions are enabled, the immediate transition will always fire before the exponential ones. In literature, this property is often named the *urgency assumption*. Figures 2.3 and 2.4 demonstrate this concept. The same GSPN can be seen in both figures, and at first sight, t_0 and T_0 appear to be in conflict as both are enabled by the single token in P_0. However, because immediate transitions always fire before exponential ones, t_0 will always be the transition that fires.

The weight function, $W(.) : T \rightarrow \mathbb{R}_{\geq 0}$ maps a transition t_k to a real positive number, and is denoted by $W(t_k)$. This value w_k denotes the firing rate that characterizes the exponential distribution of exponential transitions. The probability of transition t, with weight $W(t) = \lambda$ fires with delay d is the following:

$$P(d|\lambda) = \frac{1}{\lambda} e^{-\lambda d} \tag{2.1}$$

The weight of a given transition dictates the probability of that transition firing among all the enabled transitions of the same conflict set. The probability that a (immediate or timed) transition t_k, shall fire first in an enabled marking $\boldsymbol{m}_i$ is given by the Equation 2.2. As stated previously, here w_k can take two different meanings, according to the type of transition.

$$P\{t_k|\boldsymbol{m}_i\} = \frac{w_k}{q_i} \ ; q_i = \sum_{t_k \in E(\boldsymbol{m}_i)} w_k \tag{2.2}$$

With the partitioning of the transition set into immediate and exponential transitions, markings can be divided into vanishing and tangible markings. A vanishing marking is one where at least one immediate transition is enabled $VM = \{\forall \boldsymbol{m} \in VM \mid \exists t \in E(\boldsymbol{m})$ and $t \in T_I\}$, and a tangible marking is one where all enabled transitions are exponential ones - $TM = \{\forall \boldsymbol{m} \in TM \mid \forall t \in E(\boldsymbol{m})$ and $t \in T_E\}$. The symbols VM and TM denote the sets of all vanishing and tangible markings, respectively, and because of the

urgency assumption, we can assume$VM \cap TM = \emptyset$.

The reachability set of a GSPN $G = (P, T, I, O, W, \boldsymbol{m}_0)$ is the set of all markings reachable from its initial marking $\boldsymbol{m}_0$. The reachability graph of GSPN G, $R(G)$, is the labeled directed graph (V, E) with each vertex $m \in V$ being a reachable marking, and each edge $(m_i, m_j) \in E$ being labeled with transition t that leads the GSPN from marking m_i to marking m_j: $m_i \xrightarrow{t} m_j$.

A place in a GSPN is bounded if it does not contain more than k tokens in all reachable markings, including the initial marking. A GSPN is bounded if all its places are also bounded. A bounded GSPN implies that the reachability set and graph is finite.

2.1.3 Generalized Stochastic Petri Nets with Rewards

GSPNs can be further extended by including place rewards and transition rewards. Place rewards are a mapping between places and real numbers, $r_P : P \to \mathbb{R}$. They represent reward accumulated for each time unit that the place is marked by at least one token. A marking reward function is defined, mapping between markings and rewards $r_S : M \to \mathbb{R}$. This function maps each reachable marking $\boldsymbol{m} \in R(G)$ to a real number which is the sum of all place rewards in a given marking:

$$r_S(\boldsymbol{m}) = \sum_{m_i > 0} r_P(p_i) \tag{2.3}$$

Transition rewards represent a reward accumulated when a given transition fires, and so is a mapping between transitions and real numbers: $r_T : T \to \mathbb{R}$. Thus, GSPNs with rewards (GSPNRs) can be defined, by augmenting the GSPN tuple with the place and transition rewards, r_P and r_T, respectively:

$$\text{GSPNR} = (P, T, I(.), O(.), W(.), \boldsymbol{m}_0, r_P, r_T)$$

2.2 Markov Decision Processes

A Markov decision process (MDP) can be used to model a sequential, decision-making problem [7]. An MDP is constituted by the tuple:

$$\mathcal{M} = (S, A, p, r, s_0)$$

S is a set of finite states, and A is a set of finite actions. The probability transition function $p : S \times A \times S \to [0, 1]$, usually in the form of $p(s'|s, a)$ and can be interpreted as the probability of transitioning to state s' given the current state s and having selected action a. The set of enabled actions in state s is denoted by A_s. Additionally, MDPs have a reward function $r : S \times A \to \mathbb{R}$, defined over state-action pairs: $r(s, a)$, i.e. reward received for selecting action a in state s. The initial state is s_0.

A very simple example of a MDP can be seen in figure 2.5. The MDP represented has four states $S = \{S_1, S_2, S_3, S_4\}$ and two possible actions $A = \{a, b\}$. If action a is taken the system will transition to either state S_2 or S_3, with an even chance of transitioning to either state ($p(S_2|S_1, a) = 0.5$ and $p(S_3|S_1, a) = 0.5$). If action b is chosen, the systems always transitions to state S_4.

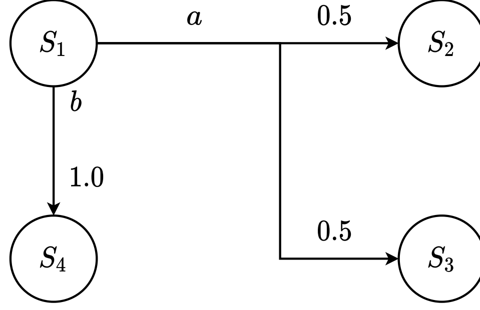


Figure 2.5: Small example of a MDP

2.3 Policies

Non-determinism is a characteristic of a system where arbitrary choices can be made. In particular, in any non-deterministic system, the decisions taken affect how the system transitions between any two given states. MDPs are an example of such systems: it is not sufficient to specify a state s to determine a probability distribution over the possible next states s'. An action a for a given state s is needed to determine the possible states s' that the system can transition to, and with what probability.

A policy, or schedule, π defines an action to select for each state in an MDP. It resolves the non-determinism of an MDP, as for a given state only the action dictated by the policy can be selected. A policy is stationary if it does not depend on time $\pi(s)$, and is deterministic if it maps each state to a single action. For an MDP $\mathcal{M} = (S, A, p, r, s_0)$, a deterministic and stationary policy $\pi \in \Pi$ is a mapping $\Pi : S \to A$. An infinite path in the MDP is denoted by $\omega^\pi = (s_0, s_1, ...)$. If rewards are considered additive, the value of a path ω^π, $V(\omega^\pi)$ is defined as follows:

$$V(\omega^\pi) = \sum_{k=1}^{\infty} r(s_k, a_k) \tag{2.4}$$

This is designated the *total accumulated reward criterion*. For the expected total reward value to converge, the MDP must have at least a single unavoidable state, s_T, where any infinite path eventually reaches s_T. Additionally, this state s_T must be terminal: it can only contain a self-loop with zero reward - $A_{s_T} = a_T, p(s_T, a_T, s_T) = 1, r(s_T, a_T) = 0$. Nonetheless, reward accumulated can be weighted by a discount factor γ, to guarantee convergence for MDPs without unavoidable terminal states:

$$V(\omega^\pi) = \sum_{k=1}^{\infty} \gamma^k r(s_k, a_k), \text{ with } 0 < \gamma < 1 \tag{2.5}$$

This is designated the *total discounted reward criterion*. For both criteria, the optimal policy $\pi^\star$ is the one that maximizes the expected value of the policy:

$$\pi^\star(s) = \operatorname*{argmax}_{\pi} \mathbb{E}\left[V(\omega^\pi)\right] \tag{2.6}$$

Policies can be also be defined over GSPNRs. In this case, instead of mapping between states and actions, a policy π maps between vanishing markings and immediate transitions: $\pi \in \Pi : VM \to T_I$.

Chapter 3

Related Work

The current chapter discusses related research work that has been done relevant to this research. It is divided into three main sections, each detailing pertinent work to different aspects of this dissertation: the first section is an overview of different approaches for general task plan representation and modelling using Petri Nets, for both single and multi-robot homogeneous or heterogeneous teams; the second section provides some examples of different approaches that focus on maintaining persistent robotic presence in a system with multiple locations; and the third and final section summarizes available tools to model and execute multi-robot systems with Petri nets, and other similar formalisms.

3.1 Petri Net Approaches

Robot Task Plan Representation by Petri Nets

One approach found in literature was the framework initially proposed in [8] and later in [9]. The work done introduces a powerful tool to model and analyse robotic tasks, while also using a layered structure, that builds the overall model from the composition of modular, simple models. The paper goes on to define an algorithm to build a single GSPN model, using top-down composition, which can be used for qualitative analysis (e.g. check for deadlocks) or for quantitative analysis. Another important contribution is model identification. A method is developed to build the environment and action models automatically from real data, however, the authors were unable to identify separately the action and environment models. These contributions are also extendable to a multi-robot team, as seen in [10]. The main contribution of this work is two-fold: the development of three communication models of different complexities (to be used in the environment layers), and the use of communication actions. With these tools, coordinated behaviour can be modelled and analyzed.

While enabling the modelling of complex tasks with a representation of uncertainty, this modelling approach is cumbersome to use with multi-robot teams. Each robot must have its own replicated action models, and with the explicit representation of communication, if there are more than two robots, there must be a selection mechanism to choose to which robots each message is sent. Furthermore, there is no automatic planning or scheduling when action selection is needed, and results and data were

collected with manually built task plan models.

Petri Net Plans

A different but complementary approach to task planning using MOPNs can be found in [11]. The framework proposed builds high-level plans using MOPNs, denominated *Petri Net Plans* or PNPs. The framework proposed consists of elementary structures that are combined using different operators. The elementary structures are simple MOPNs that model *ordinary* and *sensing* actions. These blocks can be then composed together using different operators to build the overall PNP. Some of the defined operators codify for example sequential, conditional or concurrent execution logic. From the execution perspective, transitions are labelled according to the additional conditions necessary for the firing to occur. When extending the framework to multi-agent planning, the authors [11] follow a centralized planning approach, where a global plan is shared to all agents. The global PNP of n different agents is defined as the union of n single agent's PNP, while adding synchronization operators to ensure the desired interactions are done in a safe way. An application of this PNP framework to a heterogeneous robot team can be seen in [12]. The authors consider a team made up of a single UAV and a single UGV, where the former must land on the latter, while the UGV is still moving and executing its own mission. They build the PNP according to the specifications set in [11], and then simulate and monitor the mission execution. They concluded that the PNP framework was appropriate in order to achieve the desired collective behaviour. Another example of successful application of PNPs can be seen in [13]. Although the main subject of the paper is the development of a multi-objective multi-robot planner, when validating the results in simulation, PNPs were successfully used for the generation of the distributed set of single-robot plans.

Although PNPs are a good solution for modelling task plans, and verifying qualitative properties of those plans, they are geared towards execution. By proving that the PNP is safe (the Petri net is 1-bounded), minimal (every transition fires at least once) and effective (the goal marking is reachable), the plan is proven as executable by the algorithm proposed. Despite admitting non-controllable transitions that depend on external conditions to fire, and sensing actions with multiple outcomes, PNP plans do not provide for the modelling of uncertainty in a stochastic manner.

Compact Approaches and Optimal Policies

The research work done by [4] proposes a novel methodology to solve multi-robot planning problems. By modelling the robot team and the environment using a GSPN where places are locations and most importantly uncertain navigation times and events are modelled with exponentially delayed transitions, this approach takes into account uncertainty when planning. Furthermore, taking advantage of using a homogeneous robot team, individual robots are represented anonymously as tokens, reducing the exponential blow-up that usually happens with multi-agent models. After modelling in this manner, the team-level constraints are represented as undesirable markings, and the team performance is represented as rewards given when immediate transitions (representing decisions) are fired. Finally, the

GSPN is interpreted as a MDP, that is solved to generate an optimal policy that maximizes the reward accumulated before reaching any undesirable marking outlined previously.

The authors in [5] extend this methodology by formalizing the concept of rewards into GSPNs into GSPNRs. Moreover, by interpreting the GSPNR as an equivalent Markov reward automaton (MRA), they are able to build a method that obtains the long-run average reward over the MRA, enabling the extraction of an optimal policy for the MRA, and thus for the original GSPNR.

Another relevant approach is [14]. The authors transform a known static environment into a grid of discrete occupancy cells where some of these cells are defined as regions of interest. This grid is interpreted as a Robot Motion Petri Net, where each place represents one of the cells in the grid, tokens in a given place represent that the cell is occupied by a robot and transitions represent navigation actions between cells. The authors carry on to establish algorithms that can determine the optimal sequence of transition firings that satisfies a given Linear Temporal Logic (LTL) formula. These algorithms minimize a cost weighted by several factors, such as the number of navigation actions (equivalent to minimizing the number of transitions fired) and avoiding possible collisions between robots (by constraining the number of tokens that can simultaneously be at the same place).

When comparing with the modelling approaches in the previous section [9, 11], this class of approaches [4, 5, 14] where robots are represented as tokens provide a more compact representation of a multi-robot team, but is currently limited to a homogeneous team. Additionally, these models do not support the tracking of robot-specific attributes such as battery, attributes that can impact team performance or condition decision-making.

3.2 Other Approaches

The authors in [15] attempt to solve a very similar problem to the one proposed, and also use a PN model. Considering a homogeneous robot team comprised of UAVs, the authors set out to calculate the optimal number of agents used in order to maintain uninterrupted presence in the defined target areas. To this purpose, Decision-Free PNs are used, and some of their particular cyclic properties are then used to come up with an objective function to optimize. The paper proceeds by using a genetic algorithm to overcome the optimization problem complexity. While it is a novel approach with good results, some strong assumptions were taken, i.e. deterministic travel time, and constant battery consumption.

Another recent approach can be found in [16]. The considered problem is also very similar to one proposed for this thesis. A UAV fleet is given a set of locations to be persistently monitored, and an additional ”home base” is considered where UAVs can have their batteries replaced. The problem is tackled by defining a set of well-defined drone replacement schemes so that the constraints are met. This *Minimal Spare Drone for Persistent Monitoring Problem* or MSDPM is equated to a variant of the classic Bin-Packing Problem, defined as Bin Maximum Item Double Packing. This is presumably computationally hard, and a first fit greedy approximation algorithm to solve this problem is provided, both for an offline version (where all the locations are known in advance) and an online version (in which location are given one at a time). This approach suffers from the same weaknesses as [15], as it does

not model uncertainty in any way. The main consequence is that while the planner proposed formally respects team-level requirements (i.e. each target area must always be monitored by a UAV), it does not take into account failures or uncertainty in travelling and charging times.

3.3 Software for Modelling, Analysis and Execution of Multi-Robot Systems

In regards to modelling using Petri Nets and GSPNs, there are many tools available. The most commonly used ones are PIPE [17] and GreatSPN [18]. Both contain a GUI where an user can interactively build various types of Petri nets, including GSPNs, by adding places, transitions, arcs, tokens, etc. Additionally, both toolboxes allow the user to play the token game, firing transitions and visually seeing the flow of tokens. While the newest version of PIPE (PIPE 5) has not been integrated with the analysis back-end found in previous versions, the current version of GreatSPN contains a wide range of analysis and model-checking capabilities. Users can graphically check for P- and T- invariants (see [6]), and the reachability graph can be generated and observed to check for deadlocks and liveliness. GreatSPN also features Computational Tree Logic (CTL) model checking, that enables verifying the truth-value of user-defined formulae written with an established syntax. SNAKES, [19] provides a general purpose Petri nets library, to be primarily used with Python. It offers greater flexibility than PIPE and GreatSPN, by using a variant of Python-coloured Petri nets that allow tokens to be arbitrary Python objects, transitions weights to be arbitrary expressions, and input/output arcs to be annotated with arbitrary Python variables and expressions. All three tools lack support in bridging between models and their corresponding real systems. There is no functionality that enables the user to run their multi-robot Petri net model in real robots using these software packages.

On the other hand, there is a software package to execute Petri Net Plans in ROS. It can automatically generate valid PNPs from plans written in Planning Domain Definition Language (PDDL), and it allows visualizing the generated PNPs in JARP (a simple graphical tool to visualize Petri nets). The package also allows for the elaboration of execution rules, defined simply as if-then conditions that are checked during the execution of the determined action. By interfacing with user-defined ROS action servers, and by using the ROS parameter server to save information of the truth-value of conditions defined in the execution rules, the software is able to execute the generated PNPs in either simulated or real robots.

Another important tool to mention is SMACH [20]. It is designed to be used for task execution and coordination, and it allows users to build hierarchical state machines that interface with ROS action servers and ROS services. While SMACH provides a lot of flexibility when designing state machines, it is not particularly scalable as the number of robots increases. The user must be able to explicitly list all possible states and transitions, and when working with multi-robot systems this becomes cumbersome and possibly not feasible. It also does not take into account uncertainty.

Chapter 4

Modelling Multi-Robot Systems with GSPNRs

Chapter 4 introduces an extension to the modelling framework found in [4, 5]. The first section defines the modelling framework for a homogeneous robot team, explaining how the non-determinism of robots' decision making and uncertainty regarding action duration is captured. It extends this to heterogeneous teams and the possibility of synchronized or asynchronized cooperative actions, and finally provides a way to model both system-wide resources or individual robot attributes.

4.1 Homogeneous Systems

In multi-robot homogeneous systems where all robots have the same capabilities, GSPNRs models can provide a compact way of representing the overall team state. Robots are represented as tokens, and each place in the GSPNR represents a particular state in which a robot may be at. If such a place is marked by a token, this means that a robot is currently in that particular state. The overall team state is represented by the marking of the GSPNR, specifying all the states in which each robot composing the team is at.

Transition firing is the mechanism with which tokens transverse through the places in a GSPNR. In a multi-robot problem where tokens represent robots, GSPNR transitions model any event in which the robot represented by a token changes state. The place in which the token was previously at represents the local state the robot was before the event took place. The place where the token is created by the transition firing models the state in which the robot lands after the event takes place.

Immediate and exponential transitions model different types of events. An instantaneous event such as a decision is modelled with an immediate transition. Thus, action selection and its non-determinism is captured with several immediate transitions connected to the same place by an input arc. When a token reaches a place connected to several immediate transitions, every conflicting immediate transition represents a different decision that the robot can choose to take. Exponential transitions represent an uncontrollable timed event and so are chiefly used to model uncertainty regarding action execution.

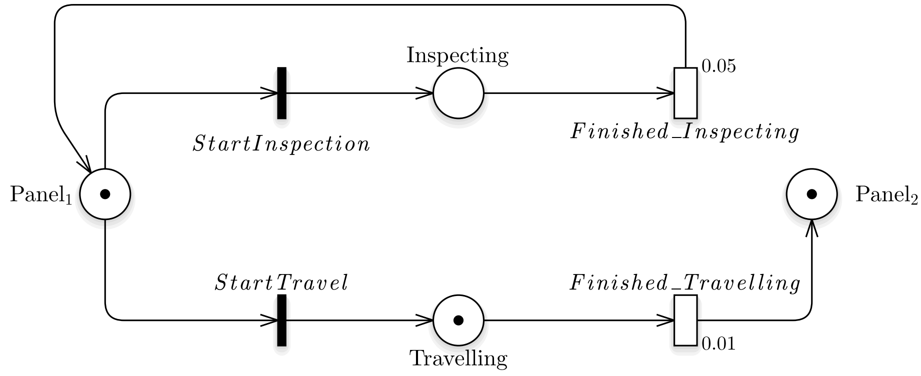


Figure 4.1: Example of a Multi-robot system with 2 robots that can travel between solar panels and inspect one;

When a token reaches a place connected to an exponential transition by an input arc, the time spent by the token in that place is a random variable with probability distribution according to Equation 2.1.

A partial GSPNR illustrating these concepts can be seen in Figure 4.1. Two robots are represented with the two tokens. The place "Panel1" represents a state where the robot must decide between inspecting and travelling, and these decisions are modelled with the immediate transitions "StartInspection" and "StartTravel". The places "Inspecting" and "Travelling" represent states where a robot is inspecting the panel or travelling. Thus, the token in the "Travelling" place represents a robot executing the travel action. When the exponential transition "Finished_Travelling" fires, the token representing this robot will be consumed. Because this transition fires with an random exponentially distributed delay, mean action duration can be modelled by tuning the weight of the exponential transition. In this particular example, the average time of the inspection action is 20 seconds ($\frac{1}{0.05}$) and the average time of the travelling action is 100 seconds ($\frac{1}{0.01}$).

4.1.1 Decision-Action Graphs

In most multi-robot problems involving sequential decision-making, the states that a robot can occupy belong into one of two categories. A robot can be in a state where it must decide between executing several available actions, or it can be in a state of executing an action. After finishing action execution, the robot transitions again into a state where it must make another decision to execute another action. This process occurs repeatedly for all robots in the system.

GSPNR models that capture this kind of "Decision/Action/Decision/..." problem contain an underlying structure. Formally defining this underlying structure provides a more direct interpretation between the multi-robot problem and its GSPNR model. Moreover, by defining this concept, algorithms to build GSPNR models are easier to understand and implement, and it becomes simpler to extend these GSPNR models to heterogeneous teams down the line.

This structure in GSPNR models is described using decision-action graphs (*DecAct-G*). Formally, a decision-action graph is a bipartite directed graph $\mathcal{DA} = (D, A, E)$, where D is the decision nodes set, A is the action nodes set, and E is the edges set. Each decision node represents a state where the robot must choose between various possible actions, and each action node represents an action that

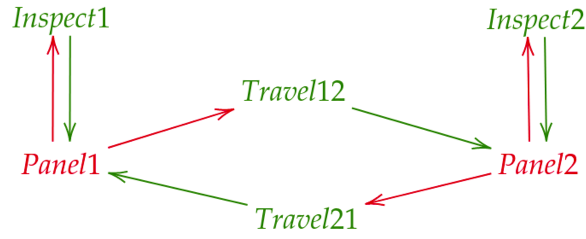


Figure 4.2: Decision-action graph for an inspection scenario; in green are decision nodes, and in red are action nodes;

takes non-zero time to execute.

Edges that begin in a decision node and end in an action node are denominated *decision edges* and represent all the possible actions that the robot can execute in that decision node; edges that begin in an action node and end in a decision node are denominated *outcome edges* and represent the decision state that the robot ends up in when the action has finished executing. $Dec(a)$ and $Out(a)$ are functions that have as input an action a, and an as output the set of decision or outcome edges connected to that action node, respectively. The number of decision edges of an action node must be the same as the number of outcome edges, as the number of robots remains constant before and after executing an action.

> **Multi-Robot Scenario 1:** A homogeneous robot team composed of small UGVs that can inspect or travel between two solar panels: {Panel1, Panel2}

To illustrate, the decision-action graph for Scenario 1 can be seen in Figure 4.2. Drawn in red are decision nodes {Panel1, Panel2} and decision edges. Drawn in green are action nodes {Inspect1, Inspect2, Travel12, Travel21} and all outcome edges.

Action nodes directly represent states where the robot is executing an action. Each action state is associated with two events: 1) the controllable event of deciding to start action execution; 2) the uncontrollable event of finishing executing the action. This leads us to define a function over the action nodes $ActionModel : A \rightarrow G$, where A is the set of all action nodes and G the set of all possible GSPNRs. $ActionModel(a)$ is the GSPNR that models the execution of action a. $ActionModel()$s have only three elements:

- *Decision transition* - an immediate decision representing the controllable event of deciding to execute the action;
- *Action place* - the place that models the robot state of actually executing the action;
- *Final transition* - an exponential transition modelling the uncontrollable event of finishing executing the action;

An action model for an arbitrary action a is represented in Figure 4.3 and is formally defined as a GSPN $G_A = (P, T, I(.), O(.), W(.), \boldsymbol{m}_0)$ with the following restrictions $P = \{p_A\}$; $T = \{t_D, t_F\}$; $I(p_A, t_F) = 1$;

$O(t_D, p_A) = 1; \boldsymbol{m}_0 = (0)$.

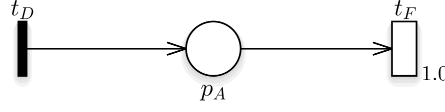


Figure 4.3: An action model for arbitrary action A; the immediate transition t_D models the decision to execute A, and the exponential transition t_F models finishing the execution of A;

4.1.2 Building GSPNR Models from Decision-Action Graphs

*DecAct-G*s allows us to build a GSPNR model in a precise manner. Each node and arc in the graph is substituted by a corresponding GSPNR element, and they are merged together to create an overall GSPNR model that captures the multi-robot problem considered. The algorithm to form the GSPNR model from the *DecAct-G* can be found in Algorithm 1. To build the full GSPNR model from a $\mathcal{DA} = (D, A, E)$, each decision node is interpreted as a place denominated *decision place*, and each action node a is replaced by its GSPNR model $ActionModels(a)$. Any decision edge (d, a) is substituted by an input arc from the corresponding decision place to the decision transition $t_D \in ActionModel(a)$ of the action model corresponding to the action node. Any outcome edge (a, d) is replaced by an output arc from the final transition $t_F \in ActionModel(a)$ to the decision place corresponding to the decision node.

```
procedure G_F = FULLGSPNR(DA, ActionModel(.))
    DA = (D, A, E)
    ActionModel(a) → G, ∀a ∈ A
    G_F = (∅, ∅, ∅, ∅, ∅, ∅, ∅, ∅)

    P_F = D                                   ▷ Decision nodes are interpreted as places
    for a ∈ A do
        G_F = MergeGSPNR(G_F, ActionModel(a))              ▷ All action models are added
    end for
    for e = (d, a) ∈ E do
        t_D ∈ ActionModel(a)                            ▷ t_D is the decision transition
        I_F(d, t_D) = 1     ▷ Adding input arc between decision place and decision transition
    end for
    for e = (a, d) ∈ E do
        t_F ∈ ActionModel(a)                               ▷ t_E is the final transition
        O_F(t_F, d) = 1       ▷ Adding output arc between final transition and decision place
    end for
end procedure
```

Algorithm 1: Building a homogeneous GSPNR from a Decision-Action Graph

The algorithm that builds a GSPNR model from a *DecAct-G* needs a function that merges two GSPNRs in a correct and unambiguous manner. The algorithm to do so can be found in Algorithm 2. Two GSPNRs can be merged by taking the union of the corresponding sets or functions that constitute each Petri net. However, there are some restrictions that need to be placed, as to maintain unambiguous

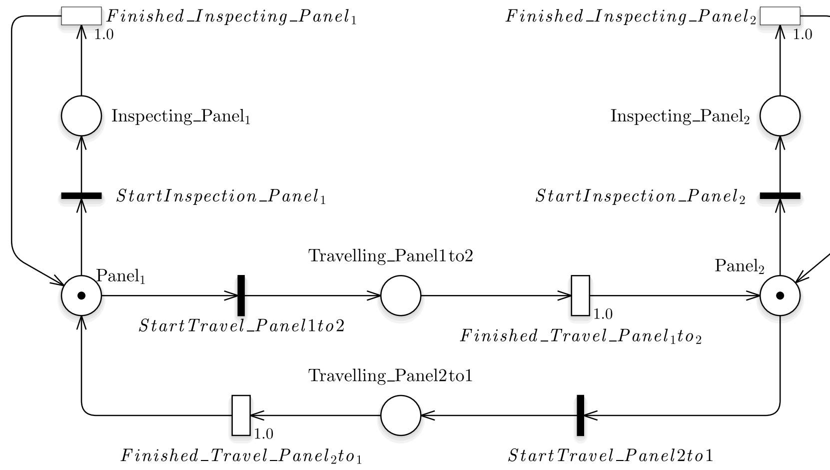


Figure 4.4: Model of two solar panels with inspection and travelling tasks, built with *DecAct-G* in Figure 4.2

mappings. Every transition that is common to both GSPNRs needs to have the same weight for the merging to be possible. The same happens with input/output arc multiplicity functions, and with the place and transition reward functions.

function G_M = MERGEGSPNR(G_1,G_2)
$P_M = P_1 \cup P_2$
$T_M = T_1 \cup T_2$
$I_M = I_1 \cup I_2$, with $\forall(p,t) \in \{I_1 \cap I_2\} \rightarrow I_1(p,t) = I_2(p,t)$
$O_M = O_1 \cup O_2$, with $\forall(t,p) \in \{O_1 \cap O_2\} \rightarrow O_1(t,p) = O_2(t,p)$
$W_M = W_1 \cup W_2$, with $\forall t \in \{T_1 \cap T_2\} \rightarrow W_1(t) = W_2(t)$
$\boldsymbol{m}_{0M} = \boldsymbol{m}_{01} \cup \boldsymbol{m}_{02}$
$r_{PM} = r_{P1} \cup r_{P2}$, with $\forall p \in \{P_1 \cap P_2\} \rightarrow r_{P1}(p) = r_{P2}(p)$
$r_{TM} = r_{T1} \cup r_{T2}$, with $\forall t \in \{T_1 \cap T_2\} \rightarrow r_{T1}(t) = r_{T2}(t)$
$G_M = (P_M, T_M, I_M, O_M, W_M, \boldsymbol{m}_{0M})$
end function

Algorithm 2: Merging two GSPNRs with common places and transitions

Figure 4.4 shows an example of a GSPNR model obtained from the *DecAct-G* depicted in Figure 4.2. The GSPNR captures a problem where the robots can decide to either inspect the solar panel where it is at, or travel to the other solar panel. Thus, there are two decision places: 'Panel1' and 'Panel2'. Each solar panel can be inspected, and so there are two action places: 'Inspecting_Panel1' and 'Inspecting_Panel2'. The two remaining action places model the travel between locations, 'Travelling_Panel1to2' and 'Travelling_Panel2to1'.

4.2 Heterogeneous Systems

Until now, all tokens represented robots that were the same type, with the same capabilities. This means that robots could be represented anonymously as tokens in the GSPNR model. To extend this framework to model a heterogeneous robot team, the GSPNR model must unambiguously determine

which type of robot each token represents. This is done by partitioning the GSPNR place set into several mutually exclusive subsets. Each subset is associated with a single robot type. By doing so, each token can be directly mapped to a specific robot type by examining which subset the marked place belongs to.

Suppose we have a GSPNR $G = (P, T, I, O, W, r_P, r_T)$, and the multi-robot problem involves n different robot types $R = \{r_1, \cdots, r_n\}$. The place set P is partitioned into n subsets: $P = \bigcup_n P_i$ which are all mutually exclusive $\forall i, j : P_i \cap P_j = \emptyset$. Tokens in a place belonging to the subset P_i represent robots of type r_i. To illustrate: take a GSPNR with place set $P = \{p_1, p_2, p_3, p_4\}$, representing a heterogeneous system with two types of robots: UAVs and UGVs. The UAV place subset could be $\{p_1, p_2\}$ and the UGV place subset $\{p_3, p_4\}$. If the current marking is for example $\boldsymbol{m} = (2, 0, 0, 1)$, the 2 tokens in p_1 are recognized as UAVs, and the single token in p_4 is recognized as a UGV.

For homogeneous systems, the GSPNR model had to be conservative: the number of tokens needed to remain constant for all possible markings. This translates the natural constraint of keeping the number of robots constant, as robots cannot be created or destroyed. When extending the framework to heterogeneous systems, this constraint needs to be stronger. The number of tokens in each place subset representing different types of robots must remain constant. In practical terms, this forbids transitions connecting places belonging to different robot type subsets. Such a transition would imply transforming a robot of a particular type into another robot type.

Decision-Action graphs can also be used to build heterogeneous GSPNR models. By partitioning the decision node set into separate sets that are each associated with a single robot type, the mapping between places and robot types defined previously can be established when building the full GSPNR model. In similar manner as previously defined with GSPNR models, suppose $\mathcal{DA} = (D, A, E)$ is a *DecAct-G* representing a multi-robot system with n different robot types, $R = \{r_1, \cdots, r_n\}$. The decision node set D must be partitioned into n subsets: $D = \bigcup_n D_i$, each being mutually exclusive $\forall i, j$: $D_i \cap D_j = \emptyset$. A decision node d belonging to subset D_i, $d \in D_i$ is associated with robot type $r_i \in R$.

Furthermore, cooperation actions involving two or more robots can be represented using *DecAct-G*s. An action done in cooperation by multiple robots is represented by an action node with multiple decision edges. Each decision edge connected to this cooperation action node represents a single robot that is involved in cooperatively executing the action. As with homogeneous systems, each decision edge that connects a decision node $d_1 \in D$ to action node $a \in A$, must have a matching outcome edge connecting the action node to a decision node $d_2 \in D$. When working with a heterogeneous system, these two decision nodes must belong to the same type subset: $d_1, d_2 \in D_i$. This corresponds to respecting the constraint of keeping the amount of each robot-type constant.

Multi-Robot Scenario 2: A heterogeneous robot team composed of small UGVs that can inspect or travel between two solar panels: {Panel1, Panel2}, and a single larger UGV that can also travel between the panels. The small UGVs can choose to cooperatively recharge its battery with the help of the large UGV, if they are in the same location.

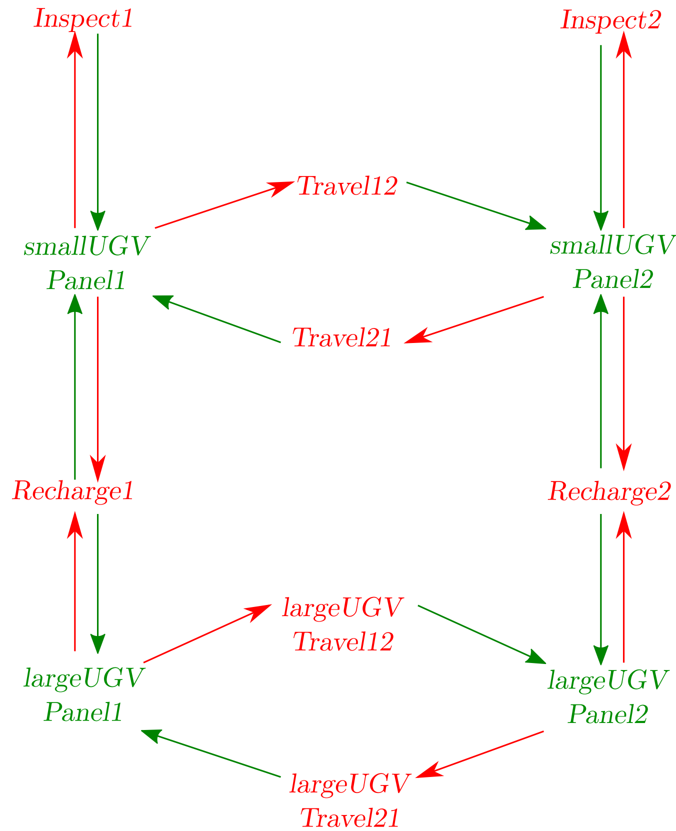


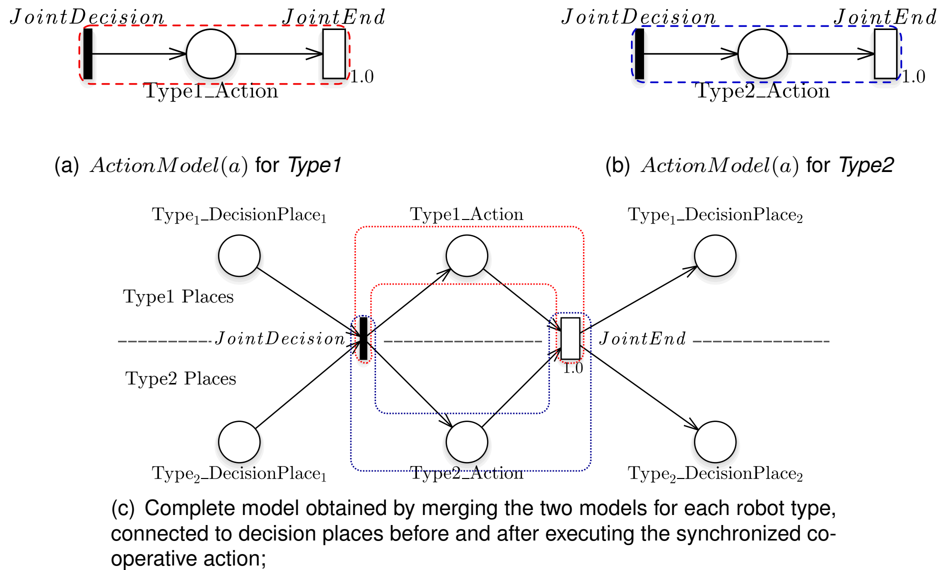
Figure 4.5: *DecAct-G* of a heterogeneous system with a recharging task done in cooperation by the two robot-types

The *DecAct-G* that captures Scenario 2 can be seen in Figure 4.5. It extends the system seen in Figure 4.2 to a scenario with two robot types: small UGVs and a large UGV. The small UGVs can inspect and travel between panels, but now they are also able to recharge in cooperation with the larger UGV present if it is in the same panel. The "Recharge" action nodes are connected by two decision edges to the corresponding decision nodes "smallUGVPanel" and "largeUGVPanel", representing this cooperation. The decision node set can be partitioned into the two sets corresponding to the two robot types: the set {smallUGVPanel1, smallUGVPanel2} pertains to the small UGVs, and the set {largeUGVPanel1, largeUGVPanel2} pertains to the larger UGV.

Actions done with cooperation require to define as many action models as there are robots executing the action. There are two possible ways to model cooperation: 1) synchronized cooperative actions, where all robots start and end the execution of the action at the same time; and 2) asynchronized cooperative actions, where all robots involved begin the action at the same time, but are independent to finish execution at different moments.

Synchronized Cooperative Action Models

To model a synchronized cooperative action that is done in cooperation by n robots, n action places need to be used. Each action place corresponds to one of the robots executing its component of the cooperative action. These actions begin and finish at the same time. Thus, all action places are connected



(a) $ActionModel(a)$ for *Type1*

(b) $ActionModel(a)$ for *Type2*

(c) Complete model obtained by merging the two models for each robot type, connected to decision places before and after executing the synchronized co-operative action;

Figure 4.6: Separate and merged models for a synchronized cooperative action between two different robot types

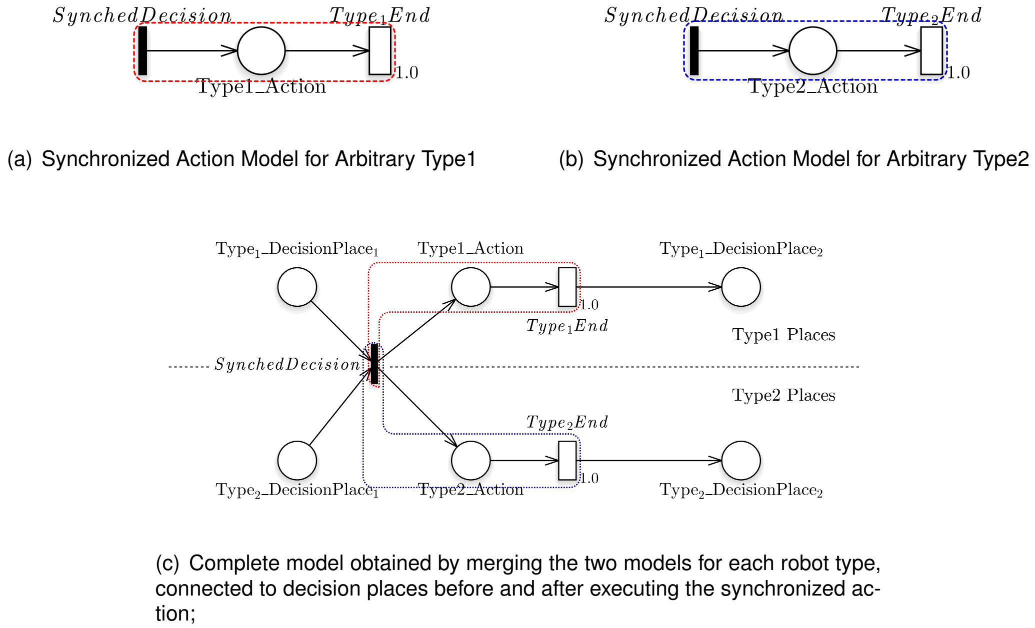
by an input arc to the same immediate transition. This immediate transition represents the decision to start the synchronized action. When the immediate transition fires, tokens are created in all action places and action execution begins. In similar manner, all action places are connected by an output arc to the same exponential transition. This constrains all robots to finish executing the action at the same time. When this transition fires, all tokens in the action places are consumed.

A synchronized cooperative action between n robots requires n separate action models. Every action model must contain the same *decision* and *final* transitions. Consequently, when these action models are merged, all action places are connected to same immediate or exponential transition, by an output or input arc, respectively.

An example for a synchronized cooperative action done by two robots can be seen in Figure 4.6. Each robot type has its own action model for the same action a, comprising of its decision transition, action place, and final transition. The example considers two arbitrary robot types *'Type1'* and *'Type2'*. When transition *JointDecision* is fired, two tokens are created in each robot-type's action place. Finally, the two robots finish executing the synchronized action at the same time, and so both action places are connected to the same exponential transition *JointEnd*.

Asynchronized Cooperative Action Models

An asynchronized action differs from a synchronized one because robots are unconstrained to end their component of the action independently from other robots. An asynchronized action done by n robots also requires n action places. As with synchronized actions, each action place is connected to the same *decision* transition by output arcs. However, in asynchronized actions, each action place is connected to an unique *final* exponential transition that fires when a robot finishes their component of the cooperative task.



(a) Synchronized Action Model for Arbitrary Type1

(b) Synchronized Action Model for Arbitrary Type2

(c) Complete model obtained by merging the two models for each robot type, connected to decision places before and after executing the synchronized action;

Figure 4.7: Separate and merged models for an asynchronized action between two different robot types

An asynchronized action between n robots also requires n separate action models. Every action model must contain the same immediate transition. Consequently, when these action models are merged, all action places are connected to the immediate transition by output arcs. Each action model must have an unique *final* exponential transition, so that each robot can finish executing their part of the cooperation independently.

A small example for asynchronized actions of two different robot types can be seen in Figure 4.7. When the *SynchedDecision* transition fires, two tokens are placed in the two action places for each robot type. However, each of these places is connected to a separate exponential transition, so that for example the action modelled by *Type1_Action* place can finish before the *Type2_Action*.

4.2.1 Algorithm to Build GSPNR Model

To build a GSPNR model from a *DecAct-G* of a heterogeneous system we have to adapt the algorithm that does so for homogeneous systems (Algorithm 1). The main difference between models of homogeneous and heterogeneous systems is that the final GSPNR model of a heterogeneous system must have the mapping between places and robot types that specifies which type of robot a token in any particular place represents. In order to do this, we propose a novel algorithm that requires the usual inputs $\mathcal{DA}$ - the decision-action graph, $ActionModels(a, r)$ specifying the action models for each action node a and each robot type r, a robot list R, and mapping between decision nodes and robot types $RobotType()$. The three extra inputs (when compared with a homogeneous multi-robot problem) are:

1. A set of unique robot types: R;

2. A mapping from decision nodes to the robot type represented by the decision node: $RobotType : D \rightarrow R$. $RobotType(d)$ represents the type of robot in decision node d. Action nodes are not

mapped to a particular robot type. This is determined by the decision edges connected to it;

3. The function $ActionModel(a)$ needs to be extended, so that action nodes modelling cooperation actions specify which GSPNR to use for each action and for each type. $ActionModel(a, r)$ represents the GSPNR model for action $a \in A$ for the specific robot type $r \in R$;

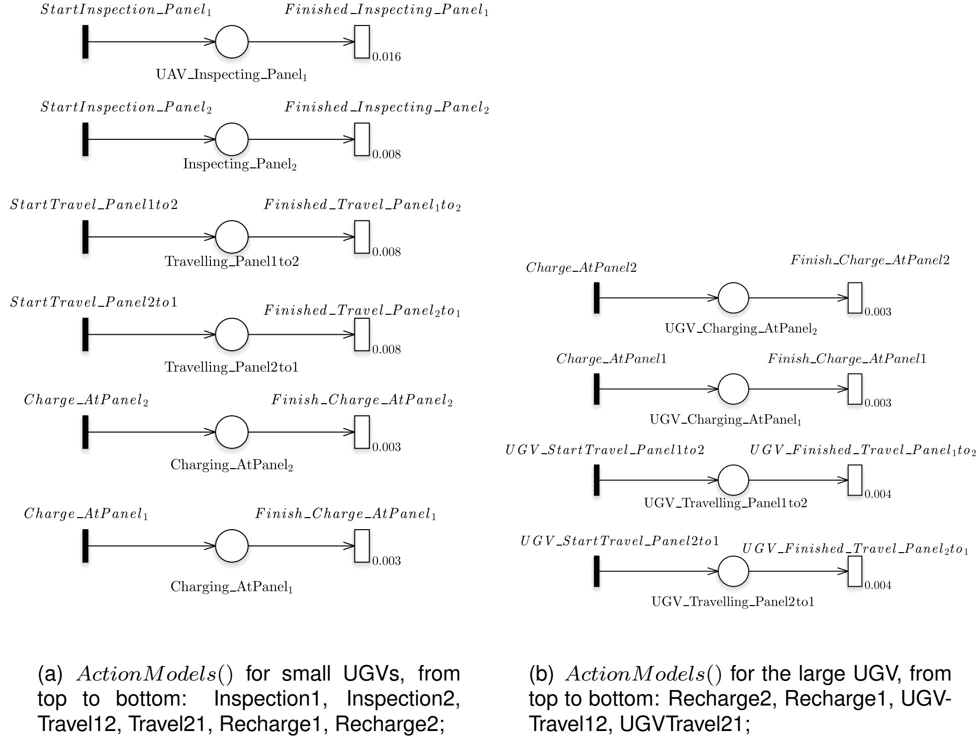
```
procedure G_F = FullGSPNR(DA, R, RobotType(.),ActionModel(.,.))
    DA = (D, A, E)
    R = {r_1, ..., r_K}                                   ▷ K different robot types
    RobotType(d) → r, ∀d ∈ D, and ∀r ∈ R
    ActionModel(a, r) → G, ∀a ∈ A, and ∀r ∈ R
    G_F = (∅, ∅, ∅, ∅, ∅, ∅, ∅, ∅)

    P_F = D                        ▷ Decision nodes are interpreted as places
    for a ∈ A do
        for (d, a) ∈ Dec(a) do
            r = RobotType(d)
            G_F = MergeGSPNR(G_F, ActionModel(a, r))      ▷ All action models are added
        end for
    end for
    for e = (d, a) ∈ E do
        r = RobotType(d)
        t_D ∈ ActionModel(a, r)                     ▷ t_D is the decision transition
        I_F(d, t_D) = 1     ▷ Adding input arc between decision place and decision transition
    end for
    for e = (a, d) ∈ E do
        r = Type(d)
        t_F ∈ ActionModel(a, r)                        ▷ t_F is the final transition
        O_F(t_F, d) = 1       ▷ Adding output arc between final transition and decision place
    end for
end procedure
```

Algorithm 3: Building a heterogeneous GSPNR model from a Decision-Action Graph

The algorithm for building heterogeneous GSPNR models from *DecAct-G*s can be seen in Algorithm 3. Decision nodes in decision-action graph are interpreted as decision places and added to the GSPNR model. Unlike with homogeneous systems, in heterogeneous systems a single action node a can have multiple action models. All action models for each action node, $ActionModel(a, .)$ are instantiated and merged into the GSPNR model. Afterwards, the algorithm connects the action models to the appropriate decision places, according to the edges in the *DecAct-G*. With heterogeneous systems, the algorithm has to check that it is connecting the transitions in each action model with the decision place of the correct robot type. It does so by verifying the function $RobotType()$.

Figure 4.9 shows a GSPNR model that captures an heterogeneous robot system. This model was built from the *DecAct-G* depicted in Figure 4.5 and the action models in Figure 4.8. The weight of the exponential transitions in each action model were chosen to model different average durations for each action: ”Inspection1” action has a mean duration of 60 seconds, so the weight of the exponential transition is $\frac{1}{60} = 0.016$, ”Inspection2”, ”Travel12”, ”Travel21” actions have a mean duration of 120 seconds so the weight of their exponential transitions are $\frac{1}{120} = 0.008$. Supposing the large UGV used is slower than the small UGVs, and the mean travel time is 240 seconds, the ”UGVTravel12” and ”UGVTravel21” action models have an exponential transition with weight $\frac{1}{240} = 0.004$. Finally, if the small UGVs take an



(a) $ActionModels()$ for small UGVs, from top to bottom: Inspection1, Inspection2, Travel12, Travel21, Recharge1, Recharge2;

(b) $ActionModels()$ for the large UGV, from top to bottom: Recharge2, Recharge1, UGV-Travel12, UGVTravel21;

Figure 4.8: Action models for every action node in *DecAct-G* depicted in Figure 4.2 and for the two types of robots involved. These action models were used to build full GSPNR model shown in Figure 4.9;

average of 300 seconds to recharge, the weight of the exponential transition in actions ”Recharge1” and ”Recharge2” (for both small UGVs and the large UGV) is $\frac{1}{300} = 0.003$. As can be seen in Figure 4.8, the immediate and exponential transitions in the small robots and the large UGV ”Recharge” models are the same, meaning that this is a synchronized cooperative action, and robots executing this task both start and end at the same time.

4.3 Modelling Attributes

It is uncommon that a multi-robot problem only involves robots and the actions that they are capable of executing. Robots can influence the environment in which they are deployed, e.g. after a robot inspects a solar panel, the panel no longer requires an inspection for a certain period of time. At the same time the operating environment can itself disallow actions that the robots are capable of doing, conditioning their decision-making. This would happen, if for example, the route to navigate between two locations could only be used by a single robot at the same time. Any such attribute which is associated with the overall system we consider to be a system attribute.

Beyond the operating environment, robots themselves can also have attributes that influence their capabilities. These robot-specific attributes are more challenging to capture in a GSPNR model, as robots are represented anonymously with tokens. An example of a robot-specific attribute is battery. Robots with a discharged battery might not be able to execute any action, becoming stranded in place,

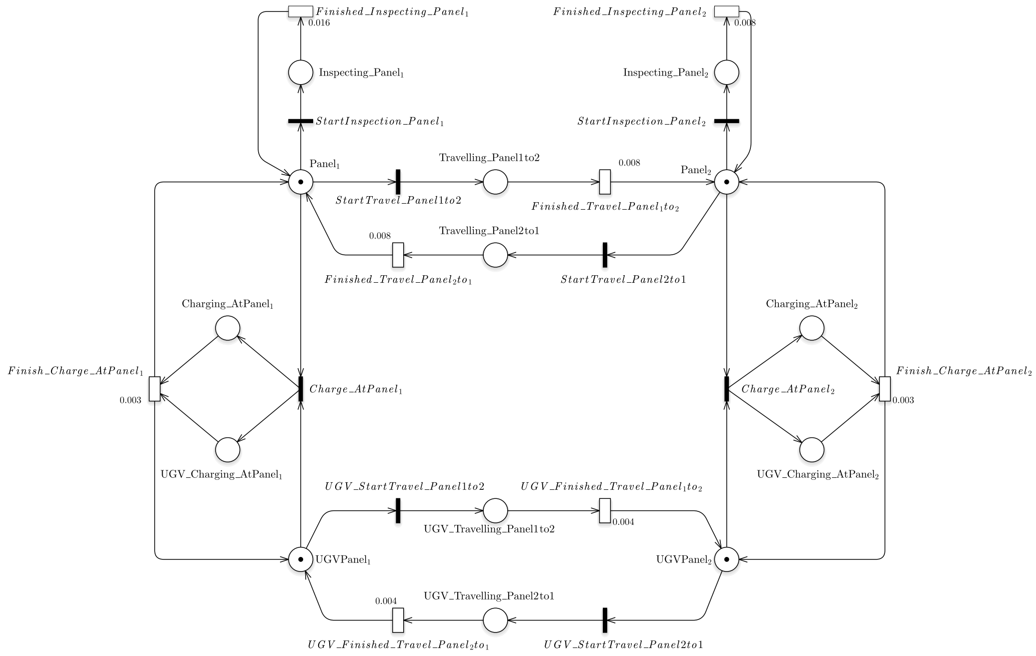


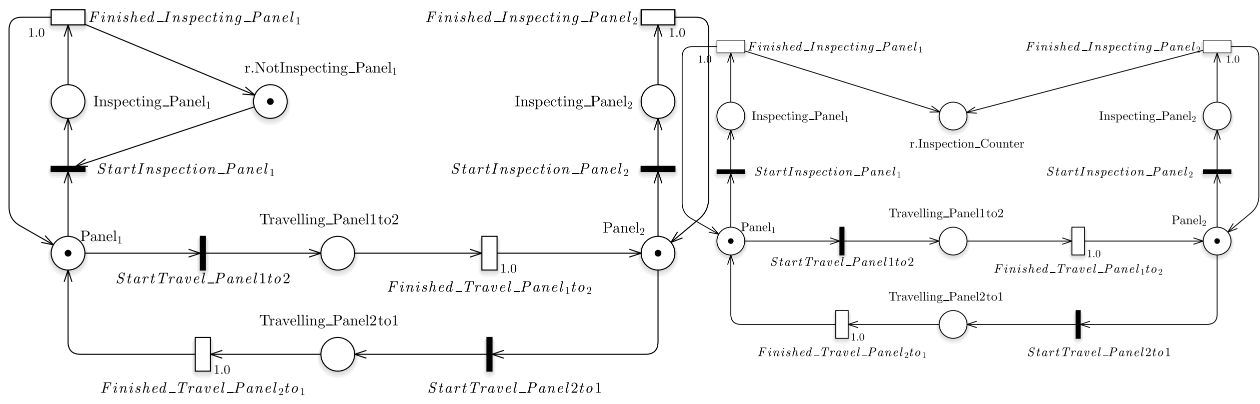
Figure 4.9: GSPNR model built with Algorithm 3 using the heterogeneous *DecAct-G* in Figure 4.5, and action models depicted in Figure 4.8;

or robots with low battery might not be able to execute a specific power-intensive action. A model that captures the evolution of each robot's battery is important in order to obtain robust policies in a problem involving robots with limited autonomy.

4.3.1 System Attributes

In order to model overall system resources or attributes that can condition the decision-making of the robots, a new type of place is used - *Resource places*. In our approach, until now, all tokens represent robots, and the place where the token is specifies the state of the robot. Tokens in *resource places* do not follow this interpretation and they represent a particular component of the state of the environment. A direct consequence is that the sub-GSPNR composed of all resource places does not need to be conservative. The number of tokens in all resource places can increase or decrease. Nonetheless, when we later discuss policy synthesis over GSPNR models, we conclude that this sub-GSPNR made up of *resource* places must still be bounded, and these considerations are elaborated further in section 5.3. We chose to distinguish *resource* places from *decision* and *action* places by adding the label **"r."** to any *resource* place, as this distinction is necessary for execution, also discussed further ahead.

Consider the example described in Scenario 1 and its GSPNR model depicted in Figure 4.4. In this case, no resource places were used. If a further restriction is included specifying that the first solar panel can only be inspected by one robot at a time, there is a need for a *resource* place that is marked when 'SolarPanel1' is not being inspected. This additional place can be seen in Figure 4.10(a), as place 'r.NotInspecting_Panel1'. The decision to inspect 'SolarPanel1' consumes a token from this resource



(a) Same model as found in Figure 4.4, with an additional inspection restriction using a resource place;

(b) Same model as found in Figure 4.4, with an additional resource place that serves as a counter of overall inspections done;

Figure 4.10: Two examples of resource places usages based in the GSPNR model of Figure 4.4;

place, and if this place is not marked, the robots can not take the decision to inspect.

Another example of the use of *resource* places can be seen in Figure 4.10(b), also based on the Scenario 1. There is an additional *resource* place, ”r.Inspections_Counter”, that is connected by an output arc from both exponential transitions that fire when the inspection actions finish executing. And so the number of tokens in this general place represent the number of total inspections already done by the robots on either solar panel. This is an example of a GSPNR that is unbounded: there is no limit to the number of tokens in the place ”r.Inspections_Counter”, and it will keep on increasing as long as the robots inspect the panels.

When building a GSPNR model from a *DecAct-G*, resource places can be directly included in action models, and can connect to its immediate or exponential transition. By including the same general place in multiple action models, specifications involving different actions can be made.

4.3.2 Robot Attributes

Modelling a multi-robot problem with a GSPNR where robots are represented anonymously as tokens presents a difficult challenge when including a robot-specific attribute. The GSPNR formalism offers no manner of distinguishing between tokens, and this is essential when capturing the evolution of a robot-specific attribute. The model must distinguish between robots where the attribute is at different states. Our approach to solving this challenge is including information on the current attribute state of a robot within the GSPNR places. Up to now, the place in which a token is specifies the state of the robot, but to model an attribute, each place must also specify the specific state of the attribute.

The modelled attribute must have a discrete set of possible states. For example, if the robot's battery is modelled, it must be approximated into a finite set of states such as {B0, B1, B2}. The battery state ”B0” would be a battery with a low state of charge, ”B1” a medium state of charge, and ”B2” battery with a state of charge close to 100%. A coarser approximation would discretize the battery into only two states {discharged, charged} that indicates if the robot still has battery or if it is discharged.

Suppose a robot-specific attribute L has N discrete states: $L = \{l_1, \cdots, l_N\}$. Broadly speaking, our

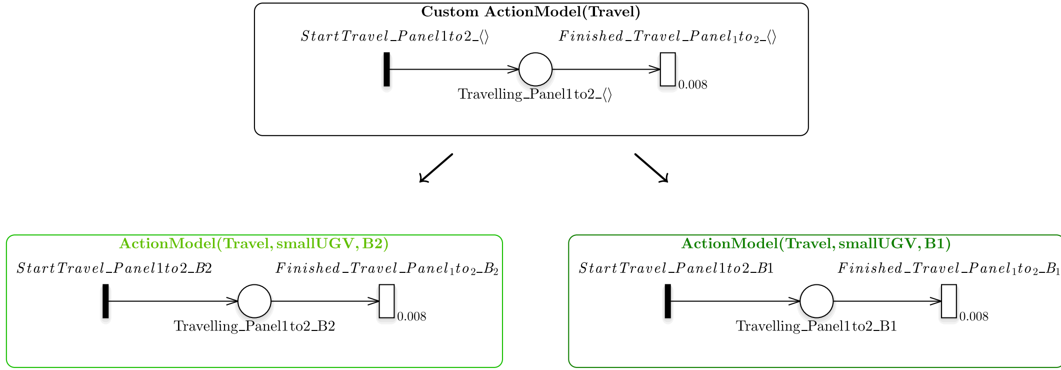


Figure 4.11: Three GSPNR action models for $Travel$: template action model (above), action model for battery level B2 (left), action model for battery level B1 (right)

approach to capture the attribute's evolution is to replicate N times all *decision* and *action* places that pertain to the robot type with an attribute. In this new GSPNR model, each of these places represents not only the state of the robot (in terms of decisions available and action execution, for *decision* and *action* places respectively), but also contains information about the particular state of the robot attribute that the robot has. For example, if a UGV robot had a battery discretized into three states {B0, B1, B2}, all of its decision places d_i would be replicated three times $\{d_i^0, d_i^1, d_i^2\}$. A token in place d_i^0 would represent a robot with a low battery state of charge, a token in place d_i^1 would represent a robot with medium state of charge, and a token in place d_i^2 would represent a robot with almost full state of charge battery. The same reasoning applies to *action* places. By replicating the *action* places, each token represents a robot with a specific attribute level executing a particular action.

But it is not enough to replicate only the *action* places. The transitions that model the beginning and end of each action also need to replicated. Thus, every $ActionModel$ involving a robot with an attribute must to be replicated for each level the attribute can take. These need to be unique: the $ActionModel$ for executing the $Travel$ action at battery level ”B2” cannot be the same as the one when battery level is ”B1”. The simplest way to achieve this is by using a template $ActionModel$. This template can be then copied and customized for each specific attribute level. This way, we avoid having to explicitly list each $ActionModels(a, t, l)$ for each possible attribute level.

We define a function ”FormatActionModel()” that takes as input a template $ActionModel(a, t)$, and a particular level of the robot attribute $l \in L$. This function formats the template and returns an $ActionModel(a, t, l)$ containing the partial GSPNR that models action execution for robots with the attribute at a particular level l. An example can be seen in Figures 4.11 and 4.12, where the template action model contains the special characters ”<>”. When the formatting function is applied, these special characters are replaced with the particular battery state specified.

When actions are done in cooperation with the robot type which has an attribute, the action models of the robot with no attributes must also be replicated, and the restrictions on the decision transition and the final transitions remain the same for synchronized and asynchronized actions.

After a robot with an attribute finishes executing an action, its attribute may evolve to a different

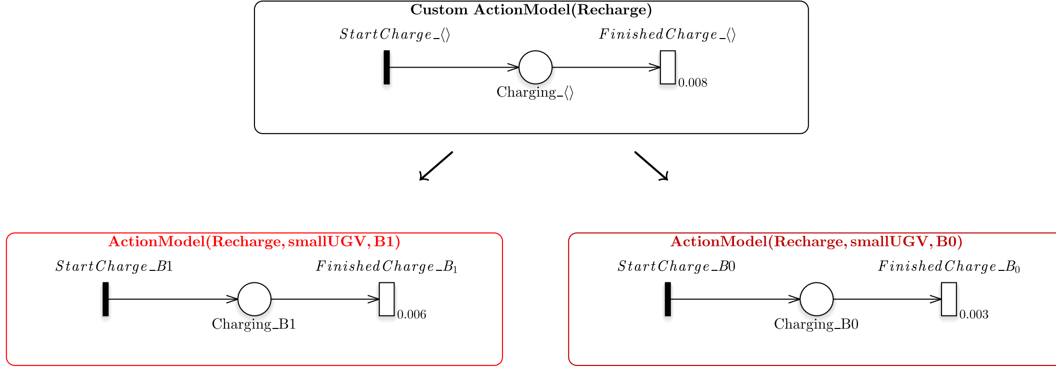


Figure 4.12: Three GSPNR action models for the small UGV $Recharge$ action: template action model (above), action model for battery level B1 (left), action model for battery level B0 (right)

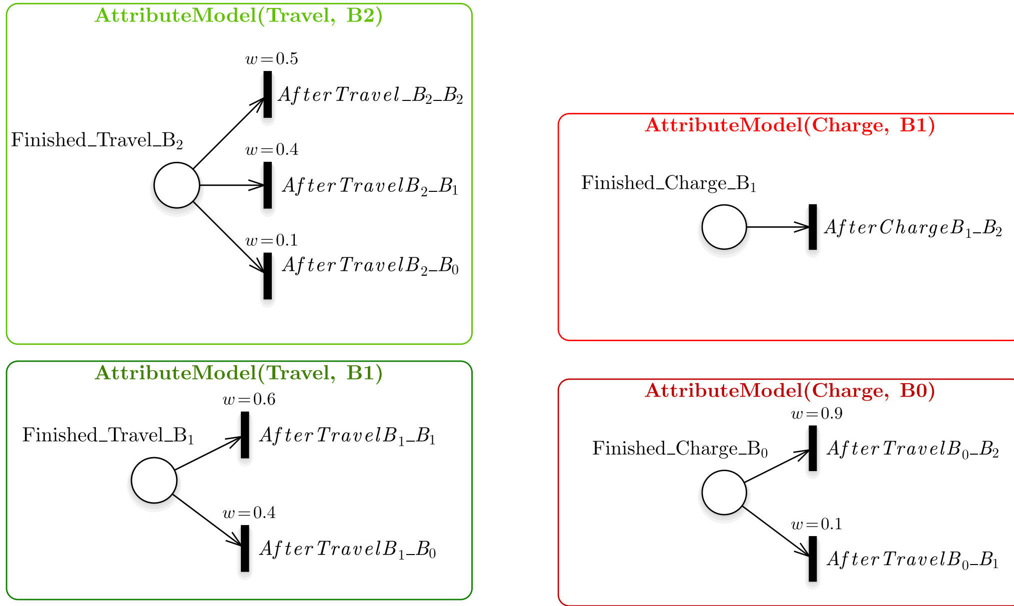
state. This is modelled with an *AttributeModel*. An *AttributeModel* is a GSPNR with a single place, denominated *attribute* place. This place is connected to multiple immediate transitions, each modelling the event of the robot's attribute progressing to a different state. All of the immediate transitions in an *AttributeModel* can be associated with transitioning to a particular attribute state. For an attribute with N possible states, $L = \{l_1, \cdots, l_N\}$, each *AttributeModel* will have at most N immediate transitions $\{t_F^1, \cdots, t_F^N\}$. The transition t_F^i is associated with the robot's attribute transitioning to state l_i.

Consider, for example, the *AttributeModel* depicted in Figure 4.13 and outlined in dark green. This is the GSPNR that models the battery discharge for a robot that has just finished travelling having battery at state "B1". A token reaches the *attribute* place "Finished_Travel_B1", and the immediate transition that fires dictates which battery state the robot transitions to. If the robot discharges and its battery transitions from state "B1" to "B0", the transition "AfterTravelB1_B0" fires, and the token representing the robot is consumed from the *attribute* place, and created in the decision place that represents the robot having battery "B0".

Considering an arbitrary action a, and an attribute with N possible states, all of the *AttributeModels* can be specified as a series of conditional probabilities: $p(l_j|a, l_i)$. This expresses the probability of the robot's attribute transitioning to level l_j after executing action a and having started out at level l_i.

$$P_a = \begin{bmatrix} p(l_1|a, l_1) & \cdots & p(l_N|a, l_1) \\ \vdots & \ddots & \vdots \\ p(l_1|a, l_N) & \cdots & p(l_N|a, l_N) \end{bmatrix}$$

Each row i of matrix P_a specifies a single *AttributeModel* for executing action a starting in a particular level l_i. Considering for example, that the robot's battery is discretized into three levels: {B0, B1, B2}. Two examples of full transition models can be seen in Table 4.1. In the left, the full transition model for a $Travel$ action can be seen, and in the right, a full transition model for a $Recharge$ action can be seen. Some rows are not specified, and this means that the action cannot be executed if the robot's attribute is at that level. For example, in the $Recharge$ action, it does not make sense to recharge the robot when its



(a) Attribute Models built according to $Travel$ transition model: when robot executes action in battery level B2 (above) and level B1 (below);

(b) Attribute models built according to $Charge$ transition model: when robot executes action in battery level B1 (above) and level B0 (below);

Figure 4.13: Attribute models for transition models specified in Table 4.1;

battery is already at the highest level B2 and so the third row is left unspecified. For the $Travel$ action, the first row corresponding to travelling starting with the battery in level B0 is also left empty: this means that the robot cannot execute the $Travel$ action when at the lowest battery level.

Table 4.1: Examples of Full Transition Models for $Travel$ (left) and $Recharge$ (right) action

$Travel$	B0	B1	B2
B0	-	-	-
B1	0.4	0.6	0
B2	0.1	0.4	0.5

$Recharge$	B0	B1	B2
B0	0	0.1	0.9
B1	0	0	1.0
B2	-	-	-

To build a specific *AttributeModel* for action a executed at level i, the row $P_a(i,:)$ is used. Each nonzero probability in the row corresponds to an immediate transition, with its weight equal to the probability specified. These are all connected to the *attribute* place by input arcs, and so the probability of each transition firing mirrors the probability defined in the transition model. We define a function $TransitionModel : A \times L \rightarrow G$, where A is the set of all possible actions, L is the set of all possible attribute states, and G is a GSPNR. This function outputs the *AttributeModel* that corresponds to the attribute evolution after executing action $a \in A$ at a particular attribute level $l \in L$.

From *DecAct-G* to Heterogeneous GSPNR Models with a single robot attribute

We propose a novel algorithm that can build the full GSPNR model for a heterogeneous multi-robot problem described as *DecAct-G*, where a single robot-type has a single attribute with finite and discrete

state space (only one total attribute can be modelled). It is set out in Algorithm 3. The inputs that the algorithm uses are the following:

1. $\mathcal{DA}$, a *DecAct-G*, with decision node set D, action set A, and edge set E. Another subset A^* is defined that includes any action node involving the type of robots with an attribute: $A^* = \{\forall a \in A \mid \exists(d,a) \in Dec(a)$ with $d \in r_L\}$
2. $R = \{r_1, \cdots, r_L, \cdots, r_K\}$, a set of the different types of robots, and a function $RobotType(d)$ defining which robot-type each decision node pertains to. The robot type with an attribute is denoted r_L;
3. $L = \{l_1, \cdots, l_N\}$, a set of N possible attribute states.
4. An action model template $ActionModel(a, r)$ for each action node $a \in A$, and for each type of robot $r \in R$. This template can be instantiated into a particular attribute state $ActionModel(a, r, l)$.
5. A transition model function $TransitionModel(a, l)$ that maps all action nodes involving the type of robot with an attribute $a \in A^*$ to a *AttributeModel*.

To build the full GSPNR model each decision node d_i that belongs to the robot with an attribute is unfolded into N decision places. Every other decision node does not need to be replicated, and is taken as an ordinary decision place. This is done by iterating through all the decision nodes in the *DecAct-G*, in the **for** loop in lines 2-8.

All action nodes a are replaced with a corresponding $ActionModel$ (see **for** loop in lines 9-25). If the action involves the robot type with the modelled attribute (checked with the **if** condition in line 10) the action model template $ActionModel(a, r)$ needs to be replicated for each possible attribute state, and for each different type of robot involved r. The correct *AttributeModels* is added and connected to the *final* transition of the custom $ActionModel(a, r, l)$. If all of the robots involved in executing the action do not have the attribute, action models $ActionModel(a, r)$ do not need to be replicated, and are simply merged into the full GSPNR model.

Now all the action models or *AttributeModels* need to be connected to the appropriate decision places by input and output arcs. The algorithm iterates through all of the decision edges in the **for** loop in lines 26-36, and adds input arcs between the decision places and the *decision* transition. The same is done for the output edges of the *DecAct-G* (in the **for** loop in lines 37-48) adding output arcs between either the *AttributeModel* and the appropriate decision places, or if the robots involved do not have an attribute, directly from the $ActionModel(a, r)$ to the decision place.

To further illustrate this algorithm, a GSPNR model that discretizes the small robots' battery into three states {B0,B1,B2} can be seen in Figure 4.15. It is based on *DecAct-G* in Figure 4.14, where small UGVs can travel from location "Panel1" to "Panel2", and recharge in cooperation at "Panel1" with the help of a large UGV. The transition models for these two actions are the ones defined previously (see Table 4.1), and so are the $ActionModels$ (see Figures 4.11 and 4.12).

As can be seen, the small UGV nodes {Panel1, Panel2} were replaced with three decision places, one for each battery level. Action node "Travel" was replicated into only two action models, because the

Figure 4.14: *DecAct-G* representing system where small UGVs can travel between location ”Panel1” and ”Panel2”, and recharge in cooperation with UGVs at ”Panel1”

TransitionModel specified does not allow the small UGVs to travel when their battery is at battery state ”B0”. Each of these $ActionModels$ was connected to a $AttributeModel$, with the immediate transitions of these attribute models connected to the correct decision place, according to the state of the battery to which the robot transitions.

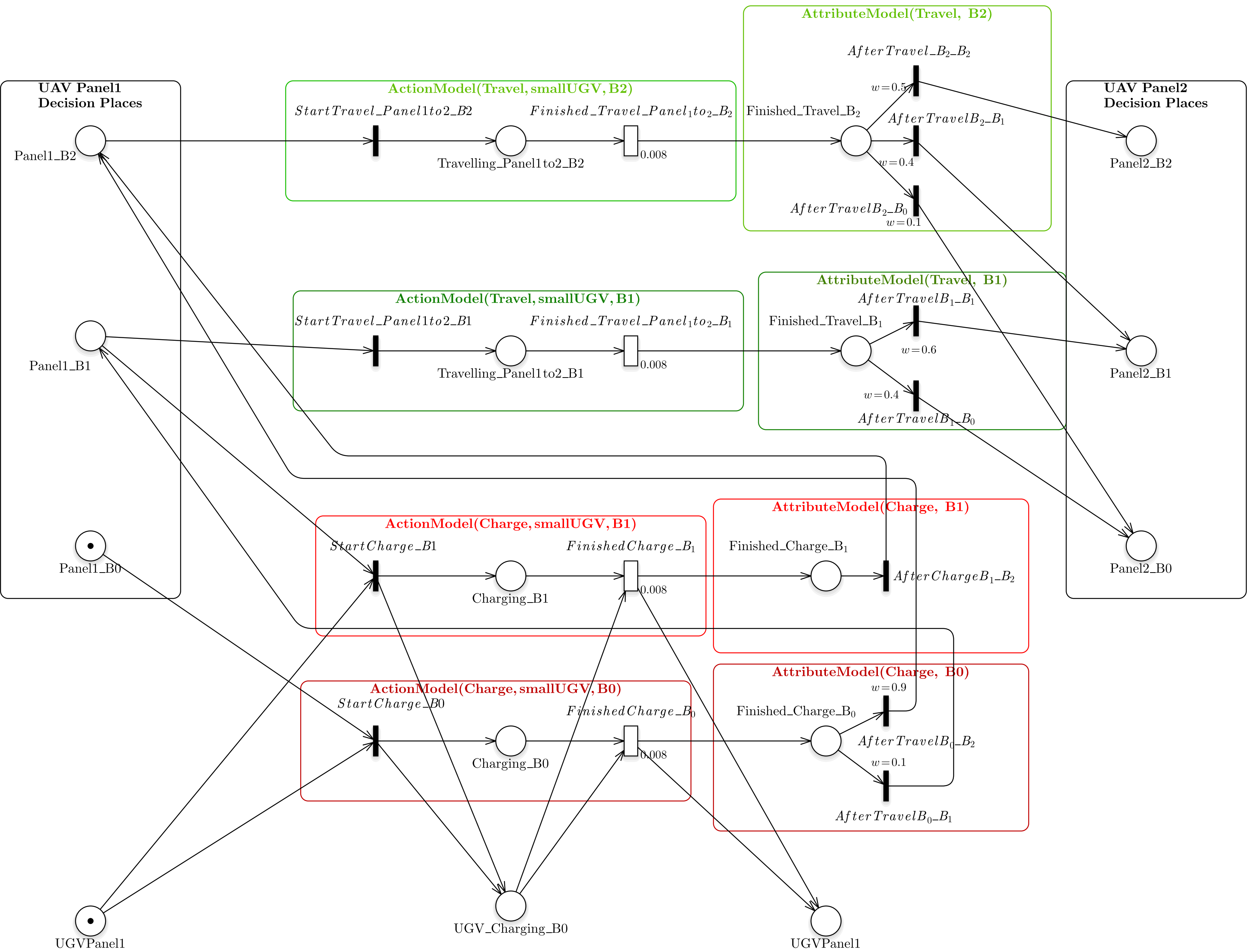


Figure 4.15: Full GSPNR model for *DecAct-G* found in Figure 4.14, with the smaller UGV's battery discretized into three levels.

1: **procedure** G_F = FULLGSPNR($\mathcal{DA}$, R, $RobotType(.)$, $ActionModel(.,.)$, $TransitionModel(.,.)$)
$\mathcal{DA} = (D, A, E)$
$R = \{r_1, \cdots, r_K\}$
$L = \{l_1, \cdots, l_N\}$ for robot-type $r_L \in R$
$RobotType(d) \rightarrow r, \forall d \in D,$ and $\forall r \in R$
$ActionModel(a, r) \rightarrow G_C, \forall a \in A,$ and $\forall r \in R$
$TransitionModel(a, l) \rightarrow G_T, \forall a \in A^*,$ and $\forall l \in L$

$G_F = (\emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset)$

2: **for** $d_i \in D$ **do**
3: **if** $RobotType(d_i) == r_L$ **then**
4: Replicate the decision node into N different decision places - $d_i = \{d_i^1, d_i^2, ..., d_i^N\}$
5: **else**
6: Add decision node as a single decision place d_i
7: **end if**
8: **end for**
9: **for** $a \in A$ **do**
10: **if** $a \in A^*$ **then**
11: **for** $(d, a) \in Dec(a)$ **do**
12: $r = RobotType(d)$
13: **for** $l = 1, ..., N$ **do**
14: Format the template of the $ActionModel(a, r, l)$ and merge it into G_F;
15: Build $AttributeModel(a, l)$ and merge it into G_F;
16: Connect $t_F \in ActionModel$ to $p_A \in AttributeModel$ with an output arc;
17: **end for**
18: **end for**
19: **else**
20: **for** $(d, a) \in Dec(a)$ **do**
21: $r = RobotType(d)$
22: Merge $ActionModel(a, r)$ into G_F
23: **end for**
24: **end if**
25: **end for**
26: **for** $e = (d, a) \in E$ **do**
27: **if** $RobotType(d) == r_L$ **then**
28: **for** $l = 1, \cdots, N$ **do**
29: $t_D \in ActionModel(a, r, l)$
30: $I_F(d^l, t_D) = 1$
31: **end for**
32: **else**
33: $t_D \in ActionModel(a, r)$
34: $I_F(d, t_D) = 1$
35: **end if**
36: **end for**
37: **for** $e = (a, d) \in E$ **do**
38: **if** $RobotType(d) == r_L$ **then**
39: **for** $l = 1, \cdots, N$ **do**
40: **for** $t_F^i \in AttributeModel(a, l)$ **do**
41: $O_F(t_F^i, d^i) = 1$
42: **end for**
43: **end for**
44: **else**
45: $t_F \in ActionModel(a, r)$
46: $O_F(t_F, d) = 1$
47: **end if**
48: **end for**
49: **end procedure**

Algorithm 4: Building a heterogeneous GSPNR model from a Decision-Action Graph

Chapter 5

Multi-Robot GSPNR Toolbox

Chapter 5 provides an outline of the architecture, algorithms and capabilities of the software package developed. It first showcases the different possible ways of creating GSPNR models. Next, the module that obtains optimal policies on a GSPNR is described. Section 5.4 describes the execution component of the package: it outlines the algorithm used to execute a GSPNR task plan, and how integration with ROS is achieved. Finally, the last section characterizes the toolbox's computational performance and how it scales when increasing the size of the GSPNR or the number of robots modelled.

5.1 Overview and Architecture

We developed the software package as an open-source add-on toolbox for the MATLAB platform. The MATLAB platform was chosen for several reasons: 1) it has a large, established user community; 2) the company behind MATLAB provides long-term support and backwards compatibility as the platform is updated; 3) the simplicity of MATLAB's scripting language allows new users to take advantage of our toolbox without having having to learn complicated syntax.

Conceptually, the toolbox can be divided into three separate modules, that can be used together or independently. All of these modules and their interactions can be seen in Figure 5.1. The main component is the *GSPNR* module, and it provides an implementation of GSPNR models and various ways to create and edit them. This module also allows the user to import and export these models from/to two external software packages: PIPE [17] and GreatSPN [18]. The *Policy Synthesis* module provides functions to compute optimal policies over GSPNR models. The remaining module *Execution Manager* allows users to execute GSPNR policies on real robots. It does so by utilizing Matlab's ROS Toolbox, that allows communication with ROS action servers.

The package is available in a public repository [21]. It includes interactive tutorials (using the MATLAB's LiveEditor feature) that explain exactly how to use each of the three modules. The work done in this chapter was supported by funding through a Mathworks-RCF project.

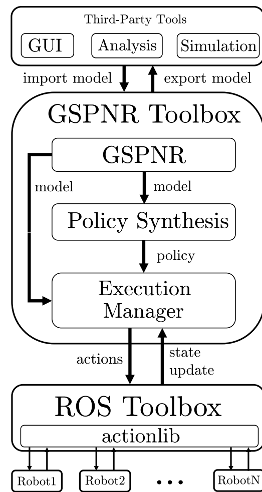


Figure 5.1: Overview of the toolbox's architecture

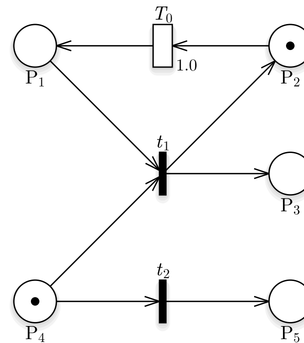


Figure 5.2: GSPNR model built in code snippets

5.2 Creating GSPNR Models

The *GSPNR* module provides an API that exposes methods to create GSPNR models. The API consists in a class that can be instantiated to create a GSPNR object. This class is equipped with methods that can add places, transitions, and arcs. The weight of each added transition can be specified, and so can the arc multiplicity (specifying the amount of tokens consumed/created when the connected transition fires, for input and output arcs, respectively). The initial marking of the GSPNR can be set, and a method to retrieve enabled transitions for the current marking is also exposed. Another method allows users to fire a given enabled transition, and this method updates the current marking. There is a method to define place and transition reward functions.

There are three main ways to create GSPNR models: 1) Directly using the API; 2) Using an external software tool that has a GUI, and then importing the GSPN into MATLAB; 3) Using a topological map. The next three subsections focus on each of these three ways of creating a GSPNR model.

5.2.1 Using the API

The code snippets in this section recreate the GSPNR model found in Figure 5.2. To create a GSPNR model with the toolbox's API the user must first create an empty instance of the class `GSPNR`:

```
example = GSPNR();
```

To add places the user can use the `GSPNR` method `.add_places(places, tokens)` that takes two inputs:

```
example.add_places(['P1', 'P2', 'P3', 'P4', 'P5'], ...
                   [0   , 1   , 0   , 1   , 0   ]);
```

The first input is the string vector that lists all the places in the GSPNR, and the second input an integer vector of the same length, which specifies how many tokens are in each corresponding place in the

GSPNR's initial marking m_0.

To add transitions to a GSPNR instance, the method `.add_transitions(transitions, types, rates)` is used:

```
example.add_transitions(['t1' , 't2' , 'T0'], ...
                        ['imm', 'imm', 'exp'],...
                        [0    , 0    , 1]);
```

This method takes three inputs, all vectors of the same length. The first input is the list of transitions to add to the GSPNR, the second input specifies the type of each transition (either immediate or exponential) and the third input specifies the weight of each transition added.

To add input and output arcs to the model, the method `.add_arcs(places, transitions, types, multiplicity)` is used. The four inputs are vectors of the same length: `places`, `transitions` defining which place and transition each arc is connected to, and `types` specifying if the arc is an input or output arc. The `multiplicity` input specifies the multiplicity of each arc added.

```
arc_places =       ['P1','P1', 'P2' ,'P2','P3', 'P4','P4','P5'];
arc_transitions = ['t1','T0', 't1' ,'T0','t1', 't1','t2','t2'];
arc_type =         ['in','out','out','in','out','in','in','out'];
arc_multiplicity= [1,    1 ,   1 ,   1 ,   1 ,   1 ,  1 , 1];
example.add_arcs(arc_places,arc_transitions,arc_type,arc_multiplicity);
```

Rewards can also be added with the method `.set_reward_functions(elements, values, types)`, where the input `elements` is the list of elements that rewards will be added to (either places or transitions), `values` are the actual rewards (negative or positive real numbers) and the `types` vector specifies if the corresponding element is a place or a transition.

```
reward_names = ['p2', 't1'];
reward_values = [1, 5];
reward_types = ['place', 'transition'];
example.set_reward_functions(reward_names, reward_values, reward_types)
```

5.2.2 Using a GUI

From all software tools that work with GSPNs, PIPE [17] and GreatSPN [18] stand out as having two desired features. Both work with a GUI that allows users to graphically design their GSPN models, and both have powerful analysis engines that can, for example, check if a GSPN model is bounded. Thus, these two programs complement our toolbox's capabilities, and so methods that interface with these external tools were developed. The two functions that allow users to import GSPNs from PIPE or GreatSPN are the following:

```
function [nGSPN, gspn_struct_array] = ImportfromGreatSPN(PNPRO_path)
function gspn = ImportfromPIPE(xml_filepath)
```

The function that imports models from GreatSPN, `ImportfromGreatSPN(PNPRO_path)` takes as input the string containing the path to the ".PNPRO" file that GreatSPN uses. It imports all the GSPNs found in the path and returns them in the structure array `gspn_struct_array`, where each fieldname is equal to name of the GSPN in GreatSPN.

```
[nGSPN, gspn_struct_array]=ImportfromGreatSPN('example_GreatSPN.PNPRO')
gspn1 = gspn_struct_array.GSPN
gspn2 = gspn_struct_array.other_GSPN
```

The code above imports the GreatSPN project seen in Figure 5.3. The output `nGSPN` in this case will be 2, because two GSPNs were imported.

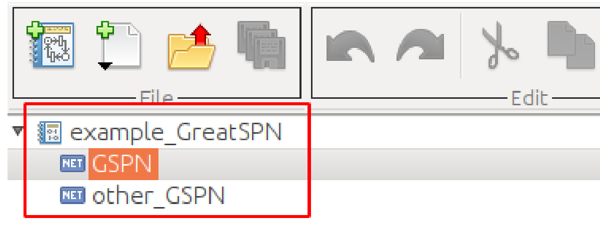


Figure 5.3: Example of GreatSPN ".PNPRO" file

Importing from PIPE is as simple, but only one GSPN can be imported at a time (as PIPE only allows one Petri net per save file):

```
gspn = ImportfromPIPE('PIPE_Example.xml')
```

Imported GSPN instances are exactly the same as the ones created programmatically: users can add places, transitions, arcs, etc. after importing GSPNs. Any rewards have to be added manually, as neither PIPE or GreatSPN allow for the inclusion of place or transition rewards.

GSPNR instances can also be exported to PIPE or GreatSPN using the functions `ExportToPIPE(GSPNR, save_path, size)` and `ExportToGreatSPN(GSPNR,savepath, size)` respectively. The `save_path` input is a string defining the path where the ".PNPRO" and ".xml" files will be saved to afterwards open with the applications. The third input `size` is an integer vector of length 2, defining the length and width of the area in which the GSPNR elements will be drawn. These two functions will create either an ".xml" or ".PNPRO" file that can be opened with either PIPE or GreatSPN, respectively. This file will contain the GSPN object specified as input `GSPNR`.

5.2.3 Using a Topological Map

Most multi-robot inspection problems require the use of a topological map. The environment in which robots operate is usually abstracted into a set of discrete locations represented as nodes in a topological map and edges of the map specify how the robots are able to navigate between locations. Furthermore, each location has a set of particular actions that the robot can choose to execute when in that location. The toolbox provides a function that assists the user in creating a GSPNR model for this type of multi-robot problem. Any problem that departs from this format cannot be modelled with this function.

The implementation of this function was done following Algorithm 5. The algorithm requires 4 input arguments. $TopologicalMap = (V, E)$ - a directed graph with nodes $v \in V$ representing locations, and edges $e = (s, t) \in E$ representing all possible navigation actions between locations. $Actions : V \rightarrow A$ - a function mapping between locations and actions that the robots can execute in each location. $Models : A \rightarrow G_T$ - a function specifying the GSPNR G_T that is used to model action $a \in A$. Each of these GSPNR models is a template that can be formatted to represent the same action done in multiple locations. $InitialLocations : R \rightarrow V$ - specifying how many robots have their initial location $v \in V$.

All GSPNRs in the $Models$ argument must include the decision place to which the decision transition is connected by an input arc, and it also must include the decision place connected to the final exponential transition. As these models will be replicated when the same action is done in different locations, these models must make use of tags "$<n>$", where n is an integer. The `GSPNR` class has a method `.format(tags)`, that will substitute each tag "$<n>$" with the n-th element of input argument `tags`. This allows to define a single GSPNR for each action, and then format it and add it to the full model without conflicts. Additionally, a template GSPNR for the $navigation$ action must be specified in the $Models$ input. All edges in the topological map will be replaced by copies of this model. Normally, the average time to navigate between locations is not the same for every navigation action. We allow the edges of the input $TopologicalMap$ to have a weight defined by function $W : E \rightarrow \mathbb{R}_{>0}$. This weight specifies the average time taken to navigate between the locations specified by the edge, and our algorithm includes a function that changes the rate of the exponential transition of the $navigation$ model template. This way each formatted $navigation$ model can already include an estimation of this parameter.

The algorithm begins by iterating through all locations, and adding the GSPNR models for all possible actions in each location. This is done by copying an instance of GSPNR model G_T, formatting it so it represents the action done in that particular location, and merging the model with the overall GSPNR. Next, all navigation models are added according to the edges in the topological map. Finally, the initial marking of the full GSPNR model is set according to input $InitialLocations$, adding one token in the place corresponding to each location that a robot starts out in.

This combination of a topological map and a function that defines which actions can be executed in each location is equivalent to a homogeneous *DecAct-G* as defined in Section 4.1.1. Each edge in the topological map is interpreted as a navigation action node in the *DecAct-G* and so is each action available at each location. The locations themselves are interpreted as decision nodes: each robot in a location must choose between traveling to a different location, or executing one of the available actions. This function can also be used to create a GSPNR model for a heterogeneous multi-robot problem. Suppose we have a problem involving two different types of robots. The function is used twice, creating two GSPNR models, one for each type of robot. These two GSPNR models can then be merged to create a single model. To model synchronized and asynchronized cooperative actions, the $Models$ input must include common *decision* and *final* transitions, just as explained in Sections 4.2 and 4.2 of Chapter 4.

To use this algorithm, the user calls the following function:

```
function gspn = GSPNRCreationfromTopMap(top_map,...
```

```
function G = GSPNRCreationFromTopMap(TopologicalMap, Actions, Models, InitialLocations)
    for node in TopologicalMap do
        PossibleActions = Actions(node)
        for action in PossibleActions do
            G_T = copy(Models(action))
            Format(G_T, [node])
            G = MergeGSPNR(G, G_T)
        end for
    end for
    for source, target = edge in TopologicalMap do
        G_T = copy(Models(navigation))
        Format(G_T, [source, target])
        G = MergeGSPNR(G, G_T)
    end for
    for robot in InitialLocations do
        p_i = InitialLocations(robot);
        m_G(p_i) = m_G(p_i) + 1;
    end for
end function
```

Algorithm 5: Building a homogeneous GSPNR Model with a topological map

```
                                        action_dict ,...
                                        models ,...
                                        robot_marking)
```

This function takes four input arguments:

- `top_map` : a Matlab `digraph` object, defining the topological map of the locations, with named nodes, equivalent to input $TopologicalMap$ in Algorithm 5;
- `action_dict`: a structure array, where each fieldname is a node in the `top_map` argument, and the corresponding value is a string array, where each element is the name of a possible action that the robots can execute in that specific location. It is equivalent to input $Actions$ in Algorithm 5;
- `models`: a structure array, where each fieldname is the name of an action defined in `action_dict` argument, and the value of each field is the GSPNR model corresponding to that particular action. It is equivalent to input $Models$ in Algorithm 5;
- `robot_marking`: a structure array, where each fieldname is a node in the `top_map` argument, and the value is the number of robots that start out in the corresponding location, equivalent to input $InitialLocations$ in Algorithm 5;

Example

To further explain this function with a small example, we will use it to model the multi-robot problem described in Scenario 1. The robots can travel between these two panels, and the topological map for this system can be seen in Figure 5.4 and it is equivalent to the *DecAct-G* found in Figure 4.2. Figure 5.5 shows the GSPNRs part of the `models` argument. The code to build this system is the following:

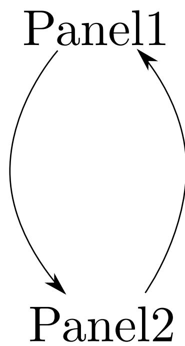
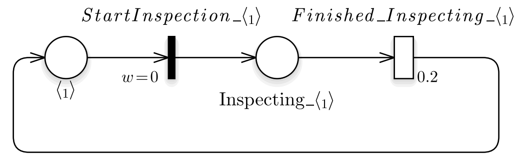
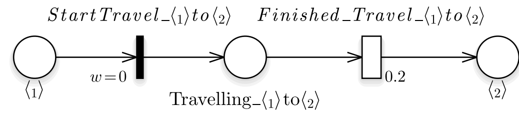


Figure 5.4: Visualization of input topological map;



(a) `models.inspection`



(b) `models.navigation`

Figure 5.5: Visualization of GSPNRs in input `models`

```
adjacency_matrix = [0 1;
                    1 0];
top _map = digraph(adjacency_matrix,{'Panel1' 'Panel2'},'omitselfloops');

actions_available.Panel1 = ['inspection'];
actions_available.Panel2 = ['inspection'];

robot_marking.Panel1 = 2;

PNPRO_path = 'solarfarm.PNPRO';
[nGSPN, models] = ImportfromGreatSPN(PNPRO_path);

solarfarm = GSPNRCreationfromTopMap(top_map, actions_available, ...
                                    models, robot_marking)
```

The resulting GSPNR `solarfarm` is exactly the same as the one found in Figure 4.4.

5.3 Policy Synthesis and Evaluation

The *Policy Synthesis* module of the toolbox provides methods to obtain optimal policies over GSPNR models. Figure 5.6 outlines the architecture of this module, and how to obtain an optimal policy for a GSPNR. The first step in obtaining any policy is to convert the GSPNR model into an equivalent MDP model. Alongside building the equivalent MDP, this step obtains a mapping between GSPNR markings and the MDP states. The second step is applying value iteration to the equivalent MDP, and returns the optimal policy for the MDP model. The final and third step uses the mapping between markings and MDP states to translate the MDP policy into a GSPNR policy. The following three subsections each focus on one of these steps taken to compute an optimal policy. Section 5.3.3 describes an additional

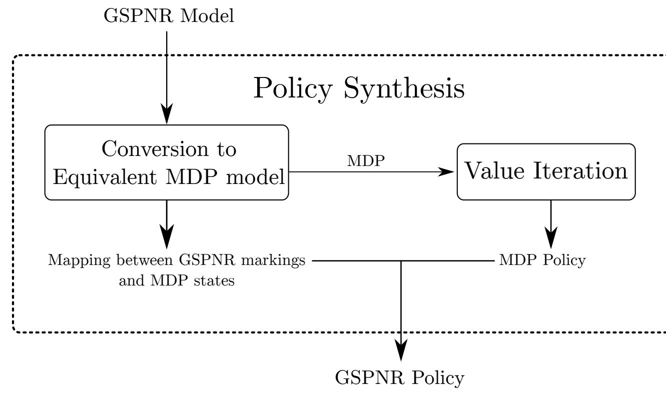


Figure 5.6: Outline of *Policy Synthesis* module

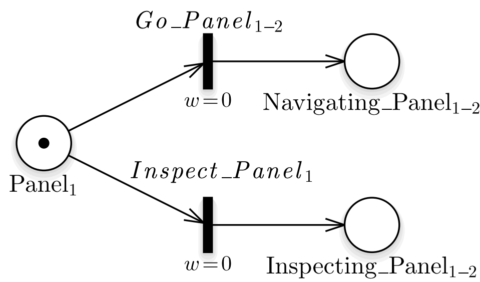


Figure 5.7: Partial GSPNR model with two immediate transitions that represent arbitrary choices;

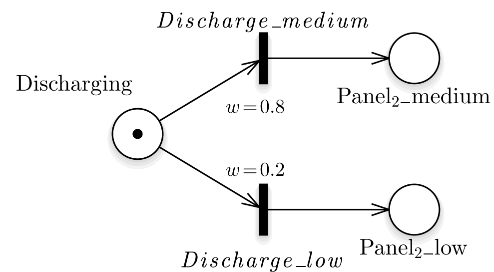


Figure 5.8: Partial GSPNR model with two immediate transitions that represent a random switch;

method that aids users in obtaining results from calculated policies in order to evaluate their efficiency, without having to execute the GSPNR plan in real robots.

5.3.1 Conversion to MDP

To convert GSPNRs to MDPs, each marking is associated to a state in the MDP. The state space of the equivalent MDP corresponds to all markings that can be reached from the initial marking - $R(G)$. For a given vanishing marking, there is a need to distinguish between two different types of immediate transitions: transitions that the system can arbitrarily fire (such as the decision transitions defined in Chapter 4), and immediate transitions that are fired probabilistically according to their weight (such as the ones found in *AttributeModels*).

Figure 5.7 contains a partial GSPNR with two immediate transitions that represent decisions that the multi-robot system can arbitrarily fire. They both have a weight of 0 to denote this characteristic. When the system decides to fire one of these transitions, the marking evolves in a deterministic manner. If for example, the transition ”Go_Panel_1-2” fires, the token will be unequivocally created in place ”Navigating_Panel1-2”. On the other hand, the GSPNR depicted in Figure 5.8 contains two immediate transitions that have a non-zero weight and are in conflict. This is considered a ”random switch” because the transition that fires cannot be chosen by the robotic system. The transition that fires is determined prob-

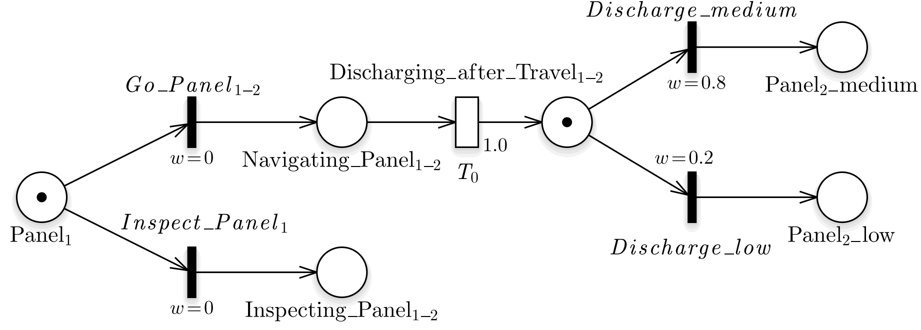


Figure 5.9: Partial GSPNR model with both types of immediate transitions;

abilistically by the weights of each transition. In this case, ”Discharge_medium” will fire with probability of 0.8 ($\frac{0.8}{0.8+0.2}$) and transition ”Discharge_low” will fire with probability of 0.2. Both cases can occur in the same GSPNR, as can be seen in Figure 5.9. Here, the multi-robot system must choose between firing either one of the immediate transitions representing decisions (the ones with weight 0), or executing the random switch, and probabilistically determining which of the non-zero weight transitions will fire.

Thus, for a given marking $\boldsymbol{m}$ each immediate transition $t \in T_I$ with weight $W(t) = 0$ in the GSPNR corresponds to an action in the MDP, and any random switch $(t_1, \cdots, t_N) \in T_I \mid W(t_i) > 0$ corresponds to action denoted with symbol $\top$.

For a given tangible marking, the race condition between all enabled exponential transitions corresponds to an action $\perp$ in the MDP. The rate between two states (s, s'), $R(s, s')$ is the sum of the weight of all exponential transitions that drive the system from state s to s'. The exit rate of state s, $Ex(s)$ is the sum of all of its rates. Constant η is the maximum exit rate plus one:

$$R(s, s') = \sum_{t \to s' = (s,t)} W(t)$$

$$Ex(s) = \sum_{s'} R(s, s')$$

$$\eta = \max\{Ex(s)\} + 1$$

In each state corresponding to a tangible marking, the only action enabled is $\perp$, and the probabilities of transitioning to any other markings are uniformized by constant η. The uniformization process also includes adding a self-loop representing the possibility of staying in each tangible marking [7], [22], that fixes the mean sojourn times in each marking.

Transition rewards $r_T(t)$ in the GSPNR are directly equivalent to the reward pair $r(s, a)$ when the action is t. As place rewards are given for each unit of time that the place has at least one token, each tangible marking has an associated total reward $r_S(s)$. This allows the definition of the reward $r(s, \perp)$ for these markings, normalized by the constant η as a consequence of uniformization.

GSPNR $G = (P, T, I, O, W, \boldsymbol{m}_0, r_p, r_t)$ can be converted to its equivalent MDP $\mathcal{M}_G = (S, A, p, r, s_0)$, where its state space is equivalent to the reachability graph of the GSPNR - $S = R(G)$. The MDP action set is the union of all transitions with weight equal to zero, the $\top$ action equivalent to random switches,

and $\perp$ action equivalent to race conditions between exponential transitions:

$$A = \{\forall t \in T_I \mid W(t) = 0\} \cup \{\perp, \top\}$$

The probability transition function $p : S \times A \times S \to [0, 1]$ is defined as:

$$p(s, a, s') = \begin{cases} 1.0 & s \in VM, \forall a \in T_I, W(a) = 0 \\ \frac{W(t)}{\sum_{t \in E(m)} W(t)} & s \in VM, a = \top \\ \frac{R(s,s')}{\eta} & s \in TM, a = \perp, s \neq s' \\ 1 - \frac{Ex(s) - R(s,s)}{\eta} & s \in TM, a = \perp, s = s' \end{cases}$$

The reward function $r : S \times A \to \mathbb{R}$ is defined as:

$$r(s, a) = \begin{cases} r_T(t) & s \in VM, \forall a \in T_I, W(a) = 0 \\ \frac{r_S(s)}{\eta} & s \in TM, a = \perp \end{cases}$$

The initial state of the MDP is defined as the initial marking of the GSPNR $s_0 = \boldsymbol{m_0}$.

The toolbox implements this conversion to MDP with a method of class GSPNR, .toMDP_without_wait ():

```
[emb_MDP,...
 covered_marking_list,...
 covered_state_list,...
 covered_state_type] = solarfarm.toMDP_without_wait(GSPN)
```

There are four output arguments of this method: emb_MDP which is an instance of the MDP class, containing the equivalent MDP; covered_marking_list an array where each row is a reachable marking in the GSPNR; covered_state_list is a column-vector with the names of the states corresponding to each row in the previous output; and finally covered_state_type is also a column vector indicating if each marking found is a tangible or vanishing marking.

With WAIT States

The urgency assumption disallows the firing of enabled exponential transitions whenever at least one immediate transition is enabled. In multi-robot GSPNR models where robots are represented as tokens, this forces the robot's decision-making to be instantaneous, and this corresponds to removing the exponential transitions in the MDP whenever any immediate transition is also enabled. However, it might be advantageous for the system to wait for any of the enabled exponential transitions to fire. For example, in an inspection problem, if there are no wait states, the robots must be constantly either navigating or inspecting. In certain problems it might be useful to allow robots to wait for another robot to finish inspecting to save battery.

Formally, a marking is hybrid when the enabled transition set includes both immediate and exponen-

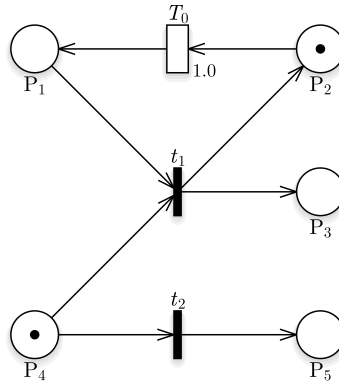


Figure 5.10: Example GSPNR model equivalent to MDP in Figures 5.11 and 5.12, without and with wait states, respectively;

tial transitions: $\{\forall \boldsymbol{m} \mid t_i, t_j \in E(\boldsymbol{m})$ and $t_i \in T_I, t_j \in T_E\}$. To allow robots for robots to wait for another, such hybrid markings are unfolded into two states: 1) the original state where only its immediate transitions are enabled, plus a special $wait$ action; 2) a second state, the wait state, with only the $\perp$ action enabled, corresponding to the system waiting for a race condition to finish. The $wait$ action leads with a probability of 1 to the wait state, and so the system can choose to wait for a race condition to finish [5].

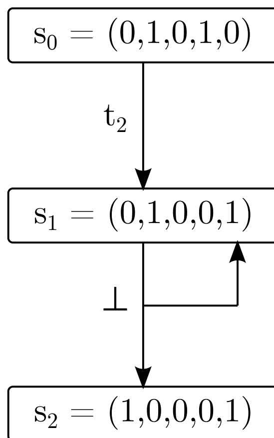


Figure 5.11: States and Actions of Equivalent MDP of GSPNR found in Figure 5.10, without wait states;

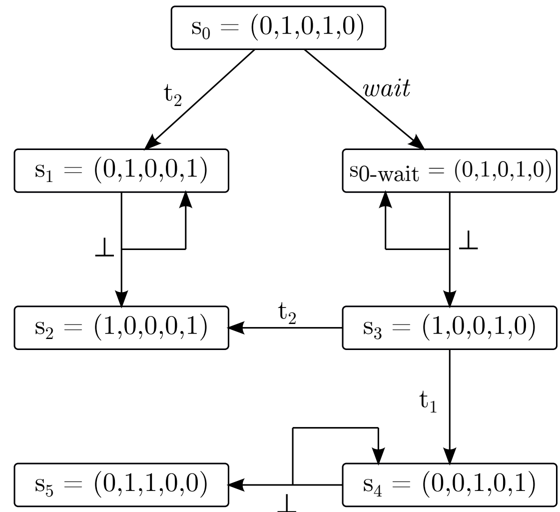


Figure 5.12: States and Actions of Equivalent MDP of GSPNR found in Figure 5.10, but with wait states;

The equivalent MDP to the GSPNR found in Figure 5.2 can be found in figures 5.11 and 5.12, respectively. For simplicity reasons, the probabilities of the action outcomes were omitted. The $\perp$ action replaces all exponential transitions, and self-loops were introduced in states corresponding to tangible markings. As can be seen, the introduction of wait states can significantly increase the state space of the equivalent MDP, even accounting for the duplication of hybrid states.

The toolbox implements this conversion to MDP with wait states with a method of class `GSPNR`, .

toMDP():

```
[emb_MDP,...
covered_marking_list,...
covered_state_list,...
covered_state_type] = solarfarm.toMDP(GSPN)
```

5.3.2 Value Iteration

Value iteration is a dynamic programming algorithm that iteratively approximates the value function of each state in an MDP. By checking which enabled action has the best value, this algorithm can also return the optimal policy for the MDP [7]. A solver was implemented that can run value iteration on a given MDP object:

```
function [values, policy, max_res, timed_out] = value_iteration(MDP,...
                                                    max_min, ...
                                                    gamma, ...
                                                    epsilon, ...
                                                    max_duration)
```

The five input arguments are the following: 1) MDP is the instance of the MDP on which to run value iteration; 2) max_min must be set to 1 if value iteration should maximize the reward accumulated, and 0 if it should be minimized; 3) gamma sets the discount factor used, it should be a value between 0 and 1 (if the criterion optimized is the total accumulated reward); 4) epsilon sets the convergence criteria: when the calculated value of all states changes by less than epsilon the algorithm stops; and finally 5) max_duration sets a time limit to the algorithm in seconds - if this argument is equal to 0, the algorithm runs until it has converged according to epsilon.

The outputs of this function are the following: 1) values is a vector that specifies the value of each state in the MDP as approximated by the value iteration algorithm; 2) policy is also a vector that specifies the optimal policy action for each state in the MDP - this is given according to the action indexes as determined internally by the MDP instance; 3) max_res defines the maximum difference in value for the last iteration done by value iteration; 4) timed_out argument can be 0 or 1 - it is 0 if the algorithm converged as defined by epsilon, and it is 1 if the algorithm timed out.

To facilitate the use of this policy synthesis module, GSPNR class instances have a method .policy_synthesis() that automatically converts the GSPNR to an MDP and runs value iteration on the resulting MDP:

```
[policy_struct, error, nStates] = solarfarm.policy_synthesis(timeout,...
                                                 discount_factor,...
                                                 wait)
```

The method takes three input arguments: 1) timeout establishes an upper time limit for running value iteration; 2) discount_factor establishes the discount factor to be used in value iteration; and 3) wait that must be 1 if the conversion to MDP should include wait states and 0 if not. The output arguments

of this method are the number of states in the equivalent MDP `nStates`, the maximum error in value iteration's last iteration `error`, and most importantly `policy_struct`, a data structure that saves the obtained policy in a format compatible with the toolbox's execution module.

The policy contained in this output `policy_struct` is already the GSPNR policy. Formally, the process to convert a GSPNR model to an MDP allows us to build a mapping between reachable markings in the GSPNR G and states in its equivalent MDP $\mathcal{M}$: $Map : R(G) \to \mathcal{M}$. After running value iteration, we obtain the MDP policy $\pi_{\mathcal{M}}$, and with the data from Map we can build the GSPNR policy π_G: $\pi_G = \pi_{\mathcal{M}}(Map(m))$. As the actions in the MDP correspond to immediate transitions in the GSPNR, we have the correct formulation of a GSPNR policy as defined in Chapter 2.

5.3.3 Policy Evaluation on GSPNRs

Evaluating the efficiency of a policy in a multi-robot system normally involves measuring how much time the system occupies particular states, or how frequently an action is executed. For example, in an inspection problem we might want to know how much time elapses, on average, between having all solar panels inspected. Or we might want to measure the proportion of time that robots are discharged, and thus not able to inspect.

To allow users to do this policy evaluation without actually executing the policy on real robots, the `GSPNR` class has a method to simulate the policy on the GSPNR model. This consists of firing a given number of transitions and keeping track of how much time the GSPNR stays in each marking, and which transitions were fired. This is an approximated sampling approach. It is expected that as the simulated time approaches infinity, the mean time spent in each marking converges to its true value. An exact method would entail applying the policy to the GSPNR and finding its equivalent continuous-time Markov chain (CTMC). From this CTMC model the mean sojourn times could be extracted analytically, such as done in [23]. This analytical closed-form approach has not yet been implemented in our toolbox and is left as future work.

It is important to note that our GSPNR model simulation is done in computer time so that a high number of transitions can be fired in a relatively short amount of time. This is done with the `.evaluate_policy()` method:

```
results = solarfarm.evaluate_policy(policy_struct, nTransitions, n_report)
```

This takes as input the `policy_struct` as returned by the policy synthesis method, the number of transitions to fire `nTransitions` and a third input that defines the frequency with which the method will report back to the Matlab command window so that the user can see the progress. The output of this method defines a structure array `results` with the following fields: 1) `results.markings` - each row contains a marking that the GSPNR went through; 2) `results.transitions` each row specifies the transition that was fired to lead to the corresponding marking; finally 3) `timestamps` - each element specifies the instant the corresponding transition was fired.

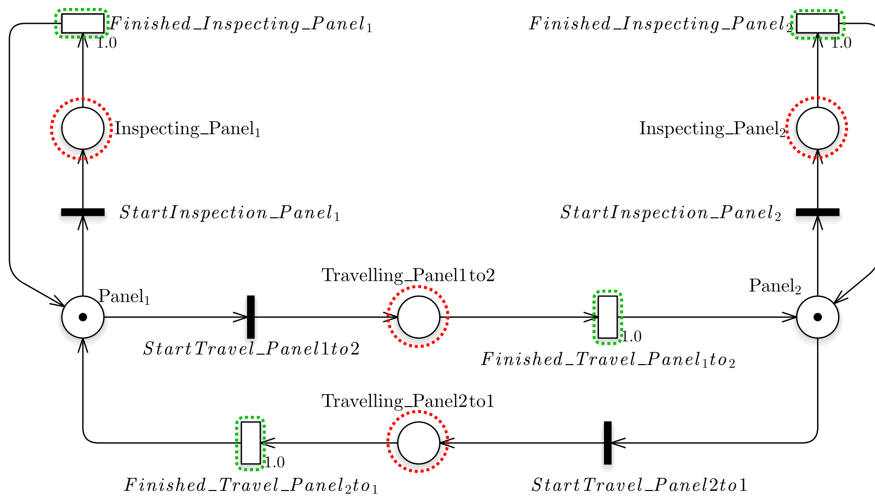


Figure 5.13: The same GSPNR Model of Figure 4.4, with the action places outlined in red and their exponential transitions outlined in green;

5.4 Execution Manager

This section focuses on the *Execution Manager* module and the methods it provides that enable users to apply GSPNR policies to real multi-robot systems. By applying a specific policy to a GSPNR model, we build a GSPNR task plan. This task plan resolves the non-determinism of the GSPNR model, as each marking/state has a single action available. The section is divided into two subsections. Section 5.4.1 provides a novel algorithm that executes this GSPNR task plan in a real multi-robot system, while Section 5.4.2 explains how to use the API to execute this plan and provides some implementation details about ROS integration.

5.4.1 Executing GSPNR Task Plan

A GSPNR policy is a mapping between markings and transitions, indicating which immediate transition to fire for a certain state/marking of the multi-robot system. By keeping track of the state of each robot, we can update the GSPNR marking so that it represents the real state of the multi-robot team. When a robot must make a decision, the GSPNR policy can be looked up for the current marking, and the transition that models the optimal decision is fired.

After the *decision* fires, a token representing a robot reaches an action place. When this happens, the corresponding robot must start executing the action modelled by the action place, and the exponential transition that models the time spent executing the action must only fire when the robot has finished executing the action. Each token must be mapped to a specific robot, so that the correct robot executes each action as the tokens move through the GSPNR. This can be seen in Figure 5.13, where the action places are highlighted in red and its exponential transitions in green. If, for example, the token representing ”Robot1” reaches the ”Inspecting_Panel1” place, the robot must actually start inspecting the first panel, and the transition ”Finished_Inspecting_Panel1” will only fire when the robot finishes inspecting. Figures 5.14 and 5.15 show some special cases that can be found in GSPNR models built according to the framework outlined in Chapter 4. These were taken into account when implementing the execution

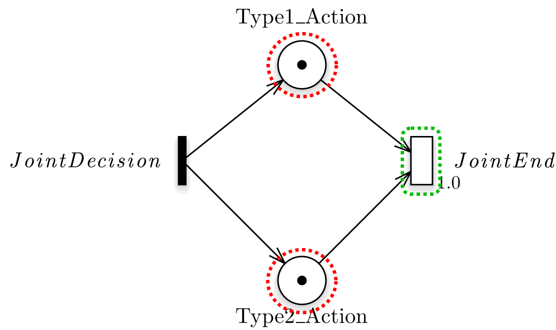


Figure 5.14: GSPNR model of a synchronized cooperative action where the transition outlined in green can only fire when both robots have finished their actions;

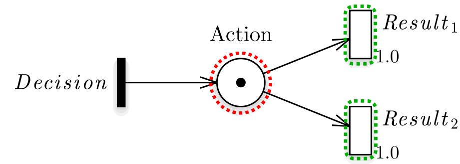


Figure 5.15: GSPNR model of an action with result, where according to the result of the action execution, one of the transitions outlined in green will fire;

algorithm: in synchronized cooperative actions (Figure 5.14), the exponential transition that represents the end of the action can only be fired when all robots have finished executing their part of the action. In action places connected to multiple transitions (Figure 5.15), the exponential transition fired depends on action execution outcome.

Algorithm 6 summarizes the implementation done to apply a policy on a GSPNR model. It takes as inputs a GSPNR model, a list of robots, a list detailing the places where each robot starts out at, and a policy mapping between markings and transitions. To do the mapping between each robot and each token we use a list called $RobotPlaces$. This list saves the place at which each robot is at, at all times. This list is initialized with the input variable $InitialPlaces$ (see line 2), and every time a transition is fired, this variable must be updated with a function - $Update(RobotPlaces)$ (lines 10, 15 and 35). Additionally, to execute the GSPNR, the algorithm must save the execution status of each robot, and this is done with the variable $RobotFlags$. If a robot is not currently executing anything, its flag will be set to $RobotFlags(robot) = FINISHED$, and if the robot is currently executing an action, its flag must be set to $RobotFlags(robot) = EXECUTING$. It is necessary to allow another value for a robot's flag: $RobotFlags(robot) = WAITING$ - a robot has this flag value if it has finished its own action, but depends on other robots so that the connected transition can fire (as shown in Figure 5.14 for two robots).

When the current marking is vanishing, depending on the policy, three possible cases can happen:

- The policy is undefined, and so a transition is randomly chosen to fire according to the weights of all enabled immediate transitions (line 7);
- The policy chooses to wait and no transition is fired (line 11);
- The policy defines an immediate transition, so that transition is fired (line 13),

If the current marking is tangible, either the robots are executing their actions, or new goals need to be sent to the respective robots. This distinction is done by checking each of the robot's flags, and only sending goals to robots which are not executing anything, and therefore have the $RobotFlags(robot) = FINISHED$ (see line 21).

The remaining time, the execution algorithm is waiting to receive a result message from any single robot (function WaitForResults). It can determine which transition it should fire (one of the transitions connected to the action place by an input arc), and include information on the robot's execution to determine a single transition. If the transition to fire is connected to multiple action places, the algorithm makes sure that all robots are finished with their respective actions, and if so, the transition is fired, and the flags of all robots involved are reset to the $FINISHED$ value, and the algorithm begins another loop of firing immediate transitions and sending new goals.

Require: G - GSPNR model; $Robots$ - list of N robots; $InitialPlaces$ - list of N places where robots begin; $Policy$ - policy for GSPNR model

1: **procedure** EXECUTION(G, $Robots$, $InitialPlaces$, $Policy$)
2: $RobotPlaces \leftarrow InitialPlaces$
3: $RobotFlags \leftarrow [FINISHED, \cdots, FINISHED]$ ▷ Set all robot execution flags as finished
4: **while not** $CTRL - C$ is pressed **do**
5: **while** $G.marking = vanishing$ **do**
6: $transition \leftarrow Policy(G.marking)$
7: **if** $transition == \emptyset$ **then** ▷ No policy is defined
8: $transition \leftarrow G.random_switch()$ ▷ Choose probabilistically transition to fire
9: $G.fire(transition)$
10: $Update(RobotPlaces)$
11: **else if** $transition == wait$ **then** ▷ Wait action, treat marking as tangible
12: $break$
13: **else** ▷ Policy defines a specific transition to fire
14: $G.fire(transition)$
15: $Update(RobotPlaces)$
16: **end if**
17: **end while**
18: **for** $robot \in Robots$ **do**
19: **if** $RobotFlags(robot) == FINISHED$ **then**
20: $RobotFlags(robot) \leftarrow EXECUTING$
21: $SendGoal(robot)$ ▷ Send goals to appropriate action server
22: **end if**
23: **end for**
24: WaitForResults()
25: **end while**
26: **end procedure**
27: **function** WAITFORRESULTS( )
28: $robot, transition \leftarrow ReceiveResult()$ ▷ Blocking function that waits for any result message
29: $RobotFlags(robot) \leftarrow WAIT$
30: $InputPlaces \leftarrow G.input_places(transition)$
31: $ready, robots \leftarrow CheckRobots(InputPlaces)$ ▷ All robots have flag "WAIT"
32: **if** $ready == True$ **then**
33: $RobotFlags(robots) = FINISHED$
34: $G.fire(transition)$
35: $Update(RobotPlaces)$
36: **end if**
37: **end function**

Algorithm 6: Algorithm to execute GSPNR task plans

5.4.2 Using the API

The *Execution Manager* module integrates with ROS as it provides several advantages when building robotic systems. In addition to being open-source, ROS has a large active community of user-contributed

packages. Furthermore, it is built in a modular manner, that allows users to pick-and-choose available components to build their own robotic system. Our software package uses an additional MATLAB toolbox [24] to be able to interact with ROS from within the MATLAB environment and using MATLAB code.

The ROS *actionlib* package [25] provides a protocol to build and run ROS nodes that can receive and execute long-running goals. These nodes are called *ActionServers* and they send a result message over ROS network when the action has finished executing. These are particularly suited to use our toolbox, because each *action* place has a direct correspondence to executing a particular action.

Each ROS node must have an unique name by which it identifies itself over the ROS network. However, in multi-robot systems, each robot must have its own version of the same *ActionServer* so that all robots can execute each action simultaneously. ROS 2 is geared towards working with multi-robot applications, and it natively solves this issue of duplicate nodes for different robots. However, when we first started developing the toolbox, ROS 2 was still in an initial development phase and it had many bugs and faults. Beyond that, most user packages were not yet migrated to this new ROS 2 version. Consequently, we chose to integrate with classic ROS 1, allowing our toolbox to have a more immediate impact and giving us more opportunities to use off-the-shelf packages for testing and drawing results for our toolbox.

To bypass this problem in ROS 1 we made use of the *namespaces* feature. Each robot uses an unique *namespace*. Each robot's *ActionServers* live within the robot's *namespace* and so each *ActionServer* has an unique name. By associating each token with a specific *namespace* and associating action places with goals for *ActionServers*, the execution module sends the specified goal to the correct robot's *ActionServer*, and each result message received is processed so that the correct exponential transition is fired when the *ActionServers* return their result message.

The execution module implements a class `ExecutableGSPNR` that inherits from the base class `GSPNR`. This class has a method that initializes an empty executable instance with all the information needed to execute a previously created `GSPNR` model: `.initialize(GSPNR, YAML_filepath, action_map)`. As input it takes at most two variables: the non-executable model `GSPNR`, and then either the path to a YAML file, `YAML_filepath`, or a structure array `action_map`.

```
executable_solarfarm = ExecutableGSPNR()
executable_solarfarm.initialize(solarfarm, YAML_filepath, [])
```

The purpose of these two arguments is to map action places to their corresponding action servers, and also specify the goal message to be sent through the action protocol. Additionally, the name of the ROS package where the actions are defined and the proper action names must also be specified for each action place. Figures 5.16 and 5.17 show how to build these inputs.

Robots must be added to the executable GSPNR with their ROS namespaces and the place in which they start out:

```
executable_solarfarm.add_robots(['jackal0', 'jackal1'],...
                                ['Panel1' , 'Panel2'])
```

To set a policy in an executable GSPNR, the method `.set_policy()` must be used:

```yaml
...
  Inspecting_panel1:
    -action_server_name: "InspectWaypointActionServer"
    -package_name: "multi_jackal_tutorials"
    -action_name: "InspectWaypointAction"
    -message_fields: {"waypoint":"'panel1'"}
  Inspecting_panel2:
    -action_server_name: "InspectWaypointActionServer"
    -package_name: "multi_jackal_tutorials"
    -action_name: "InspectWaypointAction"
    -message_fields: {"waypoint":"'panel2'"}
  Travelling_Panel1to2:
    -action_server_name: "NavigateWaypointActionServer"
    -package_name: "multi_jackal_tutorials"
    -action_name: "NavigateWaypointAction"
    -message_fields: {"destination":"'panel2'","origin":"'panel1'"}
  TravellingPanel2to1:
    -action_server_name: "NavigateWaypointActionServer"
    -package_name: "multi_jackal_tutorials"
    -action_name: "NavigateWaypointAction"
    -message_fields: {"destination":"'panel1'","origin":"'panel2'"}
...
```

Figure 5.16: Example YAML file that specifies input `YAML_filepath` for GSPNR model in figure 5.13, outlined in red are the action places;

```
action_map.Inspecting_panel1.
    server_name =
    'InspectWaypointActionServer';
action_map.Inspecting_panel1.
    package_name =
    'multi_jackal_tutorials';
action_map.Inspecting_panel1.
    action_name =
    'InspectWaypointAction;
action_map.Inspecting_panel1.
    message_fields =
    {"waypoint":"'panel1'"}
```

Figure 5.17: Code example that defines input `action_map` for a single place of GSPNR model in Figure 5.13, "Inspecting_panel1";

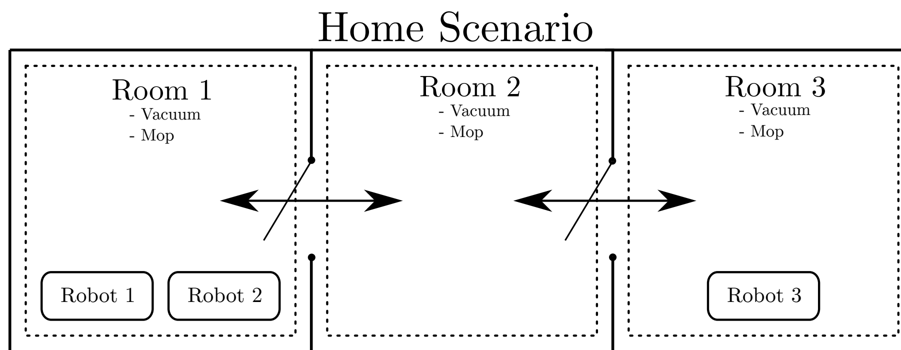


Figure 5.18: Example of domestic multi-robot scenario, with three separate locations to clean and three robots;

```
executable_solarfarm.set_policy(policy_struct)
```

To start executing the model, the method `.start_execution()` is used:

```
results = executable_solarfarm.start_execution()
```

Execution stops when either a terminal marking is reached (a marking with no enabled transitions) or the user interrupts the process by pressing the key "Control-C". The `results` output is of the same format as the one defined in the policy evaluation function (a structure array containing markings, transitions fired, and corresponding timestamps).

5.5 Computational Performance and Scalability

To test the computational performance of the toolbox and the scalability of each module, we devised a virtual multi-robot scenario where multiple robots operate in a indoors environment inside a house. The robots can travel between each subdivision of the house, and each of these rooms must be vacuumed and mopped by the robots. Figure 5.18 shows an example of such a scenario, where three robots must vacuum and mop three distinct divisions of a house.

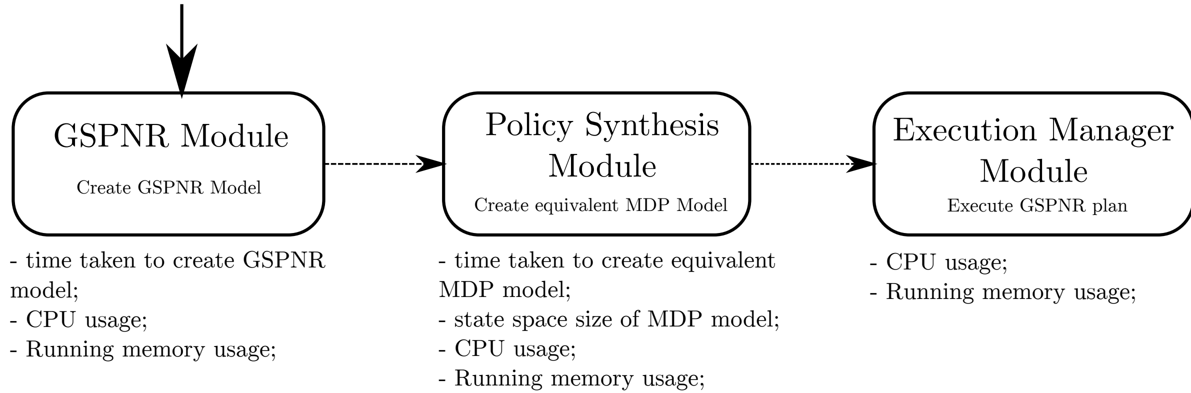


Figure 5.19: Summary of measurements taken for each experiment;

We ran two different experiments. The first experiment evaluated how the toolbox's performance scaled with modelling multi-robot problems where a fixed amount of robots could execute an increasing amount of actions. We did this by evaluating the performance of each module for a domestic scenario with an increasing number of rooms within the house. Adding rooms equates to allowing the robots to vacuum and mop more locations, and also adding possible navigation actions. The second experiment evaluated how the toolbox's performance scaled with modelling multi-robot problems with an increasing number of robots, for a fixed environment with the same number of locations.

Figure 5.19 summarizes the tests done for each multi-robot scenario evaluated. For the *GSPNR* module, we measured how much time was taken to build the GSPNR model of the multi-robot problem, and the CPU and memory used. To characterize the *Policy Synthesis* module, we measured how much time time it took to build the equivalent MDP model, the number of states in this MDP model, and the CPU and memory used. Finally, the *Execution Manager* module was tested by executing a GSPNR plan for a duration of 5 minutes. We simulated the robots by creating mock-up *ActionServers* for the vacuuming and mopping action. Each time one of these *ActionServers* received a goal, it would block for a certain amount of time and then return the result message as if the robots succeeded carrying out the action.

All measures of CPU and memory usage were taken using an external Python script. This script measured the CPU and memory of the entire MATLAB process, which can have considerable overhead. CPU measurements were taken in relation to usage of a single core. Thus, 100% usage corresponds to fully utilizing one CPU core, 200% corresponds to fully utilizing two CPU cores, etc. On the other hand, all time measurements were done internally in MATLAB. The computer where all the tests were run has the following specifications: an i74690 CPU, 32GB RAM and a NVIDIA Geforce RTX 2080 Super graphics card.

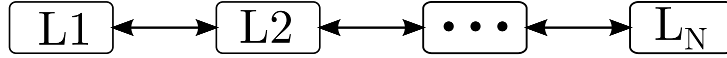
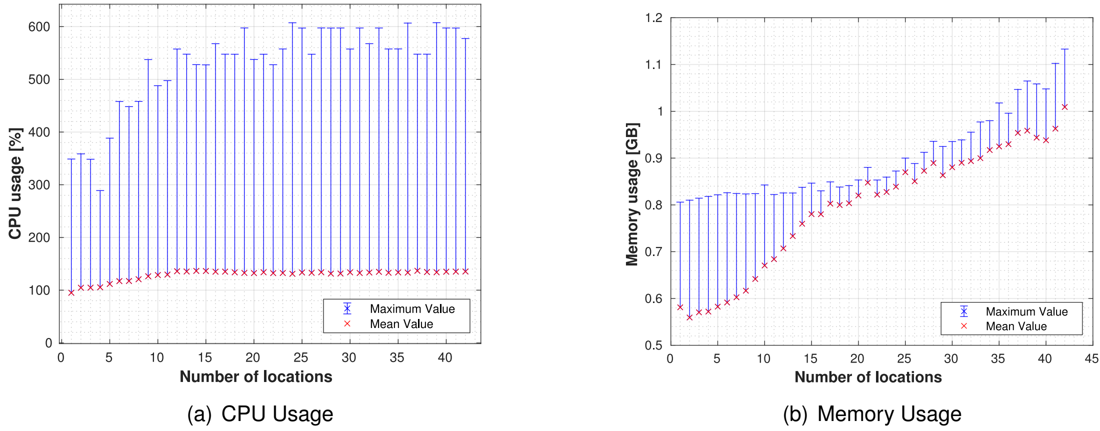


Figure 5.20: Topological map of the domestic scenario with N locations



(a) CPU Usage

(b) Memory Usage

Figure 5.21: Computational performance when creating GSPNR model with an increasing number of locations and building equivalent MDP

5.5.1 Increasing number of locations

To model multi-robot scenarios with an increasing amount of locations we used the function described in Section 5.2.3 that builds a GSPNR model from a topological map. It is particularly suited for this type of multi-robot problem, as the robots can execute the same vacuuming and mopping action in each possible room of the home. As more locations were added to the scenario, nodes were added to the topological map used. The locations were arranged in a linear manner: when in location L_i, the robot can choose to mop, vacuum, or travel to locations L_{i-1} or L_{i+1}. For every test, the number of robots was fixed at two: each one starting out in the edge locations L_1 and L_N.

We ran this experiment until we reached a multi-robot scenario with 42 different locations. We found no significant limitations when using the *GSPNR* module. As can be seen in Figure 5.21 the average running memory used increases in a linear trend as the number of locations was increased. Meanwhile, average CPU usage slightly increases until the scenario with 12 locations is tested. CPU usage then remains constant at around 110% of a single core. Each point was obtained by averaging the results of 10 runs. The time taken to create these GSPNR models can be seen in Figure 5.22. The time taken to create the GSPNR model increases as the number of locations in the multi-robot scenario grow. As the number of locations is increased, the GSPNR model grows larger with more places and transitions.

The toolbox's main bottleneck is in its *Policy Synthesis* module. Figure 5.24 shows the evolution of the number of states in the MDP model as the number of locations increases. For a conservative Petri net with p places and n tokens, the maximum number of reachable markings is "p multichoose n" (dashed black line). We verified that for the equivalent MDP model without wait states the number of states (in blue) follows this expression. When considering wait states, the number of states is even bigger (in red), as every hybrid marking is unfolded into two states. This is still considerably better than the case where each individual robot is discriminated. If every place represents a local state of each

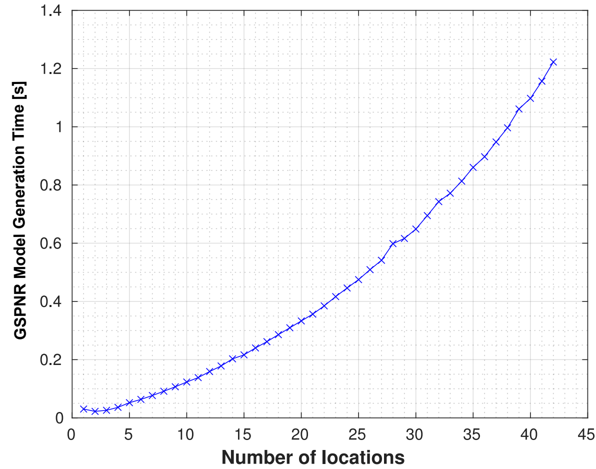


Figure 5.22: Time taken to create GSPNR model of a scenario with an increasing number of locations, using a topological map;

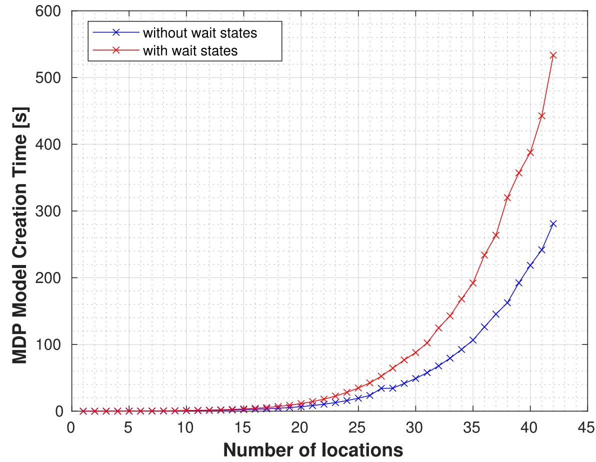


Figure 5.23: Time taken to create equivalent MDP model for a multi-robot scenario with increasing amount of locations, MDP with wait states (in red) and without (in blue);

individual robot, and the team consists of two robots, in this case, the number of states would be p^2 (line in black). As seen in Figure 5.23 as the number of states in the MDP model becomes larger, so does the time needed to create the model.

To characterize the *Execution Manager* module, we executed a GSPNR plan for each multi-robot scenario with a different number of locations. The plan was defined with a random policy, and the robots were simulated with mock-up action servers. Each plan was executed for a duration of 5 minutes, and each data point was an average of 10 runs. The CPU and memory usage of the MATLAB process running the execution function can be seen in Figure 5.25. The average of both measures remain constant as the locations increase, but the maximum value for CPU usage was significantly higher than the average value. This may happen because measurements begin to be taken as soon as the MATLAB process starts up, which consumes a great deal of CPU at start-up.

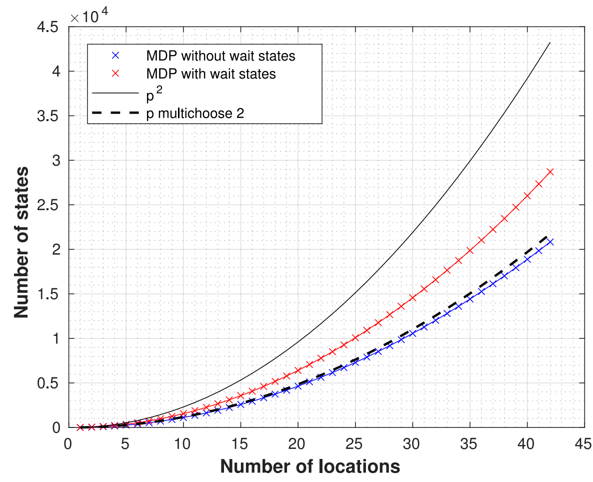
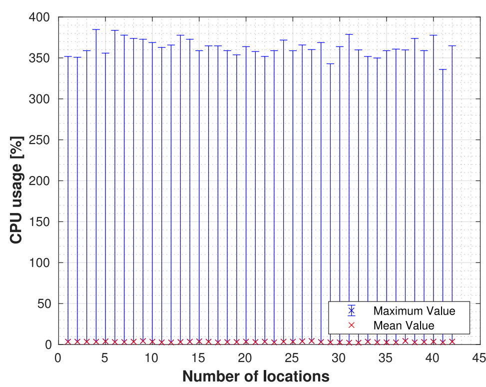
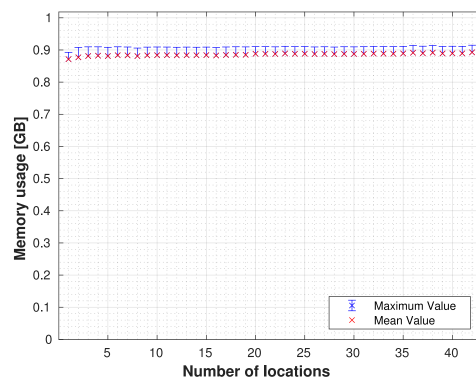


Figure 5.24: For an increasing amount of locations, the number of states in the equivalent MDP without wait states (blue), with wait states (red), and a comparison with the number of places squared (in black);



(a) Mean and maximum CPU usage



(b) Mean and maximum memory usage

Figure 5.25: Computational performance when executing GSPNR model with an increasing number of locations

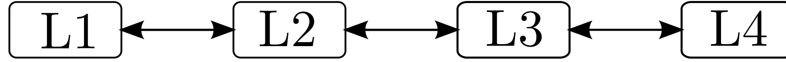
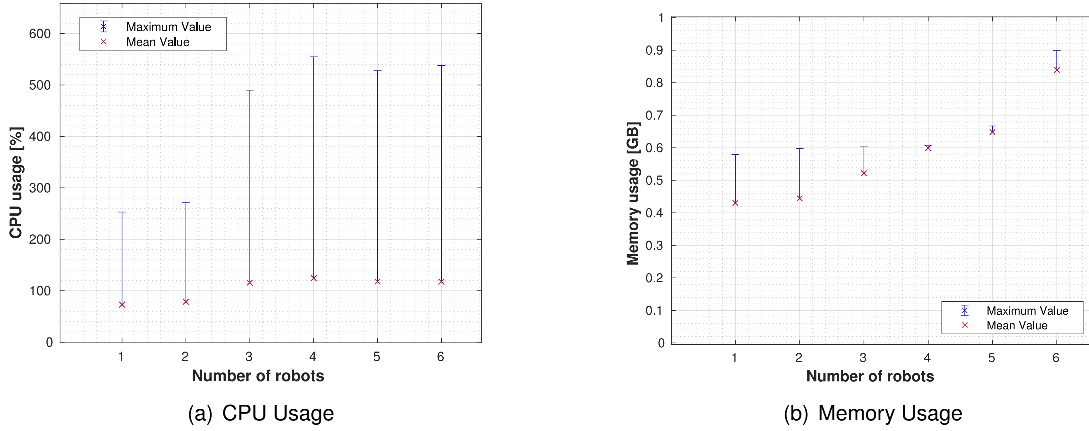


Figure 5.26: Topological map of the domestic scenario with 4 locations



(a) CPU Usage

(b) Memory Usage

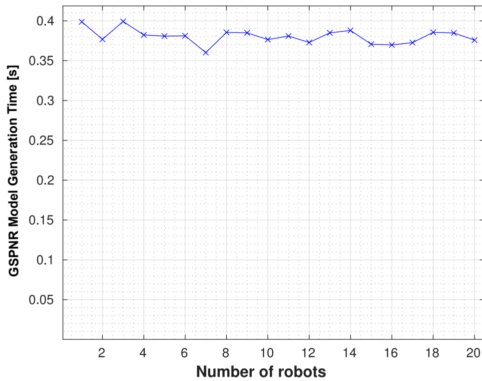
Figure 5.27: Computational performance when creating GSPNR models up to 6 robots;

5.5.2 Increasing number of robots

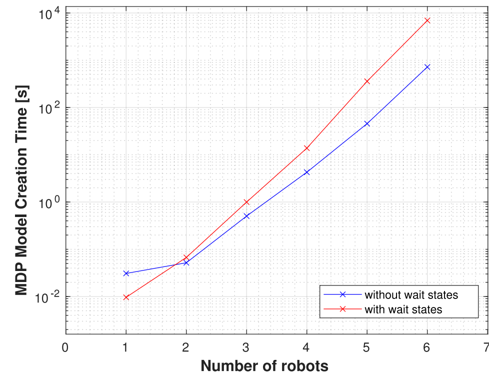
For the experiment that increases the number of robots, the number of locations was fixed at four locations, arranged according to the topological map in Figure 5.26. Just as done previously, the robots can mop and vacuum each location, or decide to travel to a neighbouring one. We again used the function that uses a topological to create the GSPNR model of each multi-robot scenario. In this experiment, each test modelled the same environment with an increasing number of robots. This translates into adding a token to the GSPNR model for each robot modelled. These tokens were divided equally and placed into $L1$ and $L4$. This corresponds each half of the robot team starting at the outermost locations.

The CPU and memory usage for creating the GSPNR model and converting it to an MDP can be seen in Figure 5.27. This mirrors Figure 5.21, with the CPU usage plateauing at 110%, while memory usage increases linearly. This can be verified in Figure 5.28. The time to create the GSPNR stays constant as the number of robots increase, while the time to create the equivalent MDP grows almost exponentially, and the experiment was stopped after creating the MDP for a team with 6 robots.

Figure 5.29 shows how the state space of the equivalent MDP grows as the number of robots modelled increase. This figure explains why the time taken to create the equivalent MDP model grows almost exponentially. As more robots are modelled, the corresponding state space also increases in an almost exponential manner. However, there is still a substantial advantage in modelling a multi-robot system with a GSPNR where robots are represented anonymously. Suppose we have a system with r robots, and each robot can occupy P different states. If the team's state is the concatenation of each robot's state $s_{team} = (p_1, p_2, ..., p_r)$, with $p_i \in P$, the maximum state space size is P^r. The GSPNR model of the exact same multi-robot scenario consists of P places, each representing a robot's local state. The maximum number of reachable markings in such a GSPNR is "P multichoose r". This is confirmed by verifying the number of states of the equivalent MDP without wait states in Figure 5.29 (in blue). Even when including wait states, the GSPNR model still has a significantly smaller state space.



(a) GSPNR creation



(b) MDP Creation

Figure 5.28: Time taken to create GSPNR with 1-20 robots (left) and time taken to create equivalent MDP model, up to 6 robots (right)

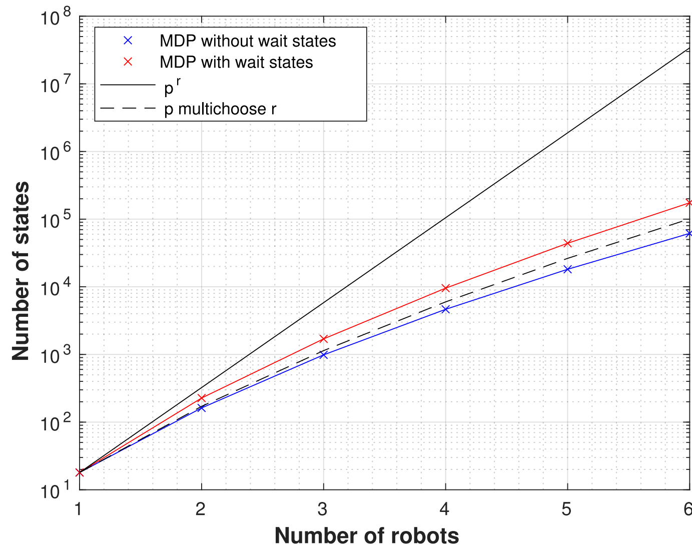
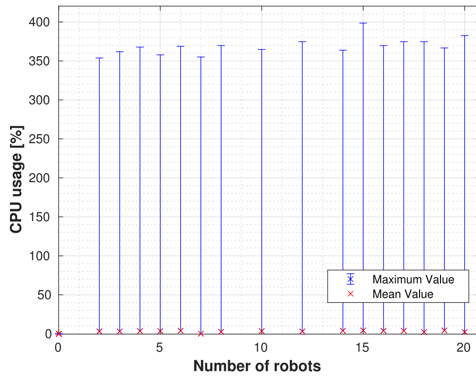
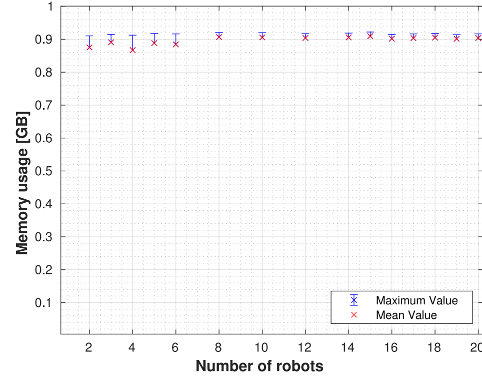


Figure 5.29: For an increasing amount of robots, the number of states in the equivalent MDP without wait states (blue), with wait states (red), and a comparison with the number of places squared by the number of robots (in black)



(a) Mean and maximum CPU usage



(b) Mean and maximum memory usage

Figure 5.30: Computational performance when executing GSPNR model with an increasing number of robots

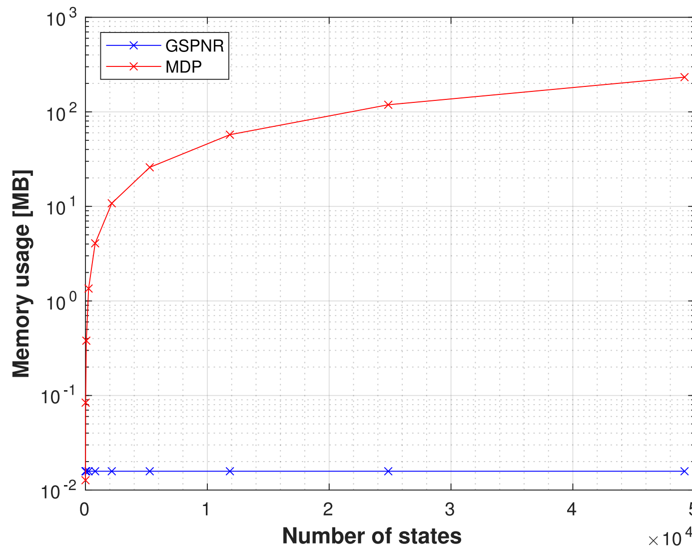


Figure 5.31: Size in megabytes of the `GSPNR` object (in blue) and the equivalent `MDP` object (in red) for an increasing amount of reachable states/markings;

To test the *Execution Manager* module, a GSPNR plan was executed for each of the multi-robot scenarios for a duration of 5 minutes. The plan was executed with a random policy, and the experiment showed that computational requirements stay constant as the number of robots in the system increases. Figure 5.30 shows the maximum and mean values for CPU and memory usage for up to 20 robots. This showcases an important advantage in using a GSPNR plan to coordinate a multi-robot system: the computational requirements to execute a GSPNR plan do not increase as more robots are coordinated. Figure 5.31 further reinforces this point. For scenarios up to 10 robots, we measured the memory occupied by the MATLAB object that holds the GSPNR model and its equivalent MDP model without wait states. As the number of reachable states/markings increased, the GSPNR size stays constant, while the MDP model grows linearly. This is due to the fact that an MDP model has to contain all possible states, and all possible transitions for each state. A GSPNR model only holds the current state of the system, how the system evolves between states is encoded in its place/transitions/arcs structure.

Chapter 6

Experiments and Results

Chapter 6 focuses on applying the work done in chapter 4 and 5 to solve two multi-robot problems. The first problem involves a domestic scenario with an homogeneous team, where two robots must vacuum different locations inside a house, whilst minimizing the time spent charging at a fixed base station. The second problem is an inspection scenario where two small robots must travel and inspect several photo-voltaic panels. This team was extended with a larger robot that can charge the smaller robots that execute the inspection. We built two Gazebo worlds, one for each multi-robot problem. We then used the Gazebo worlds to simulate the multi-robot problem and execute the GSPNR task plans that we obtained. For each problem, the GSPNRs that model the system are presented, broken down into the specific models for each robot type, and for each particular action that the robots can execute. The simulated environment in which the robots operate in is described, along with the robots used and their characteristics. Finally, results from simulation are discussed.

6.1 Home Vacuuming Scenario

6.1.1 Problem Description

This scenario was developed for RoboCup 2021, and presented at the Open Challenge of RoboCup@-Home. The problem involves coordinating a multi-robot homogeneous team in order to vacuum all rooms of a small house as efficiently as possible. The house must be vacuumed in an even manner, meaning that the robots have to vacuum all other rooms at least once before being able to vacuum the same location again. The robots have limited autonomy due to their battery, and every so often they must return to a base station to recharge their depleted battery.

The Gazebo world we developed that simulates the home vacuuming scenario can be seen in Figure 6.1. This figure shows the topological map used overlapped on the Gazebo world, and outlined in red are the areas vacuumed in each discrete location. A clip of this scenario can be seen in [26]. The robots used were two identical *Turtlebot3 WafflePi* with horizontally-mounted laser range finders and can be seen in Figure 6.2. We simulated the turtlebots using the ROS package provided by the company that commercializes these robots. Robot localization was implemented using the *AMCL* ROS package with a

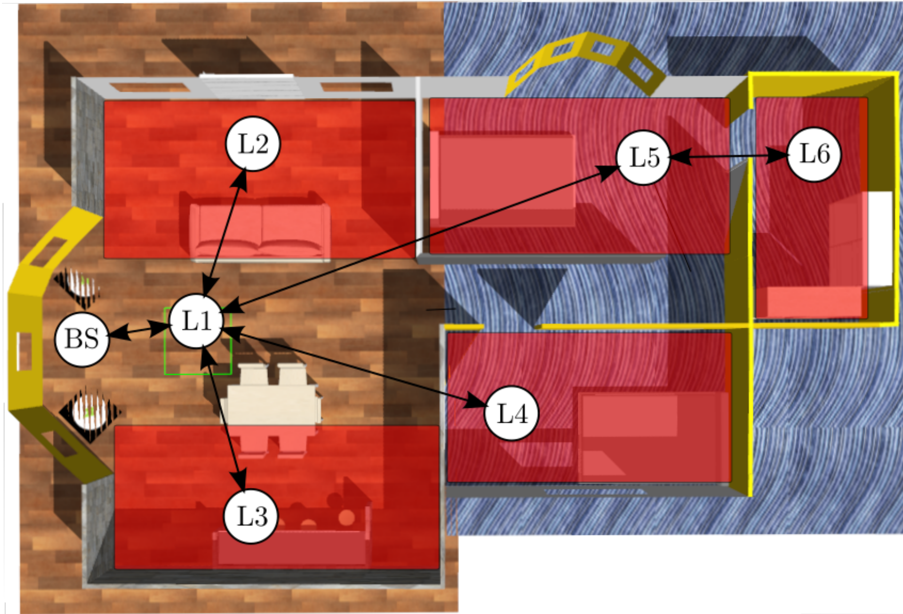


Figure 6.1: Gazebo world used, with the topological map superimposed, and the corresponding vacuumed areas for each location in red;

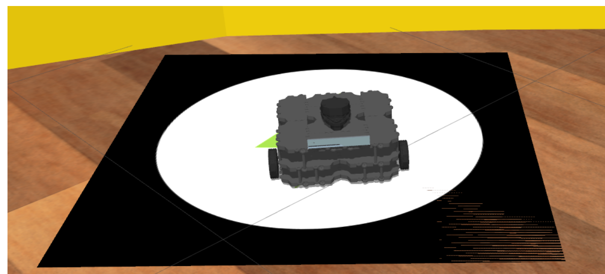


Figure 6.2: Turtlebot robot charging at location ”BS”;

previously obtained occupancy grid map (built with package *gmapping*). Navigation is done by integrating with the *move-base* package, that includes a local and global planner. Additionally, laser-scan data was included in an "Obstacles layer" in the move-base component so that the two robots can avoid colliding with each other. These localization algorithms were used off-the-shelf without any customization from our part. We limited ourselves to the parameter tuning usually done in any application.

In regards to simulating each robot's battery, the ROS package found in [27] was used. This package provides a node that can simulate the state of a battery continuously, as well as providing a topic to control if the battery is currently charging or discharging. By expanding the package to the possibility of using namespaces, it was adapted so that both robots could have their own battery. Additionally, some further work was done to build a ROS service that returns the current battery percentage remaining.

To allow integration with the MR-GSPNR toolbox, action servers were implemented for all possible tasks that the robots could execute:

1. **ChargeActionServer** - communicates with the battery mockup node so that the robot's simulated battery starts charging, and finishes charging when its current percentage is above 99%;

2. **CheckBatteryActionServer** - utilizes the ROS service developed and returns the current discrete battery level: either {low, medium, high}. The cutoffs for each discrete level were set at 33% and 66% respectively;

3. **HomeNavigateActionServer** - receives a waypoint in string format, and translates this into a *Pose()* goal by looking up the corresponding parameter in the ROS parameter server. It passes this goal on to the move-base node so that the robot moves to the desired waypoint;

4. **HomeVacuumActionServer** - also receives a waypoint that is to be vacuumed, and consecutively passes on goals to the move-base node so that the robot transverses the entire area belonging to the waypoint being vacuumed.

All action servers were implemented by utilizing custom ROS actions, defined in their corresponding action files: *Charge.action*, *CheckBattery.action*, *InspectWaypoint.action*, *NavigateWaypoint.action*.

6.1.2 GSPNR Model

To model this multi-robot problem with a GSPNR as set out in Chapter 4, the house is abstracted into discrete locations in the topological map, and each robot's battery is discretised into three levels: {low, medium, high}. The topological map representing the rooms to be vacuumed can be seen in Figure 6.3. The topological map consists of 7 nodes, where one is the base station where the cleaning robots can recharge, and the rest are rooms in the home. The rooms represented by locations {L2, L3, L4, L5, L6} need to be vacuumed. Location "L1" is an intermediate one that connects to the base station "BS" where robots can recharge. We built action models and attribute models for each possible action: navigation between locations, vacuuming a location, and the recharging action.

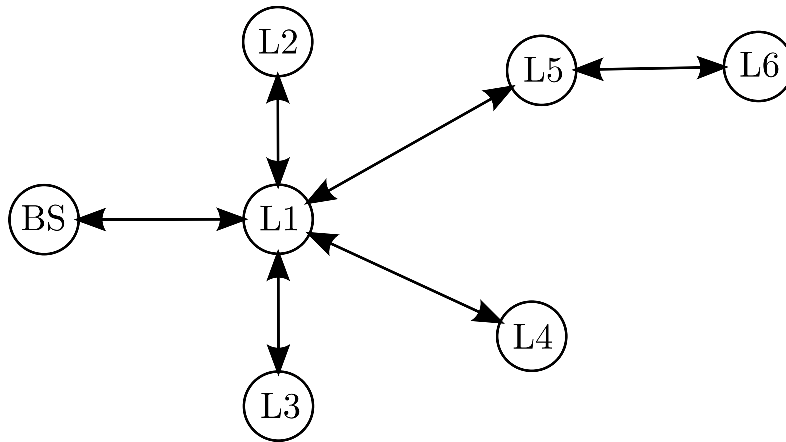


Figure 6.3: Topological map of the vacuuming scenario: 6 locations + a base station (BS) where robots can recharge;

Vacuum Action

The GSPNR model that represents the vacuuming action in location ”L2” is shown in Figure 6.4. Action and attribute models outlined in green are for robots with ”high” battery level, and outline in red are action and attribute models for ”medium” level.

When a robot is in location ”L2” a token is in either place ”L2_high”, ”L2_medium”, or ”L2_low” depending on the state of its battery. If the robot chooses to vacuum (which it cannot do so if it has low battery), the token goes through the action model corresponding to the battery level it started out with. After the robot has finished vacuuming, the token moves to an attribute model, where it has a chance to stay at the same battery level, or fall into a lower level. After vacuuming at level ”high” for example, either the transition ”Bat_Level_L2_high_high” can fire, representing that the robot stayed with ”high” battery. Otherwise, transition ””Bat_Level_L2_high_medium” fires, representing that the robot transitioned from a ”high” battery level to the ”medium” level. The same happens when the robot finishes vacuuming when having battery level ”medium”. The robot either stays at level ”medium” or transitions to level ”low”.

Each location n has a resource place associated named ”r.Requires_Vacuum_n”, which is marked if the location can be vacuumed. The transition that fires when a robot decides to vacuum consumes a token from this place, and so if there is no token in this resource place, the immediate transition is disabled and the robots can’t vacuum this particular location. After deciding to vacuum a certain location a token is created in another resource place which is common to all locations: ”r.Number_Vacuumed”, and this serves as a counter of how many vacuum actions have been done in all locations. To reset this counter, an exponential transition ”VacuumedAll” is connected to this place by an input arc with a multiplicity of five (there are five vacuumable locations). This exponential transition creates a single token in every ”r.Requires_Vacuum_n”, so that all locations are vacuumed once again. This restricts robots to vacuuming all locations evenly, and the counter connected to all vacuum transitions can be seen in Figure 6.5.

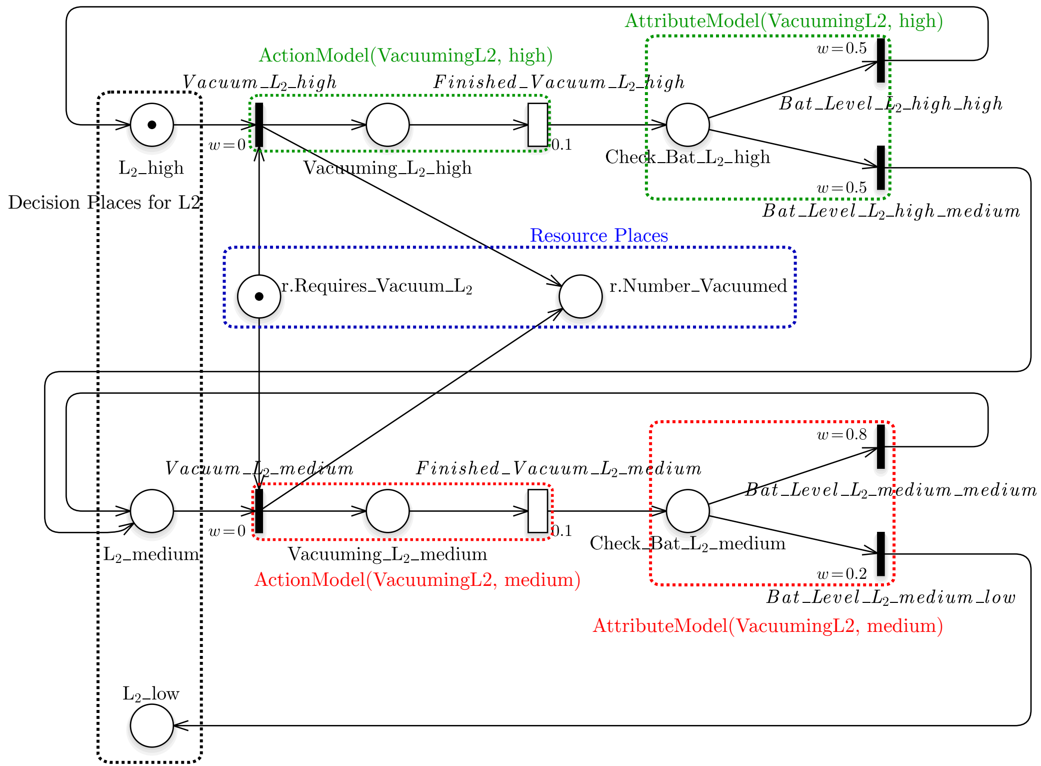


Figure 6.4: GSPNR Model for Vacuuming at *L2*;

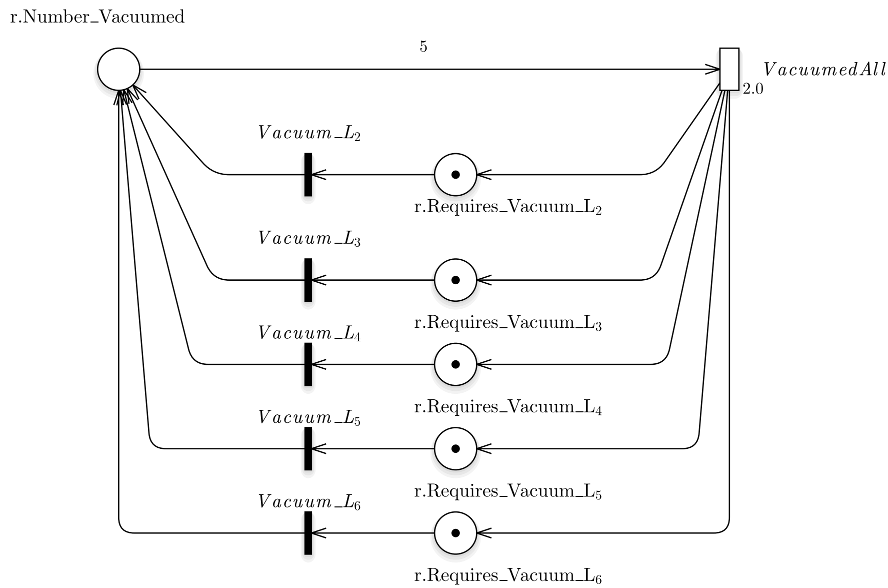


Figure 6.5: Global counter that restricts vacuums;

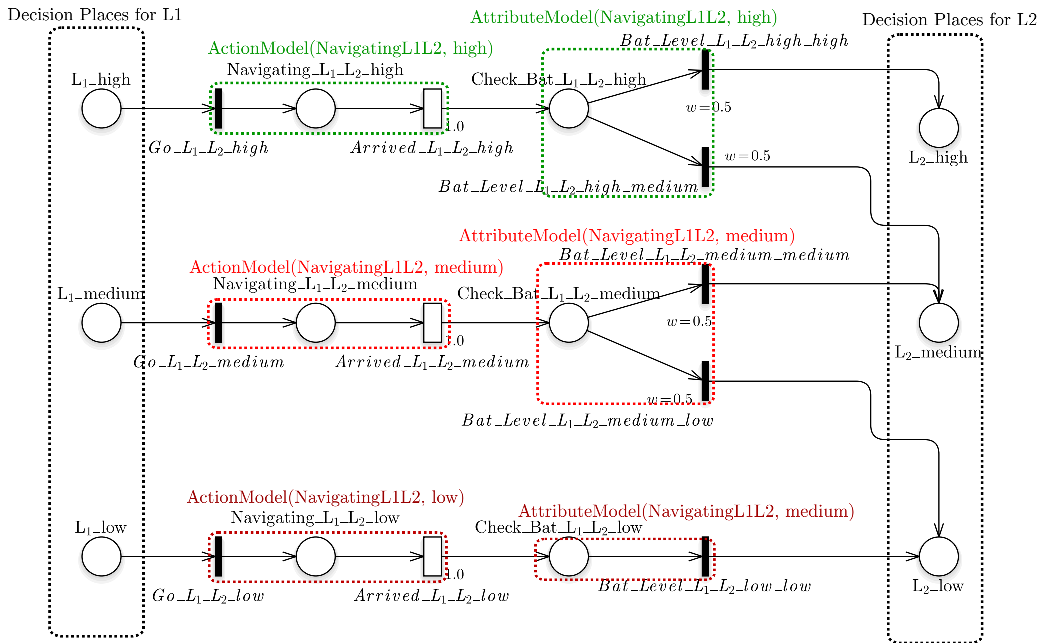


Figure 6.6: GSPNR Model for Navigating from *L1* to *L2*;

Navigation Action

An example of a navigation model can be seen in Figure 6.6. It depicts the model for navigating between locations "L1" and "L2", but the model for every navigation action is the same for all edges in the topological map. It is similar to the vacuum model, except that robots are allowed to navigate when their battery is low, and so the corresponding action model and attributed model is also added (in dark red). A robot at "L2" has a battery in either "high", "medium", or "low" state, and a token representing such a robot is in either place "L2_high", "L2_medium" or "L2_low", respectively. For every battery state, the robot can decide to navigate to "L1" and so there is an action model for each battery level representing this navigation action. After navigating to location "L1", the robot's battery might have changed state, and so an attribute model is appended to each action model. In every attribute model, the robot's battery can stay at the same level, or fall into the battery level below.

Charging Action

The part of the model representing the charging action at the base station can be seen in Figure 6.7. It is different from the previous models in two ways: because robots cannot charge while at the highest level, no action model is included for recharging at that level. The model also assumes that there is no uncertainty when charging. This means that robots that charge will transition to level *high* with 1.0 of probability. Thus the model block that usually models uncertainty regarding battery level (the attribute model) is omitted. As a result, the exponential transitions that model the end of the recharging connect directly to the decision place corresponding to the robots having *high* battery at *BS*.

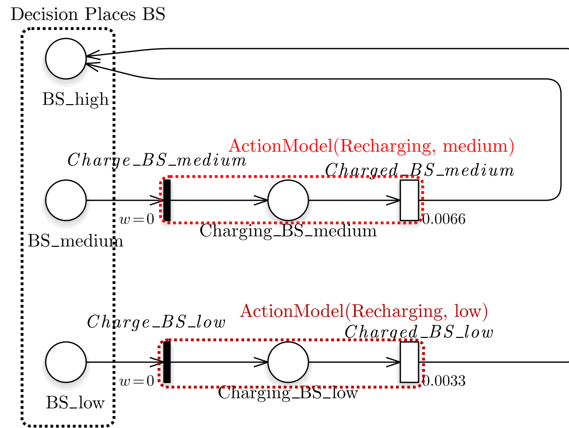


Figure 6.7: GSPNR Model for Charging at *BS*;

Overall Model and Goal Specification

The full GSPNR model of the system was designed using the procedure outlined in Section 4.3.2, using the action models and attribute models outlined previously for every action possible. Vacuuming GSPNRs were added to all nodes in the topological map, navigation GSPNRs were added for all edges in the topological map, and the recharging GSPNR was added to the "BS" location. The goal of the multi-robot team was to vacuum all locations as quickly as possible and so we added a transition reward of $+1$ to every transition that represents the decision to vacuum a location. We wanted to discourage the robots having a battery that was close to depleted, and so added place reward of -1 to every place that corresponds to robots having "low" battery level.

This was done in a semi-manual process, using the topological map and customizable action models which were instantiated for each location and merged into the overall model. The vacuum counter was added after, and so were the place and transitions rewards.

6.1.3 Results

The GSPNR model parameters can be divided into two categories: 1) exponential transition weights that model mean action duration; and 2) immediate transition weights of the attribute models that model the battery discharge. We carried out model identification to estimate these parameters by running each vacuum, navigation, and charging action repeatedly, and calculating the mean duration time of these actions. To fit this data to the distribution of the exponential transitions we used its maximum likelihood estimator: $\hat{\lambda} = \frac{n}{\sum_{i=1}^{n} x_i}$, where n is the number of samples taken $X = (x_1, \cdots, x_n)$. The discharging probabilities were also estimated for each specific action. This was done by comparing how many times each discharging transition was fired against the total number of times the action was executed. An optimal policy for this model was synthesized by optimizing the *discounted expected reward criterion*, in the equivalent MDP without wait states. The discount factor used was $\gamma = 0.99$ and the convergence criterion used was $\epsilon = 0.01$. For a system with two robots, the equivalent MDP had $139,180$ unique states. The GSPNR plan with this optimal policy was executed for 13 hours in Gazebo, and the average

Table 6.1: Rough approximation for discharging probabilities for *Navigate* and *Vacuum* actions for all locations and navigation edges used to obtain the policy with less rigorous parameters;

$Navigate$	*low*	*medium*	*high*
low	1.0	0.0	0.0
medium	0.2	0.8	0.0
high	0.0	0.5	0.5

$Vacuum$	*low*	*medium*	*high*
low	-	-	-
medium	0.2	0.8	0.0
high	0.0	0.5	0.5

	Execution in Gazebo of GSPNR plan	
	Policy obtained with **accurate** model parameters	Policy obtained with **estimated** model parameters
Average $\Delta_{Charges}$ [s]	178.62	186.34
Average $\Delta_{VacuumedAll}$ [s]	656.77	713.12
$\bar{r}$ [1/s]	-0.2384	-0.2659
Vacuums per hour	5.46	5.00

Table 6.2: Comparison between average time between charges $\Delta_{Charges}$, average time taken to vacuum the entire house $\Delta_{VacuumedAll}$, average reward $\bar{r}$, and vacuums per hour, for both policies executed in Gazebo

reward accumulated with this policy was -0.2384 per second of execution time. On average it took the two robots 656.77 seconds to vacuum the entire home, with a standard deviation of 121.54 seconds. The time between the robots charging was on average 178.62 seconds.

We also executed a GSPNR plan with a policy obtained from a model with less accurate parameters. Navigation and vacuum mean times were recorded by running the action server a single time for each navigation edge and vacuum location and recording the time taken. Battery discharge probabilities were assumed to follow the model found in Table 6.1. This policy was executed in the Gazebo for close to 17 hours, and the average reward accumulated was -0.2659 per second of execution time. On average it took the two robots 713.12 seconds to vacuum the entire home, with a standard deviation of 170.02 seconds. The time between the robots charging was on average 186.34 seconds. These results are seen in Table 6.2, compared to the results obtained with the accurate model. The policy with more accurate model parameters led to a lower average time to vacuum the entire house, and a higher average reward. In proportion to the running time, it was able to do more vacuum rounds than the policy with less accurate model parameters.

Beyond executing the two optimal policies in Gazebo, we tried to assess the differences in action duration uncertainty in the GSPNR model and the Gazebo simulation. This was done by using the function described in Section 5.3.3 to simulate the policy obtained with accurate model parameters. This is what we refer to as *model simulation* - we use the model as our world simulator and sample from it to obtain the sequence of states that the system progresses through. The policy evaluation method simulated the GSPNR model for about the equivalent of 16 hours of real time. This allows us to compare the uncertainty as it is represented in the best GSPNR model we have, with the uncertainty that was present in the Gazebo simulation. The comparison of the sample mean $\bar{x}$ and sample standard deviation s can be seen in Table 6.3, for three main quantities: 1) time taken to charge when the robot is at *low*

GSPNR plan of optimal policy obtained with accurate model parameters						
	Charge				Vacuum All Locations	
	low		*medium*			
	Model	Gazebo	Model	Gazebo	Model	Gazebo
$\bar{x}$	178.45	122.13	70.22	74.16	686.50	656.77
s	153.98	26.07	71.27	4.04	263.25	121.54

Table 6.3: Comparison between simulating GSPNR plan without robots and executing it with simulated robots in Gazebo for "Charge" action (at *low* and *medium* battery level) and time taken to vacuum all locations; These results were obtained by executing GSPNR plan obtained with model with most accurate parameters;

battery; 2) time taken to charge when the robot is at *medium* level; 3) time taken to vacuum all locations. These results show that the model can adequately capture the average time taken to execute an action, but the large differences in the standard deviation of the samples suggest that dispersion in the model is substantially higher than in the actual simulated system. This is expected when representing uncertainty with an exponential distribution: the standard deviation of this distribution is equal to its mean $\frac{1}{\lambda}$. This is confirmed by the results as the sample mean and standard deviation of the model are close in value for the charge actions. This occurs because the time taken to execute these is characterized by a single exponential distribution. The same does not happen when measuring the time taken to vacuum in the entire home: this "macro" action is a composition of several individual actions (vacuuming each location and traveling between locations) and so the overall probability distribution is not exponentially distributed (if random variables $X \sim \exp(\lambda_1)$ and $Y \sim \exp(\lambda_2)$ the sum $Z = X + Y$ is not exponentially distributed).

Finally, all these results prove that the toolbox we developed is useful when solving this type of multi-robot task planning problems. The execution algorithm allows the constant coordination of the robot team for long-running times, according to an obtained optimal policy. Our toolbox allows a faster implementation of this robot coordination, as users do not have to manually create a policy, which is usually a cumbersome and slow process that does not offer formal guarantees on team behaviour.

6.2 Solarfarm Inspection Scenario

6.2.1 Problem Description

The second scenario that was solved involved a set of 4 solar panels that need to be continuously inspected by two small, mobile robots with a limited battery life. In contrast with the home vacuuming scenario, in this application, there is no fixed location where the robots may recharge their battery, and so the team is augmented with a larger mobile robot that can recharge the inspecting agents. We assumed that this larger UGV has infinite battery-life and it can indefinitely recharge the smaller UGVs as many times as needed.

The Gazebo world simulating this scenario is depicted in Figure 6.8. It is based on the inspection world provided by Clearpath found in [28], but some modifications were done: the bridge was widened so that the warthog could pass through, terrain was smoothed to minimize slipping and some elements

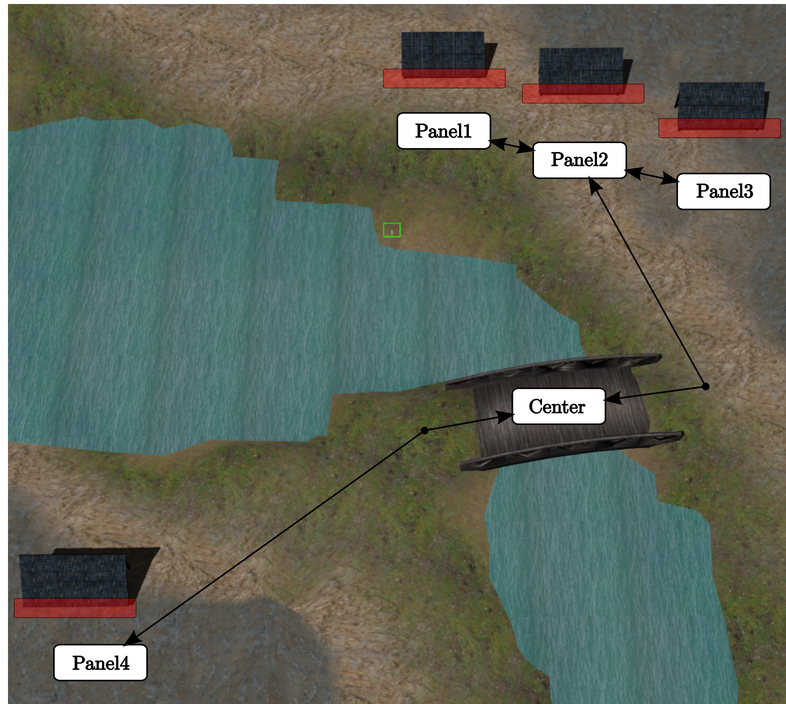


Figure 6.8: Gazebo world used in solarfarm simulation, with the topological map overlapped and the areas transversed by robots carrying out inspections outlined in red;

(such as the underground mine and the pipes) were removed. A clip of the robots in this scenario can be seen in [29].

Small UGVs

The two small UGVs were based on Clearpath's *Jackal UGVs*. They were used to inspect the solar panels, and the already existing *multi-jackal* ROS package was used. This package included a standard EKF algorithm that provides the localization using fused GPS, IMU and odometry sensor data, and a standard move base for navigation. However, the first experiments showed that due to wheel slip from the sloped terrain and the fact that *move-base* package is built for flat 2-D navigation, the robot's localization estimate progressively got worst as the simulation went on. To mitigate this, the native localization was turned off, and the ground truth data from the simulator was passed on to each jackal's move base so that they could localize themselves reliably.

Each robot had their own simulated battery in a similar manner as the turtlebots in the home vacuuming scenario. This battery is deterministic, we did not include any type of noise when measuring the current battery state of charge. This means that the time taken to discharge a battery with a full state of charge and the time taken to charge a depleted one is constant.

For obstacle avoidance, each jackal had a SICK LMS laser range finder, mounted above the jackal as seen in Figure 6.9. A square frame was added so that the robots could more easily detect and avoid each other and the back of the frame includes a charging plug that fits into a port on the warthog robot.

These robots inspect each solar panel by navigating to a set of three positions nearby to the solar panel. The robots pause in each position to orientate themselves towards the solar panel, and then

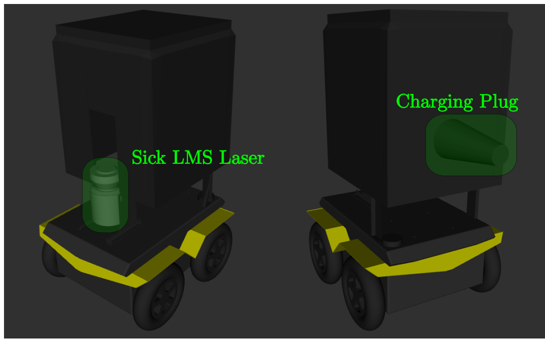


Figure 6.9: Clearpath Jackal with a standard LMS laser range finder, and a custom made frame with a charging plug;

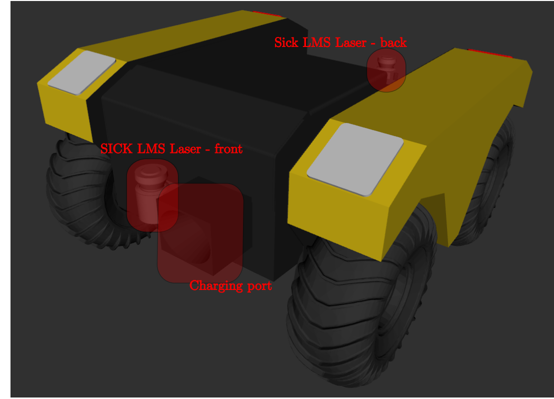


Figure 6.10: Clearpath Warthog with 2 LMS laser range finders, and custom made charging port;

Figure 6.11: Jackal coupled to the Warthog to simulate recharging;

travel to the next position. This is repeated until 90 seconds have passed, and then the inspection is finished.

Large UGVs

The single large UGV robot was based on Clearpath's warthog platform. The existing *warthog* packages provided by Clearpath [30] provide the necessary ROS interfaces to simulate a single warthog. These were adapted to use within a multi-robot problem, and a move base component was added to enable autonomous navigation. The robot's maximum speed was reduced to 1m/s to avoid wheel slip and allow tighter turns necessary to manoeuvre through the bridge. Even though the standard warthog includes GPS and IMU sensors, localization was done the same way as the jackals, by passing on ground truth data from Gazebo to the move base node. To allow for obstacle avoidance, two SICK LMS lasers were added to the robot, one attached to the front of the chassis, and the second one to the back. Additionally, the chassis itself was modified to include a charging port that the jackals can plug themselves into, as seen in Figure 6.16.

The action servers implemented to simulate each possible action that the jackals and warthogs can execute are the following:

1. **NavigateWaypointActionServer** - receives a waypoint in the topological map, and interfaces

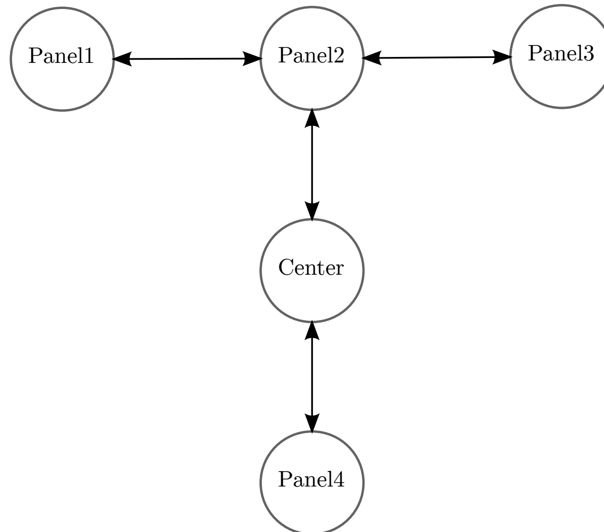


Figure 6.12: Solarfarm topological map;

with the robot's move base so that the robot navigates to the pose defined in the ROS parameter server matching the waypoint's name;

2. **InspectWaypointActionServer** - inspects the solar panel in the given waypoint, by cycling through a set of poses also defined in the ROS parameter server;

3. **JackalChargeActionServer** - navigates the jackal to a pose near to the warthog, where the charging plug and port line up, and then backups the jackal so that the its plug is inserted into the warthog's port. After this process has finished, the action server signals the simulated battery to start charging. When the battery percentage is above 99% the jackal disengages from the charging port and the action server returns;

4. **WarthogChargeActionServer** - waits for a jackal to finish charging and then returns;

5. **CheckBatteryActionServer** - returns current discrete level of battery, either "low" or "medium" and the cutoff between the two levels is at 5% battery percentage.

6.2.2 GSPNR Model

The entire environment is abstracted into a topological map with 5 locations, depicted in Figure 6.12. The topological map has 5 nodes, where four nodes represent the location of a solar panel, and an additional connecting node "Center". Because of the limitations in the size of the state space, the jackal's battery was discretized into only two levels: {low, medium}. When a jackal has "low" battery level, navigation and inspection are disallowed, and the warthog must travel to the jackal's location so they can cooperate and recharge the jackal's battery. We built action models and attribute models for each possible action: navigation between locations, inspecting a location, and the recharging action.

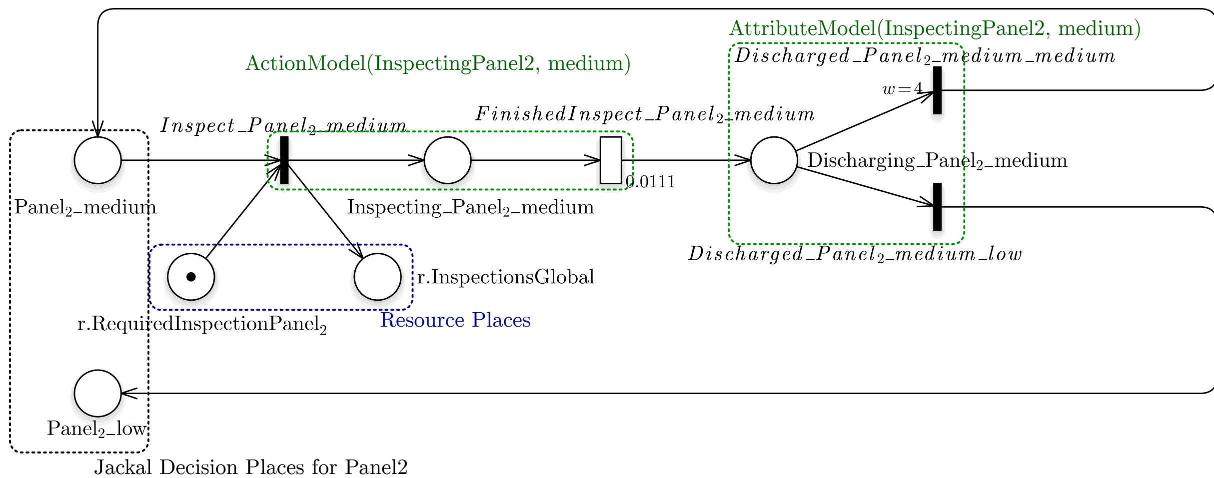


Figure 6.13: GSPNR Model for Inspecting "Panel2";

Inspection Action

The GSPNR model for the inspection action at "Panel2" can be seen in Figure 6.13. It includes the action model that models the inspection action which is connected to the attribute model that models the uncertainty in the battery state. After inspecting, the "Discharged_Panel2_medium_medium" transition can fire, representing that the robot still has battery. If the "Discharged_Panel2_medium_low" transition fires, the robot transitions to the "low" level battery and goes to the appropriate decision place representing this.

Also shown are two resources places that condition the decision to inspect this panel. The first place "r.RequiresInspectionPanel2" needs to be marked with a token for the robot to inspect the panel, and after deciding to inspect and firing transition "Inspect_Panel2_medium" consumes this token and creates one in place "r.InspectionsGlobal" (this place is common to inspection models in all locations). Only after inspecting all panels is another token created in "r.RequiredInspectionPanel2", and so inspections are carried out evenly between all panels. The mechanism that implements this is depicted in Figure 6.14, but connected to the inspection models of each panel. After all four panels have been inspected, there are four tokens in the place "r.InspectionsGlobal". The exponential transition "InspectedAll" becomes enabled, and after firing, creates a token in every "r.RequiredInspectionPanel" place, allowing the robots to begin another round of inspections.

Small Robot Navigation Action

An example of a GSPNR model for the jackals to navigate can be found in Figure 6.15. It contains the model for jackal navigation from "Panel2" to "Panel1", and because navigation is disallowed for when the battery is low, no action model or attribute model exists for battery level "low". Other than the rate of the exponential transition, equivalent action models are used to represent all navigation actions. After navigating and going through the action model, the token representing the robot reaches the place in the attribute model. With the token in this place "Discharging_Panel2_medium" either the

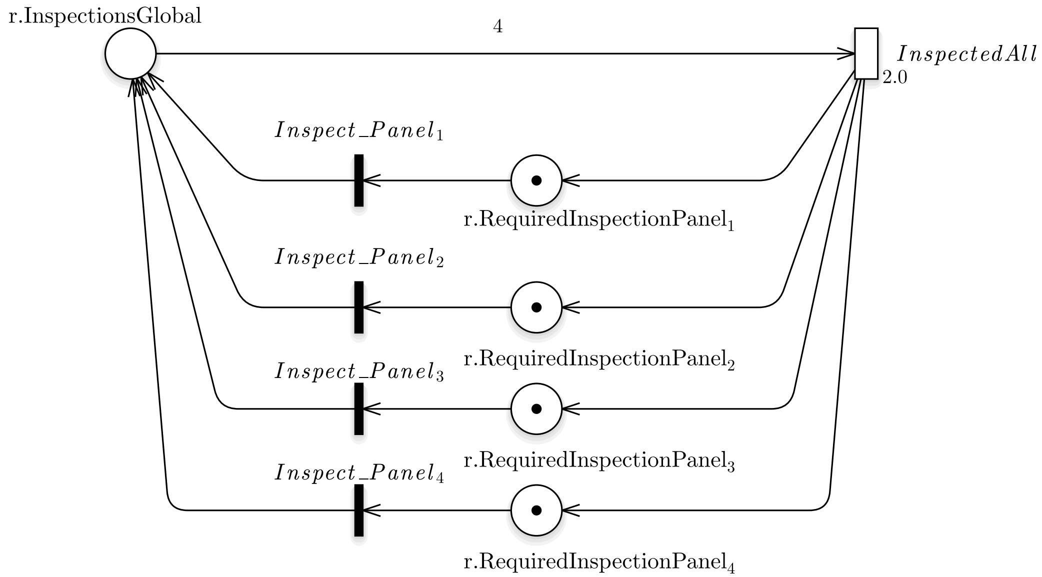


Figure 6.14: GSPNR Model for Inspecting "Panel2";

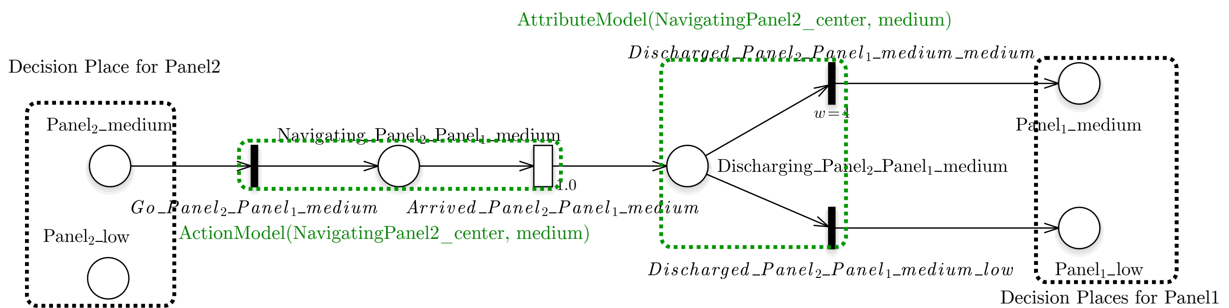


Figure 6.15: GSPNR Model for Jackal Navigation between "Panel2" and "Panel1";

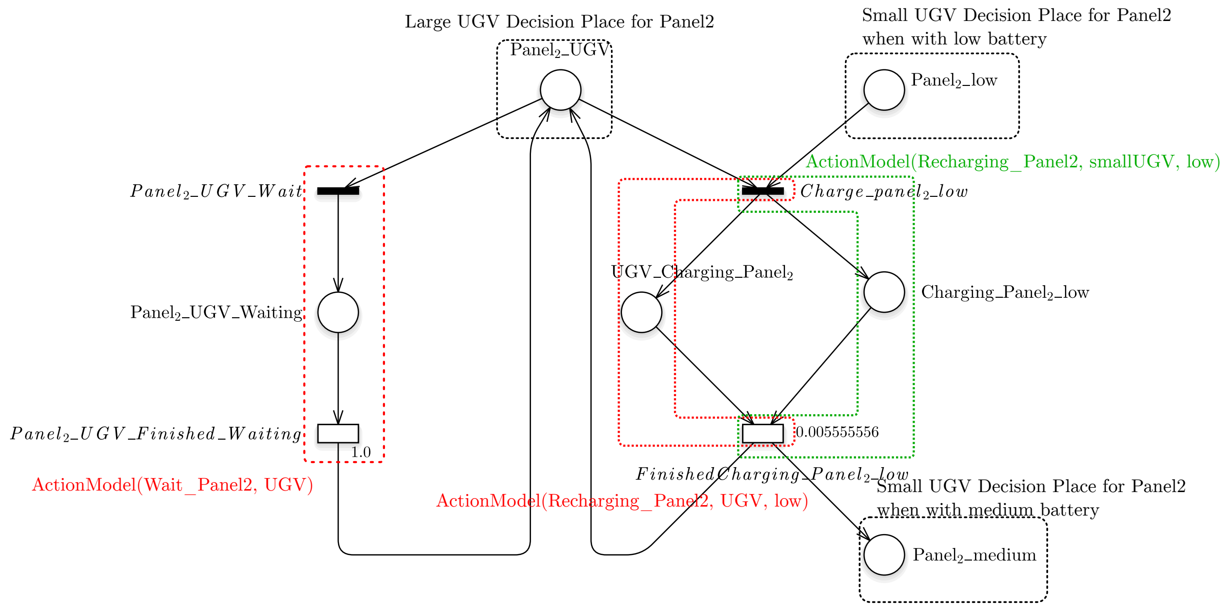


Figure 6.16: GSPNR Model for synchronized cooperative recharging action at "Panel2";

"Discharged_Panel2_Panel1_medium_medium" or the "Discharged_Panel2_Panel1_medium_low" transition fires. This will determine if the robot reaches the "Panel1" location with charged or depleted battery, respectively.

Synhcronized Cooperative Charging Action

The recharging action is a synchronized cooperative action undertaken by a jackal and the warthog. Figure 6.16 illustrates the GSPNR model for recharging at "Panel2". Two action models are shown, one for the jackal (outlined in green) and one for the warthog (outlined in red). They share the same decision transition and the same exponential transition representing the end of the recharging action. There is no attribute model because the charging is assumed to be deterministic. Thus, there is no uncertainty after the robot charges, and the token automatically goes to decision place that represents a robot in "Panel2" with a charged battery.

There is a third action model for the large UGV. This action model is composed of transition "Panel2_UGV_Wait", action place "Panel2_UGV_Waiting" and exponential transition "Panel2_UGV_Finished_Waiting". It allows the large UGV to remain in the same place for certain amount of time, waiting for a smaller UGV that might need recharging. In terms of policy synthesis, the exact same model could be achieved by obtaining the policy in an MDP with wait states. However, this increases significantly the state space size, and by including this auxiliary wait actions that are restricted to the warthog, the state space still has a manageable size.

Large UGV Navigation Action

An example of the warthog navigation model can be seen in Figure 6.17. It consists of a single action model, with its immediate decision transition, action place, and final exponential transition representing

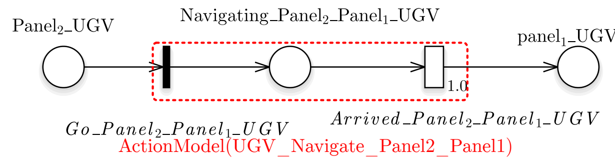


Figure 6.17: GSPNR Model for Warthog navigation between ”Panel2” and ”Panel1”;

the end of the navigation action. Other than the rate of the exponential transition, the action model for the large UGV navigation is the same for every possible navigation action according to the edges of the topological map.

Overall Model and Goal Specification

The full GSPNR model of this inspection scenario was designed using the procedure outlined in Section 4.3.2, using the action models and attribute models outlined previously for every action possible. Inspection GSPNRs were added to all nodes in the topological map except for location ”Center”, and navigation GSPNRs were added for all edges in the topological map for both the small UGVs and the large UGV. The recharging GSPNR was added to all location except the ”Center” location, because in the Gazebo world the robots had trouble coordinating the charging action within the tight space of the bridge. The goal of the multi-robot team was to inspect all panels as quickly as possible and so we added a transition reward of $+200$ to every transition that represents the decision to inspect a panel. We also wanted to discourage the system from having the small UGVs with depleted battery waiting for the large UGV to arrive, so we added a place reward of -1 to the places representing depleted small UGVs: ”Panel1_low”, ”Panel2_low”, ”Panel3_low” and ”Panel4_low”.

6.2.3 Results

The action duration parameters of the GSPNR model were estimated by running the navigation action server a single time for each navigation edge. We also estimated the mean charging time by running the charge action server a single time and recording how much time elapsed. For the battery discharge model, we chose simple parameters: every time a jackal finishes executing an action, it has 0.8 chance of staying at the same battery level, and 0.2 change of depleting its battery. We tested three different policies:

- **Random policy** - a policy where the system chooses to randomly fire a single transition from all available ones.

- **Handcrafted policy** - this policy was carefully designed by us, and it represents a greedy policy that tries to maximize the immediate number of inspections done. Both small UGVs try to inspect the closest available panel, and the larger UGV remains stationary until a small UGV is discharged. When this happens, the larger UGV navigates to closest discharged small robot and starts charging it.

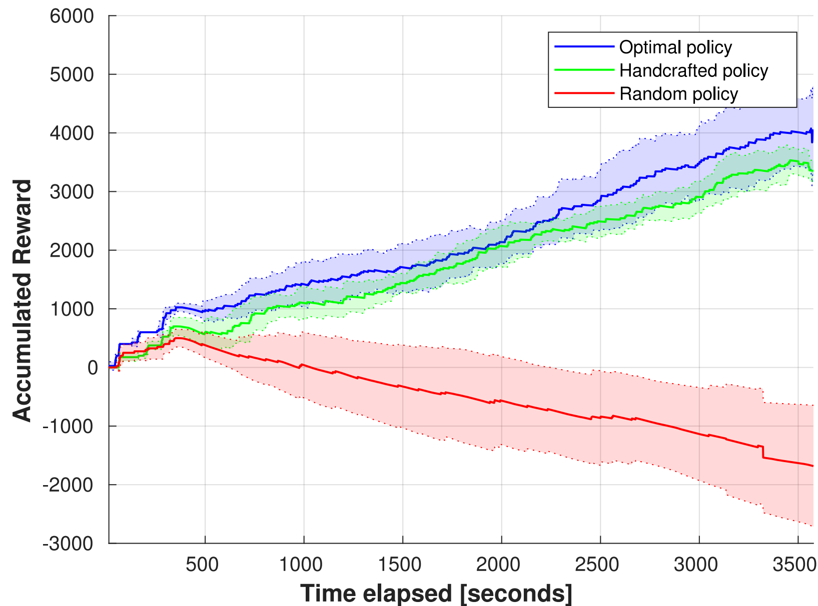


Figure 6.18: Average accumulated reward as a function of time. Obtained in Gazebo, for three different policies: the **optimal policy** (in blue), the **handcrafted policy** (in green), and the **random policy** (in red).

- **Optimal policy** - this policy maximizes the *discounted expected reward*. We used a discount factor of $\gamma = 0.99$, and a convergence criterion of $\epsilon = 0.01$ to run the value iteration algorithm on the equivalent MDP without wait states. This MDP had 84,545 states/reachable markings.

In Gazebo, we did 8 one-hour runs for each of the three policies: the optimal policy, the handcrafted and the random policy. These results can be seen in Figure 6.18, where the average accumulated reward is plotted as a function of execution time. The solid lines represent the average value, while the shaded areas represent the mean value plus and minus the measured standard deviation. The optimal policy was able to slightly outperform the handcrafted policy we designed. The handcrafted policy itself is good at coordinating the robots according to our goal specification, as it is a very large improvement over the random policy where no coordination is done.

A more direct representation of the policies' performance can be seen in Figure 6.19, where the number of solar panels inspected can be seen, instead of the accumulated reward. The optimal policy still outperforms the handcrafted one, but only slightly. The results between the optimal and handcrafted policy do not seem to diverge. This suggests that the initial coordination of the optimal policy leads to its higher performance, but that the ”steady-state” of number of panels inspected per unit of time is the same for both policies. To evaluate this point further, longer runs in Gazebo would be needed, but the fragility of the simulated robots in Gazebo and time constraints prevented us from doing so. Nonetheless, we were able to do 10 runs each with a 10 hour duration, with model simulation for both the optimal and handcrafted policy. These results can be seen in Figure 6.20, where we can verify that the optimal policy does outperform the handcrafted one, and that the number of inspected panels per unit of time (the slope of the lines) is higher for the optimal policy when compared to the handcrafted policy.

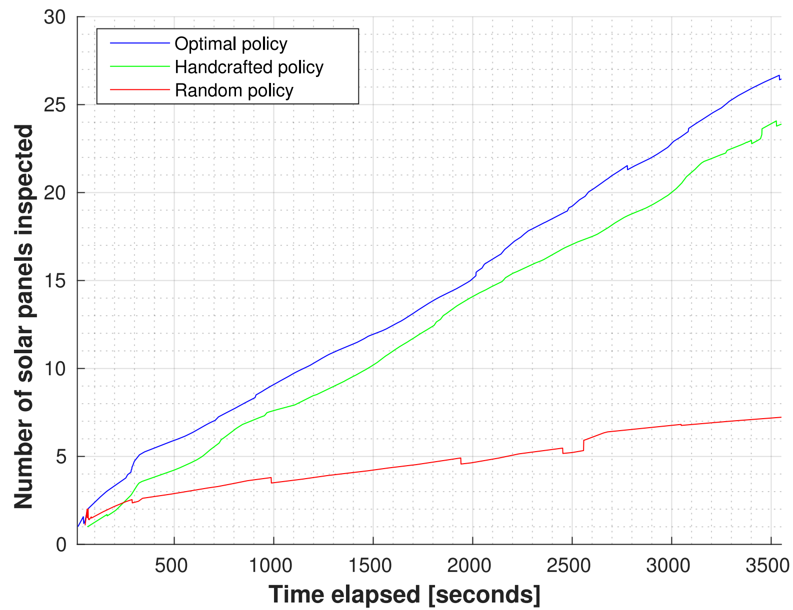


Figure 6.19: Number of inspected panels on average, along one hour runs of Gazebo simulation, for the three different policies: the **optimal policy** (in blue), the **handcrafted policy** (in green), and the **random policy** (in red).

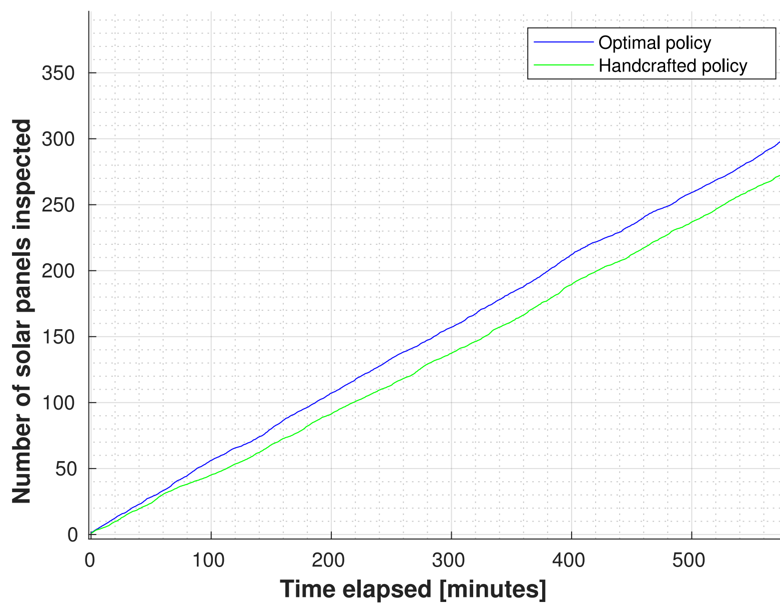


Figure 6.20: Number of inspected panels on average, along ten hour runs of model simulation, for the **optimal policy** (in blue), and the **handcrafted policy** (in green);

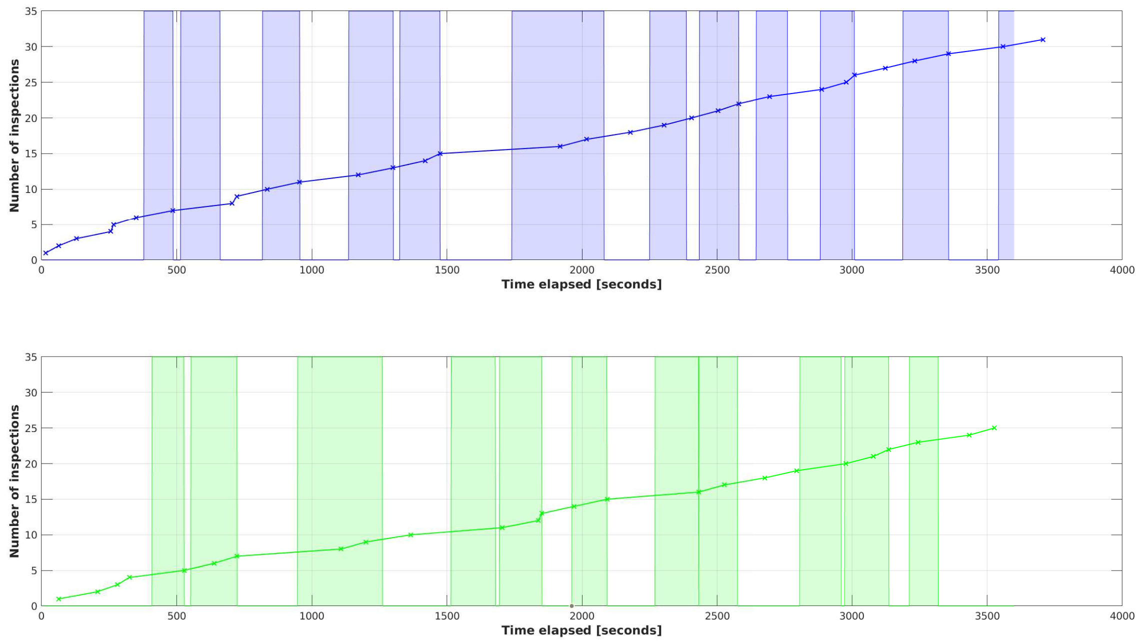


Figure 6.21: The lines depicted represent the number of inspections as a function of time and the shaded areas the periods when one of the UGVs is charging. Both graphs show the best run of the optimal policy (in blue) and the handcrafted policy (in green).

To further evaluate exactly how the optimal policy outperforms the handcrafted one, Figure 6.21 shows the number of inspections as time elapses, for the best run of both the optimal policy (in blue) and the handcrafted one (in green). The shaded areas represent the intervals where one of the small UGVs was being charged. It seems that the optimal policy distributes the charging actions more evenly (leading to a higher *downtime* proportion, as verified in Table 6.4), but exploits the fact that the system can execute two inspections almost simultaneously. The counter seen in Figure 6.14 resets at the beginning of the inspection of the last panel, allowing this last panel to be inspected simultaneously by two robots, which still counts as 2 inspections.

Just as done with the home vacuuming scenario, we also ran model simulation without robots to evaluate the accuracy of the our representation of uncertainty. This was done, again, with the function described in Section 5.3.3. Each GSPNR plan, for each of the 3 policies was executed for 10 one-hour runs. For each simulation, in both Gazebo and in the model simulation without robots, we measured three main metrics to classify the team's performance: the downtime percentage, the average time taken to inspect all 4 solar panels, and the average accumulated reward. The downtime percentage was measured as the proportion of running time where at least one robot was discharged and waiting for the warthog to be able to recharge. The average time between round of inspections was measured by counting the number of times the transition of the global counter "InspectedAll" fired, and dividing it by the running time. These results can be seen in Table 6.4. Just as in the domestic scenario, the GSPNR plan simulated without robots was a good predictor for the metrics measured in the real system simulated in Gazebo. More importantly, even though the optimal policy induces a larger downtime percentage when compared with the handcrafted policy, it was able to coordinate the small robots better. The overall time

	GSPNR Plan with optimal policy		GSPNR Plan with random policy		GSPNR Plan with handcrafted policy	
	Model	Gazebo	Model	Gazebo	Model	Gazebo
# of runs	10	8	10	8	10	8
$downtime$ [%]	42.84	33.94	94.53	80.87	40.25	31.45
$\Delta_{InspectAll}$ [s]	528	614	3155	3136	636	642
$\bar{r}$ [1/s]	1.3484	1.1074	-0.5756	-0.3622	1.0888	1.0094

Table 6.4: Comparison between the GSPNR plans simulated in the model and executed in Gazebo. Three GSPNR plans are compared: a plan with the **random policy**, with the **handcrafted policy** and the **optimal policy**. $downtime$ is the proportion of time, where at least one robot was waiting to be charged, $\Delta_{InspectAll}$ is the average time between inspecting all 4 panels, and $\bar{r}$ is the average accumulated reward.

taken to inspect all solar panels was shorter and on average it was also able to gain a greater total accumulated reward over 1 hour of execution time.

Chapter 7

Conclusions

This thesis' main objective was to solve a multi-robot inspection task planning problem using a heterogeneous robot team. To achieve this, we developed a generic framework to model multi-robot heterogeneous task planning problems as a GSPNR. Our approach falls into a model-based category that provides formal guarantees on the multi-robot team. We extend previous work where robots were represented as tokens to be able to model heterogeneous teams, and we provided a way to include a robot's battery in the GSPNR model. Furthermore, we developed novel algorithms to facilitate the creation of these models in a structured manner.

The toolbox developed is an open-source MATLAB add-on where users can create GSPNR models from scratch, and we were able to create interfaces with two of the most popular graphically-enabled software for GSPN models. We included a component that is able to synthesize optimal policies over these models. Lastly, we came up and implemented an algorithm to execute GSPNR plans that use these policies on real robotic systems, by integrating with the popular, open-source software library ROS. We characterized the toolbox's computational performance and scalability. While there is a bottleneck in regards to policy synthesis due to our use of exact approaches to calculate the optimal policy, our work is agnostic towards the methods used to compute a policy. It easily allows for integration with approximated methods that can solve more complex problems, at the expense of yielding sub-optimal policies with poorer performance. The GSPNR modelling framework is scalable, and our extensions to heterogeneous systems and the ability to model robot-specific attributes allows the representation of very complex problems. Our implementation is also efficient and it allows users to save GSPNR models that have millions or billions of unique reachable states. Furthermore, our algorithm to execute GSPNR plans scales well as the number of robots coordinated grows, and it does not consume more computational resources as the executed models become increasingly more complex.

To showcase the developed tool and disseminate it in the community we presented it during the RoboCup 2021 poster presentation and used it to solve the @Home Open Challenge. We coordinated a homogeneous multi-robot team that had to work together to vacuum a simulated house. We carried out model identification and synthesized an optimal policy for the GSPNR with the estimated model parameters. When executing the GSPNR plan with this new policy, we verified that our model was

capable of predicting how long it would take to vacuum the entire house considering battery limitations. Our execution algorithm was able to correctly coordinate the robot team according to the obtained policy for long-running times.

We have finally solved the inspection problem. We have modeled a heterogeneous robot team composed of three robots: 2 small UGVs and a single larger UGV. We have approximated the small UGVs' battery into two discrete levels, and allowed the large UGV to cooperatively recharge the small UGVs. The toolbox we developed was able to create the GSPNR model for this multi-robot problem, and the optimal policy we obtained was able to outperform a carefully handcrafted strategy which was itself a large improvement over an uncoordinated system. This demonstrated the ease-of-use of our toolbox in creating these complex models. Additionally, it provided a robust execution algorithm whereby in the 24 hours of executing GSPNR task plans, it was always reliable and it correctly coordinated the robot team.

Limitations and Future Work

The main limitation of the approaches and the toolbox we developed is associated with synthesizing optimal policies over the GSPNR models. The reachable marking set grows according to the expression "P multichoose r", where P is the number of states each robot can have, and r is the number of robots modelled. Even though this is an improvement over individually modelling each robot, where the maximum number of states grows exponentially with the number of robots, it still severely limits the size of the multi-robot scenarios where optimal policies can be synthesized in reasonable time. Any future work must tackle this limitation in order to be able to apply these approaches to large robot teams.

One possible way of mitigating this problem is by trying to optimize the *Policy Synthesis* module of our toolbox. By building an interface with established model-checking software such as PRISM or STORM, we can leverage the increased speed of these tools. Users would be able to export a GSPNR's equivalent MDP and run dynamic programming algorithms to extract an optimal value function over all possible states. Our toolbox would then be able to use this optimal value function to extract the optimal policy and use it within the *Execution* module.

Even then, it is easy to run into impossibly large state spaces with moderately sized robot teams. An alternative that might mitigate the state space problem are *approximated approaches* that use heuristics to go through the state space in a more efficient manner. However, policies obtained with these methods are not necessarily optimal or close to optimal. Additionally most of the heuristics used are not domain independent, having to come up with a good heuristic for each new problem approached.

Still, the ability of our modelling framework to model a complex multi-robot problem, and the capacity of our toolbox to save and simulate these very large models can be used to sample the state space in model-based Reinforcement Learning methods. This becomes particularly useful when simulation of the system us not possible, and real-world data is scarce.

Bibliography

[1] K. Jensen, M. Larsen, S. H. Nielsen, L. B. Larsen, K. S. Olsen, and R. N. Jørgensen. Towards an open software platform for field robots in precision agriculture. *Robotics*, 3(2):207–234, 2014.

[2] G. Orfanidis, S. Apostolidis, A. Kapoutsis, K. Ioannidis, E. Kosmatopoulos, S. Vrochidis, and I. Kompatsiaris. Autonomous swarm of heterogeneous robots for surveillance operations. In *International Conference on Computer Vision Systems*, pages 787–796. Springer, 2019.

[3] C. Gómez and D. R. Green. Small unmanned airborne systems to support oil and gas pipeline monitoring and mapping. *Arabian Journal of Geosciences*, 10(9):202, 2017.

[4] M. Mansouri, B. Lacerda, N. Hawes, and F. Pecora. Multi-robot planning under uncertain travel times and safety constraints. In *The 28th International Joint Conference on Artificial Intelligence (IJCAI19), August 10-16, Macao, China*, pages 478–484, 2019.

[5] C. Azevedo, B. Lacerda, N. Hawes, and P. Lima. Long-run multi-robot planning under uncertain action durations for persistent tasks. In *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 4323–4328. IEEE.

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

[7] M. L. Puterman. *Markov Decision Processes.: Discrete Stochastic Dynamic Programming*. John Wiley & Sons, 2014.

[8] H. Costelha and P. Lima. Modelling, analysis and execution of robotic tasks using petri nets. In *2007 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 1449–1454. IEEE, 2007.

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

[10] H. Costelha and P. Lima. Modelling, analysis and execution of multi-robot tasks using petri nets. In *International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2008)*, volume 2, pages 1187–1190, 2008.

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

[12] A. Bertolaso, M. M. Raeissi, A. Farinelli, and R. Muradore. Using petri net plans for modeling uav-ugv cooperative landing. In *Proceedings of the Twenty-second European Conference on Artificial Intelligence*, pages 1720–1721. IOS Press, 2016.

[13] F. M. Delle Fave, S. Canu, L. Iocchi, D. Nardi, and V. A. Ziparo. Multi-objective multi-robot surveillance. In *2009 4th International Conference on Autonomous Robots and Agents*, pages 68–73. IEEE, 2009.

[14] M. Kloetzer and C. Mahulea. Path planning for robotic teams based on ltl specifications and petri net models. *Discrete Event Dynamic Systems*, 30(1):55–79, 2020.

[15] H. Park and J. R. Morrison. System design and resource analysis for persistent robotic presence with multiple refueling stations. In *2019 International Conference on Unmanned Aircraft Systems (ICUAS)*, pages 622–629. IEEE, 2019.

[16] E. Hartuv, N. Agmon, and S. Kraus. Scheduling spare drones for persistent task performance under energy constraints. In *Proceedings of the 17th International Conference on Autonomous Agents and MultiAgent Systems*, pages 532–540. International Foundation for Autonomous Agents and Multiagent Systems, 2018.

[17] N. J. Dingle, W. J. Knottenbelt, and T. Suto. Pipe2: a tool for the performance evaluation of generalised stochastic petri nets. *ACM SIGMETRICS Performance Evaluation Review*, 36(4):34–39, 2009.

[18] E. G. Amparore, G. Balbo, M. Beccuti, S. Donatelli, and G. Franceschinis. 30 years of greatspn. *Principles of performance and reliability modeling and evaluation*, pages 227–254, 2016.

[19] F. Pommereau. Snakes: A flexible high-level petri nets library (tool paper). In *International Conference on Applications and Theory of Petri Nets and Concurrency*, pages 254–265. Springer, 2015.

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

[21] URL `https://github.com/cazevedo/multi-robot-gspnr-toolbox`. Last visited: 25/10/2021.

[22] Y. Butkova, R. Wimmer, and H. Hermanns. Long-run rewards for markov automata. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 188–203. Springer, 2017.

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

[24] URL `https://www.mathworks.com/products/ros.html`. Last visited: 25/10/2021.

[25] URL http://wiki.ros.org/actionlib. Last visited: 25/10/2021.

[26] URL https://www.youtube.com/watch?v=CH4iJs0rsIc&list=PLKlN8gv7SSBZpcyH_AwX9ZzPR0Dx_Bg0l&index=1. Last visited: 25/10/2021.

[27] URL https://github.com/cazevedo/battery_mockup. Last visited: 25/10/2021.

[28] URL https://github.com/clearpathrobotics/cpr_gazebo. Last visited: 25/10/2021.

[29] URL https://www.youtube.com/watch?v=SzpZBFmJZn8&t=123s. Last visited: 25/10/2021.

[30] URL https://github.com/warthog-cpr/. Last visited: 25/10/2021.