CPNP: Colored Petri Net Represention of Single-Robot and Multi-Robot Plans

Limor Marciano

Submitted in partial fulfillment of the requirements for the Master's Degree in the Department of Computer Science, Bar-Ilan University

Ramat Gan, Israel September, 2013

This work was carried out under the supervision of Prof. Gal A. Kaminka, department of Computer Science, Bar-Ilan University.

ACKNOWLEDGMENTS

First of all, I would like to express my heartfelt thanks to my teacher and advisor, Professor Gal Kaminka. It would be impossible to quantify how much I learned from him during my years as a student. Gal taught me to carry out my research to the utmost of my ability, to present challenges and overcome them and to prepare presentations at the highest level possible (excellent tips can be found on his website). He helped to develop my ideas, formulate them and present them. Gal showed me different ways to approach and solve problems. I especially want to thank him for always finding the time to see me even during busy periods.

Furthermore, I would like to thank the laboratory manager, my friend Gabriela Melamed. She was of tremendous help with all the administrative issues. Gabriela was very considerate and always tried to do the best in scheduling an appointment even if she had to squeeze me in.

Next, I would like to thank the secretaries of the Computer Science Department for their support and help whenever it was needed. They helped me to apply for and receive the proper scholarships. Their door was always open during and beyond office hours.

I also want to thank my friends and colleagues for their support, encouragement and help.

I thank G-d for exposing me to the wonderful world of robots, for letting me work with a world-renowned robotic expert and for providing me with wonderful and supportive friends.

Contents

List of Algorithms

ABSTRACT

Single-robot and multi-robot plans are steadily gaining interest in the academic community and in industry. The representation of such plans (for analysis, validation, monitoring, etc) is an important aspect of both single-robot and multi-robot systems. There are a great many challenges that should be addressed when representing robot plans in real world environments, such as dealing with interrupts, modeling concurrent events, reducing space complexity, providing validation and verification, etc. The current thesis addresses these issues. First, we introduce a framework called Colored Petri Net Plans (CPNPs) that explicitly represents single-robot plans based on Colored Petri Nets. This framework provides a comprehensive approach that addresses the mentioned challenges and proposes building blocks for representing single-robot plans. Then, we provide a space complexity analysis of existing multi-robot representations and examine their suitability for representing multi-robot plans. Finally, we extend the CPNP framework in order to represent multi-robot plans. The framework provides operators for representing either centralized or distributed plans. These operators are built based on the insights gained from the space complexity analysis in order to minimize the space complexity of the representation.

Chapter 1

Introduction

Robots are becoming part of our daily lives. They are being used in a variety of areas and mostly they work in dynamic, partially observable and unpredictable environments. As a result their plans are becoming more complex. These plans must allow representations of loops, decision making points, concurrent actions, synchronization with other robots, handling interruptions, resource usage, etc. Therefore, qualitative and quantitative formal methods to represent and analyze these plans are required in order to enable controlling, monitoring and validating of certain properties of robotic systems. Representations of robotic plans, which are not based on formal methods, tend to be tailored to a specific plan. Principled representations of either single-robot or multi-robot plans provide a systematic approach to modeling, analysis, controlling and design of robotic plans.

Representations methods should provide an explicit and efficient representations for single-robot and multi-robot plans. However, there are many challenges in representing either single-robot or multi-robot plans in real-world environments. Building an efficient representation in terms of space complexity, facilitating the readability, representing concurrency, dealing with interruptions and shared resources, representing coordination and cooperation among multiple robots, analyzing properties of the plan (e.g., reachability to goals) and preventing thrashing (i.e., behavior resets and then reselected) are common challenges which representations methods need to deal

with.

Finite State Machine (FSM), BDI (Belief, Desire, Intention) and Petri Net (PN) are common approaches for representing either single-robot or multi-robot plans. Existing approaches are described in detail in Chapter 2. However, in general they do not provide complete solutions to the challenges of modern robot systems. For instance, FSM and BDI systems do not in general, provide explicit representation of multi-robot systems.

Petri Nets (PNs) [66, 72] have recently emerged as a promising approach for modeling either single-robot or multi-robot plans. This approach provides a clear graphical representation for modeling and developing systems which are concurrent, distributed, asynchronous, nondeterministic and/or stochastic. In addition, it provides automatic analysis of certain properties of the plan (e.g., reachability to goal, avoidance of undesirable situations such as thrashing and deadlock, etc) [26, 45, 49, 66]. Petri Nets are described in detail in Chapter 2.

Lately, the interest in using Petri Nets for modeling single-robot and multi-robot plans has been increasing, and various types of frameworks and architectures have been proposed [20, 50, 53, 99–101]. However, existing Petri Nets based models do not provide satisfactory solutions for dealing with interruptions. In addition, robotic plan representations tend to be very complicated and their space complexity become very large. Therefore, reducing the space complexity is essential when representing robotic plans. While radically different approaches for representing multi-robot plans have been proposed, their relative strengths and weaknesses have not been investigated and, specifically, their space complexity.

In this thesis, we thus focus on the space complexity of Petri Nets based representations of robotic plans in our analysis. Motivated with the above issues, first, we proposed a method for representing single-robot plans (Chapter 3) based on Colored Petri Nets called *Colored Petri Net Plans (CPNP)*. CPNP is an extension of the PNP representation [101] (described in Chapter 2). We introduce basic building blocks and operators from which plans can be built. CPNP uses hi-

erarchies not only for facilitating the readability of the Petri Nets but also for reducing the space complexity. CPNP represents interruptions without the need of explicitly specifying the handling of each interrupt in each state in which it may occur. CPNP also supports sharing resources between multiple concurrent processes in the robot. Last, we proposed an algorithm for executing single-robot CPNP.

The second contribution is a space complexity analysis of the existing Petri Net based representations (Chapter 4). We show the scalability of the existing representations in two dimensions:

1. The technique which is used to represent multi-robots plans (P/T Nets or Colored Petri Nets).

2. The choice of representing either individual or joint states.

We show that using Colored Petri Nets with the combination of both individual state and joint state representations yield the best results in space complexity.

Based on this analysis, we extended the single-robot CPNP method for also representing multi-robot plans (Chapter 5). We introduced the basic operators of this representation. The operators are built according to the insight gain in the analysis of Chapter 4 in order to achieve a representation with the best space requirements. CPNP supports coordination among the robots, sharing resources and dynamic task allocation. Last, we proposed an algorithm for executing multi-robot CPNP.

The final contribution is an algorithm for automatically building a state space from a given CPNP (Chapter 6). Using this algorithm, we can analyze behavioral properties of the represented plan, such as: reachability, boundedness, home marking, liveness and fairness. These properties are specified in detail in Chapter 6.

Chapter 2

Background and Related work

This section discusses recent approaches to representation of, and reasoning about, robot plans. We begin by examining popular existing representations of single-robot and multi-robot architectures in Section 2.1. This survey reveals important open challenges. Petri Nets, which we briefly review in Section 2.2, offer a promising approach to addressing these challenges.

2.1 Representing Robot Plans for Execution

This section provides an overview of the recent approaches that have been proposed for representing single-robot and multi-robot plans for execution. We divide these approaches into three broad classes: Finite State Machines (Section 2.1.1), Belief-Desire-Intention and behavior-based approaches (Section 2.1.2), and others (Section 2.1.3).

2.1.1 Finite State Machine (FSM)

Many architectures for controlling and executing single-robot and multi-robot plans, such as [51, 57, 58, 61, 79, 80, 90], are based on Finite State Machine representation (Finite State Automata).

An FSM is a graphical representation composed of states, transitions between states (edges), and events associated with transitions. Every state represents a behavior. A *behavior* is a set of actions that are performed by robots to bring about some atomic (implicit) goal. Events in the world are matched against those associated with transitions, causing execution to stop in one state, and begin in the next. FSMs are often visualized by state diagrams. The states are represented by nodes in the graph, and edges move the agent from one state to another according to events associated with the edges.

FSM has a number of key advantages; first, it has automated methods for validation and verification [91]. Second, the rules of this representation are very simple and intuitive. Third, FSM can be hierarchical [91]. Notwithstanding these advantages, FSM's representation of concurrent events is limited and wasteful in space complexity [40]. Therefore, FSMs are inadequate for multi-robot systems [101]. Furthermore, FSMs suffer from inflexibility; there is a need to plan the order of executable tasks in advance.

2.1.2 BDI and Behavior-Based Representations

Behavior-based architectures [7,24,62,70,73] are robot control architectures in which the control of robots is shared between a set of *behaviors*. There are many variants, which differ in how behavior-based architectures execute behaviors (whether to fuse behaviors or select between behaviors), in how behaviors are selected, and in how their execution is terminated. Many robotic applications use behavior-based architectures, such as search and rescue, military missions, office automation, health care, etc. [8, 24, 46, 84].

Behavior-based architectures share the important strength in that the designer does not need to specify a predefined sequence of basic behaviors. This significantly enhances the flexibility of the framework since it allows the system to pick the executed plans from a potentially large library. On the other hand, these behavior-based robotic applications suffer from a lack of validation and

verification tools, or formal analysis.

Belief, Desire, Intention frameworks (BDI) [78] are a related method for representing and controlling behaviors. In a BDI architecture, agents select behaviors or plans to be executed (intentions), based on their goals (desires) and the current representation of the environment's state (beliefs). The behaviors are selected from a set of implemented behaviors. Each behavior has preconditions that allow its selection (the robot can select between enabled behaviors), and termination conditions that determine when its execution must be stopped. Thus, inherent to BDI systems there is a process of modeling the world (at least partially), which is not found in behavior-based systems.

BDI architectures build their plans during execution and thus they guarantee flexibility [74] and modularity. Furthermore, they are sensitive to environmental changes. However, BDI architectures do not have formal graphic representation and they lack automatic validation and verification tools.

There exist many architectures such as [39, 46, 47, 56, 67, 77, 88, 89, 97]. Those architectures are divided into single-robot architectures (e.g., [56] and [67]) and multi-robot architectures (e.g., [39,46,47,77,88,89,97]). Multi-robot architectures can be further divided into architectures represented by *individual state representations* or those represented by *joint state representations*. In individual state representations, each state represents a single robot (i.e., different robots, different states). Examples of individual state representations are [39] and [97]. By contrast, in joint state representations each state represents the joint state of all robots in the system. A variety of joint state representations are presented in [46,47,77,88,89]. The two types of representations will be explained in greater detail in Chapter 4.

To elaborate a bit further on some of the above-mentioned proposed architectures, STEAM [88, 89] is a teamwork architecture based on Cohen & Levesque's joint intentions theory [16]. STEAM uses joint state representation and facilitates reusability, which is essential for robotic teamwork. However, STEAM has not been used with real robots.

The BITE architecture [46, 47] is a behavior-based teamwork architecture that automates collaboration and coordination between autonomous robots in a real-world environment. BITE is based on BDI with joint state representation. Due to the separation between behaviors controlling a robot's interaction with its task and behaviors controlling a robot's interaction with its teammates, BITE guarantees flexibility and modularity. As opposed to STEAM, BITE enables the use of multiple synchronization and task-allocation protocols.

Teamcore [77] provides an architecture for integrating heterogeneous software agents. Each agent has a proxy which handles coordination. This teamwork model enables robust execution within dynamic environments, provides abstract task specification, and selects the appropriate agents for the mission in question. Teamcore uses STEAM as a teamwork module between the proxies. However, this architecture was built for software agents, not for robots.

2.1.3 Planning and Hybrid Approaches

We focus in this work on representations of plans for execution, and mostly ignore the important challenge of how the plans are synthesized in the first place. A vast span of literature on *automated (artificial intelligence) planning* deals with exactly this challenge. However, common wisdom in the field, originating with the work on ATLANTIS and other 3-Tier architectures [10, 34], is that representations used for planning are not often executable directly by relatively low-level controllers. Rather, an intermediate executive level is responsible for scheduling calls to instances of lower-level controllers, and request for new plans, from planner. Such a level uses a separate representation (and we will argue that Petri Nets can be useful for this purpose).

Researchers in artificial intelligence have long sought to address the need for generating a controlling algorithm for a given task, using automated means. One general approach to multi-robot planning explicit considers the uncertain in sensing and acting, and often also in communications. This approach is based on variants of decentralized Markov decision processes (DEC-MDP) and

decentralized partially-observable Markov decision processes (DEC-POMDPs). Unfortunately, such planning is NEXP-Complete [35], scaling exponentially in the number of robots. Despite recent successes in applying such planning to robotics problems, often by incorporating execution-time coordination [13, 81], most general tasks remain well outside the capabilities of existing planners.

One way to limit the complexity of planning is to utilize abstraction, e.g., using separate planning and execution layers for teamwork than for individual actions. For instance, the Skills, Tactics and Plays (STP) approach [11] treats plays as team plans, whiles skills and tactics are handled at the individual levels. This reduces the complexity of planning and execution, but creates artificial abstraction barriers, which may need to be broken to improve performance. For instance, the execution of a specific skill, which is supposed to be ignorant of the play involved and of considerations of teammates, may occasionally need to consider teammate location and decisions, for example due to failures at the individual levels, which can affect team-level plan execution. This does not happen in the representation discussed in this thesis.

T-REX [76] is a system that combines planning and scheduling at multiple levels of abstraction to allow plans to be re-worked depending on events at the different levels of abstraction. However, it only works with an individual robot, not multiple robots. In contrast, our work deals with both individual and multiple robots, but does not offer planning or re-planning capabilities.

2.2 Petri Nets: A Promising Basis for Representing Plans

Petri Nets [44, 66, 71, 72] are a graphical modeling tool for representing information processing systems that are characterized as being concurrent, asynchronous, distributed, parallel, nondeterministic, and/or stochastic [66]. We briefly describe Petri Nets in Section 2.2.1, properties of Petri Nets in Section 2.2.2, and Colored Petri Net in Section 2.2.3. We then discuss work by others

concerning the use of Petri Nets as a representation for robot plans (Section 2.2.4).

2.2.1 Petri Nets: The Basics

A Petri Net is a directed, weighted, bipartite graph that consists of two types of nodes: *places* and *transitions*. *Arcs* exist only from a place to a transition or from a transition to a place. A place may have zero or more *tokens*. Graphically, places, transitions, arcs, and tokens are typically visualized by circles, bars, arrows, and dots, respectively.

Definition 1. *A* Petri Net *(P/T Net) [14, 45, 66] is a tuple:* $PN =< P,T,A,E,M >$

(i) P *is a finite set of* places.

(ii) T *is a finite set of* transitions.

(iii) $A \subseteq P \times T \cup T \times P$ *is a set of* arcs *that connect transitions and places.*

(iv) $E : A \rightarrow \mathbb{N}_{>0}$ *is an* arc expression *function (also known as* weights*).*

(v) $M_j = [m_j(p_1),..,m_j(p_n)]$ *is the state of the net. It represents the marking of the net at time* j*, where* $m_j(p_i) = r$ *indicates that there are* r *tokens in place* p_i *at time instant* j*.* M_0 *is the* initial marking *of the net.*

The *places (P)*, *transitions (T)* and *arcs (A)* are separately grouped into three sets: P, T and A. These sets are finite and pairwise disjoint. The *arc expression* function E (also known as weight function) maps each arc $a \in A$ to a positive integer denoted as $e(a)$, (or $e(p,t)$, where $p \in P$ and $t \in T$) which defines the weight of the arc. It is customary to omit an arc expression that is equal to 1.

A distribution of tokens in the places is called a *marking* [42]. Formally, a marking maps each place $p \in P$ to a non-negative integer which defines the number of tokens that exist in p. A

marking is denoted by M such that $M(p)$ is the number of tokens in place p at marking M. One of the significant advantages of Petri Net is that it can clearly and graphically represent parallel processes. This includes a process that splits into multiple concurrent processes (fork), or multiple concurrent processes that merge into a single process (join) by the distribution of tokens in the Petri Net (i.e. the marking).

As mentioned in Definiton 1, the initial marking is denoted by M_0. It maps each place to the number of tokens at the initial state of the Petri Net execution. $M_0(p) = q$ means that there are q tokens in place p at the initial state.

Definition 2. $\forall p_i \in P, t \in T$, p_i *is called* an input place *of* t *iff* $\exists\ (p_i,t) \in A$.

Definition 3. $\forall p_o \in P, t \in T$, p_o *is called* an output place *of* t *iff* $\exists\ (t,p_o) \in A$.

Definition 4. *A transition* t *is* enabled*, iff each input place* p_i *is marked with at least* $e(p_i,t)$ *tokens.*

Figure 2.1 presents an example of a simple Petri Net at the initial marking M_0. The Petri Net consists of three places ($P = \{p_1, p_2, p_3\}$), one transition ($T = \{t_1\}$), and three arcs $A = \{(p_1,t_1),(t_1,p_2),(t_1,p_3)\}$ such that $e(p_1,t_1) = 1$, $e(t_1,p_2) = 2$, and $e(t_1,p_3) = 1$. p_1 is an input place of t_1 (since $\exists(p_1,t_1) \in A$) and p_2, p_3 are output places of t_1 (since $\exists(t_1,p_2),(t_1,p_3) \in A$). t_1 is enabled since $e(p_1,t_1) = 1$ and $M_0(p_1) = 1$.

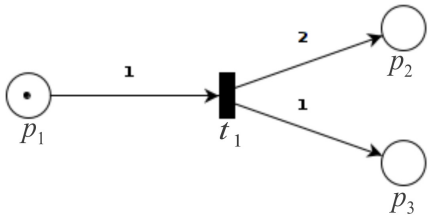

Figure 2.1 A Petri Net (initial marking M_0)

Definition 5. *Transitions* fire *the tokens (move a token from place to place) according to the following* firing rules*. A transition* $t \in T$ *fires tokens iff* t *is enabled. If* t *fires,* t *consumes* $e(p_i,t)$

tokens from each input place p_i ($e(p_i,t)$ is the arc expression of arc $(p_i,t) \in A$), and it produces $e(t,p_o)$ tokens on each output place p_o such that $(t,p_o) \in A$.

When a transition is enabled the corresponding move *may* take place. If this happens we say that the transition has been fired (Definition 5). An enabled transition does not necessarily fire the tokens; it may not fire in a situation of conflict. Conflict is a situation where one or multiple places are input places of multiple transitions. This situation causes multiple transitions to compete on firing the tokens (conflict is described in detail in Section 2.2.2).

The firing changes the marking of the Petri Net from a marking denoted as M_i to a marking denoted as M_{i+1}. In Figure 2.1, t_1 is the only transition existing in the Petri Net, so there is certainly not a situation of conflict. Consequently, t_1 satisfies the firing rules which leads to the firing of t_1. This firing transforms M_0 into the marking M_1 which is shown in Figure 2.2.

Figure 2.2 shows the marking M_1. This is the state of the Petri Net after t_1 has been fired. t_1 consumes $e(p_1,t_1) = 1$ tokens from the input place p_1 and produces $e(t_1,p_2) = 2$, $e(t_1,p_3) = 1$ tokens on the output places p_2 and p_3, respectively. In practice, the firing removed tokens from the input places (according to the arc expressions), and instead created new tokens and added them to the output places (according to the arc expressions).

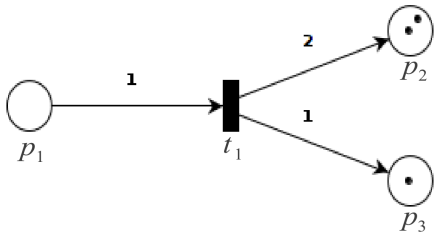

Figure 2.2 The marking M_1

2.2.2 Properties of Petri Nets

A major strength of Petri Nets is their support for analysis, validation and verification [66]. This advantage comes in the form of automated methods, which can check if a program (represented by a Petri Net) performs its goal, and if it has not gotten stuck in a deadlock. Another significant advantage is that each FSM can be transformed into Petri Nets. It has been shown that all Petri Net languages are *context-sensitive languages* [71]. This set of languages contains the regular languages; it follows from this that every FSM can be translated into a Petri Net but not vice-versa [23, 38, 66, 71]. The transformation to Petri Net gives better and more explicit representation of concurrent events [3,40,66]. Furthermore, Petri Net models are generally more compact [23,38]. For example, a composition of two Petri Nets (as will be shown in Figure 3.3) is more compact than a composition of two FSMs, since the composition of two FSMs is the product of two component state spaces [3, 40].

However, even though Petri Nets tend to be more compact than FSMs [38], still a major weakness of Petri Nets is their space complexity. Petri Net-based models tend to become too large for analysis even for modest-size systems [66]. In the next sections we will show how *Colored Petri Nets* and *Hierarchical Petri Nets* may reduce space complexity. In general, Petri Nets can clearly and graphically represent different types of executions: sequential execution, synchronization, merging, concurrency and conflict. In the remainder of the current section we show the representations of these executions.

Sequential Execution. Petri Net represents a sequential order execution by a sequential order of firing, i.e., transition t_{i+1} fires only after the firing of transition t_i. An example of sequential execution is shown in Figure 2.3. Transition t_2 can fire only after the firing of t_1. This representation imposes a precedence constraint "t_2 fires after t_1".

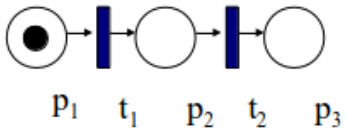

Figure 2.3 Sequential order execution (taken from [5])

Concurrency (fork). Petri Net is able to model systems of distributed control with multiple processes executed concurrently in time. Petri Net represents multiple different concurrency processes by multiple output places exiting from a single transition. When this transition fires, at least one token gets into each of the output places. A placement of at least one token in each of the output places represents multiple different concurrent processes. Figure 2.4 shows a representation of a process that forks into two concurrent processes ps_1 and ps_2 (t_1 and t_2 are concurrent).

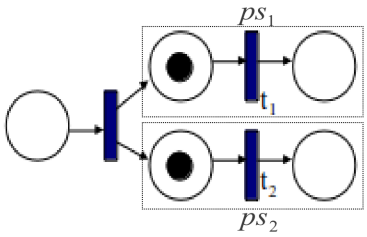

Figure 2.4 Concurrency (taken from [5])

Synchronization. According to Definition 4, a transition t can be enabled only if there is at least one token at each of its input places. Therefore, a Petri Net can represent synchronization between multiple executions by representing these executions as input places of a single transition. This transition can fire only if there is at least one token in its input places, meaning that it can fire only when the executions are synchronized. Figure 2.5 shows synchronization between two executions. Transition t_1 has two input places and one output place. It can be enabled only if there is at least one token in each of its input places; then it fires the two tokens from its two input places into a single token in its output place.

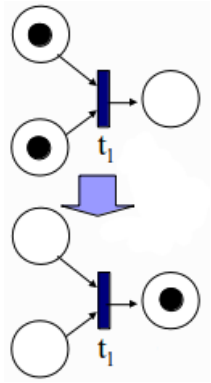

Figure 2.5 Synchronization (taken from [5])

Conflict. *Conflict* is a situation in which multiple transitions compete on firing the same tokens. This means that multiple transitions are enabled at a certain point in time; however, the firing of each transition causes the other transitions to be disabled. This situation is depicted in Figure 2.6. Transitions t_1 and t_3 are both enabled but the firing of t_1 causes t_3 to be disabled and vice versa.

The resulting conflict can be resolved in three possible ways:

1. By a guard, as will be described later on.
2. In a probabilistic way, i.e. by assigning appropriate probabilities to the conflicting transitions (e.g. Generalized Stochastic Petri Nets - GSPNs [15]).
3. By assigning time constrains to those conflicting transitions (timed Petri Net [103]).

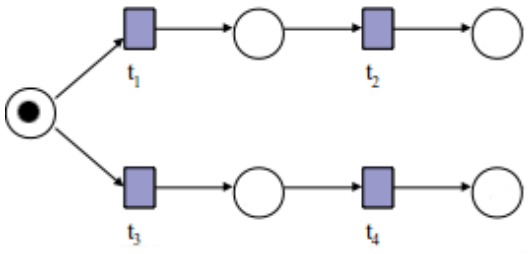

Figure 2.6 Conflict (taken from [5])

Note the difference between a conflict and a fork: in a conflict a single token satisfies multiple transitions (i.e., a token enables multiple transitions) but only a single transition can fire, while in a fork a single token is split into multiple tokens by the use of a transition that fires this token.

2.2.3 Colored Petri Nets

Colored Petri Nets (CP Nets or CPNs) [2, 4, 42–45] are a generalization of Petri Nets. They are a part of Petri Net formalisms that extend the basic Petri Net. These extensions are called *high-level Petri Nets*. CP Nets are a discrete-event modeling language combining the capabilities of Petri Nets with the capabilities of a high-level programming language (i.e. variable assignments and guard predicates) [45].

Although CPNs are computationally equivalent to Petri Nets [44], they offer greater flexibility in compactly representing complex systems. This is done by coloring the tokens with structured data. The meaning of this is that every token has attributes, whose values define its *color*. These values can change when tokens travel through transitions, and tokens can be directed towards specific places based on their color. The definition of CP Nets is as follows [45]:

Definition 6. *A* CP Net *is a tuple* $CPN = (P, T, A, \Sigma, V, C, G, E, M)$ *where:*

1. P, T, A *as defined in Definition 1.*
2. Σ *is a finite set of non-empty* color sets.
3. V *is a finite set of* typed variables *such that* $Type[v] \in \Sigma$ *for all variables* $v \in V$.
4. $C : P \to \Sigma$ *is* a color set function *that assigns a* color set *to each place. This restricts the color of tokens that can be in a given place.*
5. $E : A \to V$ *is an* arc expression *function.*
6. $G : T \to boolean$ *is* a guard function *that assigns a boolean expression to each transition* t.

7. $M_j = [m_j(p_1), .., m_j(p_n)]$ *represents the marking of the net at time j (similar to Definition 1). Note that, unlike Definition 1, all tokens have colors.*

In CP Nets, tokens store complex data structures by their colors. Similar to variables in programming languages, this information can be of multiple types (e.g integer. boolean, etc). In CP Nets the types of data are defined in the *color set* (denoted as Σ). This set contains all classes of variables that may be represented by tokens. These classes are called *colors*. The variables belong to the set V such that the class (type) of $v \in V$ is a member of Σ (denoted as $Type[v] \in \Sigma$). To summarize: in CP Nets, tokens represent collections of variables, and each variable has a color in Σ ($Type[v] \in \Sigma$).

Each place may contain tokens from a variety of classes defined in Σ. The color set function (denoted by C) is a function that assigns a color set for each place in the CP Net. As such, it defines the token colors that can be in p.

Unlike P/T Nets, in CP Nets we have a wide variety of different tokens. Hence, the arc expressions of CP Nets are much more complicated. The arc expression function of a CP Net maps each arc to a multiset (i.e. a set in which a member may appear more than once) of V. Arc expressions may involve complex calculation procedures on token *variables*. For an arc $a \in A$, consisting of the ordered pair p,t ($p \in P$, $t \in T$), it is required that the type of the arc expression (denoted as $E(a)$) is $C(p)$, i.e. $Type[a] = C(p)$. $Type[a]$ is a set of the types of variables that participate in $E(a)$.

Guards are an additional extension to Petri Nets as introduced by CP Nets. The *guard* function G maps each transition $t \in T$ to a boolean expression called *guard*. A guard is a condition on the transition. In addition to the firing rule defined in Definition 1, a guard must be satisfied in order for the transition to fire. Each guard consists of variables. The set of variables appearing in a guard is required to form a subset of V. In principle, all transitions should have a guard. If no condition is specified for a transition $t \in T$, the guard will be defined as $G(t) = true$. As an accepted notation

rule, guards are written in square brackets and positioned above or below the transition [45].

Example illustrating the use of CP Nets. Assume that we have a soccer robot R_1 that performs a simple action: *kick*. The P/T Net of this example (depicted in Figure 2.7) contains 3 places $P = \{p_1, p_2, p_3\}$. These places represent the situations where R_1 is before / now executing / after the action, respectively. In addition, the P/T Net contains two transitions $T = \{t_1, t_2\}$ that represent the events of starting and terminating the action, respectively.

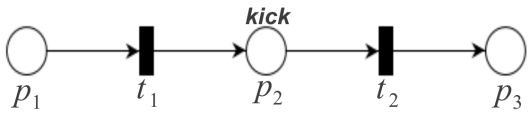

Figure 2.7 A P/T Net of a kicking action executed by a soccer robot R_1

Consider the following, slightly more complicated example: two robots, R_1 and R_2, perform the kicking action but R_2 also wants to sit down after it kicks the ball. If we would want to represents this example by the use of P/T Net, we would need to use two separate P/T Nets (one for each robot) since all P/T Net tokens are identical. Alternatively, we can represent this example using CP Net in a compact *single* CP Net (depicted in Figure 2.8). Furthermore, in CP Nets it is possible to represent constraints (for example: the robots can perform the tasks only if they are both close enough to the ball) more effectively than in P/T Nets.

Figure 2.8 shows the CP Net of the example described above. It can be seen that the tokens are distinguished by colors (black and gray) which represent R_1 and R_2, respectively. This CP Net is defined follows: $P = \{p_1, .., p_5\}$, $T = \{t_1, .., t_4\}$, A is a set that contains all the arcs in the graph, and $\Sigma = \{ROBOT, NUM\}$ such that each variable of type *ROBOT* represents a robot identification and each variable of type *NUM* is a real number greater than zero. The variables of this CP Net are $V = \{r : ROBOT, d : NUM\}$ where r represents the robot ID and d is the distance between the robot and the ball. R_1 and R_2 are the possible values of r.

All places in Figure 2.8 contain only tokens of type *ROBOT*, $C(p) = ROBOT$, $\forall p \in P$. The

expressions on the arcs are called *arc expressions*. Two kinds of arc expressions exist in Figure 2.8: arcs $\{(p_1,t_1),(t_1,p_2),(p_2,t_2),(t_2,p_3)\} \subset A$ have $1'r = R_1 \vee 1'r = R_2$, expressing that this arc can move a single token of $r = R_1$ or a single token of $r = R_2$. The remaining arcs have $1'r = R_2$ arc expressions, meaning that only a single token of $r = R_2$ can move on these arcs. The arcs expressions represent the constraint that $r = R_1$ and $r = R_2$ perform the kicking, but while robot R_1 terminates his job after *kick* (i.e in place p_3), robot R_2 continues with the sitting action and finishes in place p_5.

The expression $[d < 10]$ on transition t_1 is a guard expressing the constraint that each robot can start performing his tasks only if the distance between it and the ball is less than 10cm. M_0 is the marking represented in the figure. This example presents a Colored Petri Net Plan (CPNP) at a glance: our novel CP Net-based framework for representing single- and multi-robot systems. CPNPs will be described in detail in Chapters 3, 5 and 6.

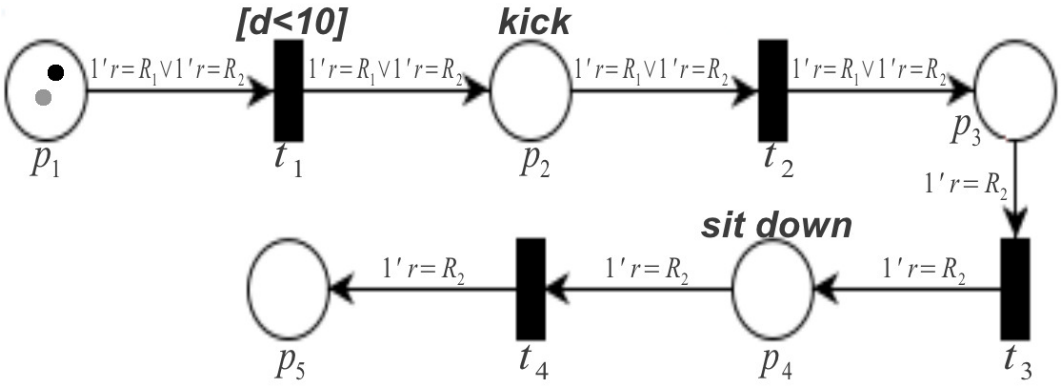

Figure 2.8 A CP Net of: 1) a kicking action executed by R_1 and R_2; 2) a sit down action performed by R_2 after it kicks the ball

2.2.4 Literature on Petri Net-Based Representations of Robot Plans

There have been recent modeling approaches on specific uses and domains, e.g., conversation protocols [17–19,36,63,75], exploration and mapping [12,85], artificial tutoring agents [64], man-

ufacturing systems [27, 95], etc. However, these models are built to a specific domain.

This section presents an overview of the main approaches that have been proposed in the past for the representation and execution of robotic behaviors and plans using Petri Nets. We discuss both single-robot and multi-robot representations. In the overview we will focus in particular on methods for dealing with interrupts (unforeseen events), the use of hierarchies, and the representation of shared resources. We conclude this section with a comparative discussion of the common strengths and weaknesses of the approaches.

A Petri Net Plan (PNP) [69, 101, 102] is a framework that can be used for representing single-robot systems; however, it is mainly associated with representing collaboration and coordination in multi-robot systems based on Petri Nets. This framework is inspired by joint intentions theory [16]. A PNP is a comprehensive language which consists of elementary building blocks and operators which allow for intuitive robot and multi-robot behavior design. These building blocks represent essential robotic developing features including sensing, interrupts and concurrency.

One of the advantages inherent to Petri Net is the ability to get a formal analysis of plans. Because of this, PNPs purport to be very expressive and easy to use and debug [101]. However, still there are key weaknesses in PNPs. First, PNPs require the programmer to connect in advance each place where an interrupt can occur to the corresponding interrupt-handling plan. In addition, PNPs do not deal with managing of shared resources among multiple robots, and they barely make use of hierarchies. In contrast, our work deals with the above-mentioned issues by expanding PNPs through the use of Colored Petri Net, an extension to Petri Nets.

Wang et al. [96] present a Petri Net Coordination Model for an Intelligent Mobile Robot. The goal of this model is to specify the integration of path planning, supervisory motion control, and vision systems in a mobile robot architecture. This work explicitly represents task precedence and information dependency among the individual systems in the Intelligent Mobile Robot System (IMRS). However, the model's Petri Nets can become very large and complex due to the lack of

hierarchies. Furthermore, no interrupt-handling method has been specified in this model.

Costelha and Lima [20, 23] introduce a framework for modeling, analysis and execution of single robotic tasks based on a Generalized Stochastic Petri Net (GSPN) [6, 48]. They extend this framework with a multi-robot cooperative task model [21,22]. In this model, events are represented by stochastic transitions. Places represent primitive actions, subtasks and predicates according to individual state representation.

Both models allows for the verification of logical properties, such as deadlocks and resource conservation, and (probabilistic) performance properties such as probability or average time to reach a desired state. Furthermore, Costelha and Lima introduce a method to identify the parameters of the Stochastic Petri Net models from real data, and analyze their method through the analysis methods of Stochastic Petri Nets is introduced. Similar to [96], they do not provide a method for trying to deal with interrupts.

Kotb et al. [53] introduce a formal framework for robotic cooperation in multi-agent systems based on sound workflow nets (another extension to Petri Nets). In their work, activities are performed by transitions, and places represent preconditions and effects. The workflow nets extension allows for dealing with more complex tasks that involve heterogeneous robots. In addition, the authors provide an algorithm to verify similarities among agent capabilities in order to determine the possibility of cooperation between multiple agents with respect to a desired task. However, in this model hierarchies are lacking altogether, resulting in very complicated representation when the system becomes more complex. Also, there is no interrupt-handling ability.

Lacerda and Lima [55] present a method to build Petri Net supervisors for Multi-Agent Systems (MAS) using linear temporal logic (LTL) formulas to specify acceptable/desirable behaviors for multi-agent systems. The construction of these supervisors is done by translating the natural language specification into LTL formulas and then translating these into Petri Net models. Similar to [20, 23], events are associated with transitions; places however are associated with state descrip-

tions. Lacerda and Lima show that building supervisors according to this method is more efficient as the number of robots rises. However, this method too does not handle interrupts and unforeseen events.

Xu et al. [99] present a formal approach for modeling and analyzing multi-agent behaviors using Predicate/Transition (PrT) Nets (a formalism of Petri Nets). In their work, transitions represent agent capabilities and places represent predicates. They also provide a reachability analysis and verification algorithms. This plan representation does not allow for unexpected events (interrupts). In addition, their modeling works only in centralized cases, as the predicates consist of the statuses of either a part, or all of the teammates.

We now move on to discuss approaches that make use of Colored Petri Nets. Yen et al. [100] developed an architecture based on Petri Nets for modeling collaborative teamwork based on a shared mental model. They used colored Petri Net with two types of places: control places, and belief places. Their model provides the ability to anticipate the information needs of teammates, and therefore proactive information exchanges. However, the framework does not deal with the managing of shared resources. Interrupts and unexpected changes in the environment are handled at group level, by an algorithm that dynamically changes roles of teammates; the framework does not provide options for repairing some interrupts by the interrupted robot itself (e.g., robot falls down).

Moldt el al. [65] introduce a representation of multi-agent systems based on Colored Petri Nets. They implement Shoham's paradigm [86] (called Agent-Oriented Programming, or AOP) by using Colored Petri Nets to better model a society of agents who are working concurrently. They enlarge the flexibility of the system by suggesting that the agent can be dynamically created and destroyed. Moreover, they achieve hierarchies by using a mechanism that folds certain behaviors into a class. However, their representation does not support dynamic changes in the environment, since a knowledge base that is represented by color cannot change dynamically. In addition, this

method cannot handle interruptions and unforeseen events nor manage shared resources among multiple agents.

Jun et al. [60] introduce a new individual state representation to model multi-agent systems with Hierarchical Colored Petri Nets (HCP Nets). Using of Colored Petri Net's properties, they analyze dynamic properties of the MAS such as reachability to goal, and deadlock detection and avoidance. Unfortunately, this method shares some of the disadvantages with the method used by Moldt: it cannot handle interruptions and unforeseen events, nor manage shared resources among multiple agents.

A non-CPN method which does provide for the managing of shared resources, is the mechanism of King et al [50] for programming, control and supervision of multi-agent systems. Plans are generated for each single robot using POP [83] and then compiled into Petri Nets for analysis, execution, control and monitoring. To avoid conflicts that may arise from the usage of shared resources among the multiple robots, supervisory control techniques are applied to the PN controller. Interruptions and unforeseen events are dealt with by replanning in real-time. On the one hand this makes the framework flexible, but on the other hand the need for replanning significantly increases the runtime of the system. Also, in cases where many interrupts occur within a short timespan, the system might encounter an interrupt exactly when it is in the middle of replanning due to a previous interrupt, leading to the current replanning becoming redundant.

So far, we have mainly discussed representations of multi-agent systems. It must be noted that there are a wide variety of additional frameworks for representing a specific type of multi-agent systems, namely multi-agent conversation protocols [17–19,63,75]. However, none of these works address the control of multi-robot teams. Instead, they focus on inter-agent communications

However, Gutnik and Kaminka's [36] examination of the scalability and suitability of Petri Net approaches for representing those conversation protocols is relevant. They show that while the run-time complexity of monitoring conversations using different approaches is the same, space

complexity undergoes significant improvements when using Colored Petri Nets with joint state representation. The authors show how to reduce space complexity by the use of hierarchies and tokens' colors. They represent multiple instances by tokens' colors. Their work is a special case of our work here.

Summary. There exists a wide variety of Petri Net-based robot architectures or frameworks which all have their own strengths and weaknesses. However, the key weaknesses of existing approaches whether single or multi-robot lie in the fact that they do not provide (satisfactory) solutions for interrupt-handling. Also, most of these approaches do not scale in terms of space complexity when the system becomes more complex, and in particular when handling multiple robots. In some cases [60, 65], this problem is dealt with visually through the use of hierarchies, but without actually reducing the space complexity [45]. In our work, we will analyze the space complexity of previous approaches and provide a real and comprehensive new approach which does significantly reduce the actual space complexity. This approach is based on Colored Petri Nets and makes use of hierarchies in an innovative way in that it changes tokens' colors instead of repeatedly creating new instances of the Petri Net.

Chapter 3

Representing the Plans of a Single Robot: CPNP

Colored Petri Net Plans (CPNPs) are a representation for complex single-robot and multi-robot plans, based on Colored Petri Nets. This representation is an extension of PNPs [101] utilizing high-level Petri Nets (i.e, Colored Petri Nets and Hierarchical Petri Nets), for a more efficient representation of plans composed of complex behaviors. This chapter introduces the CPNP representation for single robot plans.

CPNPs will be defined in Section 3.1. We will describe the structure and the basic building blocks of CPNP (Section 3.2), and then move on to hierarchical decomposition (Section 3.3). In Section 3.4 we will show how to represent interrupt handling using Colored Petri Net. We will discuss the representation of resources in Section 3.5, and finally in Section 3.6 we will introduce the algorithms for executing a single-robot CPNP.

3.1 Colored Petri Net Plan (CPNP)

A Petri Net is state- and action- oriented, meaning that it simultaneously gives an explicit description of the possible states and actions [44]. It describes the states of the system as well as the events (transitions) that can cause the system to change state [45]. Usually, events are represented by transitions and states are represented by markings.

In some of the existing literature on robot representations based on Petri Nets [53, 94, 99], places are used to represent predicates and transitions to represent actions (behaviors). In these representations, each marking shows conditions that lead to the execution of different actions. These formalisms are called Predicate/Transition (Pr/T Nets) which are a type of high-level Petri Nets.

In other works [22, 23, 50, 55, 101], transitions are chosen to represent events (i.e. the start and the termination of an action). Places are associated with the execution of actions (in addition, they can represent other parts of the system such as conditions, connections between parts of the system, etc). In this way, each marking shows which actions are executed by the robot.

Thus, there are two ways of representing executions of actions by robots:

1. Representing start, execution and termination of an action by a single transition in a Pr/T Net.

2. Representing start and termination of an action by separate transitions. Execution of an action will be represented by a place.

Since in this work we want to represent which actions are performed in each state and what is the state of each action (start, execution or termination), we choose to build our representation of robotic executions of actions conform the second option. In CPNP the states, actions and events (starting and terminating the execution of an action) are represented by markings, places and transitions, respectively.

We build on the definition of Petri Net Plans (PNPs, [101]) to which we add the use of token colors (the significance of this addition will be demonstrated). A CPNP is an augmented CP Net (Definition 6). CPNP extends CP Nets with a set of desired termination states (goal markings), denoted as L (similar to [101]).

Definition 7. *CPNP is a tuple* $(P, T, A, \Sigma, V, C, G, E, M, L)$*, where:*

1. $P, T, A, \Sigma, V, C, G, E, M$ *have been defined in Definition 6 (CP Net);*
2. L *is a set of goal markings (described below).*
3. *Following PNP [101], places are partitioned into two classes* $P^E \cup P^C$*, as follows:*
   (a) P^E *is the set of execution places, which models the execution state of actions in the CPNP; these types of places will be discussed in detail in Section 3.2.1.*
   (b) P^C *is the set of connector places, which are used to connect different CPNPs. There are two kinds of connector places:*
      - *Initial places;*
      - *Termination places.*

   These places are used for operators (discussed in Section 3.2.2).
4. *Following PNP [101], the transitions of a CPNP are partitioned into three subsets* $T = T^S \cup T^T \cup T^C$*, where:*
   (a) T^S *is the set of start transitions, which lead into a place representing an execution of an action.*
   (b) T^T *is the set of termination transitions, which lead tokens out of a place representing an execution of an action.*

(c) T^C is the set of control transitions; these transitions are used for operators (discussed in Section 3.2.2).

Goal markings are a set of markings which represent the goal states of the robots. Goals represent the final state of the program. Once a robot gets into one of such states it finishes its work. States are represented by markings. Once the current marking is equal to one of the goal markings in L, the algorithm that executes the CPNP halts. This algorithm is described in detail in Section 3.6.

In CPNP the robot's knowledge of the environment is encoded in the tokens' color by the set of variables V. CPNP assumes the existence of a perception component. The robot's knowledge is obtained through the perception component, which sets the values of the variables in V to represent the knowledge of the robot about the environment. Therefore, the colors of all the tokens are synchronized and identical.

We defined two types of tokens in CPNP: *robotToken* and *resourceToken* (described in detail in Section 3.5). These types are distinguished by their color. The first type represents the knowledge of the robot, and the second type represents a specific type of resource.

The color of the tokens from the *robotToken* type represents the robot's knowledge about the environment. It consists of the values for the following variables (defined in the V set):

1. Variables that participate in each guard conditions.

2. Boolean variables that indicate hierarchies (Section 3.3).

3. Boolean variables that indicate occurrences of interrupts (Section 3.4).

The *resourceTokens* have a single variable *type* , which indicates the type of resource that is represented by the token (e.g., battery, camera, etc). The value is taken from a set of predefined resources. The *resourceToken* is described in detail in Section 3.5.

For modeling the execution of actions a CPNP structure utilizes specific places called *execution places*. Each execution place represent the execution state of a specific action; each action has an execution place $p_e \in P^E$. $M(p_e)$, which defines the marking of the place p_e, determines whether the behavior is active or not. The set of execution places of a CPNP structure is P^E. This set models the execution state of the system.

3.2 Basic Building Blocks

This section introduces the fundamentals of our representation. CPNP is built by actions, connected together using operators. The actions are executed by the robot. The operators connect the different actions in order to determine their order of execution. We will describe the representation of actions performed by the robot in Section 3.2.1. Subsequently, we will describe the operators which connect between the different actions (Section 3.2.2).

An example of CPNP will be incrementally built up in order to illustrate CPNP building blocks. The example will introduce a single soccer robot player. This soccer robot will be built step by step, from one building block to the next.

3.2.1 Actions

Atomic actions are actions that cannot be divided into subactions. They constitute the basic building blocks of a CPNP. Structurally, the representation of actions in CPNP is similar to PNP [101]; however, CPNP is based on CP Nets. Hence, CPNP uses the components of CP Nets (e.g., colors, guard and arc expressions).

An action (depicted in Figure 3.1) is represented in CPNP by three places (initial, executed and termination, denoted as p_i, p_e and p_t respectively) and two transitions (start and termination, denoted as t_s and t_t respectively). These transitions represent the starting and terminating events. The

start transition begins or activates the actions. The termination transition deactivates the action. Usually, each termination transition has a guard. Once this guard is satisfied, the termination transition fires and deactivates the action. If no guard has been specified, the action will be terminated after a single iteration (e.g., kick). The above describes the basic CPNP structure.

Initial and termination places are connector places (defined in Definition 7). The existence of a token in the initial place means that the robot is going to start the action. A token in the execution place means that the robot is currently executing the action. Analogously, a token in the termination place signifies that the robot finished performing the action.

The outcome of a single action can vary significantly. It is determined during the execution phase, according to the environmental conditions, the robot's senses, or the success or failure in executing the action. The actual outcome can affect the robot's continued operations.

The outcome is encoded in the variables that build the tokens' color (i.e the elements of the set V defined in Definition 7) during the execution phase. The robot makes decisions and changes them according to the effect of his actions as well as changes in the environment. It does so by using the *choice operator*. This operator is explained in detail in Section 3.2.2.

It should be noted that the representation of outcomes in CPNP differs from said representation in some previous works (e.g., [21, 23, 101]). While in CPNP each action affects by an appropriate variable with a range of values that include all possible outcomes, in other works outcomes are represented by splitting the graph according to each possible outcome. The CPNP representation reduces the space complexity of representing plans.

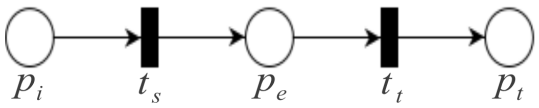

Figure 3.1 Basic CPNP structure - action

An action is illustrated in Figure 3.2 by an action called *go to ball*. In this example, the robot

moves towards the ball until he gets close enough to it. The elementary structure of this action contains three places $P = \{p_i, p_e, p_t\}$ and two transitions $T = \{t_s, t_t\}$. p_i and p_t are initial and termination places, respectively. A token in p_e signifies that the robot is currently executing the *go to ball* action. *go to ball* is the first step of a bottom up example of building a soccer robot. Through the construction of the soccer robot we will illustrate each component of the CPNP.

The robot keeps executing the *go to ball* action until t_t fires and terminates the action. t_s and t_t are start and termination transitions, respectively. The outcome of the *go to ball* action is encoded in the variable d. d is a variable of type real numbers greater than zero. t_t has a guard $[d < 10]$. This guard signifies that once the distance between the robot and the ball is smaller than 10cm, t_t fires and terminates the execution of *go to ball*.

The CPNP in Figure 3.2 has the following properties: it has one execution place ($P^E = \{p_e\}$) and two connector places ($P^C = \{p_i, p_t\}$). In addition, this CPNP has a start transition and a termination transition ($T^S = \{t_s\}$ and $T^T = \{t_e\}$). The color set Σ has a single element: NUM, which indicates a number. This means that the types of variables in V can be only NUM. The V set contains one variable d; this variable represents the distance between the robot and the ball.

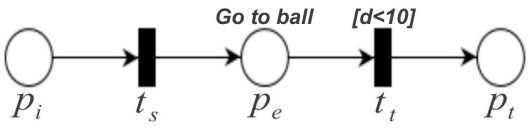

Figure 3.2 Soccer robot: ordinary action - *go to ball*

3.2.2 Operators

Operators connect between multiple CPNPs in order to build more complex ones. There are five basic operators: *serialization*, *choice*, *loop* and *parallelization* (*fork* and *join*). Operators are used as control structures, which order and sequence simpler CPNPs. Operators are thus characterized by having $P^E = \emptyset$, $T^S = \emptyset$ and $T^T = \emptyset$. In practice, operators map termination places of one CPNP

with initial places of another CPNP.

Serialization operator. The serialization operator is used for specifying a sequential order (doing A before B). It combines two structures by merging two of their places. A termination place of a first CPNP is merged with an initial place of a second one, to obtain a chain of the two CPNPs. This operator can similarly be extended to create a sequence of multiple CPNPs. Figure 3.3 shows a sequence of three CPNPs. Each CPNP represents an ordinary action. The highlighted places are the places that have merged, meaning that these places are the result of merging a termination place of a previous action with the initial place of the following action.

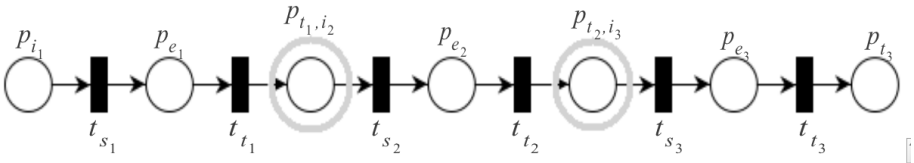

Figure 3.3 Sequence of three CPNPs

Consider the following, extended soccer robot example (taken from [101]): after the soccer robot of Figure 3.2 has reached the ball, it should kick it. The CPNP, composed of these two actions, is built using a serialization operator. It sequences between these two actions by merging the termination place of the *go to ball* action with the initial place of the *kick* action (see Figure 3.4). The actions are denoted by *g* and *k*, respectively. The highlighted circle shows the result of merging the termination place of the *go to ball* action with the initial place of the *kick* action.

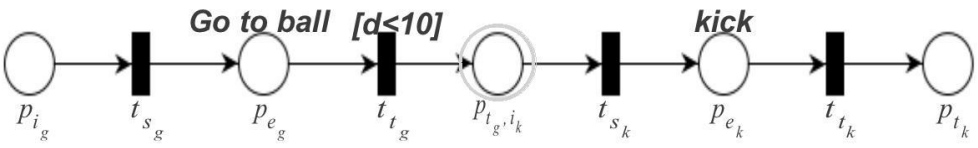

Figure 3.4 Soccer robot: the sequence of *go to ball* and then *kick*

Choice operator. Choice operators are used for specifying a decision-making point. The robot needs to choose which action to perform from among multiple choices, according to the environmental conditions, the robot's senses and desires, and the success or failure in executing a previous action (i.e., doing A or B, but not both). The selection of actions is created by merging the initial places of each optional plan (CPNP structure) and specifying a guard in each starting transition (t_{s_i}).

Figure 3.5 depicts the choice between two different actions. The robot selects either the first action or the second action according to the guards: g_1 and g_2 on transitions t_{s_1} and t_{s_2}, respectively. The highlighted circle shows the result of merging the input places of the first CPNP structure and the second CPNP structure. Note that the robot does not necessarily need to select between two actions only. The choice operator can be used for selecting between multiple CPNP structures.

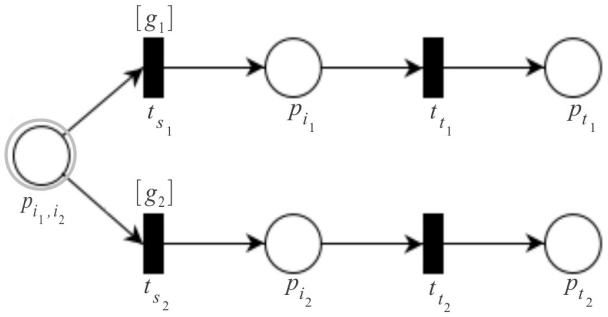

Figure 3.5 A choice between two different actions

Let us revisit the example of the soccer robot. Consider that the robot now should not only attack (i.e., *go to ball* and then *kick*, as shown in Figure 3.4) but also is required to defend in some cases. In fact, the robot should determine what to do according to the environmental conditions.

For the sake of simplicity, the current example (depicted in Figure 3.6) assumes that the choice between attacking or defending is influenced only from the distance between the ball and the robot's own goal. In order to specify appropriate guards we add another variable dgb of type NUM to the variables set V of the soccer robot CPNP; $V = \{d : NUM, dgb : NUM\}$. The highlighted

place p_{i_g,i_d} is the result of merging the initial place of the *go to ball* action with the initial place of the *defend* action (denoted by d) in order to create a *choice operator* that represents the choice between attack or defend. The robot chooses between these actions according to the guards: $[dgb > 200cm]$ or $[dgb \leq 200cm]$.

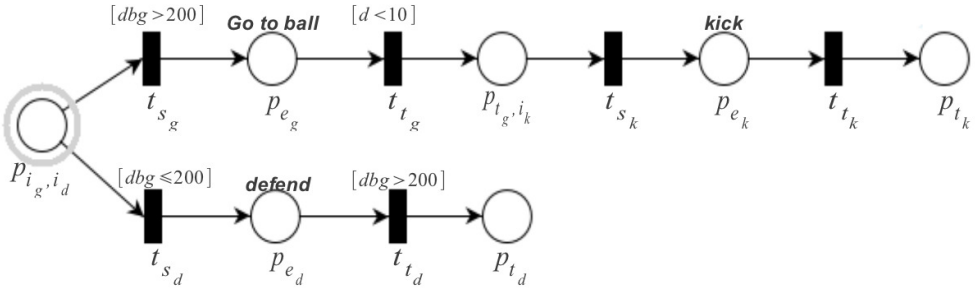

Figure 3.6 Soccer robot: a choice between attack or defend

The current example represents a robot which executes the one-time option that he has chosen, and then ends at Place p_{t_k} or p_{t_d}. Of course, in a real soccer game things work differently. Players need to repeat attack and defense actions several times until the game is over. For the purpose of representing repeated attacks and defenses, we need to use a *Loop operator*.

Loop operator. A loop is a control structure in which the same CPNP is repeatedly executed until some condition is reached (shown in figure 3.7). Basically, the input place and the output place of an action are merged into one place (denoted as $p_{i,t}$). The termination of the loop resembles the Choice operator. Two transitions exit from $p_{i,t}$. Each transition has a guard: one of the transitions has a guard that continues the loop, and the other has a guard that terminates the loop.

In essence, a loop operator consists of a serialization operator and a choice operator. However, instead of merging the termination place of one CPNP with the initial place of a second one by a serialization operator, the loop operator merges the termination place of a CPNP with the initial place of the same CPNP. In addition, the loop operator consists of a choice operator with at least

two guards: a guard on t_s (the start transition) with a condition for the start and continuation of the loop, and a guard with a loop exit condition (the "termination guard"). If the algorithm ends when the loop is terminated, the termination guard will be on a transition that fires tokens into a place that terminates the execution of the algorithm, as shown in Figure 3.7. If the algorithm does not end when the loop is terminated, the termination guard will be on a transition that starts the execution of a different CPNP component that is part of this CPNP.

Figure 3.7 shows a loop of a single action. p_i and p_t are merged into $p_{i,t}$ in order to create a loop. The loop is continued for as long as the guard in t_s is satisfied (i.e., *while* loop). The guard in t_s is marked as $[cond]$, an abbreviation of *condition*. Once this guard is not satisfied anymore, the guard $[!cond]$ that indicates the negative of $[cond]$ is satisfied and terminates the loop. Finally, the token gets into place p_f (f indicates finish) and the algorithm that executes this Petri Net is finished. Note that in this example the algorithm is finished once the loop terminates, but in general the robot can continue performing other tasks after the loop is terminated (as will be shown in Figure 3.8).

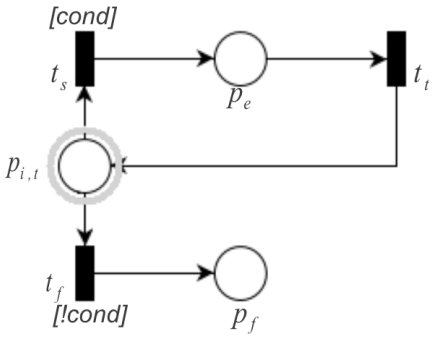

Figure 3.7 A loop of a single action

The Loop operator can also create a loop of multiple actions. For a CPNP structure which contains a sequence of actions $< a_1, a_2, .., a_n >$ that are connected by a Sequence operator (such that a_1 is the first action, a_2 is the second action and a_n is the last action), the Loop operator creates a loop from these actions (doing a_1, then a_2 then $\ldots a_n$ then back to a_1). It does so by merging the places p_{i1} (the first place in the CPNP structure, i.e., the first place of a_1) with p_{tn} (the last place in

the CPNP structure, i.e., the last place of a_n). The Loop operator can create a loop of any order of actions (i.e., sequences of actions, fork and join actions) provided that this order of actions starts with a single initial place and terminates with a single termination place.

Back to the soccer robot example. Figure 3.8 represents a soccer robot that plays until the game's end. The CPNP consists of two loops: attack (i.e., the sequence of *go to ball* and *kick*) and defend. Each loop is running until the appropriate guard is not satisfied anymore. A new variable t is added to V: $V = \{d : NUM, dgb : NUM, t : NUM\}$. This variable indicates the time that passed since the beginning of the game. Once t becomes larger than 600 seconds, the game is terminated.

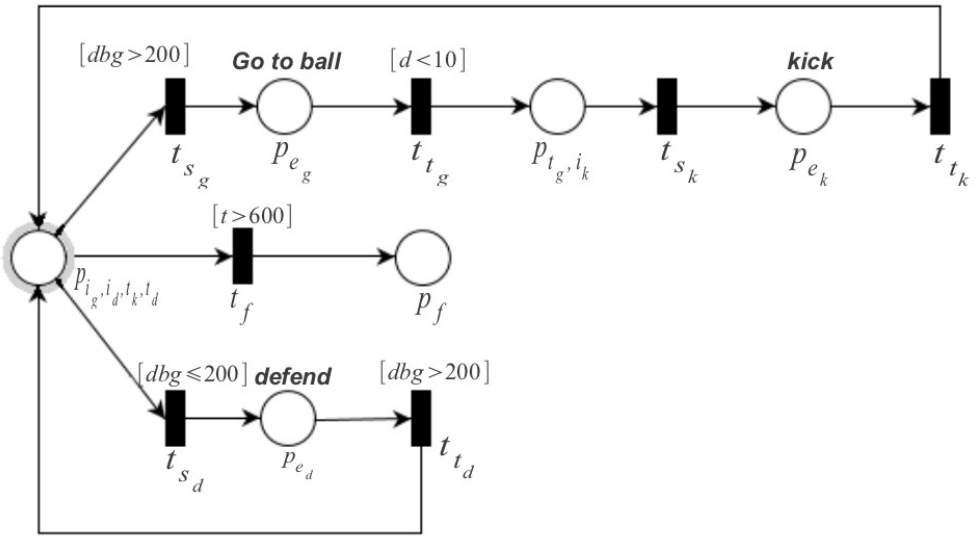

Figure 3.8 Soccer robot - loops

Specifying parallelization. Two operators exist for specifying parallelization: *fork* and *join*. The *Fork operator* splits a thread of execution into multiple actions to be performed in parallel, while the *Join operator* synchronizes multiple parallel actions back into a single thread of execution.

The *Fork operator* splits a process into multiple parallel processes. In essence, it generates multiple threads from a single thread of execution. The Fork operator creates new threads by creating new tokens. The Fork operator is characterized by a new transition, denoted as $t_{fork} \in T^C$.

This transition is added to the CPNP. It splits the tokens and fires them into multiple parallel CPNPs as depicted in Figure 3.9. The input places of this transition are the termination places of the CPNP before it was split. The output places of this transition are the initial places of the new parallel CPNPs.

Figure 3.9 shows a fork structure producing two threads of execution from a single one. The highlighted transition is t_{fork}. This transition removes a token from p_t and generates two tokens: one for p_{i_1} and one for p_{i_2} . Note that the operator can be extended to generate more threads by adding new output places.

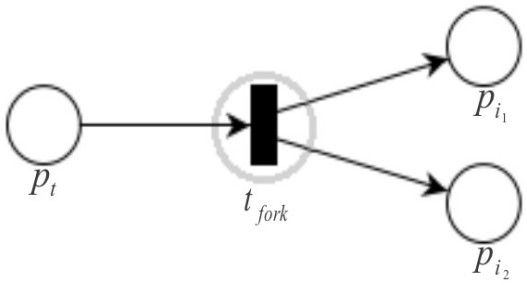

Figure 3.9 A *Fork operator* splits a process into two parallel threads

As opposed to the Fork operator, the *Join operator* merges two or more CPNPs into a single one. This allows for the synchronization (merge) of multiple threads of execution. The Join operator simultaneously consumes multiple threads of execution, and generates a single synchronized thread. Formally, this operator merges the threads by adding a new transition, denoted as $t_{join} \in T^C$. The input places of this transitions are the termination places of each of the multiple concurrent CPNPs. The output places are merged with the initial places of the following CPNPs (after the synchronization).

The join structure is shown in Figure 3.10, for the case of two threads that are synchronized into a single one. The highlighted transition is t_{join}. This transition takes two tokens, one from each input place (p_{t_1} and p_{t_2}), and fires a single token to the output place p_i. As with the Fork

operator, the Join operator can be generalized to synchronize more threads by adding new input places.

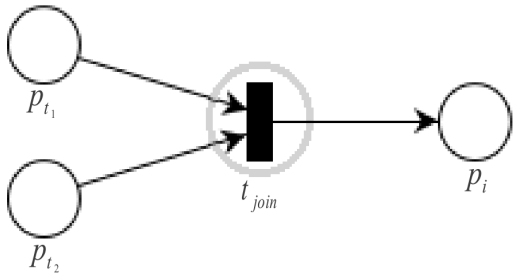

Figure 3.10 Merging two separate threads into a single one using *a Join operator*

Again we return to our soccer robot example. In order to succeed in performing the *go to ball* action, the robot needs to keep track of the ball as he moves towards it. Hence, the action *go to ball* splits into two concurrent actions: *walk to ball* and *track ball*. These actions are indicated in Figure 3.11 by w and t, respectively.

Figure 3.11 shows these concurrent actions. The splitting into two parallel actions (*walk to ball* and *track ball*) is performed by the Fork operator. This operator is expressed by the transition t_{fork}. The operator creates two separate threads, one for each action. These threads join again as the *walk to ball* action is finished. Then the robot kicks the ball.

The Join operator is performed by transition t_{t_t}. Note that in this example the action *track ball* should be terminated as the *walk to ball* action terminates. Hence, transition t_{t_t} has two jobs: it functions as a Join operator and it terminates the execution of *track ball*.

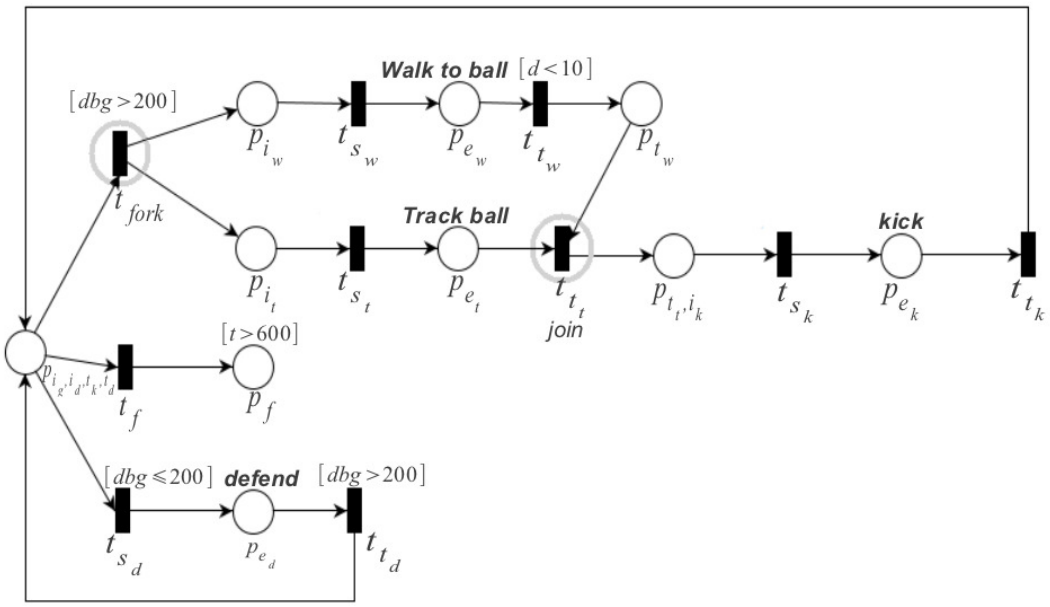

Figure 3.11 Soccer robot - *fork* and *join*

3.3 Re-use and Abstraction through Hierarchical Decomposition

CPNP as a single Petri Net can be created to represent systems which are small and not very complex. However, drawing a CPNP model of a large system as a single net is impractical, due to the risk of it becoming extremely large and inconvenient.

Hierarchical Petri Nets [31, 32, 44, 45, 104] provide the ability to organize the Petri Net into hierarchies of layers. This is done by folding a part of the Petri Net into separate subnets, and replacing these lower-layer subnets with a single transition (called *substitution transition*) to create a higher layer of abstraction in the main Petri Net. According to [32], hierarchical modeling allows for the following:

- Inspecting the modeled system at varying levels of detail;
- Visualizing selected parts of the system (e.g the subnet of one node);

- Facilitating the multiple (re-)use of parts of the model.

Jensen [45] describes how to build Hierarchical Colored Petri Nets by the use of substitution transitions. Substitution transitions add nothing fundamentally new. These transitions merely represent an entire piece of the CPN structure. The piece of CPN that is represented by a substitution transition is executed during the firing of this transition.

According to this method, we can represent a plan for performing an action in a separate CPN, and the plan will be executed when the substitution transition fires. In CPNP we represent the execution phase of an action by an execution place and not by a transition (as described in Section 3.1). Therefore, we change the method of constructing hierarchies. Instead of using substitution transitions, we define special places called *substitution places* and special variables in V. When a transition fires a token to a substitution place, the variable that represents this hierarchy is changed, and the relevant CPNP starts to execute. This methodology will be described in detail in the remainder of this section.

The current section shows how a CPNP model can be organized as a hierarchy of plans (i.e., subplans). Higher (more abstract) plans are called *superCPNPs* and lower (less abstract) plans are called *subCPNPs*. A CPNP can be both a superCPNP (that calls to some subCPNPs) and a subCPNP (which can be called by some superCPNPs). Since it is rare that a robot will have to run identical hierarchical actions concurrently, CPNP relies on the assumption that a single instance of each hierarchical action can be executed by a robot at any given moment.

Again, the CPNP that calls to a subCPNP is called *superCPNP*. The hierarchical call is represented as an ordinary action. However, rather than referring to an atomic action of the robot, the execution place (p_e) actually refers to a subCPNP that performs multiple actions. This place is called *substitution place*.

The hierarchical calls work as follows: each subCPNP has a representative variable of type boolean (denoted as h). The value of this variable will always be identical in each token and it will

be synchronized with each change. This variable is initially set to 0. Once a subCPNP is called by a superCPNP this variable is assigned the value 1 (i.e., $h = 1$ when the appropriate *subCPNP* is executed, and 0 otherwise). The assignment of a variable is done by using an arc expression on the arc that connects from t_s to p_e (shown in Figure 3.12). Each subCPNP is initially marked with tokens of $h = 0$.

Initially, each of the arcs that exit from a place on a subCPNP containing tokens at the initial marking M_0, can move tokens only if h is set to 1. This is expressed by arc expressions, and signifies that tokens can move only after the subCPNP has been called. Each subCPNP has a unique start place and goal place. After the subCPNP reaches its goal, it immediately returns to the start place using a transition which connects the goal place (the last place in the subCPNP) to the start place. The arc from this transition to the start place has an arc expression which sets h to 0. This means that the tokens are fired back to M_0 and h is set again to 0 using transitions and arcs expressions.

As an ordinary action, a hierarchical call is comprised of p_i, t_s, p_e, t_t and p_t connected with arcs (as shown in figure 3.12). In addition, the arc (t_s, p_e) contains the arc expression $h = 1$. This arc expression triggers the appropriate subCPNP. The arc (p_e, t_t) also contains the arc expression $h = 0$. This second arc expression guarantees that a token will be fired only when the hierarchical action terminates.

Figure 3.12 depicts a hierarchical call. p_e is a substitution place. The arc expressions are illustrated by dashed boxes. Figure 3.13 shows an example of a subCPNP at M_0. p_{i_1} and p_{t_2} are start and goal places, respectively. This CPNP represents two actions in a serial order, using a serialization operator. Similar to Figure 3.12, the arc expressions are represented by dashed boxes. The highlighted transition moves tokens from a goal marking to the initial marking.

To illustrate the use of hierarchies, we revisit the CPNP model of the soccer robot and develop a hierarchical CPNP model for this example. The actions *walk to ball*, *track ball* and *kick* can be

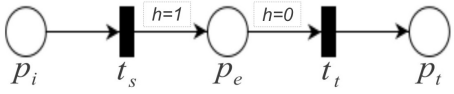

Figure 3.12 SuperCPNP - a hierarchy call

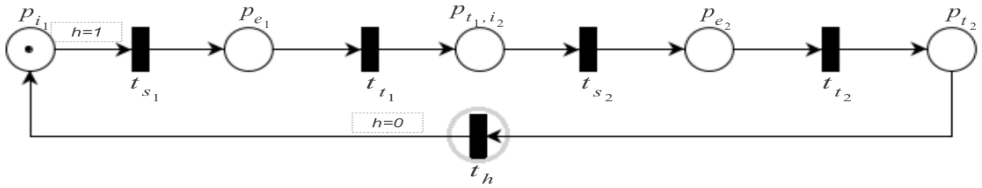

Figure 3.13 An example of a subCPNP at the initial marking

folded into a single hierarchical action named *attack*. In contrast, the *defend* action should be more specified.

Let's define the *defend* action as follows: if the robot is closer to the ball than his goal, the robot will go to the ball and kick, using the *attack* action. Otherwise, he will go to his goal and execute the *play goalie* action. For reasons of simplicity we will omit the specification of *play goalie*.

The hierarchy of our soccer robot example is depicted in Figures 3.14-3.17. Figure 3.14 shows the *play soccer* CPNP, this being the main CPNP. The *play soccer* CPNP hierarchically calls to two subCPNPs: *attack* and *defend*. When the robot executes the *attack* CPNP, he goes to the ball and kicks it towards the goal. When the robot executes the *defend* CPNP he chooses between kicking the ball to the goal (using the *attack* CPNP) or defending his goal (as a goalie). For this selection we define a new variable *drg* of type NUM. This variable indicates the distance between the robot and its own goal.

The current version of the soccer robot example shows the re-use of the *attack* subCPNP. This subCPNP is called twice: once in the *play soccer* CPNP and once in the *defend* CPNP. However, instead of copying the *attack* subCPNP twice into the main CPNP, a separate CPNP is assigned for this part (Figure 3.15). The example illustrates the reducing of space complexity using hierarchical decomposition.

The hierarchical model of this example is shown in Figure 3.17. This model is presented as a directed tree graph called the *instance hierarchy*. In this graph, each node represents an instance of actions and the arcs represent substitution places or execution places.

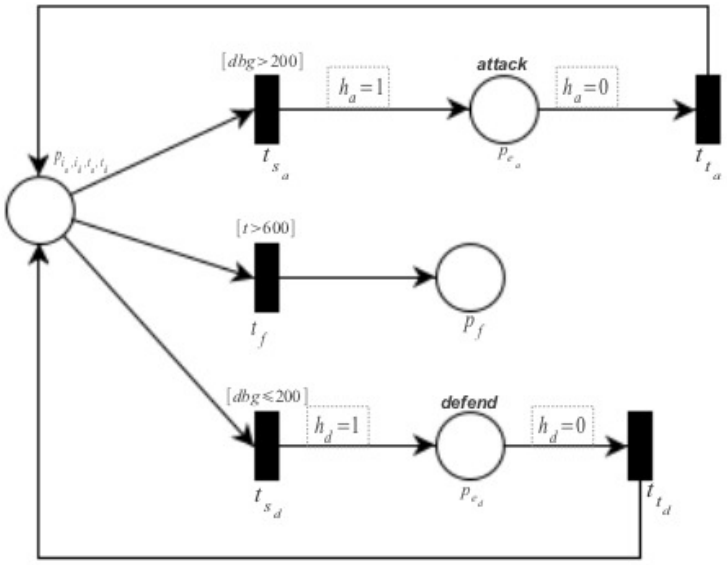

Figure 3.14 Soccer robot - *play soccer* CPNP

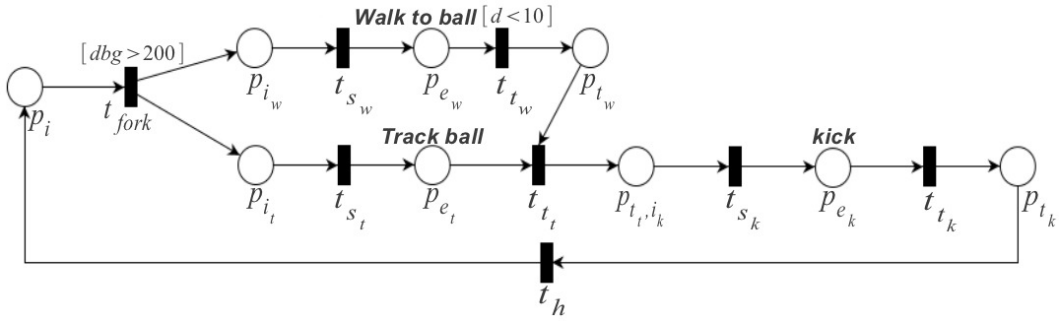

Figure 3.15 Soccer robot - *attack* CPNP

Hierarchical Petri Nets facilitate the readability of the graph and make the Petri Net representation more convenient to build and use. The existing methods for organizing CP Nets into hierarchies of layers [31, 32, 44, 45, 104] create a separate instance of each subnet each time that subnet is called. If a subnet is the value of more than one substitution transition, there will be be

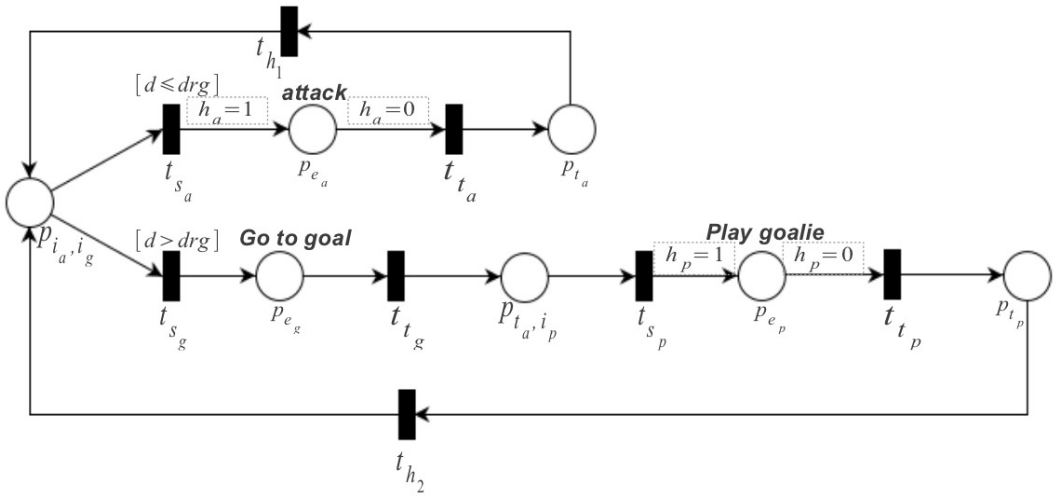

Figure 3.16 Soccer robot - *defend* CPNP

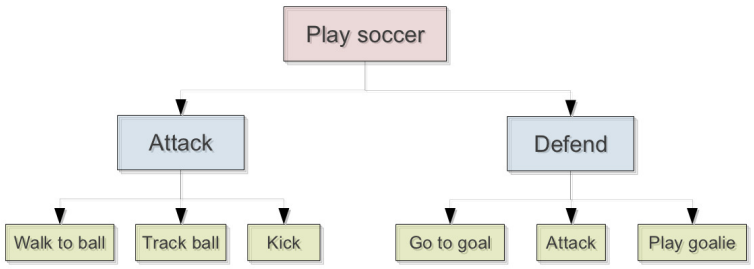

Figure 3.17 Soccer robot - *instance hierarchy*

multiple instances of the same subnet, one for each substitution transition. As a result, the space complexity of Hierarchical Petri nets which are constructed by these methods and non-hierarchical Petri Nets is almost equal.

However, the real power of CPNP in constructing hierarchies lies in the fact that a subCPNP can be the value of multiple substitution places. This advantage does not only facilitate the readability of the graph (since it does not create redundant instances), but it also reduces the space complexity.

3.4 Interrupting a Running Task

An *interrupt* is an event that may suddenly occur during execution of the robot system and whose timing cannot be anticipated. Interrupts can occur at any time during execution. An example of an interrupt would be a legged robot losing its balance and falling. Broadly speaking, two interrupt-handling methods are used to deal with unforeseen events in previous Petri Net-based representations:

1. Re-planning and building a new Petri Net according to the new plan, in execution time [50].

2. Building an interrupt-handling Petri Net for each kind of interrupt and connecting each of the interrupt plans to each of the places that should lead to the interrupt occurrence [20, 23, 101].

The first method for interrupt-handling uses planning methods for generating plans and then compiles them into Petri Nets. Interrupt occurrences lead to a re-planning of the entire system and its compiling into a new Petri Net. This method is very flexible, but a lot of execution time is spent during the processes that generate the plans and compile them into Petri Nets. This is a problem in realtime robot systems. Furthermore, during the execution of these processes, additional interrupts may occur which could cause the system to fail.

In order to handle action failures and interrupts according to the second method, we must connect each of the interrupt handling plans (also: recovery plans) to each of the places that should lead to the interrupt occurrence. The connection between the main plan and the recovery plan is established by connecting a place of the main Petri Net to the initial place of the recovery plan. This connection is established by a transition. When this transition is enabled, it fires tokens if the relevant interrupt has occurred. This constraint is represented by a *labeling mechanism*.

It should be noted that we rely on the assumption that each interrupt-handling Petri Net has an initial place and a goal place (similar to workflow nets). The initial place is the only place that is marked with a token in the initial marking of the interrupt-handling Petri Net, and the goal place

is the only place that is marked when the interrupt has been handled. If the interrupt-handling Petri Net does not have an initial place or a goal place, it should be easy to create these places by creating one place and one transition. In order to create an initial place, the input place of the new transition will be the new place and the output places of this transition will be all the places that participate in the initial marking. In a similar way, we can create a goal place by creating one more place and transition, and connecting all the places that participate in the goal marking (i.e, the marking that indicates that the interrupt has been handled) to the new transition as input places. In addition, we connect the new goal place to the new transition as an output place. More details can be found in [37, 38, 101].

The second method suffers from a number of problems. Firstly, the method is not robust. Even if a single place (in which an interrupt might occur) is not properly connected to the suitable interrupt handling plan, it could cause the whole system to fail.

Secondly, this method can lead to a problematic situation in which an interrupt-handling plan is executed concurrently in multiple places of the graph. This situation may arise when there is an interrupt that may occur in multiple places during the execution of those systems that use this method. It leads to the firing of the transitions labeled with the condition that checks if this interrupt has occurred. Those transitions fire the tokens at the same time to the relevant interrupt-handling Petri Net; therefore, multiple tokens will be in the start place of the interrupt-handling Petri Net. As a result, the interrupt-handling plan will be executed multiple times concurrently, which is highly undesired.

Thirdly, when the recovery plan (i.e., the interrupt-handling Petri Net) is terminated, usually we would expect for the execution to continue from the same place in which the interrupt occurred. In order to guarantee this constraint, we must duplicate each of the recovery plans. The initial place of each copy should be connected by a transition to a different place, that should lead to the interrupt occurrence as well as the termination place. The duplicated plans extremely enlarge the

space complexity.

Space complexity analysis of interrupt-handling according to the first method: Let C be a Petri Net representation without recovery plans. In order to handle interrupts we would add recovery plans (i.e., interrupt-handling Petri Nets) to C according to the second method. In the following we analyze the space complexity of such an addition. In order to analyze the space complexity of interrupt handling according to the second method, we define the following notations:

1. I. Number of interrupts that can occur in C.
2. $|P|$. Number of *places* in C.
3. $|T|$. Number of *transitions* in C.
4. $|H|$. Number of the *places* of a single interrupt-handling Petri Net.
5. $|F|$. Number of the *transitions* of a single interrupt-handling Petri Net.

Theorem 1. *The space complexity of handling interrupts according to the second method is:*

$$O((|P|+|T|)I(|H|+|F|)) \tag{3.1}$$

Proof. The space taken by C is $O(|P|+|T|)$ (i.e., the number of places and transitions in C). The space of each interrupt-handling Petri Net is at most $O(|H|+|F|)$. In order to handle interrupts according to this method, we must duplicate and connect by a transition each of the interrupt-handling Petri Nets to each of the places that should lead to the interrupt occurrence ($|P|$). The number of the interrupt-handling Petri Nets is I and the size of each one is less or equal to $O(|H|+|F|)$. In total: $O((|P|+|T|)I(|H|+|F|))$.

In addition, the connection between C and each interrupt-handling Petri Net is established by adding a new transition for each place that should lead to the interrupt occurrence. That place

will be the input place of the new transition, and the output place will be the initial place of the interrupt-handling Petri Net. Thus, $|P|$ transitions are added. Therefore, the space complexity is: $O((|P|+|T|)I(|H|+|F|)+|P|)$. Since all the numbers are natural numbers, $|P| \leq (|P|+|T|)I(H+F)$. Therefore: $O((|P|+|T|)I(|H|+|F|)+|P|) = O((|P|+|T|)I(|H|+|F|))$. □

Handling Interrupts in CPNP

Through the use of the Colored Petri Net extension, it should be possible to deal with interrupts more efficiently, without creating duplicate CPNPs. In this section we introduce a clear and robust method for handling interrupts using Colored Petri Nets, and we will show how this method reduces space complexity.

CPNPs deal with interrupts by building an interrupt-handling CPNP for each interrupt, and specifying a unique variable of type boolean for each interrupt. This variable is initially set to *false*. The perception component (described in Section 3.1) monitors changes in the environment.

Once an interrupt occurs (such as a case of a robot falling down), it updates the CPNP by changing the appropriate token color variable to *true*. This change will trigger the system into handling the interrupt. When the handling of the interrupt is finished, the variable will return to *false* and the system will continue from the same point at which it initially stopped.

The interrupt-handling CPNP is constructed as follows: an interrupt-handling CPNP is built for each kind of interrupt. Each one of these CPNPs must have at least the following two places: p_{start} and p_{goal}. The initial marking is a single token in the place p_{start}. The place p_{start} is the first place in the CPNP. It connects to a single transition which fires the token only if the appropriate variable is set to *true* (indicating an occurrence of the interrupt). This constraint is expressed by an arc expression on the arc that exits from p_{start}. A token in the place p_{goal} indicates that the interrupt has been handled.

In addition, each interrupt-handling CPNP contains a transition t_{goal}. This transition has a

single input place p_{goal} and a single output place p_{start}. t_{goal} fires a token back to p_{start}. During this firing the token's color is changed, and the relevant variable is set back to *false*. The variable is changed by the use of the arc expression on the arc (t_{goal}, p_{start}).

Each input arc of each transition in the main CPNP has an arc expression. Arc expressions signify that tokens can move in these arcs only if the interrupt variables are *false*. The presence of these arc expressions causes the main CPNP to stop when an interrupt occurs and to wait until the interrupt has been repaired.

To summarize, when an interrupt occurs, its appropriate variable is changed to *true*. This causes the tokens of the main CPNP to stop (since they do not satisfy the arc expressions anymore), while at the same time the token at the interrupt-handling CPNP (external to the main system) will start to move. When a token gets to p_{goal}, t_{goal} fires it back to p_{start} and changes the variable to *false* through the arc expression. In fact, t_{goal} returns the interrupt-handling CPNP back to the initial marking. At this moment, the token in the interrupt- handling Petri Net stops (since it does not satisfy the arc expression anymore) and the tokens in the main CPNP can cause transitions to fire again.

Figures 3.18 and 3.19 describe the treatment of a fallen robot interrupt, with Figure 3.18 representing the main program and Figure 3.19 the interrupt-handling CPNP - *stand up*. In order to handle this interrupt we define a new variable *fall down* of type boolean. This variable is initially set to *false*. Once the robot falls down this variable is changed to *true*. The dashed boxes are the arcs expressions.

The moment the robot falls down, the variable *fall down* is set to *true*. This change will cause the tokens in Figure 3.18 to stop (until they will satisfy the arc expression again) while the token in Figure 3.19 (which currently satisfies the arc expression), will start to move. The moment the token gets to p_{goal} it will be fired by t_{goal}. Since the arc that exits from t_{goal} has an arc expression *fall down* = *false*, the value of the variable *fall down* will change to *false*. Then the tokens in Figure

3.18 will continue running from the point they stopped.

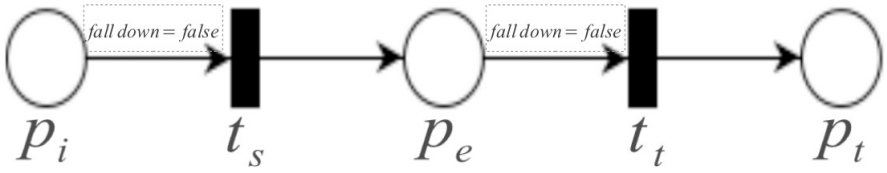

Figure 3.18 An example of a main CPNP

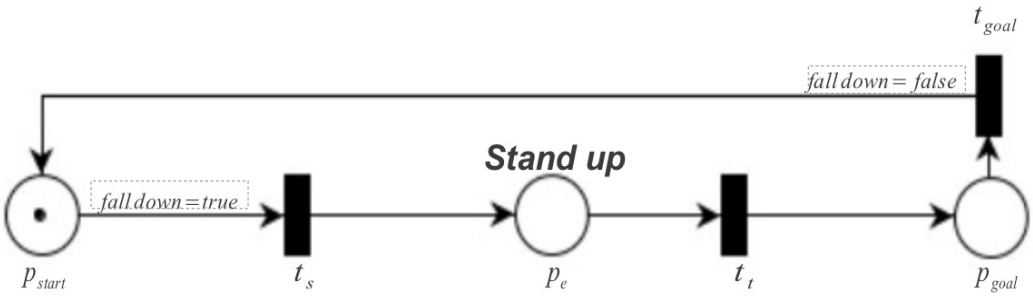

Figure 3.19 An example of an interrupt-handling CPNP

Space Complexity Analysis of Interrupt-Handling According to the New Method

Theorem 2. *The space complexity of handling interrupts according to this method is:*

$$O(|P|+|T|+I(|H|+|F|)) \tag{3.2}$$

Proof. The space complexity of C is $O(|P|+|T|)$. Since the interrupt-handling CPNPs are external CPNPs, we should add the space complexity of those CPNPs to $O(|P|+|T|)$. The space complexity of the interrupt-handling CPNP, is the multiplication of the number of interrupt-handling CPNPs and the space of each interrupt handling CPNP. Since each interrupt has a single interrupt-handling CPNP, the number of interrupt-handling CPNPs is equal to I. The length of each interrupt-handling CPNP is at most $O(|H|+|F|)$. Therefore, the total size of the interrupt-handling CPNPs is $O(I(|H|+|F|))$. In total, the space complexity is: $O(|P|+|T|+I(|H|+|F|))$. □

Theorems 1 and 2 show that the current method reduces the space complexity. It should be noted that we omit the number of arcs from the space complexity analysis, since it is bounded by the number of places and transitions. Moreover, the size of each token is omitted too. The reason is that both methods have a knowledge base and tokens; the size of the knowledge base and each token is constant and roughly equal in both methods. The analysis of the number of tokens in the CPNP (in each marking) is part of the boundedness analysis as discussed in Section 6.1.

3.5 Representing Resources

Representing shared resources (i.e. performing operation *A* only if a particular resource is available) is necessary in order to model both multi-robot and single-robot systems. The representation of shared resources in CPNP is similar to the representation shown in [42] for representing two processes that share three different resources.

In order to represent resources we define additional kinds of places and tokens (called *resource places* and *resource tokens*). These places and tokens signify types of resources. Each resource place and resource tokens' color represent different types of resources. Tokens will appear in resource places when the relevant resources are available. Examples of types of resources are: battery, camera, fuel tanks, etc. The token's color represents the type of resource. These types are taken from a list of predefined types called *Resources*. The set of variables which is associated with the *resource tokens* consists of a single variable *type* which represents each kind of resource. The number of tokens from a specific color represents the number of resources from a specific type. The number of tokens in a resource place is equal to the number of available resources from the same type.

A robot that wants to use a shared resource should fire a token by a transition from the place that represents this resource (or multiple tokens, in case it aspires to use multiple resources from

the same type). Note that the robot can use the resource only when it is available (i.e., at least one token exists in the resource place that indicates the type of this resource). When the robot finishes, a token is fired by a transition to the resource place that indicates the relevant type of resource. This firing indicates the release of the resource.

An example of a shared resource may be a battery. Two tokens of the type *"battery"* represent two batteries. Presence of these two tokens in the resource place "battery" (the place that indicates the resource of type "battery") indicates that the batteries which are represented by these two tokens are available. A firing of one of these tokens **from** the resource place indicates that one of these batteries has been allocated. In contrast, a firing of one of these tokens **to** the resource place indicates that one of these batteries has been released for use.

Consider another example of using a shared resource. Let's assume the existence of a soccer robot that is meant to perform two tasks concurrently: *track ball* (i.e., track movements of the ball) and *track goal* (i.e., observe the goal). In order to perform each one of these tasks, the robot should use a camera. However, the robot can not use the same camera when it performs these two tasks at the same time.

Figure 3.20 depicts the CPNP of this example. In this figure the black token indicates the robot and the white token indicates the camera. The place p_c is the resource place of the camera resource. The existence of a token in this place indicates that there is a camera that is available at this moment. Transitions $t_{c_{tb}}$ and $t_{c_{tg}}$ can fire a white token only. This firing indicates an allocation of a camera for one of the two tasks. In this example only one token exist in p_c, which indicates that the robot has only one camera. Therefore, transitions $t_{c_{tb}}$ and $t_{c_{tg}}$ can not fire concurrently and the tasks cannot be performed concurrently. If the robot will get another camera, one more token will be added to p_c and the robot will be able to perform these two tasks concurrently. When one of these tasks is completed, the camera is released and a white token is fired to p_c using transition $t_{r_{tb}}$ or $t_{r_{tg}}$, respectively.

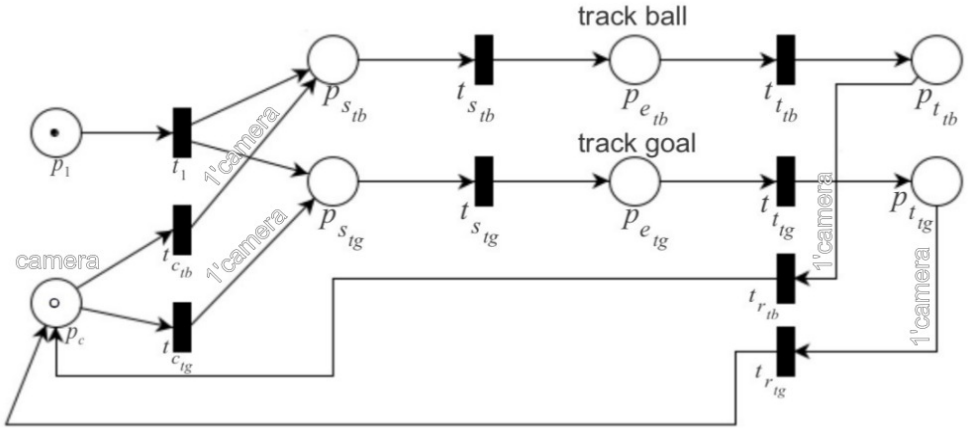

Figure 3.20 An example of shared resources modeled by CPNP

3.6 CPNP: Execution Algorithm of a Single Robot

Until now, we have described how to represent a single-robot plan according to CPNP. This section introduces the algorithms which are used in order to execute a given single-robot CPNP. These algorithms are an extension of the PNP algorithm [101]. The set of algorithms for executing a single-robot CPNP consists of a main execution algorithm 3.1 and some sub procedures (Algorithms 3.2-3.4) which are called up from Algorithm 3.1 during the execution. In Chapter 6, we will discuss the possibility of other algorithms, for reasoning and analyzing a given CPNP for monitoring purposes, and for the validation of plans.

The execution algorithm for single robot-CPNPs is algorithm 3.1 (presented further on in this section). The algorithm executes a given CPNP. It starts from the initial marking M_0 and terminates when a goal marking $M_n \in L$ is achieved. Like the algorithm in [101], the algorithm assumes the availability of a set of implemented actions $Actions = \{a_1, .., a_k\}$ that the robot can execute.

The algorithm has six structures: *input_places*, *output_places*, *Transition*, *Action*, *Token* and *Marking*. These structures are depicted in Figure 3.21. *input_places* and *output_places* are a transition's input places and output places, respectively. Each transition $t \in T$ is represented by the *Transition* structure. This structure is composed of the following:

1. type - type of t (start, end or connector).

2. action - an action $a \in Actions$ which is associated with t. t should start or terminate a according to its type. if t is of type *connector* this field will be empty.

3. $input\_places$ - a set of input places of t. Mathematically, this set is denoted as $\bullet t$.

4. $output\_places$ - a set of output places of t. Mathematically, this set is denoted as $t\bullet$

Each action $a \in Actions$ is represented by the *Action* structure that contains two methods: *start* and *end*. *start* is responsible for starting the execution of a and *end* terminates it. The events during which a is started or terminated are denoted as $t.a.start$ and $t.a.end$, respectively, where t is the transition associated with each event. The *start* and *end* methods are executed just after the transition t fires (as described in Section 3). The characters $<,>$ indicate the beginning and the termination of a list, respectively.

In Section 3.1 we defined two types of tokens in CPNP: *robotToken* and *resourceToken* (depicted in Figure 3.21, lines 5-6). These types are distinguished by their color. The first type represents the knowledge of the robot, and the second type represents a specific type of resource (denoted as e_i). Since the value of each variable is identical for each *robotToken*, it is wasteful to maintain a separate data value for each token. Another reason for not maintaining a separate data value for each token is that any change in the robot's knowledge leads to the updating of all tokens' data values.

In light of the above and in order to minimize the space complexity, we define a global knowledge base *kb* instead of having each token contain all values of the variables in V. This knowledge base is a data structure that contains the value of each variable in V. Each token has a pointer to this knowledge base.

The knowledge base *kb* is a database that contains data about the environment from the perspective of the robot (i.e., the robot's beliefs). It consists of the values of the variables in V (*execute*

procedure, Algorithm 3.1, line 1). At the beginning each variable is initialized with a predefined value. The knowledge is obtained from the robot's sensors and it changes according to changes in the environment. This knowledge defines the tokens' color. This means that the color is determined by the value of the variable in V.

Each *robotToken* is represented by a *robotToken* structure (Figure 3.21, line 5). This structure consists of a pointer to the knowledge base *kb*. The knowledge base *kb* contains values for each variable in V. The $\longrightarrow$ character indicates a pointer.

As mentioned in Section 3.5, the *resourceTokens* represent types of resources (e.g., battery, camera etc), which are taken from a list of predefined types called *Resources* (Figure 3.21, line 4), by a single variable *type*. The value of this variable is constant and can not be changed during the execution. Each marking is represented by a *Marking* structure. This is a hash table which maps each place to a list of tokens, where each token is represented by a *robotToken* structure or a *resourceToken* structure.

Domains:

1: Actions = $\{a_1, \ldots, a_k\}$: Set of implemented actions.
2: TrType = $\{start, end, connector\}$
3: $V = \{v_1, .., v_l\}$
4: Resources = $\{r_1, .., r_n\}$

Structures:

1: $input\_places$: $\{p_1, .., p_n\} \subseteq P$
2: $output\_places$: $\{p_1, .., p_m\} \subseteq P$
3: Transition: $<t \in TrType, a \in Actions, input\_places, output\_places>$
4: Action: $<$ start(), end() $>$
5: robotToken: $< o \longrightarrow kb >$
6: resourceToken: $< e_i \in Resources >$
7: Marking = hash table (keys = places, values = list of tokens).

Figure 3.21 CPNP execution algorithm - domains and structures

The main procedure is *execute* (Algorithm 3.1). This procedure is parameterized by a CPNP. The procedure starts from M_0 and generates new markings until it reaches a marking that belongs to

L (lines 2-4). New markings are generated through firing (explained below). The current marking is represented by the variable *currentMarking*. This variable is built around a *Marking* structure and it changes when the current marking itself is changed.

First, the *execute* procedure creates a knowledge base *kb* which contains an initial value for each variable in V (line 1). Note that some of the variables in V are boolean variables for representing an occurrence of interrupts, and hierarchies. These variables are initialized with *false*. The CPNP execution algorithm assumes that the *kb* is constantly updated through the robot's operating system.

For each transition $t \in T$ the *execute* procedure checks whether an interrupt occurs (line 5). If positive, the procedure sets to *true* the boolean variable in the *kb* which indicates an occurrence of this interrupt (line 6). Then, the *execute* procedure checks if t is enabled, using the *EnableTransition* procedure (line 8). If t is enabled, the *execute* procedure performs the following steps:

1. Activating or deactivating the action associated with t, according to the type of t, using the *handle transition* procedure (line 9).

2. Firing t and generating a new marking, using the *fire* procedure (line 10).

3. Terminating the execution of the algorithm once a marking that is defined as a *goal marking* is obtained (lines 11–12).

The *EnableTransition* procedure (Algorithm 3.2) is parameterized with a transition t as well as the current marking, and returns *true* if the guard on t is satisfied (line 1) and t is enabled (according to Definition 4) in the current marking (lines 2–10). If this procedure returns *true* it means that t is ready to fire.

Lines 2–10 check if t is enabled in accordance with the current marking. The procedure checks if the list of tokens in p_i contains at least the list of tokens that is defined in $E(p_i,t)$, for each input place p_i of t (line 2). This is expressed by checking if any place exists that does not contain the list

Algorithm 3.1 CPNP execution algorithm - single robot, *execute*

procedure execute (CPNP $(P,T,A,\Sigma,V,C,G,E,M_0,L)$)

```
1:  create knowledge base kb from V
2:  CurrentMarking ← M0
3:  while CurrentMarking ∉ L do
4:     for all t ∈ T do
5:        if an interrupt i occurs and kb.v_i ≠ true then
6:           kb.v_i ← true // v_i is a boolean variable that indicates an occurence of interrupt i
7:        end if
8:        if EnableTransition(t,CurrentMarking) then
9:           HandleTransition (t)
10:          CurrentMarking ← fire(t,CurrentMarking)
11:          if CurrentMarking ∈ L then
12:             exit
13:          end if
14:       end if
15:    end for
16: end while
```

of tokens defined in the arc expression (line 7). If so, the procedure returns *false* (line 8). Note that each arc expression $E(p_i,t)$ defines a list of tokens.

Algorithm 3.2 CPNP execution algorithm - single robot, *EnableTransition*

procedure EnableTransition(t,CurrentMarking)

```
1:  if G(t) = true then
1:     // checks if the guard on t is satisfied
2:     for all p_i ∈ t.input_places do
3:        if CurrentMarking(p_i) ⊉ E(p_i,t) then
4:           return false
5:        end if
6:     end for
7:  else
8:     return false
9:  end if
10: return true
```

If the transition t is enabled the *HandleTransition* procedure is started (Algorithm 3.3). The procedure activates or deactivates the action related to t (if it exists) according to the type of t.

Algorithm 3.3 CPNP execution algorithm - single robot, *HandleTransition*

procedure HandleTransition(t)

1: **if** t.t = start (// if the type of t is start) **then**
2: t.a.start() // activate action a
3: **else if** t.t = end (// if the type of t is end) **then**
4: t.a.end() // deactivate action a
5: **end if**

The *fire* procedure (Algorithm 3.4) executes the firing of t. In practice, this procedure changes the current marking according to the firing rules defined in Section 2.2. This means that this procedure consumes $E(p_i,t)$ tokens from each input place p_i (lines 1–2) and produces $E(t,p_o)$ to each output place p_o (lines 4–5). The operation $CurrentMarking(p_o)+E(t,p_o)$ (line 5) produces tokens for p_o according to the arc expression $E(t,p_o)$. This operation also changes the values of the variables according to $E(t,p_o)$.

Note that in a case of hierarchical decomposition, the firing of t starts the execution of the hierarchical action by changing the tokens' color. In fact, the arc from t to the substitution place has an arc expression that changes this color by setting the value of the boolean variable that represents this hierarchical action to *true*. Once the execution of the CPNP that represents the hierarchical action will reach its goal, it will change the boolean variable to *false* using an arc expression. Changes in the variables are recorded in the knowledge base. Since each token has a pointer to the knowledge base, the tokens get notified of any changes in the knowledge base.

Algorithm 3.4 CPNP execution algorithm - single robot, *fire*

procedure fire (t,CurrentMarking)

1: **for all** $p_i \in t.input\_places$ **do**
2: $CurrentMarking(p_i) = CurrentMarking(p_i) - E(p_i,t)$
3: **end for**
4: **for all** $p_o \in t.output\_places$ **do**
5: $CurrentMarking(p_o) = CurrentMarking(p_o) + E(t,p_o)$
6: **end for**

Summary. The current chapter introduced the CPNP representation for single-robot architectures. We elaborated on the CPNP basic building blocks; those basic CPNP structures that are used to represent each action and the operators that are used to connect between the actions. Next, we presented an efficient method for constructing hierarchies, which does not only facilitate the readability of the Petri Net but also reduces space complexity.

A mechanism to deal with interruptions was provided. This mechanism is especially robust since it allows for handling interruptions without connecting any place of the CPNP with the interrupt-handling CPNP, no matter the CPNP's state. We proved that the use of this mechanism reduces space complexity. We showed how to represent shared resources in CPNP. Finally, we introduced the algorithms for executing a given CPNP.

Chapter 4

Theoretical Analysis of Multi-Robot Petri Net Plan Representations

Before extending CPNP to handle multi-robot plans, it would be prudent to examine general approaches to multi-robot plan representations. There are a wide variety of Petri Net-based multi-robot representations [20–22, 50, 53, 55, 60, 65, 99–101]. Thus far, their relative strenghs and weakness have not been investigated however. This chapter includes an analysis of these representations that examines their space complexity when representing a given multi-robot plan.

We classify the mentioned multi-robot representations along two dimensions:

1. The type of Petri Net used for representing multi-robot plans (P/T Net or CP Net);
2. Whether individual or joint states are represented.

We will show that choices along these two dimensions involve significantly different space requirements. Furthermore, we will show in which cases either individual or joint state representation is preferable. The analysis reveals that combining Colored Petri Net with the correct choice of state representation yields the best results on space complexity. Building on the insight gained from this analysis, Chapter 5 will introduce the CPNP representation for multi-robot systems.

The current chapter is organized into three sections. Section 4.1 describes the joint and individual state representations. In order to analyze the existing representations for a given plan G, we divide G into three parts. These parts are analogous to the three existing types of robot dependencies that will also be described in this section. Section 4.2 presents the analysis of existing multi-robot representations and their scalability. Finally, Section 4.3 summarizes the chapter.

4.1 Joint State Representation vs. Individual State Representation

There are two widespread representation approaches for representing multi-robot systems: *individual state representation*, and *joint state representation*. The most popular approach is individual state representation [21, 22, 50, 53, 55, 60, 65, 100, 101]. This approach uses separate symbols to represent the separate states of separate robots. In contrast, joint state representation [99] uses a single symbol to represent the state of all participants together.

In individual state representation, each symbol represents the state of a single robot (role). Particular to Petri Net is that each robot has separate places and tokens in the shared task's net, and different markings distinguish the status of the mission. Essentially, the net for each robot is built separately and merged with the other nets. Figure 4.1 presents two different Petri Nets (a and b), each one of them executed by a different robot and each independent of the other.

By contrast, in joint state representation one symbol is required for representing the status of all team members. With joint state representations in Petri Net, multiple robots share the same place which denotes their joint state in the shared task.

Figure 4.2 shows the joint state representation based on the two individual Petri Nets shown in Figure 4.1. Each joint place indicates two individual places: one from Petri Net a and one from Petri Net b. For example: a token in place $P_{a_1} \wedge P_{b_1}$ represents a token in P_{a_1} and a token in P_{b_1} in

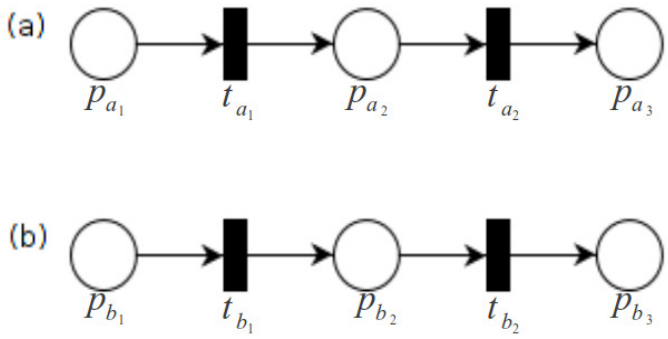

Figure 4.1 Example of two Petri Nets based on individual state representation

the two individual Petri Nets above. Each transition in Figure 4.2 represents two transitions (one from each Petri Net) in Figure 4.1. For example: transition $t_{a_1}(F) \wedge t_{b_1}(\neg F)$ represents the firing of the token in the case of transition t_{a_1} having fired the token (in Figure 4.1, Petri Net *a*) and transition t_{b_1} not having fired the token yet (in Figure 4.1, Petri Net *b*). Note that F indicates that the transition fired, while $\neg F$ indicates the opposite.

Consider the following *Open door* example: two robots are required to jointly open a door. The plan of this example consists of a single action *Open door*. This plan is performed by two robots: R_1 and R_2. Figure 4.3 depicts an individual state representation which represents the example's plan. This individual state representation is built according to PNP [101]. In individual state representation, each place represents the status of one of the robots participating in the mission (individually) and the marking represents the task progress status. The upper and lower Petri Net components of Figure 4.3 represent the states of R_1 and R_2, respectively. When the two robots are ready to open the door, one token appears in place $p_{R_{1_1}}$ and one token apears in place $p_{R_{2_1}}$. Only when each of these places contains at least one token, transition t_{sync} fires the tokens and each of the two robot starts the *Open door* action.

Figure 4.4 depicts a joint state representation that represents the plan of the *Open door* example. As mentioned previously, in a joint state representation each place represents the status of all the robots participating in the mission. Hence, each place represents the task progress status.

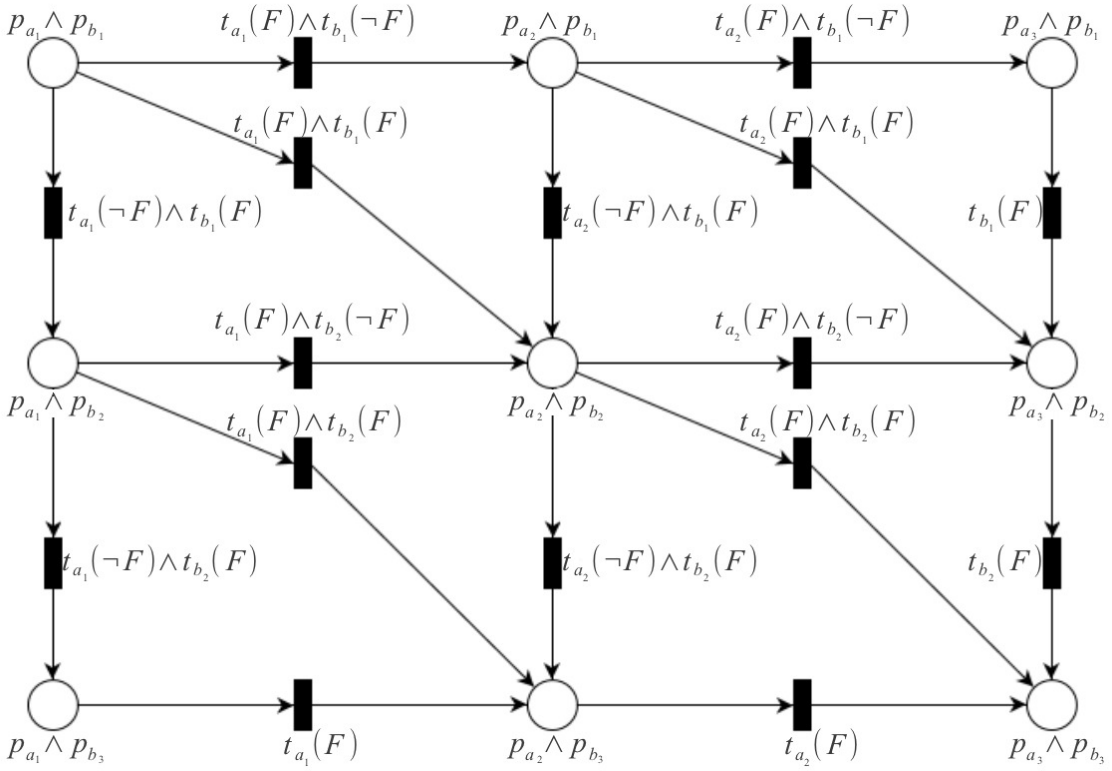

Figure 4.2 Example of joint state representation (base on the two individual Petri Nets of Figure 4.1).

Therefore, if the two robots are ready to open the door, a token should appear in the place that indicates that R_1 and R_2 are ready to open the door (place $p_{R_1 \wedge R_{21}}$). Then the token is fired to place $p_{R_1 \wedge R_{22}}$ and the robots execute the *Open door* action. Once the robots finish, the token is fired to place $p_{R_1 \wedge R_{23}}$.

Until now, most Petri Net-based multi-robot architectures made use of individual state representation [21,22,50,53,55,60,65,100,101]. Gutnik and Kaminka [36] examined the strengths and weaknesses of each type of state representation for conversation modeling. There, the Petri Nets represent the communication acts that two or more agents should take, to carry out a conversation according to a specific protocol. Their analysis showed that for tracking conversations (e.g., for decision protocols, task allocation between robots, etc.), a joint state representation that is based

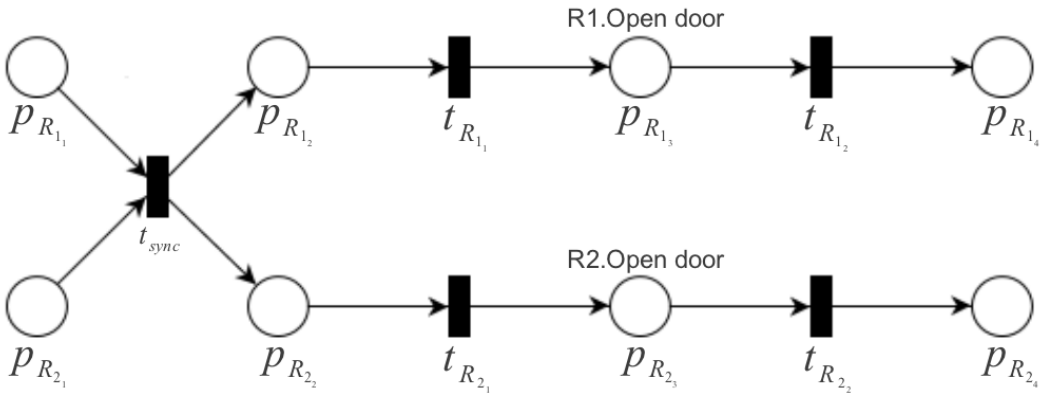

Figure 4.3 *Open door*: example of individual state representation

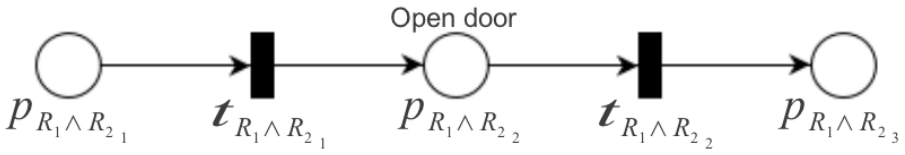

Figure 4.4 *Open door*: example of joint state representation

on Colored Petri Nets yields significant advantages in space complexity.

However, the overall strengths and weaknesses of individual and joint state representations regarding different tasks and architectures have not been examined. Figures 4.1-4.4 illustrated that the two types of state representation for representing the same multi-robot plan involve widely divergent space requirements with no obvious preferable option. In Figures 4.1-4.2 individual state representation is preferable in terms of space complexity, while in Figures 4.3-4.4 the same is true for joint state representation.

Types of Inter-Robot Dependencies. Analysis of individual and joint state representations should distinguish three existing types of robot inter-dependencies, *independence*, *weak dependency* and *strong dependency*. The dependencies between these robots define the type of coordination between them:

1. Independence: each robot acts as individual, without any commitment to the other robots. In

practice, a multi-robot plan in which each robot is independent consists only of the union of the single robot plans of each robot.

2. Weak dependence of robot *A* on robot *B*: robot *A* cannot start an action until robot *B* reaches a certain state. Weak dependency presents a precedence relation among the actions of two plans.

3. Strong dependence: robot *A* is weakly dependent on robot *B* and robot *B* is weakly dependent on robot *A*. Robot *A* and robot *B* are strongly dependent when they are executing actions *C* and *D* such that the execution of actions *C* and *D* is synchronized.

These dependencies have been defined above for two robots, but they can be expanded for any number of robots. The weak dependence can be expanded, according to the following logical dependencies:

1. A weak dependence of robot *A* on robot *B* and robot *E*

2. A weak dependence of robot *A* on either robot *B* or robot *E*

A strong dependence of a group of multiple robots, means that the robots should be synchronized while they are performing their actions. The definition of strong dependence of multiple robots is: every robot in the group is weakly dependent on all the others.

4.2 An Analysis of Petri Net Representations for Multi-Robot Systems

This section introduces the space complexity analysis of existing multi-robot representations. As mentioned, we classify the existing multi-robot representations along two dimensions:

1. The technique used for representing multi-robot plans (P/T nets or CP nets).

2. Individual state representations or joint state representations.

Given a multi-robot plan G with R robots, this section will present an analysis of the space complexity of each type of representation (i.e., individual state representations based on P/T Nets, individual state representations based on CP Nets, joint state representations based on P/T Nets and joint state representations based on CP Nets). In some plans multiple robots can have the same roles. This means that they execute exactly the same part of the plan and perform the same actions. Definition 8 defines the meaning of *role*. In our analysis we will regard the number of different roles in the plan. This number is denoted by L ($L \leq R$).

Definition 8. Role *is the specific part relative to each robot in the common task. If two robots have the same role, it means that they execute exactly the same actions (they have the same code).*

In order to compare representations based on P/T Nets to representations based on CP Nets, first we should calculate the space complexity of the colors added to the tokens. Although CP Nets are computationally equivalent to Petri Nets [44] (every P/T Net can be translated to CP Net and vise versa), the space complexity of a representation of a multi-robot plan based on CP Nets is different than a representation of the same plan based on P/T Nets. In CP Nets, tokens maintain data (color), as opposed to P/T Nets. Therefore, instead of one bit which is required to represent a token in a P/T Net, CP Nets use multiple bits per token. For instance in CP Net based representations for multi robot plans, each token may contain an ID of a robot. Therefore $\log R$ bits are required in order to differentiate R robots. Despite this in the Sections 4.2.1, 4.2.2 and 4.2.3, we will show that in some cases, choosing CP Nets to represent multi-robot plans reduces the space complexity.

Sections 4.2.1, 4.2.2 and 4.2.3 introduce the space complexity analysis of the existing representations.

The analysis is carried out according to the dependencies between the robots. Section 4.2.1 presents the space complexity analysis where the robots are independent. Section 4.2.2 presents the space complexity analysis for weak dependency and Section 4.2.3 does so for strong dependency. Finally, Section 4.3 summarizes the space complexity analysis of multi-robot representations by summing up the space complexity analyses presented in Sections 4.2.1, 4.2.2 and 4.2.3.

4.2.1 Space Complexity Analysis When Each Robot is Independent

This section introduced the space complexity analysis for representations of multi-robot plans, of R robots in which the robots are independent. In these plans robots act as individuals. They do not have commitments to the other robots and they do not need to coordinate and cooperate with other robots in order to perform a mission. The robots are divided into L different roles. The multi-robot plan consists of L independent single-robot plans (one for each role). In the following analysis we assume that the representation of a multi-robot plan has K tokens. If the representation is bounded then K is a finite number bounded by a constant natural number. The boundedness property will described in Chapter 6.

These multi robots plans consist of a union of all single-robot plans (one for each role). Each single-robot plan is made of a number of actions and a number of operators. Let P be the size of the maximal plan with the maximal size for any role. The number of actions in P is denoted as A, and the number of operators is denoted as E. The size of the representation of a single atomic action and operator is bounded. Therefore $P = O(A+E)$.

Space complexity of individual state representations

Theorem 3. *Given a multi-robot plan G with R robots, the space complexity of an individual state representation based on P/T Nets is:*

$$O\left(PR+K\right) \tag{4.1}$$

Proof. G is composed of a single-robot plan for each role. The size of the representation of each single robot plan is $O(P)$. In individual state representations which are based on P/T Nets, each place can represent only a single robot. Therefore the representation of G is composed of R separate single-robot representations (one for each robot). Thus the size of the representation of G is $O(PR)$. We should add the size of K tokens to this size. The size of each token in a P/T Net is one bit and the size of K token is K bits. In total, the space complexity is: $O(PR+K)$ □

Theorem 4. *Given a multi-robot plan G with R robots, the space complexity of an individual state representation based on CP Nets is:*

$$O(PL+K\log R) \tag{4.2}$$

Proof. G is composed of a single-robot plan for each role. The size of the representation of each single robot plan is $O(P)$. In individual state representations which are based on CP Nets, each place can represent a single role. Therefore the representation of G is composed of L separate representations (one for each role). Thus the size of the representation of G is: $O(PL)$. In addition, the representation of G has K tokens (as assumed). Each token requires $\log R$ bits (explained above). In total, the space complexity is: O $O(PL+K\log R)$. □

Space complexity of joint state representations

Theorem 5. *Given a multi-robot plan G with R robots, the space complexity of a joint state representation based on P/T Nets is:*

$$O\left(P^R+K\right) \tag{4.3}$$

Proof. In joint state representations which are based on P/T Nets, each place indicates the joint state of all the robots participating in the system. Because the robots are independent of each other, each place in a joint state representation represents a possible combination of a single place from each single-robot representation (i.e. a representation of a single-robot plan). Therefore, the

size of the representation of G is: $O\left(P^R\right)$. We should add the size of K tokens which is K bits to this size (as explain above). In total, the space complexity is: $O\left(P^R + K\right)$ □

Theorem 6. *Given a multi-robot plan G with R robots, the space complexity of a joint state representation based on CP Nets is:*

$$O\left(P^L + K\log R\right) \tag{4.4}$$

Proof. G is divided into L sub plans (one for each role). Each place in a joint state representation represents a possible combination of the state of each sub plan. Therefore, the size of the representation G is: $O\left(P^L\right)$. We should add the size of K tokens which is $K\log R$ bits to this size (as explain above). In total, the space complexity is: $O\left(P^L + K\log R\right)$ □

Table 4.1 summarized theorems 3-6. It presents the space complexity of different approaches for representing the same multi-robot plan. The plan consists of R robots and L roles. The robots are independent and act as individuals without the need of communication between the robots. From this table we can infer that the individual state representation is preferable in order to represent independent robots. Figures 4.1 and 4.2 present an individual state representation and a joint state representation of a multi-robot plan, which consists of two independent robots respectively. The figures demonstrate that individual state representations have significantly less space complexity, when representing a plan of independent robots.

Regarding the choice of either P/T Net or CP Net: The choice of CP Net is preferable in terms of space complexity for the following reasons:

1. The $O(\log R)$ factor can be reduced to $O(1)$, since $L \leq R$ and in practice, the number of robots R is bounded.

2. We can reduce the space complexity of the tokens in CP Nets from $O\left(K\log R\right)$ to $O\left(K - R + R\log R\right)$ by the following implementation: in most representations $R \leq K$, therefore we can reduce

	Non CP net	**CP net**
Individual state representation	$O(PR+K)$ (Theorem 3)	$O(PL+K\log R)$ (Theorem 4)
Joint state representation	$O\left(P^R+K\right)$ (Theorem 5)	$O\left(P^L+K\log R\right)$ (Theorem 6)

Table 4.1 Space complexity analysis when the robots are independent

the space complexity of the K tokens by building a representation in a way that R tokens will maintain the ID of all robots (cost: $O(R\log R)$), and the rest will have a pointer to the relevant ID (cost: $O(K-R)$). In total, representing K tokens according to this method will cost: $O(K-R+R\log R)$.

3. An additional reason in choosing CP Nets over P/T Nets is that by the use of CP Nets we can efficiently represent interrupts, hierarchies and shared resources, as described in Section 3.

4.2.2 Space Complexity Analysis of The Weak Dependence Operator

This section discusses multi-robot plans in which weak dependence exists. The part of the representation that represents a single weak dependence operator has a bounded size denoted as X; this size includes all Petri Net ingredients for a single weak dependence operator. This analysis regards only the number of sub plans containing weak dependence operators, which exist in total in a given multi robot plan (denoted as G). The number of unique weak dependence operators in G is denoted by W. Different weak dependence operators connect between different roles. Consider the following example: a plan consists of four robots: R_1, R_2, R_3, R_4. R_1 and R_3 have the same role and R_2 and R_4 have the same role. The plan consists of a weak dependence operator between the two different roles (i.e., a weak dependence operator between R_1 and R_2 and a weak dependence operator between R_3 and R_4). In this example $W=1$, since the plan has two identical weak

dependence operators, which equals a single unique weak dependence operator.

Space complexity of individual state representations Figure 4.5 presents an individual state representation of a weak dependence operator of the following example: two robots R_1 and R_2 and four actions A, B, C and D. R_1 performs Action A, and then performs Action B; and R_2 performs Actions C and then it performs Action D, but R_2 cannot perform Action D until R_1 finishes performing Action A.

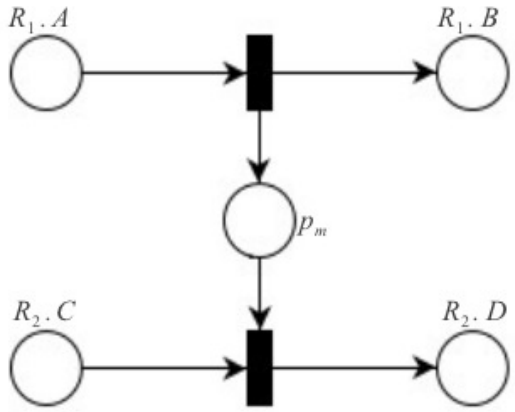

Figure 4.5 Individual state representation of weak dependence of R_2 on R_1

Unlike CP Nets, in P/T Nets, tokens do not have values and we cannot differentiate between tokens. Therefore, while we can represent two robots with the same role in a single CP Net by two different tokens (one representing the first robot and the other representing the second), we cannot do this in a P/T Net based representation. Therefore, in a P/T Net based representation, two robots with the same role will be represented by two separate but identical P/T Nets.

Back to the example of four robots and two roles and a weak dependence operator between the two different roles. A P/T Net based representation of this example is depicted in Figure 4.6. This figure has two identical P/T Net components: the left one represents the weak dependence operator of R_1 and R_2, and the right P/T Net component represents the weak dependence operator of R_3 and R_4.

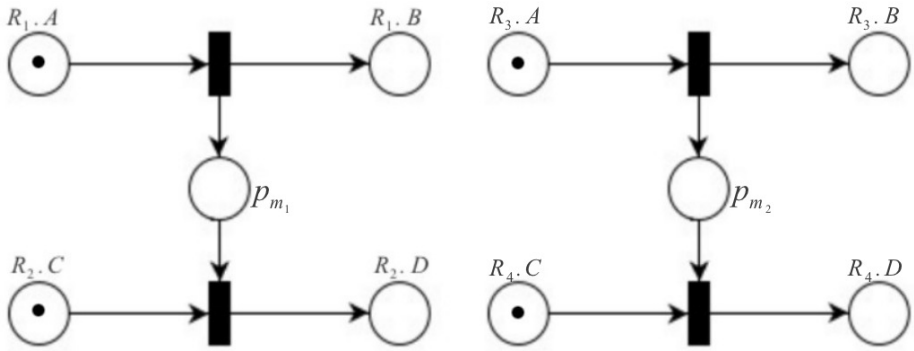

Figure 4.6 A P/T Net-based individual state representation of weak dependence of R_2 on R_1 and weak dependence of R_3 on R_4

Figure 4.7 presents a more space-efficient CP Net-based representation of this example. The figure has a single CP Net component that represents the weak dependence operator of the two roles and four token colors that represent the robots.

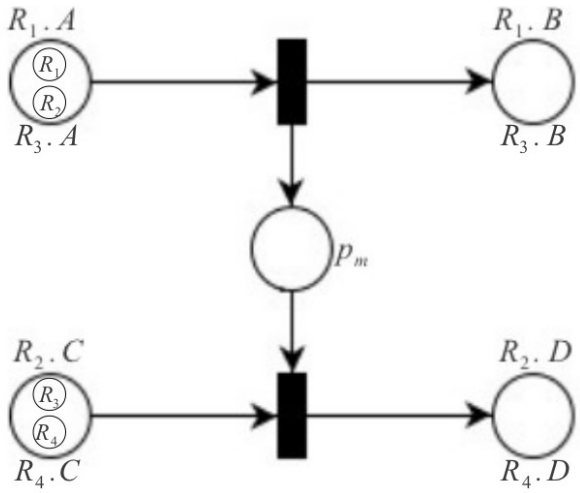

Figure 4.7 A CP Net-based individual state representation of weak dependence of R_2 on R_1 and weak dependence of R_3 on R_4

Theorem 7. *Given a multi-robot plan G with R robots, L roles, W different weak dependence operators, and X as the upper bound of the size of a single weak dependence operator, the space*

complexity of the weak dependence operators in P/T Net-based representations of G is:

$$O\left(\frac{R}{L}WX\right) \tag{4.5}$$

Proof. In individual state representations that are based on P/T Nets, each place can represent only a single robot. When there are identical (duplicate) weak dependence operators (as described in the example above), the individual state representation should represent them in separate representations (one for each operator). W weak dependence operators connect L roles to each other. To determine the total number of operators, we need to examine how many duplicates of these roles exist. In the worst case there would be W operators of each such duplication. The maximal number of duplicates occurs when the robots are divided uniformly into the L roles, thus $\frac{R}{L}$ duplicates. The worst case number of operators would therefore be $O\left(\frac{R}{L}W\right)$, and the space complexity is $O\left(\frac{R}{L}WX\right)$. Naturally, a smaller L would result in greater space complexity. □

In the example above (Figure 4.6) there are four robots ($R = 4$), two different roles ($L = 2$), and two identical weak dependence operators which equals one unique weak dependence operator ($W = 1$). Therefore, the number of the identical (duplicate) weak dependence operators is bounded by $\frac{R}{L}$=$\frac{4}{2}$=2 and the space complexity is $O(2X)$.

Theorem 8. *Given a multi-robot plan G with R robots, L roles, W different weak dependence operators, and X as the upper bound of the size of a single weak dependence operator, the space complexity of the weak dependence operators in CP Net-based representations of G is:*

$$O(WX) \tag{4.6}$$

Proof. In individual state representations which are based on CP Nets, only the unique weak dependence operator is represented separately. Therefore, the space complexity required to represent W weak dependence operators is $O(WX)$. □

Space complexity of joint state representations

Theorem 9. *Given a multi-robot plan G with R robots, L roles, W different weak dependence operators, and X as the upper bound of the size of a single weak dependence operator, representing the W weak dependence operators in a joint state representation based on either CP Nets or P/T Nets does not enlarge the space complexity of the representation of G.*

Proof. The weak dependence operator defines precedence order between actions performed by different robots. In joint state representations, each place represents the joint state of all robots, meaning that each place indicates the state of all the robots that participate in the system. Therefore the precedence order between the actions has already been embodied in the representations of those actions without adding additional symbols (places or transitions) to the representations. □

Figure 4.8 demonstrates the proof of Theorem 9 through the example previously described in this section: assume without loss of generality, that the plan consists of two robots R_1 and R_2 and four actions A, B, C and D. R_1 performs Action A and then performs Action B and R_2 performs Action C, and then it performs Action D, but R_2 cannot perform Action D until R_1 finishes performing Action A. Figure 4.8 depicts the joint state representation of the plan. The figure shows that neither additional places nor transitions have been added in order to represent the weak dependency.

Table 4.2 summarizes theorems 7–9. It presents the space complexity of different approaches for representing weak dependence operators of the same multi-robot plan. The plan consists of R robots, L different roles and W weak dependence operators. Even though the representation of the weak dependence operators in joint state representations does not enlarge the space complexity of the representation of G compared to individual state representations, these kinds of representations are not preferable in terms of space complexity, since the representation of the actions that participate in the operator require exponential space complexity (Theorems 5 and 6).

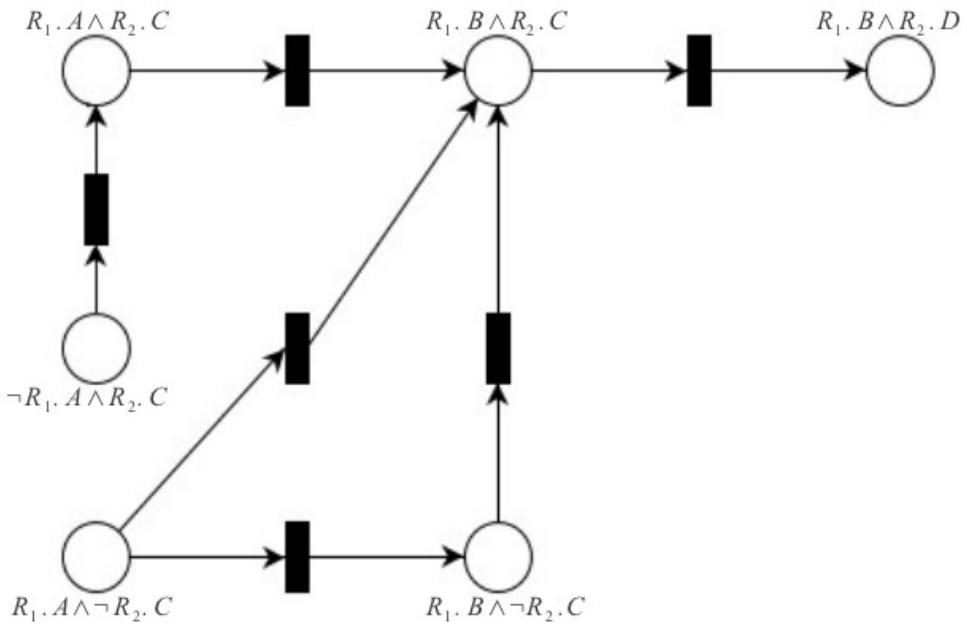

Figure 4.8 Joint state representation of weak dependence of R_2 on R_1

Regarding the example of a plan which consists of two robots R_1 and R_2 and four actions A,B,C and D. R_1 performs Action A, and then performs Action B; and R_2 performs Actions C and then it performs Action D, but R_2 cannot perform Action D until R_1 finishes performing Action A. Figure 4.5 presents a more space efficient individual state representation of this example. Compared to the joint state representation (depicted in Figure 4.8), it can be seen that even tough no additional places or transitions have been added in order to represent weak dependence in Figure 4.8, and an additional place (p_m) has been added to Figure 4.5, still the individual state representation in Figure 4.5 has smaller space requirements.

4.2.3 Space Complexity Analysis of the Strong Dependence Operator

The strong dependence operator defines synchronization between multiple actions performed by different robots. This section discusses the parts of multi-robot plans in which a strong dependence operator is used and the actions associated with it. There is no need to add additional symbols

	Non CP net	**CP net**
Individual state representation	$O\left(\frac{R}{L}WX\right)$ (Theorem 7)	$O(WX)$ (Theorem 8)
Joint state representation	$O(1)$ (Theorem 9)	$O(1)$ (Theorem 9)

Table 4.2 Space complexity analysis when the robots are weakly dependent

(i.e., places and transitions) for representing a strong dependence operator in either individual or joint state representations. Individual state representations represent this operator by merging transitions which are associated with the actions and robots that should be synchronized. Joint state representations represent this operator by merging places and transitions associated with those actions and robots.

Figures 4.9 and 4.10 show an example of an individual state representation and a joint state representation representing the following plan: the plan consists of two robots R_1 and R_2 and two actions A and B. The robots should perform actions A and B together in synchronization. Figure 4.9 presents the individual state representation of the example above. The figure shows that no additional places or transitions have been required in order to represent this example. The transition before the execution of A by R_1 ($R_1.A$) is merged with the transition before the execution of A by R_2 ($R_2.A$), as well as the transition before the execution of B by R_1 ($R_1.B$), and the transition before the execution of B by R_2 ($R_2.B$).

A joint state representation of the example above is depicted in Figure 4.10. Like the individual state representation (Figure 4.9), no additional places or transitions have been required in order to represent this example. But unlike the individual state representation, in this representation the places also have been merged. The place that represents the execution of A by R_1 has been merged with the place that represent the execution of A by R_2, and the place that represent the execution of

B by R_1 has been merged with the place that represent the execution of B by R_2.

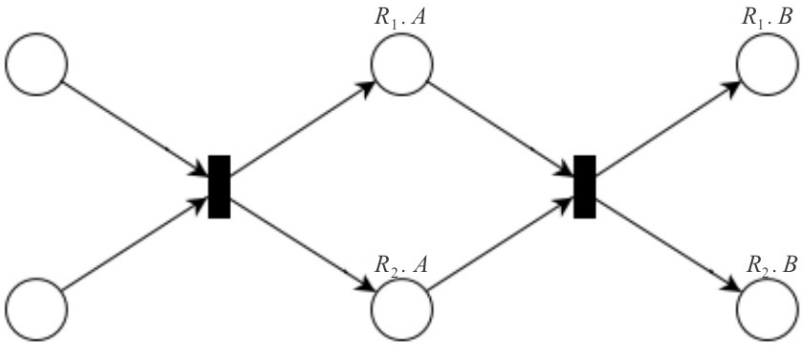

Figure 4.9 An individual state representation of the example above

Figure 4.10 A joint state representation of the example above

This section introduces an analysis of the space complexity of a plan G which consists of R robots (R_1,..,R_n) which are strongly dependent while they are performing a sequence of actions. The analysis will only focus on the space complexity of the parts in G that contain strong dependence operators. Let Q_i be the part of the representation of this sequence of actions that is related only to Robot R_i; let Q be the Q_i that has the maximum length (i.e., $Q = \max Q_i$); and let S be the size of Q. The rest of this section will show the space complexity analysis of individual and joint state representations which are based on either P/T Nets or CP Nets.

Space complexity of individual state representations

Theorem 10. *Given a multi-robot plan G with R robots, L roles, and S being the maximum length of representing a subplan that refers to a single robot (as described above), the space complexity*

of the strong dependence operators in P/T Nets based representations of G is:

$$O(SR) \tag{4.7}$$

Proof. In individual state representations, which are based on P/T Nets, each place can represent only a single robot. Therefore, a sequence of actions which is performed by R robots should be represented in separated places for each robot. For each place in Q there are R places in the representation of G (one for each robot). Thus the space complexity is: $O(SR)$. □

Theorem 11. *Given a multi-robot plan G with R robots, L roles, and S being the maximum length of representing a subplan that refers to a single robot (as described above), the space complexity of the strong dependence operators in CP Nets based representations of G is:*

$$O(SL) \tag{4.8}$$

Proof. In CP Nets we can differentiate the robots by the use of the tokens' color. Therefore, multiple representation of the same roles can be unified into a single representation (Chapter 5). The result is that, for each place in Q, there are L places in the representation of G (one for each role). Thus the space complexity is: $O(SL)$. □

Space complexity of joint state representations

Theorem 12. *Given a multi-robot plan G with R robots, L roles, and S being the maximum length of representing a subplan that refers to a single robot (as described above), the space complexity of the strong dependence operators in either a P/T Nets or CP Nets based representations of G is:*

$$O(S) \tag{4.9}$$

Proof. In a joint state representation, each place represents the joint state of all robots. When the robots are strongly dependent, each state should be synchronized between all robots and represented as a single place in joint state representation. Therefore, the representation Q of each single

	Non CP net	**CP net**
Individual state representation	$O(SR)$ (Theorem 10)	$O(SL)$ (Theorem 11)
Joint state representation	$O(S)$ (Theorem 12)	$O(S)$ (Theorem 12)

Table 4.3 Space complexity analysis when the robots are strongly interdependent

robot can be merged into a single representation Q, where each place represents the joint state of all robots. Thus, the space complexity is $O(S)$ (the length of Q). □

Table 4.3 summarizes the theorems 10-12. It presents the space complexity of different approaches for representing a plan which consists of R robots (in L different roles) and which execute a sequence of actions in synchronization (strong dependency). The table shows that the joint state representation is the preferable choice to represent strong dependency among the robots.

The *open door* example, which has been described in Section 4.1, describes a plan in which the two robots are strongly interdependent while they perform the *open door* action. Figure 4.3 presents an individual state representation of this plan and Figure 4.4 presents a joint state representation of this same plan. As expected, the figures demonstrate that the joint state representation (depicted in Figure 4.4 is preferable to the individual one in terms of space complexity.

4.3 Summary

This chapter analyzed the space complexity of existing approaches for representing a given multi-robot plan (denoted as G). We classified the existing representations under two dimensions:

1. Joint state representations or individual state representations

2. P/T Nets or CP Nets

	Non CP net	**CP net**
Individual state representation	$O\left(PR+\frac{R}{L}WX+SR+K\right)$ [21, 22, 50, 53, 55, 101] (Theorems 3, 7 and 10)	$O(PL+WX+SL+K\log R)$ [60, 65, 100] (Theorems 4, 8 and 11)
Joint state representation	$O(P^R+S+K)$ [99] (Theorems 5, 9 and 12)	$O(P^L+S+K\log R)$ (Theorems 6, 9 and 12)

Table 4.4 Space complexity analysis of multi robot representations

The analysis is divided into three parts of G: independency, weak dependency and strong dependency. The results of our analysis and the scalability of existing approaches is presented in Table 4.4. The table summarized the total space complexity of each type of representation. In fact, it sums up the space complexity of the parts where the robots are independent (Section 4.2.1), weakly dependent (Section 4.2.2) and strongly dependent (Section 4.2.3). The table also cites relevant previous work (based on Theorems 3-12).

Theorems 3-12 and Tables 4.1-4.3 show that individual state representations based on Colored Petri Nets are preferable (in terms of space complexity) for representing independent and weakly-dependent robots, and joint state representations are preferable (in terms of space complexity) for representing strongly-dependent robots.

Building from this analysis, Chapter 5 will introduce our CPNP representation for representing multi-robot plans. This is a novel representation that combined individual and join state representation by representing independency and weak dependency as individual state representation and representing strong dependency as joint state representation. according to Theorems 4, 8 and 12). This combination leads to the best space complexity: $O(PL+W+S+K\log R)$.

Chapter 5

CPNP Representation for Multi-Robot Systems

In this chapter, we extend the CPNP representation to allow for representation of multiple robots, interacting within the same system (i.e., a multi-robot system). We focus on representing a system-level (group-level) plan, that represents the plans for all robots in the system. We will examine the use of this representation both in centralized settings (a single robot overseeing others) as well as distributed settings (multiple robots, each reasoning about others).

The CPNP representation is extended following the insights gained from the analysis of Chapter 4. We combine individual and joint state representation into a hybrid approach, in which a group of robots acting together use a single symbol and separate symbols are used when robots break into individualized roles. We call this kind of representation *Partial Joint State Representation*. A joint state representation will be used in case of strong dependency between the robots, and individual state representation will be used otherwise.

The multi-robot CPNPs' building blocks for centralized and distributed execution will be described in Section 5.1. Then we will move on to dynamic task assignment (Section 5.2). Section

5.3 will introduce the algorithms for executing a multi-robot CPNP. Finally, Section 5.4 summarizes the chapter.

5.1 Building Blocks for Multi-Robot CPNPs

The main challenge of representing multi-robot systems is how to represent the interactions between the robots. CPNP models multi-robot plans as a collection of single-robot plans enriched with synchronization operators as similar to PNP [101]. But unlike PNP, CPNP is based on Colored Petri Net and models these operators as partial state representations, in order to reduce the space complexity.

In addition to the use of colors as defined in Chapter 3 (for single robots), in multi-robot CPNPs token colors distinguish between robots. Each robot has an ID that defines its tokens' color by a new variable *r*. This variable is added to the set of variables that define the color. As described in Chapter 3 there are two types of tokens: tokens that represent robots and tokens that represent resources (denoted as *robotTokens* and *resourceTokens*, respectively). The set of variables composing the color of the *robotTokens* is identical to the set of variables as defined in Chapter 3, with the addition of the variable *r*. The addition is done in such a way that each token with a different value in *r* represents a different robot, while the values of the tokens' other variables represent data which is associated with the robot (i.e., hierarchies, interrupts and robot beliefs).

The representation of shared resources in multi-robot CPNPs extended similarly, with an extra variable *r*, which holds the ID of the robot that currently uses the resource. When there is no robot currently using the resource, the variable is set to null. Whenever the resource is allocated to a robot, or when it is released, the variable will change. This means that the variable changes when the token gets out of or gets into the resource place.

The operators are defined according to the dependencies between the robots (Section 4.1). Broadly, there are two possibilities for executing a multi-robots plan: *centralized execution*, and *distributed execution* [25, 28, 29, 98]. The multi-robot operators are built according to one of these approaches.

Centralized execution architectures. Centralized execution architectures are characterized by a single control robot (master) that has a multi-robot plan and manages all the other robots accordingly. The master sends commands to the other robots (slaves) based on the plan. The slaves do not have access to the plan.

Distributed execution architectures. A multi-robot plan or a relevant subset of it, is distributed among a group of autonomous robots. Each teammate is able to see the plan and executes its part in coordination with the other teammates.

The next sections describe the CPNP's operators for the centralized and distributed executions. CPNP models multi-robot coordination by operators which are built according to weak or strong dependencies. Section 5.1.1 describes multi-robot operators in centralized settings, and Section 5.1.2 discusses these operators in distributed settings. Note that a reliable channel for robot communication is assumed.

5.1.1 Multi-Robot Operators in Centralized Settings

This section introduces the weak dependence and strong dependence operators in centralized settings. The centralized CPNP (i.e., the multi-robot plan) is executed by a single (master) robot. This robot sends commands to the other robots according to the CPNP execution algorithm (described in Chapter 6). All the robots in the plan are represented by tokens which are part of the CPNP that is maintained and executed by the master robot. As explained before, each token holds data

about a robot or resource. In order to keep this data up-to-date, each robot updates the master robot regarding changes in its beliefs or any interrupt occurrences, and the master robot in turn updates the value of the variables.

5.1.1.1 Weak Dependence Operator in Centralized Settings

A *weak dependence operator* is used to represent a precedence relation among the actions of different robots. There are two types of weak dependence. The first type represents a situation in which robot A cannot start an action (denoted as C) until one of its teammates reaches a certain state. In this case, it does not matter which of the teammates is the robot that reaches this state. In the second type of dependence, robot A depends on a specific robot. This means that robot A cannot start an action C until robot B reaches a certain state. This section introduces the weak dependence operator according to the two types of weak dependency. The operators are built as individual state representations, since according to Chapter 4 individual state representations based on CP nets yield the best space complexity. This means that every place in the operator represents a state of a single robot.

The first type of weak dependence operator is represented by CPNP according to the following method: we define a boolean variable (denoted as v_c) that represents the condition that should be fulfilled before robot A performs C. This variable is initialized to $true$ or $false$ according to the state of the environment. The master will update all the *robotTokens* with each change in this variable. In addition, we will add arc expressions that contain the expression v_c=$true$ to those arcs that move tokens to the transition that start the action C (denoted as t_{s_c}). As a result, the arc expressions will prevent robot A from performing action C while v_c is $false$. When one of the robots reaches the state that is represented by v_c, it will set the variable v_c to $true$. Following this change in v_c, robot A will start the execution of action C. Noted that this weak dependency is represented by individual state representations of the actions performed by each robot and an

additional variable v_c. Therefore, this representation is an individual state representation.

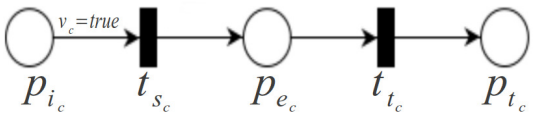

Figure 5.1 Weak dependence operator of the first type

Figure 5.1 shows a weak dependence operator for the case where a robot depends on one of the other members of the group but is not dependent on a specific member (the first type of weak dependence operator). The robot that executes the plan that is represented by the CPNP in this figure cannot start the execution of action C while v_c=$false$. Once v_c is set to $true$ by one of the robots, the robot will start to perform C.

Consider the following *Organize room* example: robot A is a vacuum machine that has to enter a room in order to clean it. However, the room is locked and robot A does not have the ability to open the room. Therefore, robot A cannot enter the room until one of the robots that can open this room does so.

Figure 5.2 illustrates this example. The upper CPNP represents the plan that is executed by one of the robots in order to open the room (*Open room* action), and the lower CPNP represents the plan that is executed by robot A. The variable v_{room} represents the state of the room, where $true$ indicates that the door is open and $false$ indicates that the door is closed. The robot that executes the plan which is represented by the upper CPNP, opens the door if the door is closed (v_{room}=$false$). When the robot finishes opening the door, the variable v_{room} is changed to $true$. Robot A that executes the lower CPNP, can enter the room only when the door is open. This condition is expressed by the arc expression v_{room}=$true$.

Regarding the weak dependence operator of the second type: this one is taken from [101]. The operator is depicted in Figure 5.3. The CPNP in Figure 5.3 has two input places $P^I = \{p_{i_1}, p_{i_2}\}$, two output places $P^O = \{p_{o_1}, p_{o_2}\}$, one connector place $P^C = \{p_m\}$ and two control transitions

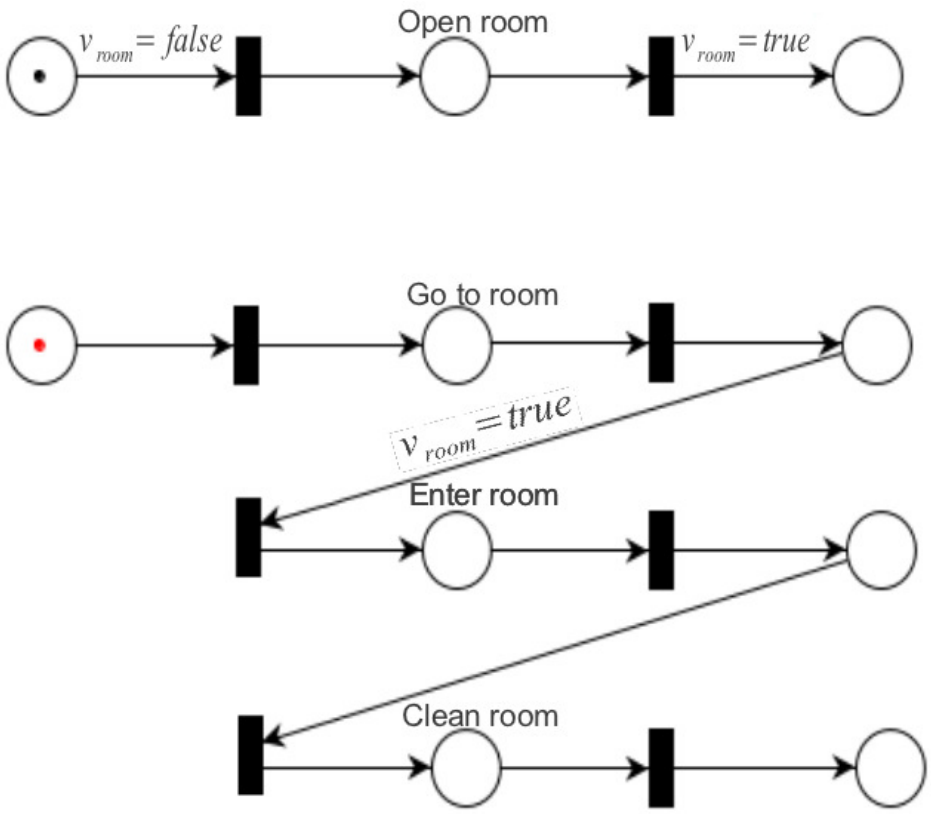

Figure 5.2 *Organize room* - weak dependence operator of the first type

$T^C = \{t_f, t_j\}$. Consider two robots, R_1 and R_2. Places p_{i_1} and p_{o_1} are part of the plan that is executed by R_1 and places p_{i_2} and p_{o_2} are part of the plan that is executed by R_2. p_m is a connector place that connects R_1 and R_2. It represents the state in which R_1 finishes executing Action C. Again we can see that each place represents a state of a single robot (R_1 or R_2 but not both) as an individual state representation. If this place contains a token, it indicates that R_1 has reached the desired state that allows R_2 to start performing the dependent action (action C). When R_2 reaches the place p_{i_2}, it waits until R_1 will reach the state that is indicated by the existence of a token in p_{i_1}. When R_1 reaches this state a token is fired to p_m, and R_2 can continue to place p_{o_2}.

The operator can be expanded to include more than two robots, according to the following logical dependencies:

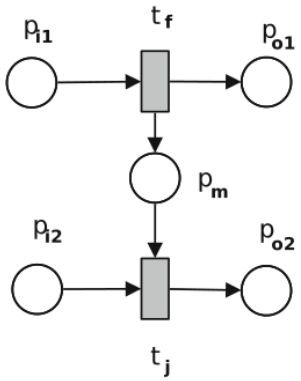

Figure 5.3 Weak dependence operator (taken from [101]), for the case of dependency on a specific robot (second type of operator)

1. A weak dependence of robot *A* on robot *B* **and** robot *D*.

2. A weak dependence of robot *A* on either robot *B* **or** robot *D*.

In order to expand the weak dependence operator to include more than two robots, for each robot R_j we add the places p_{i_j},p_{o_j}, the transition t_j and the arcs $(p_{i_j},t_j),(t_j,p_m),(t_j,p_{o_j})$. The logical dependencies are expressed by the arc expression that is defined on the arc that exits from place p_m. Figure 5.4 shows such an expanded operator for three robots, where R_3 is dependent on R_1 and/or R_2. If R_3 is dependent on R_1 **and** R_2, the arc expression on the arc (p_m,t_3) is $1'r = R_1 \cap 1'r = R_2$ (meaning that the arc moves a single token of R_1 **and** a single token of R_2). Otherwise, the arc expression is $1'r = R_1 \cup 1'r = R_2$ (the arc moves a single token of R_1 **or** a single token of R_2), where r is a variable that indicates the ID of each robot and $1'$ indicates a single token.

Now consider the *Organize room* example with a slight change: instead of the robot being dependent on a random other robot to open the door, R_2 is dependent specifically on R_1 to open the door (leaving aside the reason for this specific dependency). In order to represent this dependency we should use a weak dependence operator of the second type.

Figure 5.5 depicts the CPNP for this example. This CPNP is built by means of the second weak dependence operator (dashed box). The upper and lower parts of the CPNP are relevant to

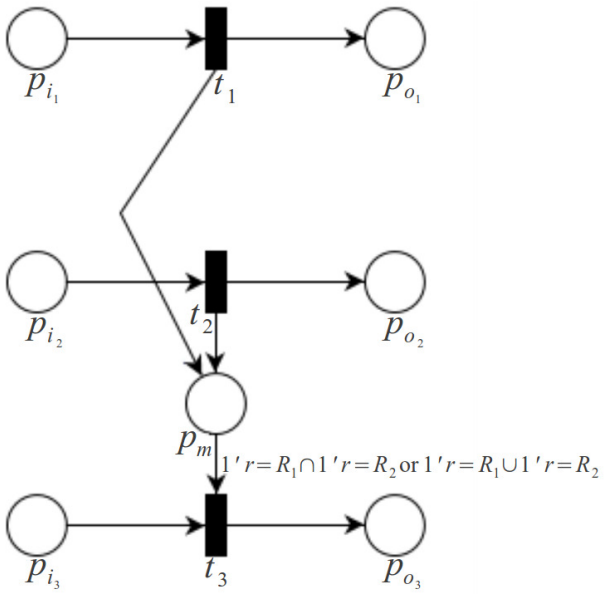

Figure 5.4 Weak dependence operator for three robots, for the case of dependency on a specific robot (second type of operator)

R_1 and R_2, respectively. After R_1 finishes opening the door (i.e., finishes performing the *Open room* action), a token is fired to place p_m. Once R_2 terminates the *Go to room* action it waits for a token to appear in place p_m, before it enters the room and cleans it.

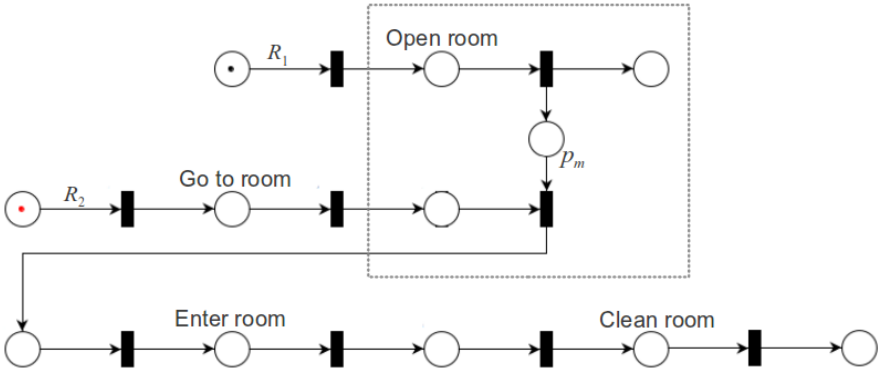

Figure 5.5 *Organize room* - weak dependence operator of second type

5.1.1.2 Strong Dependence Operator in Centralized Settings

This section introduces the *strong dependence operator* for multi-robot CPNP. This operator represents the plans of n robots which are strongly dependent, meaning that these are interdependent robots. In this operator places represent states of all robots as a joint state representation, since it is proven that joint state representation yields the best space complexity when representing strong dependency (Chapter 4).

A strong dependence operator (shown in Figure 5.6) provides time synchronization between robots $R_1,..,R_n$. Such an operator synchronizes the plans of n robots. The operator consists of a basic CPNP structure enriched by the arc expression $1'r = R_1 \wedge 1'r = R_2 \wedge .. \wedge 1'r = R_n$. This arc expression expresses that each transition can fire *iff* for each robot $R_i \in \{R_1,..,R_n\}$; there is at least one token with the *ID* of R_i. The arc expressions in the figure are written as a shortened formula.

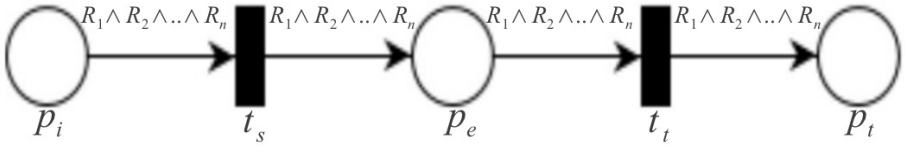

Figure 5.6 Strong dependence operator

Each place in this operator represents the joint state of all the robots that participate in the mission. This representation is a joint state representation (described in Section 4.1). Place p_i represents a state where all the robots are ready to start the synchronized mission, where transitions t_s and t_t respectively start and terminate the action of each robot according to the ID of each robot. Place p_e represents the execution phase and place p_t represents the termination of the synchronized mission.

The operator can also be used hierarchically by using hierarchical decomposition (explained in Section 3.3). Instead of starting each action associated with each robot in t_s and terminating the action in t_t, in Figure 5.6 we demonstrate how we can use hierarchical decomposition as described in Section 3.3 by defining a new variable for each robot. The new variable indicates the hierar-

chical call. This variable is initially set to 0 and is changed at the start and the termination of the hierarchical call.

The strong dependence operator that uses hierarchies is shown in Figure 5.7. The upper CPNP is a *superCPNP* and the lower is a *subCPNP*. $h_1,..,h_n$ are variables that indicate the hierarchical calls (i.e., h_i is a variable that indicates the hierarchical call of robot R_i). Once t_s fires, the values of these variables are set to 1 due to the arc expression on (t_s, p_e). t_t can only fire when all of these variables are set back to 0, due to the arc expression on (p_e, t_t). The subCPNP has n tokens in place p_{i_h}; one token from each robot *ID*, at the initial marking. When the values of $h_1,..,h_n$ are changed to 1, the condition on the arc expression on (p_{i_h}, t_{s_h}) is satisfied and the tokens are split to n CPNPs by firing t_{s_h}. Each CPNP represents the plans of one robot. This means that a token of each robot ID is fired to the appropriate CPNP according to the arc expression. When all the robots finish executing these CPNPs, a token will appear in each of the places $p_{t_1},..,p_{t_n}$ and the transition t_{t_h} will be enabled. Then transition t_{t_h} fires. This firing changes the variables $h_1,..,h_n$ back to 0 and moves the tokens back to the initial marking of the subCPNP.

Back to the *Organize room* example: consider that the robots should lift a table once R_2 finishes cleaning the room. In order to perform this action, each robot should navigate to an opposite side of the table and then the robots should start to lift it together. Figure 5.8 depicts the CPNP that represents this example. After R_1 finishes opening the room it navigates to the right side of the table and waits for R_2. R_2 enters the room and cleans it, and then navigates to the left side of the table. Transition t_{join} connects the two individual CPNPs, thereby creating a joint CPNP (using the join operator, described in Section 3) that represents the *lift* operation. This operation needs to be executed while the two robots are fully synchronized since they should lift the table together. Therefore, we use the strong dependence operator in order to synchronize between the robots (Figure 5.8, dashed box).

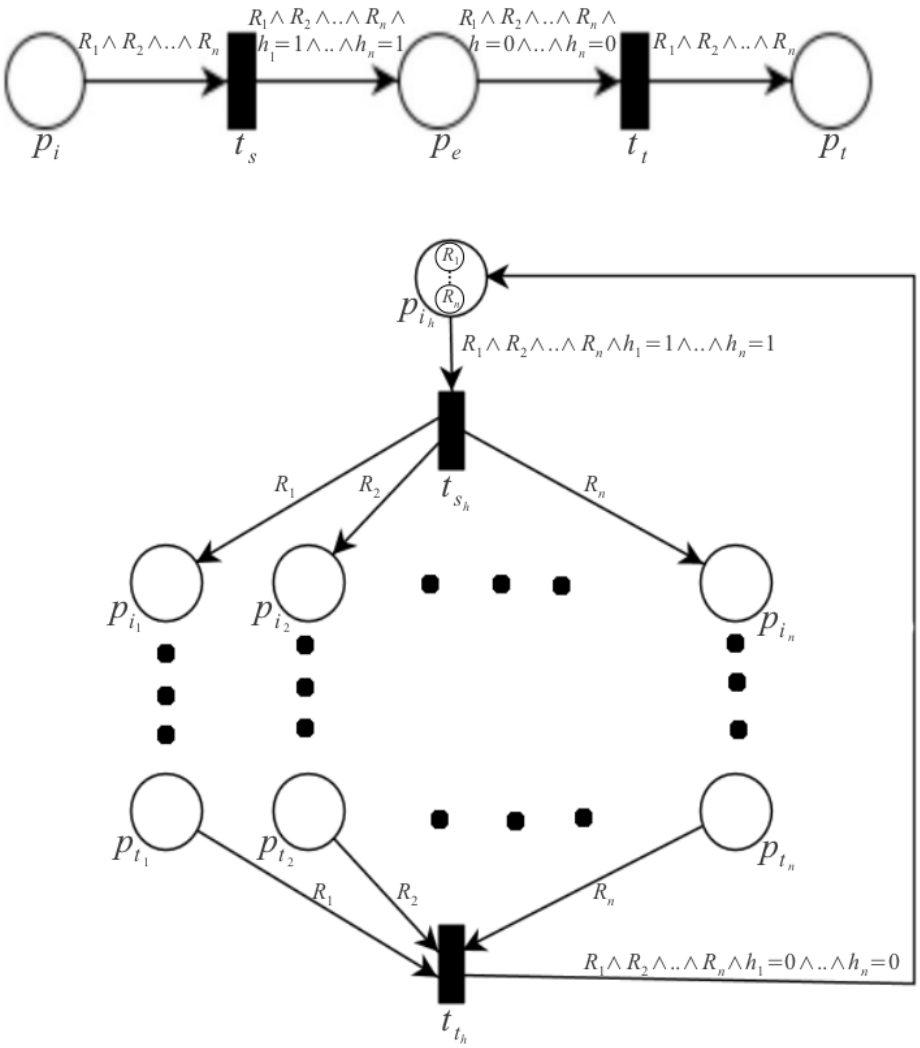

Figure 5.7 A strong dependence operator with hierarchies

It should be emphasized that the weak dependence operator is built as an individual state representation (a single place represents a state of a single robot). The strong dependence operator is built as a joint state representation (a single place represents the state of all robots). We call this combination of representing weak dependency using individual state representation and representing strong dependency using joint state representation *partial state representation*. It builds on the insights gained in Chapter 4.

Centralized execution has several advantages and disadvantages. One major advantage is that

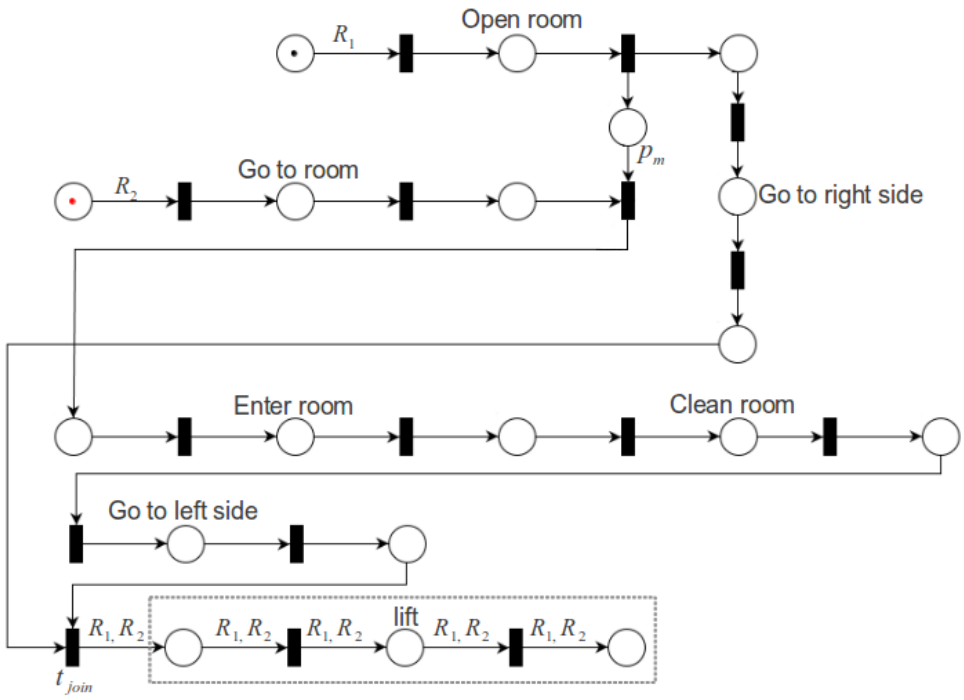

Figure 5.8 *Organize room* - strong dependence operator

its representation gives an explicit description of the execution state of the system. This advantage stems from the fact that the multi-robot CPNP is executed by a single machine (the centralized robot), and each state (marking) represent the state of all the robots, with no need to synchronize the placement of the tokens that represent the different robots. Also, this approach reduces the need for conversation and synchronization protocols between the robots, since a single robot both executes and manages the multi-robot CPNP (conversations only take place between the master and each slave). In addition, the centralized execution approach reduces the space complexity of the multi-robot system, since the whole plan is maintained by a single centralized robot, instead of each robot in the plan needing to maintain all or parts of it.

One drawback of centralized execution is that this approach has a single point of failure (i.e., the centralized robot). Also, centralized execution can lead to an extreme overload of the centralized robot who has to execute the multi-robot CPNP, update the variables that indicate the beliefs of

each robot in the plan, and communicate with the slaves. Finally, the communication network can overload since each change in a robot's beliefs requires communication with the master robot. This may lead to a bottleneck at the master robot who will consequently need a lot of time for sending and receiving messages via the network. The overload may slow down the execution of the system, resulting in a slower response to environmental changes. In light of the mentioned drawbacks of centralized executions, in the next section we will present the CPNP representation for *distributed* multi-robot architectures.

5.1.2 Multi-Robot Operators in Distributed Settings

We now introduce multi-robot CPNPs for distributed architectures. In distributed CPNPs, the multi-robot CPNP is divided into R parts where R stands for the number of robots in the system. Each robot R_i maintains that part of the plan which is relevant to it. Such a part includes the following: all the places that can have a token which represents R_i (i.e., a token in which R_i is the value of the variable r) during the execution; transitions that are connected to those places; and resource places. We denote the CPNP which is maintained by robot R_i as R_iCPNP.

At the beginning of execution, each R_iCPNP is marked with an initial marking. Each robot R_i executes its R_iCPNP separately. The dependencies between the robots are represented by the weak dependence and strong dependence operators (defined below for distributed executions). The distributed CPNP assumes the existence of a conversation and synchronization protocol which synchronizes the state of each R_iCPNP.

Weak dependence operator in distributed settings. In Section 5.1.1 two different types of weak dependence operators were presented. The first type represented a situation in which robot A could not start an action (denoted as C) until one of his teammates reached a certain state. In this case, it did not matter which of the teammates was the robot that reached this state. As for the

second type, it represented a situation where robot A depended on a specific robot, meaning that robot A could not start an action C until robot B reached a certain state.

The first type of weak dependence operator will be represented in distributed CPNPs as follows. Similar to the centralized weak dependence operator as described in Section 5.1.1, we define a boolean variable (denoted as v_c) that represents the condition that should be fulfilled before robot A performs C. This variable is initialized to *true* or *false* according to the state of the environment. The variable will be shared among the robots and it will be synchronized using a synchronization protocol (i.e., each change in this variable will be broadcast to all robots e.g., [1]). The CPNP as shown previously in Figure 5.1 will be maintained by robot A. When one of the robots reaches the state that is represented by v_c , it will set the variable v_c to *true* and broadcast the new value of v_c to the other robots in the system. Following this change in v_c, robot A will start the execution of action C.

In order to build a weak dependence operator of the second type, we divide the centralized weak dependence operator as shown in Figure 5.3 into n pieces (n being the number of robots that are involved in the dependency). Each robot maintains only that part of the operator that is relevant to it, in addition to place p_m (i.e., there are n instances of place p_m). When the execution algorithm of robot A (the dependent robot) reaches the transition that should fire the tokens from place p_m, robot A starts a conversation with the robots that are involved in the dependency (using a conversation protocol). Those robots are represented by tokens that appear in the arc expression on the arc exiting from place p_m in the R_ACPNP (the CPNP of robot A). Robot A waits until it is confirmed that the other robots have the relevant tokens in the appropriate place p_m in each R_iCPNP (where i is the index of the robot IDs involved in the dependency). Then robot A fires the tokens. This firing is performed using a synchronization mechanism in order to correctly represent the change in the markings of each R_iCPNP of those robots involved in the dependency.

Figure 5.9 presents the distributed weak dependence operator of the second type for a situation

with two robots, in which robot R_2 depends on robot R_1. This figure is divided into two parts. The upper and lower parts are associated with the R_1CPNP and the R_2CPNP, respectively. When the algorithm that executes the R_2CPNP reaches transition t_2 and there is a token in place p_{i_2}, the algorithm discovers that a token of robot R_1 should be in place p_m in R_2CPNP in order to enable t_2 (this constraint is expressed by the arc expression $1'r = R_1$ on the arc (p_m, t_2)). To follow up on this discovery, robot R_2 uses a conversation protocol e.g., [1, 17–19, 36] to ask robot R_1 to notify it regarding the existence of a said token in place p_m in R_1CPNP. When a token for R_1 exists in place p_m in R_1CPNP, the algorithm inserts said token into place p_m in R_2CPNP as well. As a result, transition t_2 fires and R_2 notifies R_1 in order for R_1 to synchronize the marking of p_m in R_1CPNP (i.e., to make $M_{R_1CPNP}(p_m) = M_{R_2CPNP}(p_m)$).

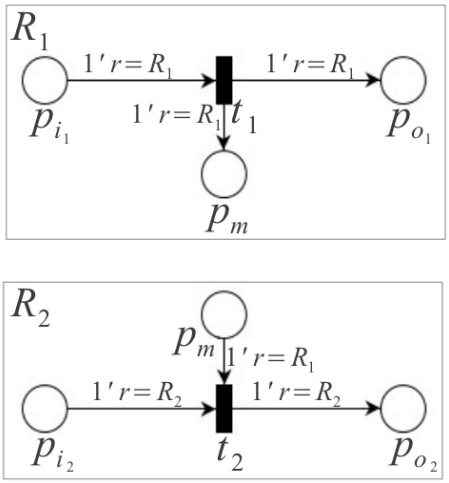

Figure 5.9 A distributed weak dependence operator of the second type, where robot R_2 is dependent on robot R_1

Figure 5.9 depicts the distributed weak dependence operator when two robots are involved in the dependency. This operator is also able to represent weak dependencies of multiple robots (similar to the centralized weak dependence operator of Section 5.1.1). When robot A is dependent on multiple robots, the dependency of robot A on those robots will be expressed by the arc expression on the arc exiting from place p_m in R_ACPNP.

Strong dependence operator in distributed settings. As described in Section 5.1.1, a strong dependence operator provides time synchronization between robots $R_1,..,R_n$. Figure 5.10 introduces the strong dependence operator in distributed settings. Each robot R_i that is involved in the strong dependency, maintains the CPNP as depicted in Figure 5.10 as a part of its own R_iCPNP. The operator consists of a basic CPNP. This basic CPNP moves a token representing R_i with each firing (according to the arc expression $r = R_i$ which means $1'r = R_i$). The strong dependence between the robots is reflected by the guards in transitions t_s and t_t.

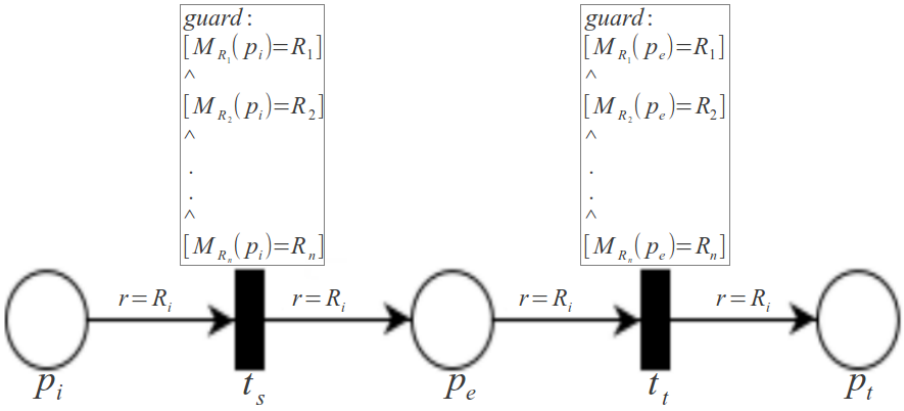

Figure 5.10 A distributed strong dependence operator

Before transitions t_s and t_t are fired, a conversation protocol starts in order to confirm that the guards are satisfied. The guard in t_s checks if each robot R_i that is involved in the strong dependency has a token with its ID (i.e., $r = R_i$) in place p_i in R_iCPNP, while the guard in t_t does the same with regards to place p_e. The guard in t_s represents the following expression: $[M_{R_1}(p_i) = R_1] \wedge [M_{R_2}(p_i) = R_2] \wedge .. \wedge [M_{R_{n(p_i)=R_n}}]$. The guard in t_t represents a similar expression but with place p_e instead of p_i. These expressions are composed of n clauses that all need to be satisfied. Each clause consists of the term $[M_{R_i}(p) = R_i]$ $(i = 1..n)$. This term is the abbreviation of $[M_{R_iCPNP}(p) = 1'(r = R_i)]$, and its meaning is that the marking of the R_iCPNP in place p has at least one token representing R_i (i.e., p has a token that represents R_i). Note that p can stand for either place p_i or p_e, in accordance with the relevant guard (the one on t_s or t_t). The guards continue to check

the term for each robot until the condition is satisfied. Each transition t_t in each R_iCPNP has additional guards that check if the actions as represented by each place p_e in each R_iCPNP have been terminated.

The relevant transition (either t_s or t_t) waits until its guard is satisfied. Once each R_iCPNP has a token representing R_i in place p_i (or p_e) and the guard in t_s (or t_t) is satisfied, transition t_s (or t_t) in each R_iCPNP is enabled and fires. Consequently, the token in each R_iCPNP moves to either place p_e or p_t depending on the case at hand.

Figure 5.11 illustrates the distributed CPNP for the *Organize room* example. The upper CPNP is the R_1CPNP and the lower is the R_2CPNP, executed by R_1 and R_2, respectively (each robot maintains and executes its own CPNP). The weak dependence and strong dependence operator are marked by dashed boxes. At the start of the CPNPs' execution, R_1 and R_2 individually execute the *Open door* and *Go to room* operations, respectively. When R_1 finishes the *Open door* operation it fires a token to place p_m (in order to inform R_2 through use of the weak dependence operator) and then navigates to the right side of the table, thus executing the *Go to right side* operation.

When R_2 terminates the *Go to room* operation, the synchronization mechanism is activated in order to check for a token in place p_m in R_1CPNP. In case such a token is present, a token with the same color (i.e., a token that represents R_1) appears in place p_m in R_2CPNP, thereby enabling the transition t_m. Then, R_2 performs the *Enter room*, *Clean room* and *Go to left side* operations. When R_1 and R_2 are positioned on their respective right and left sides of the table, they start the *lift* operation using the distributed CPNP strong dependence operator in order to synchronize between them (depicted in the lower dashed boxes of both halves of Figure 5.11).

Using shared resources in distributed CPNP. CPNPs use the synchronization mechanism not only for weak dependence and strong dependence operators, but also when using shared resources. When a token that represents a shared resource is fired and the variable *r* (the variable that indicates which robot is using the resource) is changed, the synchronization mechanism is executed and

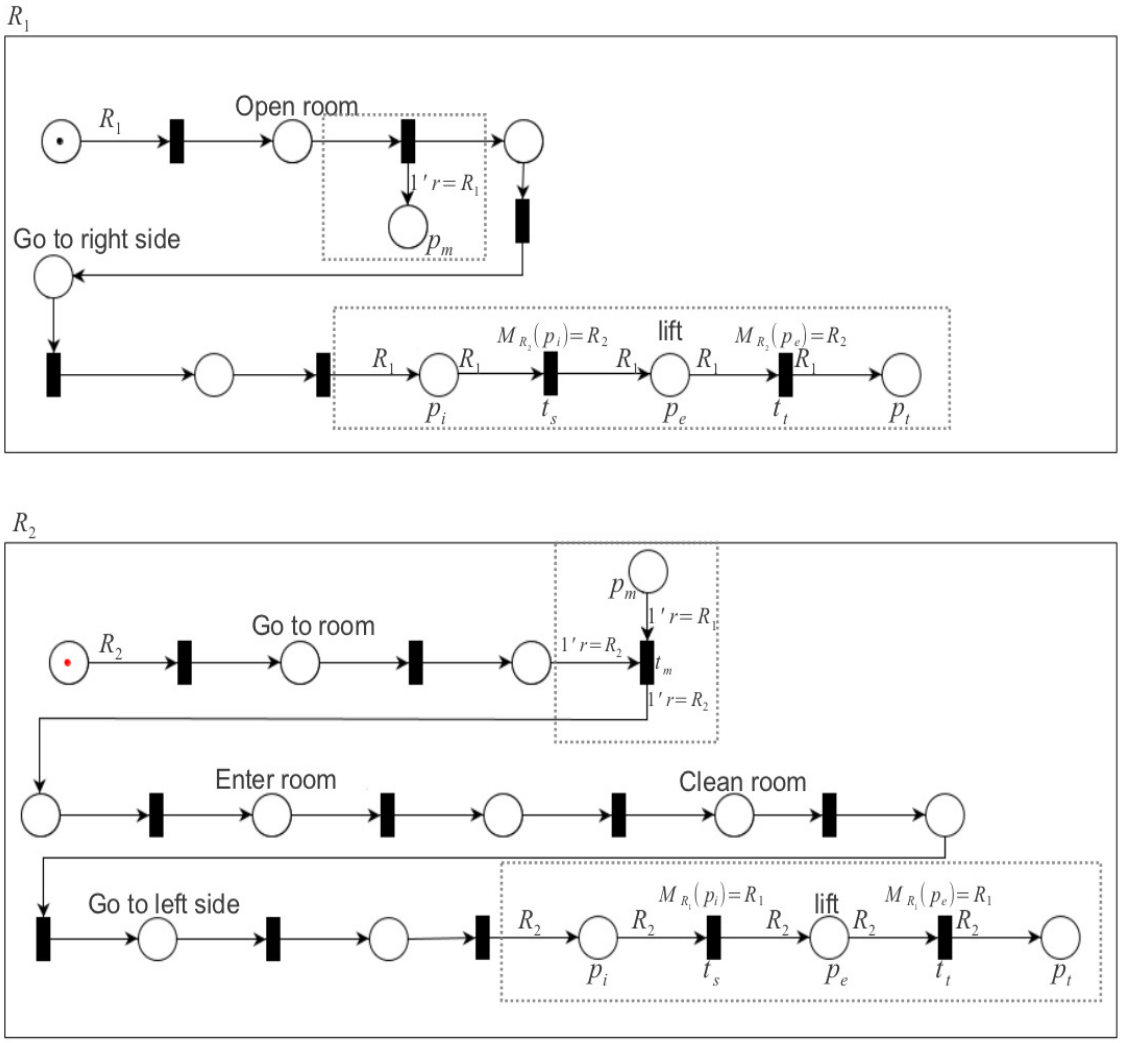

Figure 5.11 *Organize room* - distributed CPNP

updates this change in each $R_i CPNP$. In fact, the synchronization mechanism notifies the other robots that the resource is either occupied or released by R_i.

5.2 Dynamic Roles and Task Assignment

The weak dependence and strong dependence operators as described in Sections 5.1.1 and 5.1.2 assume predefined roles. In order to maximize the flexibility of the multi-robot system, dynamic

task assignment is required. In this section we will show how to use a dynamic task assignment mechanism in CPNP. Section 5.2.1 redefines the dependencies that were presented in Section 5.1 to suit situations where it is not known in advance which robot will perform each task. Section 5.2.2 describes the task allocation and the operators in centralized CPNPs when the roles are not know in advance, while Section 5.2.3 does the same for distributed CPNPs.

For each robot we define variables that represent the status of each task to be assigned to the robots (*success*, *failure*, *in progress*, or *not assigned*). *Success* means that the task has been completed successfully and *failure* logically indicates that the task failed to be executed properly. *In progress* signifies that the robot has only just started or is still in the process of executing this task, while *not assigned* indicates that the task has not been allocated to this robot yet. Each variable is initialized with the value *not assigned*. These variables will be synchronized between the tokens that belong to the same robot.

Furthermore, we define a boolean variable $task\_alloc$ which triggers the task allocation. CPNP assumes the existence of a task allocation mechanism. When the $task\_alloc$ variable is assigned with the value *true*, the task allocation mechanism will start to run. When a robot wants to activate the task allocation, it simply sets $task\_alloc$ to *true*; subsequently, the task allocation mechanism starts and once it finishes, the value of the variable is set back to *false*. The variable will initially be set to *true* because of the need to allocate tasks for each robot at the beginning of the system run. The $task\_alloc$ variable will be synchronized between all robots; therefore, when a robot decides to start the task allocation, all other robots in the team will become aware of it. It should be noted that in centralized architectures, the centralized robot is the only one which has access to this variable and allocates tasks to the other robots.

When the task allocation is active, all the robots participate in the task allocation process (not all of them get new tasks however). This process is carried out in parallel to the work of the robots. CPNP assumes that the task allocation mechanism will prefer to assign new tasks to idle robots

(i.e., robots that have already completed their tasks). If a robot gets a new task while it is currently performing a previously assigned task, it will stop performing its current task and start executing the new one. A robot currently performing a task and not being provided with a new one during the task allocation process, will logically continue towards the completion of its current task.

The task allocation mechanism assigns tasks to robots by changing the values of the variables that represent the status of those tasks. In order to assign a task i to robot R_j, the mechanism will set the value of the variable that represents task i (denoted as t_i) to *in progress*; this will be done only for those tokens of robot R_j where the the variable ID is set to R_j.

Figure 5.12 presents the CPNP of the task allocation. The task allocation is embodied in transition t_1. The initial marking consists of a single token of each robot in place p_1. Once the variable $task\_alloc$ is set to *true*, these tokens are fired by transition t_1 and the task allocation mechanism starts. The task allocation mechanism assigns tasks to robots by changing the values of the variables of those tasks that haven't terminated successfully yet (these variables are denoted as $t_1, .., t_m$ for m tasks). In practice, the task allocation assigns a single task for each robot each time it is called. Therefore, while transition t_1 fires, the task allocation changes the values of the variables that represent each task; it follows from the above that one variable for each robot is adjusted. The rule by which the task allocation assigns a single task for each robot each time, is expressed by the arc expression on (t_1, p_2). After the task allocation is completed (a state represented by tokens in place p_2), the tokens are fired back to the initial place p_1 by transition t_2 and the variable $task\_alloc$ is set back to *false*.

Each task is represented in a different CPNP. The basic CPNP structure that represents each task is shown in Figure 5.13. Similar to the CPNP structure as depicted in Figure 3.1, each task i consists of an initial place p_1, an execution place p_2, a termination place p_3, a start transition t_1 and finally a termination transition t_2. The variable representing task i is t_i. The main difference between the figures is that the initial place of Figure 5.13 contains n tokens (one token per robot),

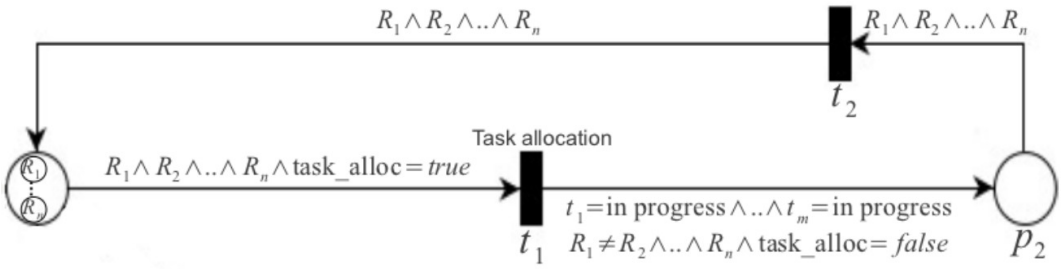

Figure 5.12 CPNP task allocation

as compared to the CPNP structure of Figure 3.1 which represents a single-robot system containing only one token at the initial marking. In addition, Figure 5.13 adds two arc expressions, one on (p_1, t_1) and one on (t_2, p_3). The arc on (p_1, t_1) expresses that only the token in which t_i has the value *in progress* will be fired by transition t_1, and that only the robot that is assigned with this task will start to perform it. Place p_3 represents the termination of task i; hence, the arc expression on (t_2, p_3) sets the variable t_i to *success* or *failure* according to the outcome of the task. Also, it sets the variable *task_alloc* to *true* in order to call the task allocation since the robot that performs task i is now idle and ready to receive a new task.

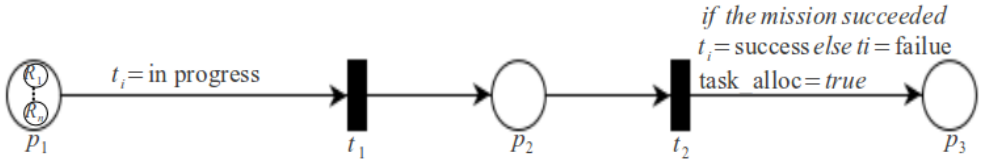

Figure 5.13 CPNP task representation

The CPNP is also able to represent a situation of dead or resurrected robots, as depicted in Figure 5.14. In order to do so, two variables are defined. The first variable is a boolean variable *live* that indicates whether the robot is "alive" or not. This variable will be synchronized between all the tokens that represent the same robot. The *live* variable is initialized with *true* and it is updated by an external mechanism which checks if the robot is alive (CPNP assumes the existence of such a mechanism).

The second variable is *live_robots*. This is a numerical variable that represent the number of "living" robots. When a robot "dies" or becomes not functional, the token that represent this robot in place p_1 moves to place p_3 by transition t_3 and stays in this place until the robot will come back to life (e.g., a robot whose battery was drained and subsequently recharged). In addition, the *live_robots* variable is decremented by one (represented by the arc expression on (t_3, p_3)). Since the dysfunctional robot cannot accomplish his task, reallocation of tasks is required. As a result, the variable *task_alloc* is set to *true* (expressed in the arc expression on arc (t_3, p_3)). When a robot is resuscitated, the token representing that robot moves back from place p_3 to place p_1 and the *live_robots* variable is incremented by one (represented by the arc expression on (t_4, p_1)). In this situation too reallocation is required in order to assign a task to the revived robot; therefore, *task_alloc* is set to true in the arc expression on arc (t_4, p_1).

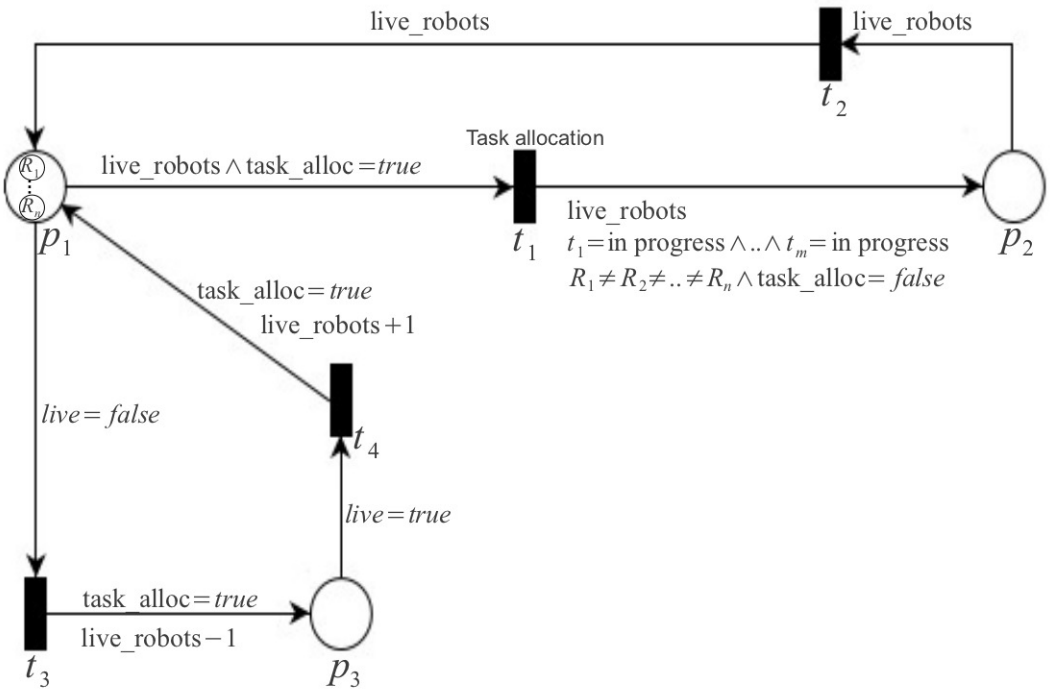

Figure 5.14 CPNP task allocation when robots may "die"

In Figure 5.12, a single token of each robot is fired by transitions t_1 and t_2 in order to allocate

tasks for all robots. However, if we take into account that robots may “die", we want only the “living" robots to participate in the task allocation. Therefore, in Figure 5.14 only those tokens that represent the living robots are fired by transitions t_1 and t_2. This constraint is represented by the presence of the *live_robots* variable on the arc expressions on (p_1,t_1),(t_1,p_2),(p_2,t_2) and (t_2,p_1). The meaning of this variable's presence on the arc expressions is that the number of tokens that are moved by the associated arcs is equal to the value of *live_robots*. Consequently, according to the structure of the CPNP in Figure 5.14, each marking consists of precisely a single token of each robot. Furthermore, the tokens that represent dead robots are stuck in place p_3. Last, the k tokens ($k = live\_robots$) moved by arcs (p_1,t_1),(t_1,p_2),(p_2,t_2) and (t_2,p_1) are exactly composed from a single token of each living robot.

It should be noted that the CPNP of Figure 5.13 assumes that all robots are “alive”. In case of the possibility that a robot “dies” during the performing of his task, we should add the expression $live = true$ to the arc expressions on each arc that exits from a place to a transition (i.e., the arcs (p_1,t_1) and (p_2,t_2).

5.2.1 Dependencies When Robot Roles Are Not Predefined

When robot roles are not predefined but are allocated dynamically (meaning that they can change continuously), the definitions of dependencies are slightly different from those as defined in Section 5.1. Instead of defining dependence as robot A being dependent on robot B (where A and B are robot IDs), the definition of dependency will be as follows: the robot that performs *task A* is dependent on the robot that performs *task B* (where A and B are *task* IDs). Again, it should be noted that two kinds of such dependency exist:

1. A *weak dependence* of a robot that performs task A on a specific state of task B, means that the robot that performs task A cannot start this task until the robot that performs task B reaches a certain state.

2. A *strong dependence* between a robot that performs task A (denoted as robot R_1) and a robot that performs task B (denoted as robot R_2), means that R_1 is weakly dependent on R_2 and R_2 is weakly dependent on R_1.

5.2.2 Centralized Execution When Robot Roles Are Not Predefined

In centralized executions, the centralized robot maintains and executes the task allocation CPNP as part of the multi-robot CPNP. The centralized robot assigns task to the other robots in the system using the task allocation CPNP (see Figures 5.12 and 5.14), executes the relevant CPNP for each robot according to the value of each task variable (Figure 5.13), and sends commands to the other robots in accordance with the CPNP.

As explained previously, the centralized robot activates the task allocation by setting the value of the *task_alloc* variable to *true*. The task allocation CPNP is executed in concurrence with the other parts of the multi-robot CPNP; the changing of the value of *task_alloc* to *true* does not prevent the firing of any tokens in the multi-robot CPNP.

The weak and strong dependence operators are defined in a similar manner to those presented in Section 5.1.1, but operators refer to tasks instead of specific robots when robot roles are not predefined.

Weak dependence operator when robot roles are not predefined. Figure 5.15 shows a weak dependence operator of a robot (let's call it A) that should perform a task (denoted as $task_3$) but cannot do so until the robot that performs $task_1$ and/or the robot that performs $task_2$ will reach a state that is indicated by a token in p_{i_1} and/or a token in p_{i_2}. The predicates that appear on the arc expressions in this figure are written in abbreviated formulas: $task_i$ means $task_i$ is in progress, $i \in \{1,2,3\}$. It should be noted that this operator is used in case of robot A being dependent on a specific state of $task_1$ and/or $task_2$. However, if robot A is dependent only on the termination or

outcome of one or more tasks, then we can use the weak dependence operator of the first type as described in Section 5.1.1.

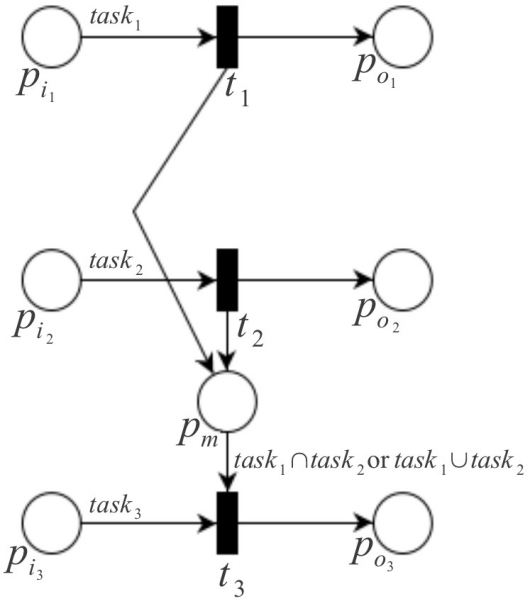

Figure 5.15 A weak dependence operator without a predefined role assignment

Strong dependence operator when robot roles are not predefined. Figure 5.16 depicts a strong dependence operator of m robots that perform m tasks. Similar to Figure 5.15, the arc expressions in this figure are written as abbreviated predicates, where $task_i$ denotes $task_i$ is in progress. This operator is the same operator as defined in Section 5.1.1, but instead of explicitly specifying which robots participate in the synchronized mission, the participation is identified by the tasks associated with the robots.

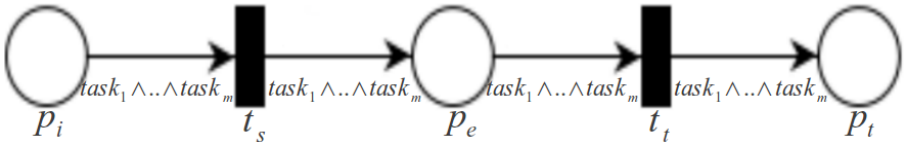

Figure 5.16 A strong dependence operator without a predefined role assignment

Returning to our *Organize room* example: in the previous section, the planner determined in

advance which robot will go to each side of the table. In practice, it is more efficient to dynamically determine which robot will go to the left side and which robot will go to the right side during the execution of the plan. Figure 5.17 depicts the *Organize room* example in such a way that the *Go to right side* and *Go to left side* tasks are allocated to R_1 and R_2 dynamically, during the execution of the plan. The centralized robot (which is either robot R_1 or R_2) maintains and executes the CPNP illustrated in this figure in addition to the the task allocation CPNP (depicted in Figures 5.12 and 5.14).

The plan starts with the execution of the *Open room* and *Go to room* actions, by R_1 and R_2 respectively. Then R_2 performs the *Enter room* and *Clean room* actions using the weak dependence operator as described in Section 5.1.1 (without dynamic task allocation). When the *Clean room* operation is terminated the *task_alloc* variable is set to *true*. This change triggers the task allocation CPNP to start the task allocation process. Two new variables have been defined for the task allocation: t_1 and t_2. These represent the *Go to right side* and *Go to left side* tasks, respectively. The robot that is represented by tokens whose variable t_1 will be set to *in progress*, will get the *Go to right side* task. Similarly, the robot that is represented by tokens whose variable t_2 will be set to *in progress*, will get the *Go to left side* task as a result of the task allocation process. When the robots finish performing these tasks, they start together the *lift* operation using the strong dependence operator that allows for dynamic allocating of roles.

5.2.3 Distributed Execution When Robot Roles Are Not Predefined

This section introduces multi-robot CPNPs for distributed execution when robot roles are not known in advance but are allocated dynamically during system execution. As mentioned previously, CPNP assumes an existence of a synchronization protocol which synchronizes the state of each personal robot CPNP. Each time the synchronization protocol is activated, it updates the markings of each robot CPNP for all the robots involved in the synchronization process. The syn-

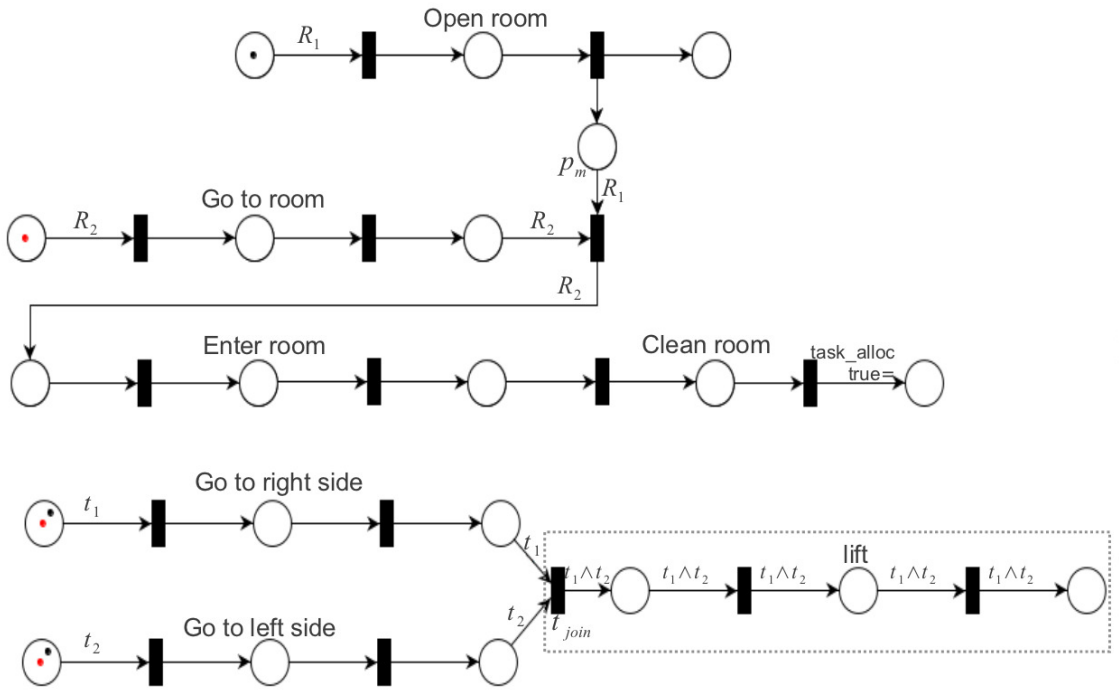

Figure 5.17 *Organize room* - centralized execution that dynamically allocates the *Go to right side* and *Go to left side* tasks

chronization mechanism updates the state of each robot CPNP as well as the color of each token (i.e., the values of the variables).

Section 5.1.2 presented the distributed CPNP in such a way that each robot maintains only that part of the CPNP that is relevant to it. However, when robot roles can be allocated dynamically, we cannot know in advance which part of the CPNP will be executed by which robot (this depends on the task allocation). Therefore, in a distributed CPNP in which robot roles can change during execution, each robot maintains the entire multi-robot CPNP (the whole plan). This means that for n robots we have n copies of the graph (one for each robot). We denoted the CPNP of robot i as $R_i CPNP$. At the start of execution, each graph is marked with the initial marking.

In fact, each robot in the distributed architecture maintains the same multi-robot CPNP which is maintained by the centralized robot in the centralized architecture (Section 5.2.2). Each $R_i CPNP$

is similar to the multi-robot CPNP which is maintained by the centralized robot and marked by the same initial marking. Also, the weak dependence and strong dependence operators in each $R_i CPNP$ are the same as the operators introduced in Section 5.2.2.

Each robot R_i executes his $R_i CPNP$ separately. There are two possible ways in which the firing can be executed. First, the firing may consist only of tokens that represent R_i. These are the tokens that have the value $r = R_i$ (i.e., no resource tokens and no tokens that represent other robots), where r is the variable that indicates the robot ID in the color. In this case, the firing will be executed according to the single-robot CPNP rules as described in Section 3. The firing rules do not change when the arc expressions that are involved in the firing consist solely of tokens that represent R_i.

However, if the arc expressions involved in the firing consist not only of the tokens that represent R_i, then the firing will consist of two phases. First, a transition t fires, and a synchronization mechanism is executed which synchronizes the markings of each robot that appears in the arc expressions of those arcs that either enter or exit t. The synchronization keeps running until t fires. Then, in the second phase, the transition t fires and the synchronization is terminated. This case represents a situation where R_i should coordinate and cooperate with other robots.

When a token that represents a shared resource is fired and the variable r (the variable that indicates which robot is using the resource), is changed, the synchronization mechanism is executed and updates each $R_i CPNP$ with this change. As mentioned before, this synchronization in fact notifies the other robots that the resource is either occupied or released by R_i.

The task allocation process in distributed multi-robot CPNPs will be as follows: when a robot wants to execute the task allocation process, first it will change the *task_alloc* variable as defined in Section 5.2. Then it will execute the synchronization mechanism in order to update the *task_alloc* variable in each robot with this change. Following mentioned change, task allocation is executed in each robot and each robot participates and negotiates in the task allocation process using the synchronization mechanism. As mentioned previously, the task allocation process is embodied in

a transition (i.e., transition t_1 in the CPNPs which are depicted in Figures 5.12 and 5.14). The task allocation process is activated when the transition in the task allocation CPNP becomes enabled, which occurs when *task_alloc* is set to *true*. As explained in Section 5.2.2, the task allocation CPNP can be executed in concurrence with other parts of the CPNP.

The result of the task allocation is an allocation of tasks to the robots that is agreed upon by all robots. This allocation is represented by the assignment of values to the variables which represent the tasks in each token (each token represents a different robot). The assignment of values is synchronized between all robots. This means that at the end of the task allocation process, each robot maintains the tokens of all the robots in his task allocation CPNP with the correct values in the variables that represent the different tasks. Each firing in the task allocation CPNP is synchronized between all robots.

In our representation we chose to represent the task allocation and the synchronization mechanism as a black box embodied in a transition (as opposed to PNP, which represents the entire task allocation process [101]). Even though this is not an explicit representation, it is still preferable because it leaves the representation of those protocols to a lower level of hierarchy and does not limit the planner to a specific task allocation protocol or synchronization mechanism.

Once more we return to the *Organize room* example. Figure 5.18 illustrates the *Organize room* example as described in Section 5.2.2 but does so for distributed execution. That is to say that instead of maintaining one CPNP in one of the two robots, the CPNP is distributed among the two robots. The upper CPNP is the CPNP maintained and executed by R_1 and the lower is maintained and executed by R_2. In addition, each robot maintains the task allocation CPNP (Figures 5.12 or 5.14). In the current example, the roles for *Open room*, *Go to room*, *Enter room* and *Clean room* operations are determined preliminary by the planner before execution, and these roles are not allocated dynamically during execution. Therefore, the CPNPs that represent these actions are built according to the description in Section 5.1.2

The *Go to right side* and *Go to left side* operations are dynamically allocated during execution while the *lift* operation should be synchronized using the strong dependence operator. In light of this, each robot maintains the CPNP of all three operations. When R_1 finishes executing the *Clean Room* operation, it sets the *task_alloc* variable to *true*. Subsequently, R_1 executes the synchronization mechanism in order to update the *task_alloc* variable in R_2 accordingly. The changes in the *task_alloc* variables trigger the task allocation CPNP of both R_1 and R_2, and each robot starts executing the task allocation process. The two robots execute the task allocation CPNP and each firing is synchronized using the synchronization mechanism. The execution of the task allocation CPNP by each of the robots represents the fact that both robots participate in the task allocation process.

In the *Organize room* example of section 5.2.2 two variables were defined: t_1, representing the *Go to right side* operation, and t_2, representing the *Go to left side* operation. The synchronization mechanism is executed during the execution of the task allocation CPNPs. The result of the task allocation is a synchronized assignment of values to t_1 and t_2. This means that the tokens that represent R_1 and R_2 get new values for either their t_1 or t_2 variables. The changes in the color of these tokens are synchronized and updated in the CPNPs of both R_1 and R_2.

When the task allocation is finished, the robots perform the *Go to right side* and *Go to left side* operations according to the values of their t_1 and t_2 variables (i.e., the robot with t_1=*in progress* in his tokens executes the *Go to right side* operation, and the robot with t_2=*in progress* in his tokens executes the *Go to left side* operation). When the two robots finish these operations, they execute together the *lift* operation using the strong dependence operator.

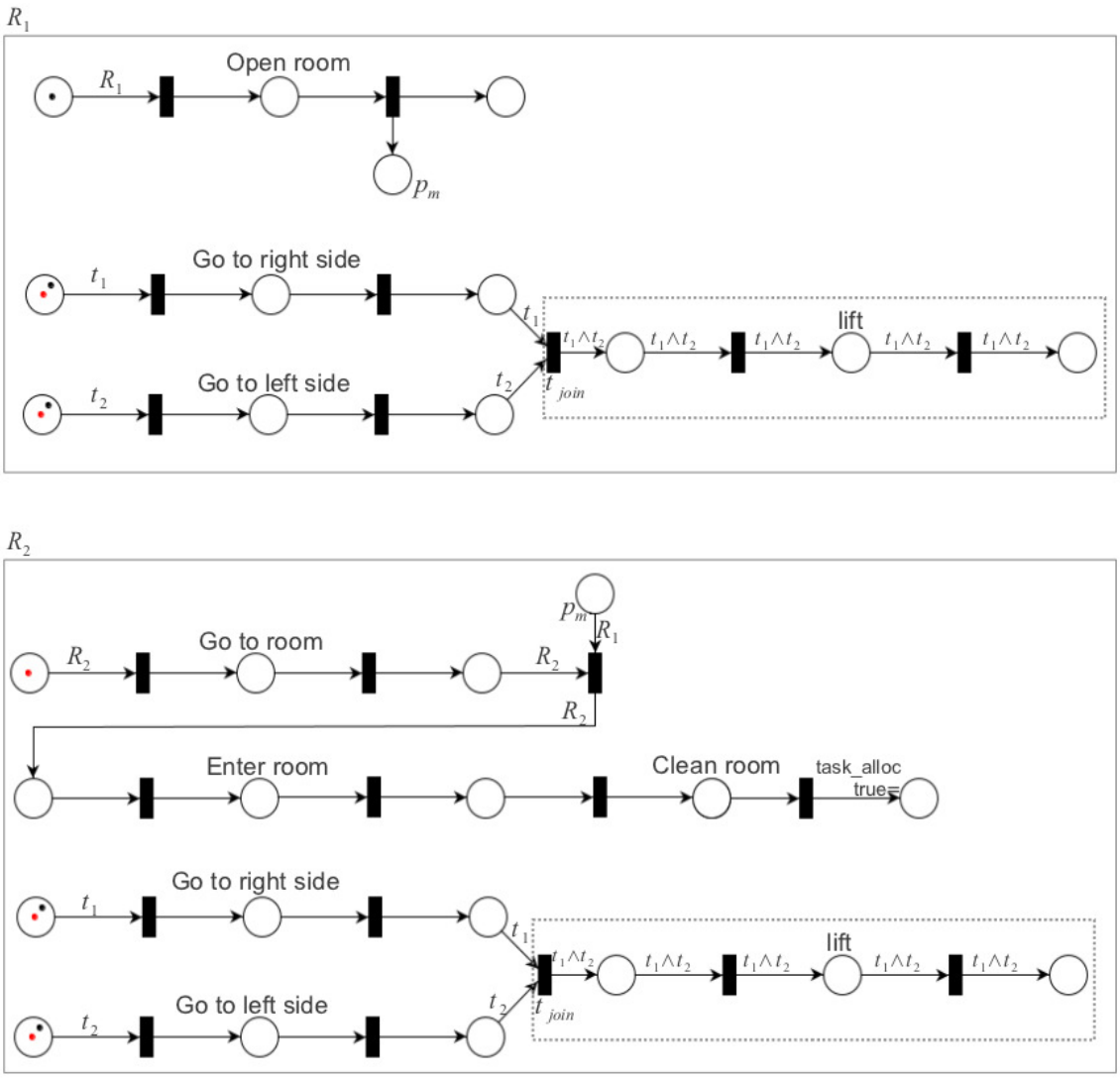

Figure 5.18 *Organize room* - distributed execution that dynamically allocates the *Go to right side* and *Go to left side* tasks

5.3 Execution Algorithm of a Multi-Robot CPNP

This section introduces the execution algorithms of a multi-robot CPNP. We describe the centralized and distributed algorithms in Sections 5.3.1 and 5.3.2, respectively. Finally, Section 5.3.3 proves that when relying on a reliable synchronization mechanism, the distributed execution algorithm produces the same results as the centralized one.

5.3.1 CPNP: Centralized Execution Algorithm Settings

In the following we introduce the execution algorithm of a centralized multi-robot CPNP. The algorithm consists of the same procedures as Algorithm 3.1, with some additions and changes which will be described in this section. The algorithm is executed by the centralized robot that sends commands to the other robots according to the CPNP.

Like Algorithm 3.1, Algorithm 5.1 starts from the initial marking M_0 and is terminated when a goal marking $M_n \in L$ is achieved. The algorithm has the same structure as was shown in Figure 3.21, with the addition of a new type to *TrType*: *task allocation*. A transition of the type *task allocation*, when enabled, executes the task allocation process. Similarly to Algorithm 3.1, Algorithm 5.1 assumes the availability of a set of implemented actions that the robots can execute: $Actions = \{a_1, .., a_k\}$.

In addition to the variables that were defined in Section 3.6, new variables will be added to the variables set (denoted as V) that is associated with the tokens of type *robotToken*:

1. A variable r that represents the robot ID;
2. A variable for each task that represents the task status (denoted as t_i);
3. A boolean variable $live$ that indicates whether the robot is alive;
4. A numeric variable $live\_robots$ that represents the current number of living robots in the system;
5. A boolean variable $task\_alloc$ that represents whether an allocation process should be executed;
6. Boolean variables that represent conditions that should be fulfilled before performing some tasks (each variable is denoted as v_{c_i} and the subset containing all variables of this kind is denoted as V_c).

Moreover, a new variable will be added to the variables set of each *resourceToken*. The new variable will represent the robot ID of the robot that currently uses the resource. If at the current moment there is no robot that is using the resource, the value of the variable will be null.

In Section 3.6 we defined a knowledge base *kb* that contains the robot's beliefs. In multi-robot CPNPs we define a knowledge base for each robot (i.e., for n robots, n knowledge bases will be created). This means that each token that represents a different robot (i.e., has a different value in the variable r) will have a pointer to a different knowledge base. The $live\_robots$, $task\_alloc$ and V_c variables should be identical in all of the knowledge bases. Therefore, these variables are maintained in the knowledge base that represents the knowledge of the centralized robot, while all the other knowledge bases merely maintain pointers to these variables.

The main procedure in a centralized multi-robot CPNP is *CMR_Execute* (Algorithm 5.1), which is executed by the centralized robot. This procedure is similar to the equivalent single-robot execute procedure (Algorithm 3.1). However, instead of creating a single knowledge base, it creates n knowledge bases (one for each robot, line: 1). The set of the knowledge bases is denoted as KB. The knowledge base is created separately for each robot and consists of the variables in V. The same variables make up the knowledge base of each robot, but their values differ from one robot to the next.

In order to preserve the updates of the knowledge base, the algorithm uses a *CMR_Listener* procedure (Algorithm 5.6). The *CMR_Execute* procedure starts the listener procedure immediately after the creation of the knowledge bases (in line 2), and terminates it at the end of execution (line 17). The *CMR_Listener* procedure listens to the messages sent by the robots. Each robot sends a message to the centralized robot when a variable in his knowledge base has been changed. Once a new message is received from robot R_i, *CMR_Listener* updates the kb_i.

Similar to Algorithm 3.1, for each transition $t \in T$ the *CMR_Execute* procedure (Algorithm 5.1) does the following:

Algorithm 5.1 Centralized CPNP execution algorithm - centralized robot, *CMR_Execute*

procedure CMR_Execute(CPNP $(P,T,A,\Sigma,V,C,G,E,M_0,L)$)

1: for each robot R_i create a knowledge base $kb_i \in KB$ from V
2: start CMR_Listener(KB)
3: $CurrentMarking \leftarrow M_0$
4: **while** $CurrentMarking \notin L$ **do**
5: **for all** $t \in T$ **do**
6: **if** $\exists$ robot R_i which is interrupted by an interrupt (denoted as j) and $kb_i.v_j \neq true$ **then**
7: $kb_i.v_j \leftarrow true$ // v_j is a boolean variable indicating an occurence of interrupt j
8: **end if**
9: **if** EnableTransition(t,CurrentMarking) **then**
10: $CurrentMarking \leftarrow CMR\_Fire(t, CurrentMarking)$
11: **if** $CurrentMarking \in L$ **then**
12: exit
13: **end if**
14: **end if**
15: **end for**
16: **end while**
17: stop CMR_Listener(KB)

Algorithm 5.2 Centralized CPNP execution algorithm - centralized robot, *CMR_Listener*

procedure CMR_Listener(KB)

1: **while** the system is executed **do**
2: Listen to messages from the robots
3: **if** a new message has been recived from robot R_i **then**
4: update kb_i
5: **end if**
6: **end while**

1. Checking if one of the robots has been interrupted and handle the interruption (lines 6-8);

2. Checking if t is enabled, using the *EnableTransition* procedure (Algorithm 3.2);

3. Firing by Algorithm 5.3 (line 10). Note that the *CMR_HandleTransition* procedure is called from the *CMR_Fire* procedure to perform the operations which are associated with the transition (i.e., executing the task allocation process, starting or terminating actions).

The *EnableTransition* procedure of the execution algorithm of the centralized multi-robot CPNP is the same as the *EnableTransition* procedure of the execution algorithm for a single-robot CPNP (described in Section 3.6, Algorithm 3.2). If the transition t is enabled, the multi-robot fire procedure is started (Algorithm 5.3).

The *CMR_Fire* procedure (Algorithm 5.3) executes the firing of t. Logically, this procedure is similar to the single-robot *fire* procedure (Algorithm 3.4), albeit with a slight difference: instead of executing the *HandleTransition* procedure and then the *fire* procedure (Algorithm 3.1, lines 10 -11), the *CMR_Fire* procedure calls the *CMR_HandleTransition* procedure with the following parameters: t, as well as a set of tokens (denoted as K) which should be fired by t (lines 4-5).

Algorithm 5.3 Centralized CPNP execution algorithm - centralized robot, CMR_Fire

procedure CMR_Fire(t,CurrentMarking)

1: **for all** $p_i \in t.input\_places$ **do**
2: $\quad CurrentMarking(p_i) = CurrentMarking(p_i) - E(p_i,t)$
3: **end for**
4: create a set of robotTokens K that contains all the robotTokens which are fired by t
5: CMR_HandleTransition(t,$CurrentMarking(p_i)$)
6: **for all** $p_o \in t.output\_places$ **do**
7: $\quad CurrentMarking(p_o) = CurrentMarking(p_o) + E(t,p_o)$
8: **end for**

The *CMR_HandleTransition* procedure (Algorithm 5.4) gets an enabled transition t and a set of tokens that should be fired (denoted as K). Unlike Algorithm 3.3, instead of activating or deactivating the action related to t (if it exists) according to the type of t, the procedure performs the fol-

lowing: if t is of type *task allocation* (line 1), then transition t is the transition that executes the task allocation process (i.e., transition t_1 in Figures 5.12 & 5.14) and the procedure executes the task allocation accordingly (line 2). When the task allocation is finished, the *CMR_HandleTransition* procedure assigns values to the tasks variables according to the task allocation. For each token $k \in K$, the procedure assigns a value to the appropriate task variable in accordance with the task allocated to the robot that is represented by k (lines 3-4).

When the type of t is *start*, each robot represented by one of the tokens in K should start the execution of the action associated with it and with t. In practice, if the type of t is *start* (line 6), then for each robotToken $k \in K$ (line 8), the procedure (executed by the centralized robot) sends a command to the robot R_i which is represented by k (i.e., $k.r = R_i$, the notation $k.r$ indicates the variable r in the token k, lines 12-13). If the robot represented by k is the centralized robot, then the robot starts the command associated with it and with t (lines 9-10). When the type of t is *end*, each robot represented by one of the tokens in K terminates the execution of the action associated with it and with t. The termination of the actions is done by sending commands to the robots that perform them, in a manner similar to the process of starting the actions as described above (lines 18-29).

So far, we have described those algorithms that are executed by the centralized robot. The following algorithms (Algorithms 5.5, 5.6 & 5.7) are executed by the slave robots. Each slave robot listens to the changes in its beliefs, informs the centralized robot, and performs the commands that it receives from the centralized robot. Algorithm 5.5 describes the activity of each slave robot. First, the slave robot builds a knowledge base *kb* with initial values that consist of all the variables in V. Initially, the slave robot sends his entire knowledge base to the centralized robot (lines 1-3). The values of the variables in V may constantly change under the influence of the slave robot's beliefs. The purpose of building a *kb* is for the slave robot to be able to inform the centralized robot of each change in the values of its variables. The slave robot simultaneously tracks the changes in

Algorithm 5.4 Centralized CPNP execution algorithm - centralized robot, *CMR_HandleTransition*

procedure CMR_HandleTransition(transition t,Tokens K)

```
1:  if t.t = task allocation then
2:     executes task allocation
3:     for all robotToken k ∈ K do
4:        assign values to the task variables according to the task allocation
5:     end for
6:  else if t.t = start then
7:     for all i = 1 : n do
8:        for all robotToken k ∈ K do
9:           if k.r = centralized robot then
10:             t.a.start()
11:          else
12:             if k.r = R_i then
13:                send to R_i a command to start the action a_i
14:             end if
15:          end if
16:       end for
17:    end for
18: else if t.t = end then
19:    for all i = 1 : n do
20:       for all robotToken k ∈ K do
21:          if k.r = centralized robot then
22:             t.a.end()
23:          else
24:             if k.r = R_i then
25:                send to R_i a command to terminate the action a_i
26:             end if
27:          end if
28:       end for
29:    end for
30: end if
```

its beliefs and listens to the messages from the centralized robot (line 4).

Algorithm 5.5 Centralized CPNP execution algorithm - *CMR_Slave*

procedure CMR_Slave(t,CurrentMarking)

1: build a knowledge base *kb* that consists of all the variables in V that are influenced from the slave robot's beliefs
2: initialize these variables according to the robot's beliefs
3: send the knowledge base to the centralized robot
4: execute concurrently *CMR_ListenerBeliefs(kb)* and *CMR_ListenerCommands()*

The *CMR_ListenerBeliefs* procedure (Algorithm 5.6) listens to the changes in the slave robot's beliefs during the execution of the CPNP (line 2). The procedure updates the *kb* after each change in beliefs (line 4) and informs the centralized robot about this change (line 5), albeit without sending the entire *kb* anew. The *CMR_ListenerCommands* procedure (Algorithm 5.7) listens to messages from the centralized robot (line 2) and performs the commands that are received from the centralized robot (line 4). Note that each message sent by the centralized robot consists solely of one command.

Algorithm 5.6 Centralized CPNP Execution Algorithm - slave robot, *CMR_ListenerBeliefs*

procedure CMR_ListenerBeliefs(kb)

1: **while** the system is executed **do**
2: Listen to changes in the robot's beliefs
3: **if** a variable has been changed **then**
4: update *kb*
5: send the new value to the centralized robot
6: **end if**
7: **end while**

5.3.2 CPNP: Distributed Execution Algorithm Settings

This section introduces an algorithm for a distributed execution of a multi-robot CPNP. This algorithm consists of the same procedures as Algorithm 3.1, with some additions and changes (similar

Algorithm 5.7 Centralized CPNP Execution Algorithm - slave robot, *CMR_ListenerCommands*

procedure CMR_ListenerCommands ()

```
1: while the system is executed do
2:    Listen to messages from the centralized robot
3:    if a new message has been received then
4:       perform the command that has been sent by the centralized robot
5:    end if
6: end while
```

to the centralized Algorithm 5.1 of Section 5.3.1). Each robot executes its own CPNP (denoted as R_iCPNP for robot R_i) using this algorithm.

Algorithm 5.8 starts from the initial marking M_0 and terminates when a goal marking $M_n \in L$ is achieved. The algorithm has the same structures as previously shown in Figure 3.21, with an addition of a new type of transition known as *task allocation* to *TrType*. The algorithm assumes the availability of a set of implemented actions $Actions = \{a_1, .., a_k\}$ that the robots can execute. The sets of the variables in distributed CPNPs consist of the same variables as in centralized CPNPs.

Each robot maintains a knowledge base *kb* that contains its beliefs. Similar to the knowledge bases which have been defined previously, this knowledge base consists of the values of the variables in V (the variables which are associated with the type of tokens known as *robotToken*). The $live\_robots$, $task\_alloc$ and the V_c variables should always be synchronized between all robots. Therefore, a synchronization mechanism is activated with each change in those variables. This mechanism informs the robots about each change in the mentioned variables. The variable r in the *resoureceTokens* is synchronized between all the robots using the same synchronization mechanism, and its value will always be identical for those *resoureceTokens* that represent the same resource.

The main procedure is called *DMR_Execute* (Algorithm 5.8). This procedure acts like the equivalent single-robot execute procedure (Algorithm 3.1), but has been adjusted to fit the distributed multi-robot execution. In order to keep the knowledge base and the value of the r variable

in the *resourceTokens* updated, a listener procedure is defined called *DMR_Listener* (Algorithm 5.8). This procedure is executed concurrently with the *DMR_Execute* procedure during CPNP execution. In practice, the *DMR_Listener* procedure is activated by *DMR_Execute* in line 2 and deactivated in line 18.

Algorithm 5.8 Distributed CPNP execution algorithm - *DMR_Execute*

procedure DMR_Execute(CPNP $(P,T,A,\Sigma,V,C,G,E,M_0,L)$)

```
1:  create knowledge base kb from V
2:  start DMR_Listener(kb)
3:  CurrentMarking ← M_0
4:  while CurrentMarking ∉ L do
5:     for all t ∈ T do
6:        if an interrupt i occurs and kb.v_i ≠ true then
7:           kb.v_i ← true // v_i is a boolean variable indicating an occurrence of interrupt i
8:        end if
9:        if DMR_EnableTransition(t,CurrentMarking) then
10:          HandleTransition(t)
11:          CurrentMarking ← DMR_Fire(t,CurrentMarking)
12:          if CurrentMarking ∈ L then
13:             exit
14:          end if
15:       end if
16:    end for
17: end while
18: stop DMR_Listener (kb)
```

The *DMR_Listener* procedure is executed by each robot separately and listens to messages from the teammates about changes in the variables that should be synchronized between the robots (line 2). For each robot R_i, it updates the R_iCPNP accordingly. If there is a change in one of the values of the synchronized variables that are part of the *kb* (i.e., $live\_robots$, $task\_alloc$ or one of the V_c variables), the procedure updates the *kb* with this change. If there is a change in the variable *r* of one of the *resourceTokens* (meaning that one of the shared resources has been allocated or released), the procedure changes the value of variable *r* in the appropriate *resourceToken* in R_iCPNP. The procedure updates the *r* variable in all *resourceTokens* that represent the same resource.

Algorithm 5.9 Distributed CPNP execution algorithm - *DMR_Listener*

procedure DMR_Listener (*kb*)

```
1: while the system is executed do
2:    Listen to changes in synchronized variables
3:    if live_robots, task_alloc or one of the V_c variables has been changed by one of the team-
      mates then
4:       update kb
5:    else if the r variable of one of the resourceToken has been changed by one of the teammates
      then
6:       update the appropriate resourceToken
7:    end if
8: end while
```

Similar to Algorithms 3.1 and 5.1, for each transition $t \in T$ the *DMR_Execute* procedure (Algorithm 5.8) does the following:

1. Checking if one of the robots has been interrupted and handling the interrupt (lines 6-7);
2. Checking if t is enabled, using the *DMR_EnableTransition* procedure (Algorithm 5.10);
3. Activating or deactivating the action associated with t, according to the type of t, using the *HandleTransition* procedure (line 10).
4. Firing t and generating a new marking, using the *DMR_Fire* procedure (line 11)

The *DMR_EnableTransition* procedure (Algorithm 5.10) checks if the transition is enabled, similar to the equivalent single robot *EnableTransition* procedure (Algorithm 3.2). The procedure checks if the guard on the transition is satisfied (line 1). Note that in some cases the guards can contain conditions which are associated with other robots (e.g., strong dependence operator, see Section 5.1.2). Therefore, the procedure assumes the existence of a conversation protocol which will be used in order to check for such conditions. Then, the procedure checks if the relevant tokens exist in each input place according to the arc expression. The arc expressions can consist of tokens that represent other robots; therefore, the markings of the CPNPs that represent those

robots should be synchronized (lines 3-6). The procedure returns *true* if the transition is enabled; otherwise, it returns *false*.

Algorithm 5.10 Distributed CPNP execution algorithm - *DMR_EnableTransition*

```
procedure DMR_EnableTransition(t,CurrentMarking)
 1: if G(t) = true then
 1:   // checks if the guard on t is satisfied
 2:   for all p_i ∈ t.input_places do
 3:     if E(p_i,t) consists of tokens that represent other robots then
 4:       synchronize the markings of those robots' CPNPs
 5:       update CurrentMarking
 6:     end if
 7:     if CurrentMarking(p_i) ⊉ E(p_i,t) then
 8:       return false
 9:     end if
10:   end for
11: else
12:   return false
13: end if
14: return true
```

The *HandleTransition* procedure for the execution algorithm of a distributed multi-robot CPNP is the same as the *HandleTransition* procedure for the execution algorithm of a single robot CPNP (described in Section 3.6, Algorithm 3.3). The *DMR_Fire* procedure (Algorithm 5.11) executes the firing of t. This procedure is similar to the single-robot *fire* procedure (Algorithm 3.4), albeit with a slight difference: when at least one of the variables that should always be synchronized between the robots is changed ($task\_alloc$, $live\_robots$, at least one of the variables in V_c, or the variable r in a *resourceToken*) the procedure broadcasts this information to all the other robots (lines 6-8). These variables are changed when a token is fired from t to one of the output places p_o through an arc that has an arc expression that changes these variables.

Algorithm 5.11 Destributed CPNP execution algorithm - *DMR_Fire*

procedure DMR_Fire(t,CurrentMarking)

1: **for all** $p_i \in t.input\_places$ **do**
2: $CurrentMarking(p_i) = CurrentMarking(p_i) - E(p_i,t)$
3: **end for**
4: **for all** $p_o \in t.output\_places$ **do**
5: $CurrentMarking(p_o) = CurrentMarking(p_o) + E(t,p_o)$
6: **if** the value of task_alloc, live_robots, or at least one of the variables in V_c has been changed, or the variable r in a resourceToken has been changed according to $E(t,p_o)$ **then**
7: inform all the robots about any change in these variables
8: **end if**
9: **end for**

5.3.3 CPNP Distributed Algorithm vs. CPNP Centralized Algorithm

In the current section it is proven that the distributed algorithm as introduced in Section 5.3.2 produces the same results as the centralized algorithm of Section 5.3.1. The proof assumes the existence of reliable conversation and synchronization protocols. As mentioned previously, in a distributed CPNP each robot R_i maintains and executes its own CPNP (denoted as R_iCPNP). Each R_iCPNP is divided into three parts: CPNP components in which R_i acts individually without commitments to other robots (independency), CPNP components in which R_i is weakly dependent on other robots, and CPNP components in which R_i is strongly dependent on other robots. We will prove in this section that the execution of each part of the distributed CPNP produces the same results as the execution of the equivalent part in the centralized CPNP.

Theorem 13. *Given a plan G that is represented by both distributed and centralized CPNP, Algorithm 5.8 for distributed CPNP produces the same results as Algorithm 5.1 for centralized CPNP.*

Proof. Assume, without loss of generality, that the multi-robot system consists of at least two robots R_i and R_j. The centralized algorithm sends commands to these robots according to the executed centralized CPNP. The distributed execution of this system consists of two execution algorithms (one for each robot). The distributed execution algorithm of R_i executes the R_iCPNP

and the distributed execution algorithm of R_j executes the R_jCPNP. The proof is divided into three parts, analogous to the three parts of each R_kCPNP ($k \in \{i, j\}$):

1. CPNP components in which R_i and R_j are not dependent on each other. Here, the distributed algorithm will simply execute these independent parts of the R_kCPNP and R_k will perform the actions according to the R_kCPNP.

   The centralized algorithm will send the same action commands to both R_i and R_j according to the part of the centralized CPNP that corresponds to R_kCPNP.

2. CPNP components in which R_i is weakly dependent on R_j. According to the definition of weak dependence (Section 4.1), R_i is dependent on R_j but R_j is not dependent on R_i. In a distributed execution, when R_j executes the part which is related to this weak dependency, it independently fires a token to place p_m (according to the weak dependence operator, Section 5.1.2). R_i also executes its R_iCPNP until it gets to the transition that should fire a token from place p_m (transition t_2 in Figure 5.9). Since this transition has an input arc with an arc expression that consists of a token that represents R_j, robot R_i starts a conversation with robot R_j and synchronizes the markings of place p_m until the transition is enabled (according to *DMR_EnableTransition* procedure, Algorithm 5.10, lines 3-6).

   Analogously, the centralized algorithm is executed according to the centralized weak dependence operator and activates the two robots in the same way.

3. CPNP components in which R_i is strongly dependent on R_j. According to the distributed strong dependence operator (depicted in Figure 5.10), each of the two transitions that participate in the distributed strong dependence operator of R_iCPNP cannot fire until its guards are satisfied. Lets denote these transitions as $t_{s_{R_i}}$ and $t_{t_{R_i}}$, respectively. The two transitions that participate in the relevant distributed strong dependence operator of R_jCPNP will be denoted as $t_{s_{R_j}}$ and $t_{t_{R_j}}$, respectively. The guards on $t_{s_{R_i}}$ are not satisfied until $t_{s_{R_j}}$ is enabled

and the guards on $t_{s_{R_j}}$ are not satisfied until $t_{s_{R_i}}$ is enabled. These guards are checked using a synchronization mechanism (*DMR_EnableTransition* procedure, Algorithm 5.10, lines 3-6). When both $t_{s_{R_i}}$ and $t_{s_{R_j}}$ are enabled, the algorithms of R_i and R_j jointly execute the firing of $t_{s_{R_i}}$ and $t_{s_{R_j}}$, respectively.

The firing of $t_{s_{R_i}}$ and $t_{s_{R_j}}$ is equivalent to the firing of transition t_s in the relevant centralized strong dependence operator (depicted in Figure 5.6). This firing is executed by the centralized algorithm. Similarly, the firing of $t_{t_{R_i}}$ and $t_{t_{R_j}}$ is equivalent to the firing of the transition t_t in the centralized strong dependence operator (Figure 5.6).

□

The above proof does not take into account the use of shared resources and dynamic task allocation. When the robots share resources, this is represented by assigning unique tokens to these resources (*resourceTokens*). Each such token has a variable that indicates the name of the resource and a variable that holds the ID of the robot that currently uses the resource. The distributed algorithm updates the other robots about any change in these variables (*DMR_Fire* procedure, Algorithm 5.11, lines 6-8), and monitors for upcoming changes from the other robots (*DMR_Listener* procedure, Algorithm 5.9, lines 5-6). Therefore, each robot always holds the real and most recent values for these variables.

In centralized executions, the values of the *resourceTokens* variables are always up-to-date thanks to the role of the centralized robot. Since the managing of shared resources relies only on the values of these variables, the managing of shared resources in the distributed algorithm is equivalent to the managing of shared resources in the centralized algorithm.

Dynamic task allocation is activated by the $task\_alloc$ variable (Section 5.2). This variable is always synchronized between the robots in the distributed execution (*DMR_Listener* procedure, Algorithm 5.9, lines 3-4). This synchronization guarantees that the task allocation mechanism in the distributed and centralized executions will be activated in the same situations. Since the same

task allocation mechanism is activated in both the centralized and distributed execution, the results of the allocation process will be identical.

We proved that for each of the three CPNP components, the distributed execution algorithm produces the same result as the centralized execution algorithm. Therefore, generally speaking, the distributed algorithm produces results identical to those of the centralized algorithm.

5.4 Summary

Building on the insights gained from the analysis in Chapter 4, this chapter introduced the CPNP representation for multi-robot architectures. CPNP combines individual state representation and joint state representation into a new approach called *partial state representation*. The Partial Joint State Representation is responsible for the space complexity of CPNP being superior to that of other representations. CPNP representations are divided into two categories: CPNP representation for centralized execution and CPNP representation for distributed execution. For each category, two operators were defined: a weak dependence operator and a strong dependence operator. These operators are analogous to the two possible kinds of dependencies (defined in Section 4.1) between robots coordinating and cooperating in a team. The current chapter also provided a CPNP representation for a task allocation process. This representation relies on the existence of a task allocation mechanism (depicted as a black box) in order to provide a generic task allocation representation. The task allocation process representation also represents a situation in which robots can "die" or become non-functional. Finally, the current chapter introduced the execution algorithms for centralized CPNP as well as distributed CPNP and it provided proof that the distributed execution algorithm produces the same result as the centralized execution algorithm, under the assumption that a reliable conversation and synchronization protocol exists.

Chapter 6

Reasoning about CPNPs

The previous chapters introduced our CPNP representation for representing either single-robot plans or multi-robots plans. But after modeling plans with Petri Net, an obvious question is: What can we do with the model? A major strength of Petri Nets is their support for analysis of many properties associated with concurrent systems [66]. These properties are called *behavioral properties*. Jensen [45] shows how to investigate the behavioral properties by the use of *state space*. In this chapter, we will give an overview about the behavioral properties and explain them in context of robotic systems (Section 6.1). Then, Section 6.2 will describe what is state space and how to build a state space for a given CPNP.

6.1 Behavioral Properties

This section gives an overview of the basic behavioral properties and explains them in the context of robotic systems. The description of the behavioral properties is based on [45] and [66]. Jensen presents some automatic tools that check these properties in CP Nets. He does this by creating a state space from a given CP Net, and then investigates these properties using the state space. These tools are specified in detail in [45].

Reachability. The reachability properties are concerned with determining whether a marking M' is reachable from another marking M, i.e., whether there is an occurrence sequence (firing sequence) starting from M which leads to the marking M' [45]. For example, we can use this property in order to check if each one of the markings that can be defined as a goal marking, can be reached from M_0.

Boundedness. The boundedness properties specify how many and which tokens a place may hold, when all reachable markings are considered [45]. This property specifies the maximal and minimal number of tokens that can reside on a place in any reachable marking. This property can be used in order to check if the number of resources in each reachable marking is correct and to validate that no new resources were incorrectly "born", as a result of a transition that fires and mistakenly creates new resource tokens (tokens that represent resources).

Home marking. A home marking [45] (denoted as M_{home}) is a marking which can be reached from any reachable marking. This means that it is impossible to have an occurrence sequence starting from M_0, which cannot be extended to reach M_{home}. In other words, we cannot do things which will make it impossible to reach M_{home} afterwards. An example of home marking, is a marking in which a token is positioned in place o and all the other places are empty in a work-flow net [9, 93, 94]. By this property, we can validate that some interrupt-handling CPNPs and goal markings can be reached from any state.

Liveness. A transition $t \in T$ is *live* [45] if, when starting from any reachable marking, we can always find an occurrence sequence containing t. In other words, we cannot do things which will make it impossible for the transition to occur afterwards [45]. A Petri Net *PN* is *live* [66] if each $t \in T$ is live. A live Petri Net guarantees a deadlock-free operation, no matter what firing sequence is chosen [66]. A *dead marking* [45] is a marking in which no binding elements are enabled. A

transition is *dead* [45] if there are no reachable markings in which it is enabled. These properties can help us know if a resource is not stuck in a dead transition, and therefore can not be released. In addition, we can validate that the robot can not be stuck in an undesirable state.

Fairness. Fairness properties give information about how often transitions occur in infinite occurrence sequences [45]. It lists the impartial transitions. A transition t is *impartial* [45] if it occurs infinitely often in all infinite occurrence sequences. This property can help us avoid a situation of starvation.

6.2 CPNP: Building State Spaces

Simulation can only be used to explore a finite number of executions of the system under consideration [45]. Therefore, we cannot guarantee that the simulations cover all possible executions. Hence, executions that terminate in an undesired state or lead to a deadlock, may still exist after a set of simulations have been conducted. A *state space*, in contrast, represents all possible executions of the system under consideration and can be used to *verify*, i.e., prove in the mathematical sense of the word, that the system possesses a certain formally specified property.

A state space is a directed graph in which each possible state of the system is represented by a vertex. Jensen [45] shows how to build, a state space for a colored Petri Net model in an automatic way and how to automatically analyze and verify the behavioral properties using state spaces. Furthermore, Jensen [45] shows advanced methods in order to reduce the space complexity of state spaces.

This section will describe how to build a state space for CPNPs. Given a state space, it can be used in order to automatically investigate the behavioral properties using Jensen's methods [45].

In previous sections, we introduced CPNP as a formal robotic representation based on colored Petri Nets. However, a CPNP is not a regular CPN. In CPNP, part of the variables in V should be

always synchronized. This means that once a transition fires and changes the value of one of these variables by firing, all other tokens change this value as well (in the algorithm presented in Section 3.6 the synchronization is expressed by a global knowledge base *kb* shared by tokens). The variables that should be synchronized are the variables that indicate hierarchies and interrupts. These variables can change not only by firing, but also by changes in the environment e.g., when they trigger interrupts. Therefore, Section 6.2.1 introduces transformation methods that bring CPNPs to be as close as possible to Colored Petri Nets. Then, Section 6.2.2 presents an algorithm for building a state space from a given CPNP.

6.2.1 Transformation Methods

As described in [45], each node in a state space represents reachable marking and each arc represents a transition firing. Since a transition firing is not the only way by which the markings in CPNP can change (as described above), this section will show some transformation methods on the CPNP. These transformations will result in a situation in which each marking (except M_0) in the state space will be followed by a firing of a transition. Building on these transformations, the method for building a state space for CPNP will be shown in the next section (Section 6.2.2).

A state space is a directed graph where we have a node for each reachable marking (a marking that can be reached from M_0) and an arc for each transformation, from one marking to another. In CPNP, one of the following factors can change the current marking:

1. An occurring binding element (i.e. changes referred to a firing of a transition according to the CP Nets' firing rules described in Section 2.2.3)
2. A change in the environment
3. A hierarchical call
4. An interrupt

The current marking can change by the four factors above. In the rest of this section, we will explain each one of these factors and some transformations on the CPNP which will help us build a state space of a given CPNP. The method of building a state space for CPNP will be discussed in the next section (Section 6.2.2).

The trivial way of changing the current marking is an *occurring binding element*. As described in [45], when a transition fires, tokens are removed from its input places and are added to its output places. In this way the current marking is changed. A *change in the environment* can changes the current marking only if there is a transition $t \in T$, such that t is enabled in this marking (according to Definition 4) and has a guard which is influenced by this change (i.e. a change which does or does not satisfy said guard).

Hierarchical calls change the marking by changing the position and color of tokens. As described in Section 3.3, CPNP models hierarchy using substitution places (as shown in Figure 3.12). A transition fires a token into a substitution place via an arc expression, which changes the value of the boolean variable that indicates this hierarchy (denoted as h). The current marking is changed to a marking which has a token in the substitution place with h set to *true*. In addition, each token that has a pointer to kb updates his color with $h = true$.

In order to build a state space, we will make the following transformation: first, we will remove the transition that connects the goal place with the start place in each of the subCPNPs. Then we will clone each subCPNP, according to the number of substitution places that refer to this subCPNP. Each substitution place will be replaced with the relevant subCPNP according to the following method:

1. Connect each input arc of the substitution place to the start place of the subCPNP.

2. Connect the goal place of the subCPNP to each output arc of the substitution place.

Figure 6.1 depicts the transformation of the hierarchy shown in Figures 3.12 and 3.13. The dashed box represents the subCPNP as shown in Figure 3.13. Note that the transition t_h is omit-

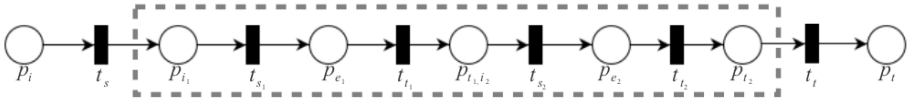

Figure 6.1 The result of the transformation of the CPNPs in Figures 3.12 and 3.13

.

ted from the subCPNP as part of the transformation method. The substitution place p_e from the superCPNP described in Figure 3.12, is replaced with the subCPNP (the dashed box).

Interrupt changes the current marking by changing the color of tokens. Once an interrupt takes place, the flag (boolean variable) that represents this interrupt is changed to *true* (as described in Section 3.4). This change is saved in the knowledge base *kb*. Therefore, all the tokens that have a pointer to *kb* are notified of this change. It causes the token in the CPNP that handles this interrupt to start moving, while the rest of the tokens get stuck. Hence, the current marking (denoted as M_{curr}) is changed to a marking in which all the tokens that have a pointer to *kb* change their color. As shown in Figure 3.19, when the recovery CPNP reaches the goal state, a transition fires a token to the start place via an arc expression, which changes the flag to *false* and the marking is changed back to M_{curr}.

Algorithm 6.1 is a transformation method that has been used in order to model interrupts in a state space. The algorithm gets a CPNP and returns this CPNP after transformation. In essence, the algorithm connects each place with an instance of the recovery CPNPs of the interrupts which may happen when a token exists in this place. The algorithm assumes an existence of a set of interrupt-handling CPNPs called I. Each element in this set is a CPNP which handles a specific kind of interrupt that may occur during the execution.

Lines 4-5 create a choice operator by creating a new transition t_{start} and connecting p to t_{start}. This transition has a guard which checks if the variable v_i that represents the interrupt i has been assigned to *true* by the world model listener. Note, that we slightly change the variable v_i; instead of a boolean variable that only expresses if the interrupt occurred or not, now v_i represents the

status of the interrupt. This change prevents a problematic situation in which the same interrupt will be handled concurrently, in two different places in the CPNP. While an interrupt handling CPNP i is executed, t_{start} fires a token and start the execution of the duplication of i.

Lines 6-7 clone the interrupt handling CPNP i, and then connect t_{start} to the new instance of i. As defined in Section 3.4 the first place in each interrupt-handling CPNP is called p_{start} and the last place is called p_{goal}. p_{goal} is an input place of the transition t_{goal}. Line 12 replaces the expression $v_i = true$, with the expression $v_i = in\_handling$, in each of the arcs expressions which contain this expression in the duplicated interrupt-handling CPNP. In the last step of Algorithm 6.1, line 12 connects t_{goal} to p to ensure that the plan will return to the state that was present before the interrupt.

Algorithm 6.1 CPNP Transformation Algorithm for Interrupts

procedure transformation (CPNP $(P,T,A,\Sigma,V,C,G,E,M_0,L)$)

1: **for all** $i \in I$ **do**
2: **for all** $p \in P$ **do**
3: **if** the interrupt that i handles can occur when a token exists in p **then**
4: Create new transition t_{start} with a guard that checks if $v_i = true$
5: Create an arc (p,t_{start}) // connect p to t_{start}
6: Create new instance i' of i (clone i)
7: Connect t_{start} to the first place in i', which is denoted as p_{start}
8: Set the arc expression $v_i = in\_handling$, on the arc (t_{start}, p_{start})
9: **for all** arc expressions in i' that contain the expression $v_i = true$ **do**
10: Replace the expression $v_i = true$ with the expression $v_i = in\_handling$
11: **end for**
12: Replace the arc (t_{goal}, p_{start}), with the arc (t_{goal}, p) // the new arc contains the same arc expression as the old one.
13: **end if**
14: **end for**
15: **end for**

6.2.2 State Space Building Method for CPNP

This section introduces how to build state spaces for a CPNP model. The basic idea of state spaces is to calculate all reachable states (markings) and state changes (firing or changes in colors) of the CPNP model and to represent this in a directed graph where the nodes correspond to the set of reachable markings and the arcs move from one marking to another, according to a transition's firing.

According to [45], a state space is a directed graph where we have a node for each reachable marking and an arc for each occurring binding element. There is an arc labeled with t from a node representing a marking M_1 to a node representing a marking M_2, iff t is enabled in M_1 and the firing of t in M_1 leads to the marking M_2. An arc from one marking to another represents only a single firing.

Algorithm 6.2 builds a state space from a given CPNP. The algorithm starts with creating a node from M_0 (Line 2). Then, it calculates all of the enabled markings. An *enable marking* is a marking that is derived from a firing of an enabled transition. Therefore, each node in the state space will be followed by an arc which indicates a firing of an enabled transition. A node is called *processed*, when all of its immediate successor markings have been calculated (Line 15). The algorithm is continued until all of the reachable markings have been processed (Line 5).

Note that Algorithm 6.2 ignores guards since they are influenced by environmental conditions. As a result, there can be markings that appear in the state space but will not actually be evaluated during the execution of CPNP. This means that the state space contains all of the reachable markings that can be reached from M_0, but the evaluation of some of these markings during the execution of CPNP depends upon the environmental conditions.

It should be emphasized that even if a CPNP model has a finite state space, the size of the graph may still be very large and impossible to store in a computer memory. This problem is called the *state explosion problem* [92]. Therefore, methods for reducing the space complexity of state spaces

Algorithm 6.2 CPNP Building a State Space

procedure BuildingSP (CPNP $(P,T,A,\Sigma,V,C,G,E,M_0,L)$)

```
1: Create new graph G
2: Create a node (in G) that represents M_0
3: current_marking ← M_0
4: Mark current_marking as unprocessed.
5: while ∃ unprocessed node do
6:    for all t ∈ T do
7:       if t is enabled then
8:          if ∄ a node M which represents the marking after the firing of t then
9:             Create new node M (in G) which represents the marking after the firing of t
10:            Mark this node as unprocessed.
11:         end if
12:         Create an arc from current_marking to M, labeled with t
13:      end if
14:   end for
15:   Mark current_marking as processed.
16:   if ∃ an unprocessed node M then
17:      current_marking ← M
18:   end if
19: end while
20: return G
```

are developed, for example: sweep-line method, symmetry method, equivalence method, etc. For more details refer to [45, 54].

Summary. This chapter described the behavioral properties (Section 6.1). These properties can be validated automatically in Petri Nets and give important information about the system. Jensen [45] built automatic tools which analyzes these properties in CP Nets, using state spaces. Therefore, Section 6.2 showed how to automatically build a state space for a CPNP model and presented an algorithm for doing so. Given a state space, we can automatically verify and analyze the behavioral properties using Jensen's tools. The automatic analysis methods for analyzing these properties are specified in details in [45]. The existence of these automatic analysis methods is a significant advantage of modeling robotic systems with Petri Nets.

Chapter 7

Summary and Future Work

7.1 Summary and Conclusions

Our work addresses the following two aspects:

1. Representing single-robot plans.

2. Representing multi-robot plans.

We describe a representation which is comprehensive, represents explicitly single-robot and multi-robot plans, is readable easily by humans, has minimum space requirements, provides validation and verification, and is suitable for real time execution (especially providing a satisfactory solution for dealing with interruptions). Firstly, we introduced our CPNP framework for modeling, analysis and execution of single-robot plans. CPNP is based on Colored Petri Nets. CPNP provides an explicit and comprehensive graphical representation and specified formal building blocks and operators for modeling robotic plans. CPNP allows modeling controllable events, decisions made by the robot, multiple concurrent processes and shared resources between multiple processes running concurrently in the robot. In addition, the framework provides a methodology to modeling robotic

plans in hierarchies. Modeling by the use of this methodology not only facilitates the readability of the representation but also reduces the space complexity.

Handling interrupts and uncontrollable events is a challenging task in Petri Nets based representations. The reason is that a Petri Net is a predefined model and the modeler should refer to each situation that may occur in advance [59]. This methodology can lead to a lot of problems and mistakes. Therefore, we introduced a smart mechanism to modeling and handling interrupts through the use of Colored Petri Nets. By changing tokens' color an interrupt handling plan is started or terminated. This methodology frees the modeler from specifying the interrupt handling representation in each state in which it may occur, facilitates the readability of the representation and reduces the space complexity. Furthermore, we provided an execution algorithm for executing a given CPNP representation.

The second part discussed the representations of multi-robot plans. Influenced by Gutnik and Kaminka's space complexity analysis [36] of representations for conversation protocols, we introduced a space complexity analysis of existing representations frameworks for multi-robot plans. The analysis examined the scalability of the existing representations. We classified the existing representations in two dimensions:

1. Representation according to individual state representation or joint state representation.

2. The Petri Net based technique (i.e., P/T Net or CP Net).

Results of the analysis are:

1. The choice of modeling according to individual state or joint state representation should be according to the dependencies between the robots.

2. CP Nets yield the best results when representing most of the multi-robot plans.

Based on the insights gained from the analysis, we have extended the CPNP framework for representing multi-robot plans. CPNP can model both centralized and distributed multi-robot plans

and provides the basic operators for representing multi-robot plans. These operators are built according to the result of the space complexity analysis in order to minimize the space requirements. In addition, CPNP allows for representing shared resources among the robots. Furthermore, a representation for a task allocation process that supports a situation in which robots can “die" or become non-functional is provided. Moreover, we provided two algorithms: one for executing CPNP representation of centralized multi-robot plans and the other for executing CPNP representation of distributed multi-robot plans.

Finally, we provided an algorithm for transforming a given CPNP representation to state space. This transformation leads to the ability of using Jensen’s tools [45] in order to validate behavioral properties of the plan. It gives the CPNP framework the ability to provide a design-analysis-design approach, which leads to improving the plan even before executing it in simulation or in real robots.

7.2 Open Challenges

We believe that this work can assist and motivate continuing research on representations of single-robot and multi-robot plans. In the future, we plan to extend the CPNP framework with the ability to represent temporal constraints (e.g., representing a constraint in which an Action B should start after Action A, but in parallel). We also plan to extend CPNP to support proactive information exchange in a dynamic environment (similar to [100]) and start conversations before the robots reach the states in which they are required. This improvement gives the ability to receive important information in advance and influences the decision making. In addition, we plan to investigate an algorithm for translating BDI representations into Petri Nets. This will be a major improvement for BDI representations since it will give them formal validation of behavioral properties of the represented plan.

Bibliography

[1] CCNx synchronization protocol. https://www.ccnx.org/.

[2] Colored petri nets. http://www.daimi.au.dk/CPnets/intro.

[3] Petri nets. http://users.abo.fi/lmorel/MoCs/slides/03-pn.slide.pdf, 2006.

[4] Embedded control systems design finite state machines and petri nets. http://en.wikibooks.org/wiki/Embedded_Control_Systems_Design/Finite_State_Machines_and_Petri_Nets, 2010.

[5] Computer simulation. http://www.cise.ufl.edu/fishwick/cap4800/, 2012.

[6] M. Ajmone Marsan, G. Conte, and G. Balbo. A class of generalized stochastic petri nets for the performance evaluation of multiprocessor systems. *ACM Transactions on Computer Systems (TOCS)*, 2:93–122, May 1984.

[7] R. C. Arkin. *A Behavior-based Robotics*. MIT Press, Cambridge, MA, USA, 1st edition, 1998.

[8] T. Balch and R. C. Arkin. Behavior-based formation control for multirobot teams. *IEEE Transactions on, Robotics and Automation*, 14(6):926–939, 1998.

[9] K. Barkaoui, R. Ayed, and Z. Sbaï. Workflow soundness verification based on structure theory of petri nets. *International Journal of Computing and Information Sciences*, 5(1):51–61, 2007.

[10] P. Bonasso. Issues in providing adjustable autonomy in the 3T architecture. In *Proceedings of the AAAI Spring Symposium on Agents with Adjustable Autonomy*, 1999.

[11] B. Browning, J. Bruce, M. Bowling, and M. Veloso. Stp: Skills, tactics and plays for multi-robot control in adversarial environments. *IEEE Journal of Control and Systems Engineering*, 219:33–52, 2005.

[12] D. Calisi, A. Farinelli, L. Iocchi, and D. Nardi. Multi-objective exploration and search for autonomous rescue robots. *Journal of Field Robotics*, 24(8-9):763–777, 2007.

[13] J. Capitán, M. T. J. Spaan, L. Merino, and A. Ollero. Decentralized multi-robot cooperation with auctioned POMDPs. *International Journal of Robotics Research*, 32(6):650–671, 2013.

[14] C. G. Cassandras and S. Lafortune. Petri nets. In *Introduction to Discrete Event Systems*, pages 223–267. Springer US, 2008.

[15] G. Ciardo, R. German, and C. Lindemann. A characterization of the stochastic process underlying a stochastic petri net. *IEEE Transactions on, Software Engineering*, 20(7):506–515, July 1994.

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

[17] R. S. Cost. *A framework for developing conversational agents*. PhD thesis, University of Maryland at Baltimore County, Department of Computer Science, 1999.

[18] R. S. Cost, Y. Chen, T. Finin, Y. K. Labrou, and Y. Peng. *Using Colored Petri Nets for Conversation Modeling*, volume 1916 of *Lecture Notes in AI*, pages 178–192. Springer-Verlag, September 2000.

[19] R. S. Cost, Y. Chen, Y. K. Labrou, and Y. Peng. Modeling agent conversations with colored petri nets. In *Working notes of the Autonomous Agents '99 Workshop on Specifying and Implementing Conversation Policies*, Seattle, Washington, May 1999.

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

[21] H. Costelha and P. Lima. Modelling, analysis and execution of multi-robot tasks using petri nets. In *Proceedings of the 7th international joint conference on Autonomous Agents and Multi-Agent Systems (AAMAS)*, volume 3, pages 1187–1190. International Foundation for Autonomous Agents and Multiagent Systems, 2008.

[22] H. Costelha and P. Lima. Petri net robotic task plan representation: Modelling, analysis and execution. *Autonomous Agents*, pages 65–90, 2010.

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

[24] L. De Silva and H. Ekanayake. Behavior-based robotics and the reactive paradigm, a survey. In *The 11th International Conference on Computer and Information Technology (ICCIT).*, pages 36–43. IEEE, 2008.

[25] M. de Weerdt and B. Clement. Introduction to planning in multiagent systems. *Multiagent and Grid Systems*, 5(4):345–355, 2009.

[26] J. Desel, A. Oberweis, T. Zimmer, and G. Zimmermann. Validation of information system models: Petri nets and test case generation. In *Systems, Man, and Cybernetics, IEEE International Conference on Computational Cybernetics and Simulation.*, volume 4, pages 3401–3406. IEEE, 1997.

[27] K. A. D'Souza and S. K. Khator. A survey of petri net applications in modeling controls for automated manufacturing systems. *Computers in industry*, 24(1):5–16, 1994.

[28] E. H. Durfee. Distributed problem solving and planning. In *Multiagent systems: A modern approach to distributed artificial intelligence*, pages 121–164. MIT Press, 1999.

[29] E. H. Durfee. Distributed problem solving and planning. In M. Luck, V. Marik, O. Stepankova, and R. Trappl, editors, *Multi-Agent Systems and Applications*, volume 2086 of *Lecture Notes in Computer Science*, pages 118–149. Springer Berlin / Heidelberg, 2006.

[30] X. Fan and J. Yen. R-cast: Integrating team intelligence for human-centered teamwork. In *proceedings of the national conference on artificial intelligence*, volume 22, page 1535. Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999, 2007.

[31] R. Fehling. A concept of hierarchical petri nets with building blocks. In *Proceedings of the 12th International Conference on Application and Theory of Petri Nets*, pages 370–389, June 1991.

[32] R. Fehling. A concept of hierarchical petri nets with building blocks. *Lecture Notes in Computer Science; Advances in Petri Nets 1993*, 674:148–168, 1993.

[33] R. E. Fikes and N. J. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial intelligence*, 2(3-4):189–208, 1972.

[34] E. Gat. Integrating reaction and planning in a heterogeneous asynchronous architecture for mobile robot navigation. *ACM SIGART Bulletin*, 2(4):70–74, 1991.

[35] C. V. Goldman and S. Zilberstein. Decentralized control of cooperative systems: Categorization and complexity analysis. *Journal of Artificial Intelligence Research*, 22:143–174, 2004.

[36] G. Gutnik and G. A. Kaminka. Representing conversations for scalable overhearing. *Journal of Artificial Intelligence Research*, 25(1):349–387, 2006.

[37] M. Hack. Petri net language. Technical report, Cambridge, MA, USA, 1976.

[38] L. E. Holloway, B. H. Krogh, and A. Giua. A survey of petri net methods for controlled discrete event systems. *Discrete Event Dynamic Systems*, 7(2):151–190, 1997.

[39] N. Howden, R. Rönnquist, A. Hodgson, and A. Lucas. Jack intelligent agents-summary of an agent infrastructure. In *5th International conference on autonomous agents*, 2001.

[40] W. Hseush and G. E. Kaiser. *Modeling concurrency in parallel debugging*, volume 25. ACM, 1990.

[41] B. Innocenti, B. Lopez, and J. Salvi. Resource coordination deployment for physical agents. In *From Agent Theory to Agent Implementation, 6th Int. Workshop AAMAS*, 2008.

[42] K. Jensen. Coloured petri nets. *Journal of Petri nets: central models and their properties*, pages 248–299, 1987.

[43] K. Jensen. An introduction to the theoretical aspects of coloured petri nets. In *of A Decade of Concurrency, Lecture Notes in Computer Science*, pages 230–272. Springer-Verlag, 1994.

[44] K. Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use*. Springer, 1997. Three Volumes.

[45] K. Jensen and L. M. Kristensen. Coloured petri nets. *Basic Concepts, Analysis Methods and Practical Use. Berlin: Spring-Verlag*, 2009.

[46] G. A. Kaminka and I. Frenkel. Flexible teamwork in behavior-based robots. In *Proceedings of the national conference on artifical intelligence (AAAI)*, pages 108–113, 2005.

[47] G. A. Kaminka and I. Frenkel. Integration of coordination mechanisms in the bite multi-robot architecture. In *IEEE International Conference on Robotics and Automation*, pages 2859–2866, 2007.

[48] D. Kartson, G. Balbo, S. Donatelli, G. Franceschinis, and G. Conte. *Modelling with Generalized Stochastic Petri Nets*. John Wiley & Sons, Inc., New York, USA, 1st edition, 1994.

[49] V. Khomenko and M. Koutny. LP deadlock checking using partial order dependencies. *Journal of CONCUR 2000 - Concurrency Theory*, pages 410–425, 2000.

[50] J. King, R. K. Pretty, and R. G. Gosine. Coordinated execution of tasks in a multiagent environment. *IEEE Transactions on Systems, Man and Cybernetics, Part A: Systems and Humans*, 33(5):615–619, 2003.

[51] K. Konolige. Colbert: A language for reactive control in sapphira. In *KI-97: Advances in Artificial Intelligence*, pages 31–52, 1997.

[52] K. Konolige. Saphira robot control architecture. *SRI International, Menlo Park, CA, Tech. Rep*, 2002.

[53] Y. T. Kotb, S. S. Beauchemin, and J. L. Barron. Petri Net-Based cooperation in Multi-Agent systems. In *Fourth Canadian Conference on Computer and Robot Vision, 2007. CRV'07*, pages 123–130, 2007.

[54] L. M. Kristensen. *State space methods for coloured Petri nets*. PhD thesis, PhD Dissertation, Department of Computer Science, University of Aarhus, Denmark, 2000.

[55] B. Lacerda and P. U. Lima. Designing petri net supervisors for multi-agent systems from LTL specifications. In *Proceeding of the 10th International Conference on Autonomous Agents and Multi-Agent Systems (AAMAS)*, volume 3, pages 1253–1254. International Foundation for Autonomous Agents and Multiagent Systems, 2011.

[56] J. Lee, M. J. Huber, E. H. Durfee, and P. G. Kenny. UM-PRS: An implementation of the procedural reasoning system for multirobot applications. In *Nasa Conference Publication*, pages 842–842. Citeseer, 1994.

[57] M. Loetzsch, H. D. Burkhard, and T. Rofer. *XABSL-a behavior engineering system for autonomous agents*. PhD thesis, Diploma thesis. Humboldt-Universität zu Berlin, 2004. Available online: http://www. martin-loetzsch. de/papers/diploma-thesis. pdf, 2004.

[58] M. Loetzsch, M. Risler, and M. Jungel. XABSL-a pragmatic approach to behavior engineering. In *Proceedings of IEEE/RSJ International Conference of Intelligent Robots and Systems (IROS)*, pages 5124–5129, 2006.

[59] M. Loewe, D. Wikarski, and Y. Han. *Higher order object nets and their application to workflow modeling*. Technische Universität Berlin, Fachbereich 13, Informatik, 1995.

[60] B. Ma. Modeling multi-agent systems with hierarchical colored petri nets. In *Artificial Intelligence Applications and Innovations II: IFIP TC12 and WG12. 5-Second IFIP Conference on Artificial Intelligence Applications and Innovations (AIAI-2005), Sept. 7-9, 2005, Beijing, China*, volume 187, page 167. Springer, 2005.

[61] A. Marino, L. Parker, G. Antonelli, and F. Caccavale. Behavioral control for multi-robot perimeter patrol: A finite state automata approach. In *IEEE International Conference on Robotics and Automation, 2009. ICRA'09.*, pages 831–836. IEEE, 2009.

[62] M. Matarić. Behavior-based robotics as a tool for synthesis of artificial behavior and analysis of natural behavior. *Trends in cognitive sciences*, 2(3):82–86, 1998.

[63] H. Mazouzi, A. Seghrouchni, and S. Haddad. Open protocol design for complex interactions in multi-agent systems. In *Proceedings of the First International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS-02)*, pages 517–526, Bologna, Italy, 2002. ACM.

[64] M. Miranda and A. Perkusich. Modeling and analysis of a multi-agent system using colored petri nets. In *Workshop on Applications of Petri Nets to Intelligent System Development*, Williamsburg, USA, 1999.

[65] D. Moldt and F. Wienberg. Multi-agent-systems based on coloured petri nets. *Journal of Application and Theory of Petri Nets*, pages 82–101, 1997.

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

[67] K. L. Myers. User guide for the procedural reasoning system. *SRI International, Menlo Park, CA*, 1997.

[68] J. Mylopoulos, A. Borgida, M. Jarke, and M. Koubarakis. Telos: Representing knowledge about information systems. *ACM Transactions on Information Systems (TOIS)*, 8(4):325–362, 1990.

[69] P. F. Palamara, V. A. Ziparo, I. Iocchi, D. Nardi, and P. Lima. Teamwork design based on petri net plans. In *RoboCup 2008: Robot Soccer World Cup XII*, pages 200–211, 2008.

[70] L. E. Parker. On the design of behavior-based multi-robot teams. *Advanced Robotics*, 10(6):547–578, 1995.

[71] J. L. Peterson. *Petri net theory and the modeling of systems*. Prentice-Hall, Inc., Englewood Cliffs, N.J 07632, 1981.

[72] C. A. Petri. *Communication with automata*. PhD thesis, Rome Air Development Center, Rome, NY, 1966.

[73] P. Pirjanian. Behavior coordination mechanisms-state-of-the-art. *Institute for Robotics and Intelligent Systems, School of Engineering, University of Southern California, Tech. Rep. IRIS-99-375*, 1999.

[74] A. Pokahr, L. Braubach, and W. Lamersdorf. A flexible bdi architecture supporting extensibility. In *Intelligent Agent Technology, IEEE/WIC/ACM International Conference on*, pages 379–385. IEEE, 2005.

[75] M. Purvis and S. Cranefield. A layered approach for modeling agent conversations. In *Proceedings of the Second International Workshop on Infrastructure for Agents, MAS and Scalable MAS, the Fifth International Conference on Autonomous Agents*, pages 163–170, Montreal, Canada, 2001.

[76] F. Py, K. Rajan, and C. McGann. A systematic agent framework for situated autonomous systems. In *Proceedings of the 9th International Conference on Autonomous Agents and Multi-Agent Systems (AAMAS)*, volume 2, pages 583–590. International Foundation for Autonomous Agents and Multiagent Systems, 2010.

[77] D. V. Pynadath and M. Tambe. An automated teamwork infrastructure for heterogeneous software agents and humans. *Autonomous Agents and Multi-Agent Systems*, 7(1):71–100, 2003.

[78] A. S. Rao and M. P. Georgeff. BDI agents: From theory to practice. In *Proceedings of the first international conference on multi-agent systems (ICMAS-95)*, pages 312–319. San Francisco, 1995.

[79] M. Risler, M. Loetzsch, M. Jungel, T. Krause, and B. Schmitz. XABSL web site. http://www.xabsl.de, 2009.

[80] M. Risler and O. von Stryk. Formal behavior specification of multi-robot systems using hierarchical state machines in XABSL. In *International Conference on Autonomous Agents and Multi-Agent Systems (AAMAS) - Workshop on Formal Models and Methods for Multi-Robot Systems,(Estoril, Portugal)*, 2008.

[81] M. Roth, R. G. Simmons, and M. M. Veloso. Reasoning about joint beliefs for execution-time communication decisions. In *Proceedings of the Fourth International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS-05)*, pages 786–793, 2005.

[82] S. Russell and P. Norvig. *Artificial Intelligence - A Modern Approach*. Prentice-Hall, Inc, Englewood Cliffs, NJ, 1995.

[83] S. J. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach, 2nd Ed.* Prentice Hall, Englewood Cliffs, NJ, 2002.

[84] Y. Sakagami, R. Watanabe, C. Aoyama, S. Matsunaga, N. Higaki, and K. Fujimura. The intelligent asimo: System overview and integration. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, volume 3, pages 2478–2483. IEEE, 2002.

[85] W. Sheng and Q. Yang. Peer-to-peer multi-robot coordination algorithms: Petri net based analysis and design. In *Proceedings of the IEEE/ASME International Conference on Advanced Intelligent Mechatronics.*, pages 1407–1412. IEEE, 2005.

[86] Y. Shoham. Agent-oriented programming. *Artificial intelligence*, 60(1):51–92, 1993.

[87] J. Sudeikat, L. Braubach, A. Pokahr, and W. Lamersdorf. Validation of BDI agents. In J. D. A. E. F. S. R. Bordini, M. Dastani, editor, *The 4th International Workshop on Programming Multi-Agent Systems (PROMAS-2006)*, pages 185–200, Berlin, Heidelberg, 2006. Springer.

[88] M. Tambe. Agent architectures for flexible, practical teamwork. In *Proceedings of the National Conference on Artificial Intelligence*, pages 22–28, 1997.

[89] M. Tambe. Towards flexible teamwork. *Journal of Artificial Intelligence Research (JAIR)*, 7:83–124, 1997.

[90] S. Tousignant, E. V. Wyk, and M. Gini. An overview of xrobots: A hierarchical state machine based language. In *Proceedings of the ICRA-2011 Workshop on Software development and Integration in Robotics*, 2011.

[91] I. Toyn and A. Galloway. Formal validation of hierarchical state machines against expectations. In *18th Australian Software Engineering Conference, (ASWEC)*, pages 181–190. IEEE, 2007.

[92] A. Valmari. The state explosion problem. *Lectures on Petri Nets I: Basic Models*, pages 429–528, 1998.

[93] W. van der Aalst. Verification of workflow nets. *Application and Theory of Petri Nets 1997*, pages 407–426, 1997.

[94] W. van der Aalst. The application of petri nets to workflow management. *Journal of Circuits Systems and Computers*, 8:21–66, 1998.

[95] N. Viswanadham and Y. Narahari. Coloured petri net models for automated manufacturing systems. In *Proceedings of IEEE International Conference on Robotics and Automation*, volume 4, pages 1985 – 1990, mar 1987.

[96] F. Y. Wang, K. J. Kyriakopoulos, A. Tsolkas, and G. N. Saridis. A petri-net coordination model for an intelligent mobile robot. *IEEE Transactions on Systems, Man and Cybernetics*, 21(4):777–789, 1991.

[97] M. Winikoff. Jack intelligent agents: An industrial strength platform. In R. Bordini, M. Dastani, J. Dix, and A. Fallah Seghrouchni, editors, *Multi-Agent Programming*, volume 15 of *Multiagent Systems, Artificial Societies, and Simulated Organizations*, pages 175–193. Springer US, 2005.

[98] M. J. Wooldridge. *An introduction to multiagent systems*. Wiley, 2002.

[99] D. Xu, R. Volz, T. Ioerger, and J. Yen. Modeling and verifyiüng multi-agent behaviors using predicate/transition nets. In *Proceedings of the 14th international conference on Software Engineering and Knowledge Engineering*, pages 193–200. ACM, 2002.

[100] J. Yen, J. Yin, T. R. Ioerger, M. S. Miller, D. Xu, and R. A. Volz. Cast: Collaborative agents for simulating teamwork. In *International joint conference on artifical intelligence*, pages 1135–1144, 2001.

[101] V. A. Ziparo, L. Iocchi, P. U. Lima, D. Nardi, and P. F. Palamara. Petri net plans. *Autonomous Agents and Multi-Agent Systems*, 2010.

[102] V. A. Ziparo, L. Iocchi, D. Nardi, P. F. Palamara, and H. Costelha. Petri net plans: a formal model for representation and execution of multi-robot plans. In *Proceedings of the 7th international joint conference on Autonomous Agents and Multi-Agent Systems*, volume 1, pages 79–86, 2008.

[103] W. M. Zuberek. Timed petri nets and preliminary performance evaluation. In *Proceedings of the 7th annual symposium on Computer Architecture*, ISCA '80, pages 88–96, New York, NY, USA, 1980. ACM.

[104] W. M. Zuberek and I. Bluemke. Hierarchies of place/transition refinements in petri nets. In *Proceedings of Conference on Emerging on Technologies and Factory Automation*, pages 355–360, 1997.

תקציר

מערכות המורכבות מרובוט יחיד ומערכות מרובות רובוטים צוברות עניין רב הן באקדמיה והן בתעשייה. ייצוג של המערכות הללו לצורך ניתוח, בקרה ואימות של תכונות מסוימות הוא חשוב מאוד עבור שני סוגי המערכות גם יחד, בין אם מדובר על מערכת עם רובוט יחיד ובין אם על מערכת מרובת רובוטים. לאחרונה הוצעו שיטות רבות לייצוג המערכות הנ״ל אך היתרונות והחסרונות של השיטות הללו ומידת התאמתם לייצוג של מערכות רובוטיות עדיין לא נחקרו. בנוסף, ישנם אתגרים רבים ששיטת ייצוג טובה צריכה להתמודד איתם ולהתייחס אליהם בעת ייצוג של מערכות רובוטיות בסביבה דינאמית ולא צפויה כגון: טיפול יעיל בהפרעות לא צפויות, מידול של תהליכים מקביליים, הקטנת סיבוכיות המקום של הייצוג, ניתוח אוטומטי ואימות של מאפיינים מסוימים במערכת ועוד.

עבודה זו מטפלת בשתי הסוגיות הנ״ל גם יחד. ראשית, היא מציגה שיטת ייצוג חדשה המבוססת על *Colored Petri Nets*, הנקראת *Colored Petri Net Plans (CPNPs)* המשמשת לייצוג מערכות המורכבות מרובוט יחיד. השיטה החדשה יודעת להתמודד באופן יעיל עם כל האתגרים שהוצגו לעיל ומציעה אבני בניין מפורשות לצורך בניית הייצוג. שנית, העבודה מציגה ניתוח סיבוכיות מקום של שיטות ייצוג קיימות ובוחנת את מידת ההתאמה שלהן לייצוג מערכות מרובות רובוטים. לבסוף, העבודה מציגה הרחבה ל - CPNP לצורך ייצוג של מערכות מרובות רובוטים, הרחבה הכוללת אופרטורים מתאימים לייצוג מערכות מרובות רובוטים, ריכוזיות ומבוזרות. האופרטורים הללו נבנו על פי ניתוח סיבוכיות המקום שהוצג וכתוצאה מכך הייצוג המתקבל ע״י שיטה זו הוא ברור יותר, מובן יותר וחסכוני במקום.

עבודה זו נעשתה בהדרכתו של פרופסור גל א .קמינקא מן המחלקה למדעי המחשב של אוניברסיטת בר-אילן.

בס"ד

CPNP: שיטה לייצוג מערכות המורכבות מרובוט יחיד ומערכות מרובות רובוטים

לימור מרציאנו

עבודה זו מוגשת כחלק מהדרישות לשם קבלת תואר
מוסמך במחלקה למדעי המחשב של אוניברסיטת בר-אילן

רמת גן

תשע״ג