University of Évry
Laboratory IBISC

PhD Thesis

in Computer Science

Modelling and Analysing Open Reconfigurable Systems

Defended by:
Viet Van PHAM

Supervisors:
Hanna KLAUDEL
Frédéric PÉSCHANSKI

Jury:

Reviewers:	Laure PETRUCCI
	Fabrice MOURLIN
Examinators:	Romain DEMANGEON
	Franck POMMEREAU
	Cinzia DI GIUSTO
Supervisors:	Hanna KLAUDEL
	Frédéric PÉSCHANSKI

September 26, 2014

Abstract

Today we witness the rapid spread of highly dynamic reconfigurable and distributed infrastructures that we group under the common name of *open reconfigurable systems*. The communication topology of those systems can dynamically change – or reconfigure – as a consequence of an internal or external concurrent activity. Labelled transition semantics for open systems allow to take into account the external activities in an implicit way.

Open reconfigurable systems are commonly modeled using formalisms inspired by the π-calculus. Name passing makes it possible to model dynamic communication topologies. In this thesis, we introduce the π-graphs, a variant of the π-calculus that among other things enjoys a natural graphical interpretation. Moreover, the formalism has been designed to serve as an intermediate language between the abstract π-calculus and more concrete realizations, especially in the realm of high-level Petri nets.

We first propose a faithful encoding of π-graphs into high-level Petri nets that are supported by common modelling and verification tools. We show that the translation can be lifted to an isomorphism between the two formalisms. Hence, the prototype tools that are developed specifically for π-graphs can be used in conjunction with more mature and general tools for Petri nets.

Based on this bidirectional encoding, we develop a high-level variant of the linear temporal logic – namely the *open system logic* – to specify properties about the dynamic evolution of systems taking into account interactions within open environments. The atomic propositions of the logic precisely capture the state properties of π-graphs processes.

The whole framework has been implemented during the course of this thesis. Our prototype tool provides a simulator for π-graphs models. These models can also be compiled into high-level Petri nets and then manipulated using the SNAKES framework. Finally, we provide an encoding of open system logic into linear temporal logic with state properties to be used with the NECO model checker for the automated verification of properties. Thanks to the bidirectional encoding from-and-to high-level Petri nets, counterexamples provided by the model checker for Petri nets can be easily reconstructed in terms of the original π-graphs.

Keywords: Reconfigurable systems, Petri nets, π-calculus, temporal logic.

Résumé

Les *systèmes ouverts reconfigurables* sont aujourd'hui omniprésents dans le paysage informatique : réseaux mobiles, calculs et données dans «le nuage», etc. Une particularité de ces systèmes est que leur topologie de communication évolue dynamiquement – nous parlerons de *reconfiguration* – en conséquence d'activités concurrentes internes ou externes. Les systèmes de transitions étiquetées pour les systèmes ouverts permettent de prendre en considération l'environnement extérieur de façon implicite.

Les systèmes ouverts reconfigurables sont souvent modélisés par des formalismes inspirés ou dérivés du π-calcul. Le passage de nom permet de modéliser la dynamique des topologies de communication. Dans cette thèse, nous introduisons les π-graphs, une variante du π-calcul qui possède, entre autre, une interprétation graphique naturelle. De plus, le formalisme a été conçu pour servir de langage intermédiaire entre le π-calcul abstrait et des formalismes plus concrets, en particulier dans la famille des réseaux de Petri de haut-niveau.

Nous proposons tout d'abord une traduction formelle et prouvée des π-graphes vers des réseaux de Petri de haut niveau supportés par des outils de modélisation et de vérification courants. Nous montrons que cette traduction peut-être élevée au rang d'isomorphisme entre les deux formalismes. Ainsi, les outils prototypes que nous avons développés dans le cadre des π-graphes peuvent travailler de concert avec des outils plus stables et plus généraux basés sur les réseaux de Petri.

En se basant sur cette traduction bi-directionnelle, nous développons une extension de la logique temporelle linéaire (LTL) – la logique des systèmes ouverts reconfigurables – permettant de spécifier des propriétés portant sur la dynamique d'évolution de la topologie de communication dans le cadre d'environnements ouverts. Les propositions atomiques de cette logique caractérisent précisément les propriétés d'état des π-graphs.

Un prototype d'outil a été développé dans le cadre de cette thèse pour valider expérimentalement l'approche proposée. Cet outil fournit un simulateur pour les modèles exprimés dans le formalisme des π-graphes. Ces modèles peuvent être compilés en réseaux de Petri de haut niveau et manipulés dans le cadre de l'outil SNAKES. Enfin, nous proposons une traduction de la logique des systèmes ouverts reconfigurables vers la logique de plus bas niveau supportée par le vérificateur de modèle NECO. Grâce à notre preuve constructive d'isomorphisme entre les π-graphes et leur traduction en réseaux de Petri, les contre-exemples générés pour les réseaux de Petri en cas d'invalidation de proposition par NECO peuvent être réinterprétées et expliquées dans les termes des π-graphes.

Mots-clés: Systèmes reconfigurables, Réseaux de Petri, π-calcul, logique temporelle.

Acknowledgments

First and foremost, I would like to thank my PhD supervisors, Hanna Klaudel and Frédéric Péschanski, for supporting me during the past three years. They patiently provided the vision, encouragement and advise necessary for me to proceed through the doctoral program and complete my thesis.

I also wish to thank the reviewers, *Laure Petrucci* and *Fabrice Mourlin*, for their valuable comments and suggestions to improve the quality of the thesis.

I would like to express my sincere thanks to *Franck Pommereau* for answers to my technical questions about the tool SNAKES, to *Lukasz Fonc* for answers about the tool NECO. In addition, I would like to thank *Aurelien Deharbe* for his survey of the π-calculus tools.

To my colleagues at laboratory IBISC – University of Évry, I am grateful for the chance to become a member of the lab. Thank you for welcoming me as a PhD student and also as a friend.

I would like to acknowledge the Ministry of Education and Training – Vietnam for providing the scholarships for my doctoral studies.

Finally, I sincerely thank to my parents for their encouragement through these years. Special thanks to my wife for her great support, patience and understanding.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Today we witness the rapid spread of highly dynamic reconfigurable and distributed infrastructures that we group under the common name of *open reconfigurable systems*. The communication topology of those systems can dynamically change – or reconfigure – as a consequence of an internal or external concurrent activity. We can find out many examples of this type of system in the real life, such as in *ad-hoc networks* where nodes can dynamically appear or disappear or in the context of *cloud computing* where resources such as computation tasks or virtual machines are spawned and destroyed dynamically. Indeed, the overall structure of such systems is highly flexible and involves many dynamic reconfigurations. Another example is in *banking systems* where the server must establish secure channels before performing money transfer transactions, and then close these channels in a secure way. The dynamics of the structure in such systems play a fundamental role to enforce the security of the transactions.

These kinds of large-scale distributed systems are of a very complex nature. Hence, the design phase of these systems is a very difficult part of the development process, and it exhibits a very high level of risks. To support this design phase, modelling tools represent a very important requirement. At the other end of the development process, it is unavoidable that even in high-assurance systems some defficiency will occur during the maintenance phase. To understand the cause of such a defficiency and the ways the system could be corrected, modelling tools can also play a significant role. Indeed, they can be used to reproduce the defficiency in the models and examine how these models could be corrected. Correcting the models probably represents the best starting point for the correction of the system itself.

Given the extreme complexity of the designs, one specific modelling tool cannot fulfill all the design requirements: the tools must be able to let engineers focus on specific aspects of the systems. In this thesis, we discuss the tool requirements focusing on the aspects related to dynamic reconfigurations.

There are two different and complementary ways to approach the modelling of systems.

In the *closed system* approach, the objective is to describe the system itself together with the environment in which it is supposed to operate. In the *open system* approach, the objective is instead to model the system and the ways it can interact with its environment. The important difference here is that the environment is kept as general as possible. Closed systems are in general easier to formalize and it is also more likely to be supported by modelling tools. However, as the interactions within the system and its environment grow in complexity, the closed systems approach becomes less and less adapted. Indeed, one must describe explicitly the system, the environment and the complex interactions between the two. In general, we must consider some specific instances of the environment so that the approach remains feasible but then we loose generality. Comparatively, in open systems we focus on the complexity of the model exclusively. The complexity of the environment must be taken into account by the formalism itself. This makes the models simpler at the price of more complex formalisms. It is thus more difficult to develop modelling tools in this setting. It is clear that dynamic reconfigurations represent quite complex interactions between the systems in which they occur and their external environment.

Another argument in favor of the open systems approach is defended by the creator of the π-calculus Robin Milner in [9]:

> *You can't do behavioural analysis with the chemical semantics [...] I think the strength of labels is that you get the chance of congruential behaviours.*

This says, in a way, that only the open systems approach allows to decompose the models into sub-components, analyze the behaviors of these sub-components separately while still understanding the behavior of the whole composition. Put in other terms, the open systems can be modelled in a *compositional* way.

In this thesis, our starting point is the formalism of π-graphs [47], a variant of the π-calculus that among other things both enjoys a traditional process algebra representation as well as a visual representation in the spirit of Petri-nets [53] and the Petri-box calculus [11]. Moreover, the formalism has been designed to serve as an intermediate language between the abstract π-calculus and more concrete realizations in terms of high-level Petri nets [31]. Our main objective in this thesis is to develop enough the theory of π-graphs so that modelling and verification tools based on this formalism can be developed in practice.

1.1 Contributions

Our first contribution is to propose a faithful encoding of π-graphs into high-level Petri nets that are supported by common modelling and verification tools, especially the

SNAKES framework [52] (but also HELENA [25], Maria [36], etc.). Unlike previous attempts, we show that this translation can be lifted to an isomorphism between the two formalisms. Hence, we can go back-and-forth between the π-graphs and the translated net. Compared to other translations of π-calculi into Petri nets, the proposed translation is the only one to support both the open systems approach of modelling and the automated verification of behavioural properties.

Based on this bidirectional encoding, we also develop a high-level variant of the linear temporal logic – namely the *open system logic* – to specify properties about the dynamic evolution of systems taking into account an open environment. The atomic propositions of the logic precisely capture the state properties of π-graphs processes. This logic is much more concrete than the usual logics developed in the realm of the π-calculus [18, 58, 16]. As such, we think it contributes to the general question of what it is to reason about open reconfigurable systems. Moreover, thanks to our isomorphism, the logic is at the same time a logic about the π-graphs and a logic about the translated Petri nets. This way, a counterexample for a property that fails on the Petri nets side can be easily reinterpreted on the initial π-graph model. This also means that the tools that are developed specifically for π-graphs can be used in conjunction with more mature and general tools for Petri nets.

As a final contribution, the whole framework has been implemented during the course of this thesis. Our prototype tool provides a simulator for π-graph models. This model can also be compiled into high-level Petri nets and then manipulated using the SNAKES framework [52, 51]. Finally, we provide an encoding of open system logic into linear temporal logic with state properties to be used with the NECO model checker [28, 27] for the automated verification of properties.

1.2 Structure of the thesis

The structure of the thesis is as follows.

In chapter 2, we discuss the scientific context of the thesis. We present some background about open reconfigurable systems and how the π-calculus can be used to model such systems. We also recall the main Petri nets feature on which our translation is based. Most importantly, we survey the related work concerning translations of π-calculus variants into Petri nets. Finally, in a synthesis we discuss the scientific positioning of this thesis.

In chapter 3, we present at an informal level the formalism of π-graphs and how to use it to model open reconfigurable systems. We illustrate this with a simple but non-trivial example of an open client/server system. We also describe some user-level interactions with our prototype tool.

In chapter 4, we present formally the syntax and the operational semantics of π-graphs.

In chapter 5, we present the translation of π-graphs into high-level Petri nets, and prove that the translation is an isomorphism.

In chapter 6, we introduce the *open system logic* (OSL) which is used to specify properties of open reconfigurable systems. We show how to translate π-graphs properties into equivalent ones on the translated Petri nets, and then how to check these properties using the model checker NECO. Some experimental results are presented at the end of the chapter.

Finally, the conclusion and perspectives for this research work are presented in chapter 7.

Chapter 2

Background and related work

The goal of this chapter is to describe the scientific context of this thesis. First, in Section 2.1 we see how the notion of open reconfigurable systems is presented, especially in connection with the π-calculus. Then, in Section 2.2, we provide some background information about Petri nets, especially the high-level Petri net features that we require in this thesis. Section 2.3 surveys the main related work concerning translations of π-calculus variants intro Petri nets. Finally, in Section 2.4, we synthesize our scientific positioning and provide an overview of our approach.

2.1 Open reconfigurable systems in the π-calculus

As mentioned in the introduction, open reconfigurable systems are systems whose communication topologies can dynamically change (reconfigure) as a consequence of an internal or external concurrent activity.

The π-calculus [41, 55, 44], developed by Robin Milner, is a process algebra that can be considered as the continuation of the calculus of communication systems (CCS) [40]. The main difference is that in π-calculus, channels can be transferred along channels – this is called *channel passing* – which is quite useful to describe dynamic reconfigurations in systems.

Let us consider the two simple definitions of π-calculus processes below:

$$Recp \stackrel{\text{def}}{=} c(x).\overline{d}\langle x\rangle.Recp \quad \text{and} \quad Proc \stackrel{\text{def}}{=} d(y).\tau.\overline{y}\langle a\rangle.Proc,$$

The definition *Recp* is for a *receptionist* of requests from external clients, and the *Proc* is a *processor* for such requests. In the receptionist, the action prefix $c(x)$ represents an input on channel c bound to a variable x, and $\overline{d}\langle x\rangle$ emits the value bound to x along channel d. The τ prefix allows as in CCS to describe that "something happened" in the process without precisely describing the nature of this internal event. The two definitions

are recursive which means that the receptionist and processor always accept/handle new requests from the clients.

The system consists in making these two processes run in parallel, which is simply denoted by:

$$Recp \mid Proc. \tag{2.1}$$

which through unfolding yields:

$$c(x).\overline{d}\langle x\rangle.Recp \mid d(y).\tau.\overline{y}\langle a\rangle.Proc$$

There are two basic operational semantic settings for the π-calculus: the *reduction semantics* and the *labelled transition semantics.*

In the reduction semantics, the transitions are unlabelled and of the form $P \rightarrow Q$ with P a process and Q its continuation after a single action. The possible actions in reduction semantics are either *silent steps* (explicit τ's) as in the following example:

$$\tau.P \rightarrow P$$

or through *synchronization*, which only occur when two processes communicate on the same channel, as in the following example:

$$\overline{c}\langle a\rangle.P \mid c(x).Q \rightarrow P \mid Q\{a/x\}$$

The latter example illustrates the evolution of processes after a communication between an output $\overline{c}\langle a\rangle$ and an input $c(x)$, where $Q\{a/x\}$ denotes the substitution of x by a in process Q. To illustrate channel passing, consider the following system:

$$(\nu a)\,[\overline{c}\langle a\rangle.P \mid c(x).\overline{x}\langle b\rangle.Q \mid a(y).R]$$

The name a is restricted to the three processes P, Q and R, which means the external environment cannot interact using the channel a. Note also that the process Q in the middle has no reference to a. The only possible reduction is a synchronization between processes P and Q, yielding :

$$\rightarrow (\nu a)\,[P \mid \overline{a}\langle b\rangle.Q \mid a(y).R]$$

Now the channel a has been passed from P to Q, which means now Q and R can communicate, as follows:

$$\rightarrow (\nu a)\,[P \mid Q \mid R\{b/y\}]$$

The system has been reconfigured so that the processes Q and R can interact.

This reduction semantics is only suitable for capturing the internal behavior of a system. Put in other terms, it is only suitable for *closed systems* modelling. For example, there is no possible reduction from the following processes:

$$(\nu a)\overline{c}\langle a\rangle.P \not\rightarrow \text{ and } d(x).Q \not\rightarrow \text{ and } \overline{c}\langle a\rangle.P \mid d(x).Q \not\rightarrow$$

From the point of view of reductions, the three processes above are identified with the *deadlock* process 0. This is also the case for the system specified in (2.1). We may say that from the point of view of analyzing process behaviours this is not satisfactory.

In labelled transition semantics, on the contrary, all the "deadlocked" processes above have an interpretation that distinguish them from 0. Compared to reduction semantics, each transition in the labelled case is of the form $P \xrightarrow{\mu} Q$, where μ is an action label (*i.e.*, input, output, bound output or silent). Our first example yields:

$$(\nu a)\overline{c}\langle a\rangle.P \xrightarrow{c\nu a} P$$

The action $c\nu a$ is called a *bound output* and represents some difficulty in analyzing π-calculus behaviors. The idea is that the channel a was private before being sent to the external environment. Clearly it cannot be considered private anymore since it is known externally. The second example is an input from the environment, which also represents some difficulty. A first interpretation is called the *early semantics* consists in replacing the name received from the environment by "all the possible names":

$$d(x).Q \xrightarrow{du} Q\{u/x\} \text{ for any name } u$$

In the *late semantics* the idea is to keep the variable x bound in the transition labels:

$$d(x).Q \xrightarrow{d(x)} Q$$

There are many implications for this choice, and we will see that it is a non-trivial issue when translating to Petri nets.

The first transition is :

$$\overline{d}\langle k\rangle.Recp \mid d(y).\tau.\overline{y}\langle a\rangle.Proc$$

in the case that process *Recp* receives the input k on channel c. Now, both processes can communicate on the same channel d, and the system may evolve to

$$Recp \mid \tau.\overline{k}\langle a\rangle.Proc$$

Finally in:

$$\overline{c}\langle a\rangle.P \mid d(x).Q$$

the processes are deadlocked in both the early and late case since they try to synchronize on a distinct channel. The *symbolic semantics* would trigger the following transition:

$$\overline{c}\langle a\rangle.P \mid d(x).Q \xrightarrow{c=d,\tau} P \mid Q\{a/x\}$$

which means the transitions is under the condition that the names d and c are equal after the synchronization between $\overline{c}\langle a\rangle$ and $d(x)$.

The early, late and symbolic semantics are all suitable to reason about *open systems* in that they all consider implicitly the most general external environment possible. The definition for *open reconfigurable systems* that we will thus follow in this thesis is roughly the class of systems that can be modelled by π-calculus variants in late symbolic semantics, i.e., the interpretation that is the "most open" of all.

2.2 Petri nets

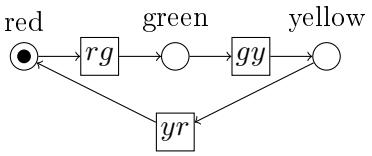
Petri nets [42, 53, 54, 49] are a mathematical and graphical model invented by Carl Adam Petri and introduced in his doctoral dissertation "Communication with automata" [50]. It is suitable for modelling and verification of distributed systems, in particular, thanks to several decidability results [24, 29]. Moreover, many support tools [4] are developed for Petri nets for the purpose of verification. In this section we present place/transition Petri nets and their extensions (with read arcs and inhibitor arcs [14]), which are used for the translation of π-calculus in [39, 38, 15], high-level Petri nets, which are used in our translation and the framework SNAKES+NECO [51, 27] which allows us to specify and verify high-level Petri nets.

2.2.1 Place/transition Petri nets

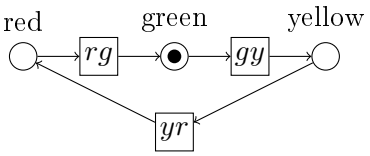
Place/transition Petri nets (or P/T Petri nets for short) are a widely used formalism for modelling and analyzing distributed systems and provided with many support tools. We consider first an example of modelling using P/T Petri nets, and then give their formal definition.

Consider the traffic light system modelled by a Petri net depicted in Figure 2.1. The system has three states which are represented by three circles with labels red, green and yellow, called *places*. Initially, suppose that the light is red, represented by a bullet, called a *token*, inside the place red. The action of changing states of the system is represented by rectangles, called *transitions*, with labels rg, gy and yr. The transition with label rg changes the system from red to green, gy changes the system from green to yellow and yr changes the system from yellow to red. These transitions are connected to places by arcs that represent the flow of changing states.

Transition rg is enabled if the system is in state red, *i.e.*, the token is present at place red. If the transition is enabled then the system may change to state green represented by "consuming" the token from place red and "produce" a token to place green. The firing of transition rg changes the system state from red to green, depicted in Figure 2.1b. Similarly, the transition gy is enabled if the system is in state green, yr is enabled if the system is in state yellow.



(a) Initial state (red)



(b) Next state (green)

Figure 2.1: A Petri net model of the traffic light system

More generally, a place may contain several tokens and a transition may consume and produce several tokens. A P/T net is composed of a static and a dynamic part. The static part is defined by a bipartite directed graph, called a *net structure*, while the dynamic part is given by a mapping from the set of places to natural numbers, called a *marking*.

Definition 2.1. *A* P/T net *is a 5-tuple,* $PN = (P, T, F, W, M_0)$ *where:*

- P *and* T *are finite sets of places and transitions, such that* $P \cap T = \emptyset$*;*
- $F \subseteq (P \times T) \cup (T \times P)$ *is a set of arcs (flow relation);*
- $W : F \to \mathbb{N}^+$ *is a weight function;*
- $M_0 : P \to \mathbb{N}$ *is the initial marking which associates with each place the number of tokens that it carries.*

The system behaviour can be described in terms of markings (system states) and their changes. In order to simulate the dynamic behaviour of a system, a marking M of a Petri net evolves according to the following transition (firing) rule:

1. A transition t is said to be *enabled* if each input place p is marked with at least $W(p,t)$ tokens, where $W(p,t)$ is the weight of the arc from p to t, *i.e.*, $W(p,t) \leq M(p)$.
2. If transition t is enabled, then it may *fire* (depending on whether or not the event actually takes place).
3. A firing of an enabled transition t at a marking M removes $W(p,t)$ tokens from each input place p and adds $W(t,p)$ tokens to each output place p, where $W(t,p)$ is the weight of the arc from t to p, *i.e.*, for each place p the new marking M' is $M'(p) = M(p) - W(p,t) + W(t,p)$.

A firing of t at a marking M is denoted by $M[t\rangle M'$.

These notations may also be extended to *occurrence sequences* defined as follows:

Definition 2.2 (Interleaving semantics of Petri nets)**.** *Let $PN = (P, T, F, W, M_0)$ be a P/T net, $t_1, t_2, \cdots \in T$ be transitions of PN, and $M_1, M_2, \ldots$ be markings.*

- $\sigma = M_0 t_1 M_1 \ldots t_n M_n$ *is a* finite occurrence sequence *of PN if and only if* $\forall i, 1 \leq i \leq n : M_{i-1}[t_i\rangle M_i$*;*
- $\sigma = M_0 t_1 M_1 \ldots$ *is an* infinite occurrence sequence *of PN if and only if* $\forall i, 1 \leq i : M_{i-1}[t_i\rangle M_i$*.*

A marking M is said to be *reachable* in a P/T net $PN = (P, T, F, W, M_0)$ if there is a finite occurrence sequence $M_0 t_1 M_1 \ldots t_n M_n$ such that $M_n = M$.

(a) Inhibitor (b) Read arc

Figure 2.2: P/T nets with inhibitor and read arcs

P/T nets can be extended by inhibitors or read-arcs which are illustrated in Figure 2.2. An inhibitor permits to test the absence of tokens in an input place of a transition. In Figure 2.2a, transition t_2 is enabled if place p_1 has no token (which is the case in the given marking). On the contrary, read arcs permit to check for presence of tokens in input places of a transition. In Figure 2.2b, transition t_2 is enabled if place p_1 has at least one token (which is not the case in the given marking). The firing of a transition in both variants may produce a new marking like in P/T nets.

Definition 2.3. *A P/T Petri net is said to be* k-safe *for some $k > 1$ if and only if in any reachable marking M, the number of tokens in each place p is $M(p) \leq k$. It is called just* safe *if it is 1-safe.*

2.2.2 High-level Petri nets

In P/T Petri nets tokens are not distinguishable, they are considered as "black tokens". In our translation, we use high-level Petri nets in which tokens are distinguishable and structured, arcs are annotated with multisets of expressions, and transitions have firing conditions, called the *guards*. High-level Petri nets used in this thesis are defined as follows.

Definition 2.4. *A* high-level Petri net *is a tuple* $N = (\mathsf{P}, \mathsf{T}, \mathsf{U}, \mathsf{G}; \mathsf{M})$*, where:*

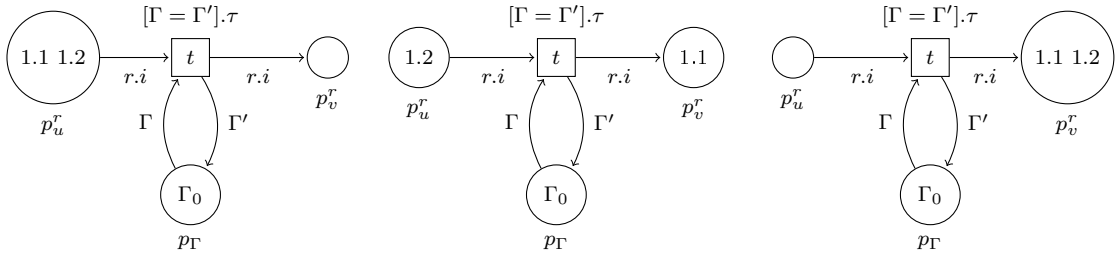
- P *and* T *are sets of places and transitions, such that* $\mathsf{P} \cap \mathsf{T} = \emptyset$*;*
- $(\mathsf{P} \times \mathsf{T}) \cup (\mathsf{T} \times \mathsf{P})$ *is the set of arcs;*
- U *is a labelling mapping for each element of* $\mathsf{P} \cup \mathsf{T} \cup (\mathsf{P} \times \mathsf{T}) \cup (\mathsf{T} \times \mathsf{P})$*, such that*
  - *for each place* $p \in \mathsf{P}$*,* $\mathsf{U}(p)$ *gives its type (the set of admissible tokens);*
  - *for each transition* $t \in \mathsf{T}$*,* $\mathsf{U}(t)$ *is its (possibly empty) label;*
  - *for each arc in* $(\mathsf{P} \times \mathsf{T})$*,* $\mathsf{U}((p,t))$ *is a (possibly structured or empty) expression with variables compatible with* $\mathsf{U}(p)$*, and analogously for arcs in* $(\mathsf{T} \times \mathsf{P})$*;*
- G *is a mapping associating a guard (decidable Boolean formula) with each* $t \in \mathsf{T}$*;*
- M *is a marking associating with each place* $p \in \mathsf{P}$ *a set of (possibly structured) tokens in* $\mathsf{U}(p)$*.*

As usual in high-level Petri nets [32, 31, 34], a transition t in T is *enabled* at a marking M if the following two conditions are true:

1. There exists a *binding* ρ for the variables in its guard and in the arc inscriptions adjacent to t such that ρ is compatible with the type of each place p adjacent to t, and makes the guard $\mathsf{G}(t)$ true.
2. For each input place p of t, there are enough tokens to satisfy the flow, *i.e.*, $\mathsf{M}(p) \supseteq \rho(\mathsf{U}(p,t))$.

The *occurrence* of t under ρ has the label $\rho(\mathsf{U}(t))$ and produces a new marking M' by consuming the tokens in all input places of t and producing the new ones according to the arc annotations and conditions expressed in the guard on the output places. More precisely, for each $p \in \mathsf{P}$, the firing of t removes tokens in $\rho(\mathsf{U}(p,t))$ from p and put tokens in $\rho(\mathsf{U}(t,p))$ on p. Such an occurrence of t is denoted $\mathsf{M}[t{:}\rho\rangle\mathsf{M}'$, or $\mathsf{M}[t{:}\rho(\mathsf{U}(t))\rangle\mathsf{M}'$ if we are only interested in the effect of the occurrence.

Let us consider an example of transition rule for high-level Petri nets. Suppose that we have a high-level Petri net with its initial marking M_0 as in Figure 2.3a. The net consists of places p_1^1, p_2^1 and p_Γ, and a unique transition t. The type of both places p_u^r and p_v^r is a Cartesian product of sets of identifiers, which means the tokens in those places are tuples of the form (j,k), denoted $j.k$. We deliberately use structured tokens, for example, the token "1.2" may be understood as a thread 2 in replicator 1. So, the arc labels connecting p_1^1 and p_1^2 to t are also of the form $r.i$, where r and i are variables. Place p_Γ contains only one structured token Γ_0 (which may be the whole name context of π-graphs we will see in the next chapter) and the adjacent arcs are labelled with variables Γ and Γ'. Transition t represents the action τ and it is enabled at marking M_0 if there exists a binding for variables r, i, Γ and Γ' such that its guard $[\Gamma = \Gamma']$ is true.



(a) A high-level Petri net with its initial marking M_0 (b) Marking M_1 after firing t with binding ρ_1 from M_0 (c) Marking M_2 after firing t with binding ρ_2 from M_1

Figure 2.3: High-level Petri nets and transition rule

The high-level Petri net and its initial marking is given in Figure 2.3a. There are two possibilities for firing transition t corresponding to two bindings $\rho_1 = [r = 1, i = 1, \Gamma = \Gamma_0, \Gamma' = \Gamma_0]$ and $\rho_2 = [r = 1, i = 2, \Gamma = \Gamma_0, \Gamma' = \Gamma_0]$. Suppose that we choose to fire the transition with binding ρ_1, then the marking M_0 evolves to M_1, as shown in Figure 2.3b.

After firing the transition with binding ρ_1, there is only one possibility for the next firing, which is with binding ρ_2 (actually, recomputed at marking M_1). Similarly, if t fires with ρ_2, then the marking M_1 evolves to M_2 as shown in Figure 2.3c.

The notions of reachability and occurrence sequence naturally extend to high-level nets. Similarly, a high-level Petri net is said to be k-safe ($k \in \mathbb{N}^+$ is a natural number not zero) if and only if for any reachable marking M, the number of tokens in each place p is less or equal than k. For example, it can be checked that the net in Figure 2.3 is 2-safe.

2.2.3 Petri nets vs. process algebras

Petri nets and process algebras are two widely known and well studied formalisms dealing with concurrency. However, their advantages are rather different and sometimes complementary.

The advantages of Petri nets include the following:

- they make a natural distinction between states and activities, this corresponds to the distinction between net places (local states) and transitions (local activities);
- they are well suited to express distributed systems as their global states and global activities are derived from the local ones;
- they are easy to understand and manipulate thanks to their graphical representations;
- they are also formally defined making them usable for verification purposes.

On the other hand, the advantages of process algebras are the following:

- they are directly related to programming languages;
- they are compositional, meaning that they make it possible to construct in a structural way larger systems from smaller ones;
- they facilitate reasoning about important properties of systems using related logics;
- they are often provided with a variety of algebraic laws making it possible to manipulate systems or to prove them correct with respect to some specifications.

The idea of combining the advantages of both formalisms in a unified way is not new and several works have been published in this field [35, 43, 30, 57]. Petri Box Calculus (PBC) [10] (and Petri Net Algebra (PNA) [11], which is its generalisation) appears in this context as perhaps one of the most original. Like a process algebra, it is composed of a syntax in terms of *box expressions* and has a semantics in terms of safe low-level Petri nets. The main operators of PBC are on one hand those allowing to model control flow, *i.e.*, the sequential composition, the parallel composition, the nondeterministic choice, the recursion or an asynchronous link, and of the other hand those dedicated to model communications, *i.e.*, a synchronization and a restriction, which have the particularity to be separated from the parallel composition. These operators have their counterparts in the semantic domain of *boxes*, *i.e.*, safe low-level nets provided with composition interfaces on places and transitions. Actually, a composition operation on boxes is defined for each operator of the syntax.

PBC and PNA have been used to give the semantics of CCS process algebra [40, 6] and many dedicated specification languages. High-level variants of PBC, the family of M-nets [34, 12] have been used to give the semantics of complex programming constructs, such as procedures, exceptions or threads in the context of distributed systems. They have also been used in [19, 20] to give one of the first syntactic non-reduction semantics to mobile systems, such as a variant of π-calculus. However, the net class used for these approaches are different, and often significantly more complex, than the class of high-level nets used in this thesis.

2.3 Translation of π-calculus variants into Petri nets

In the literature, there are several works related to the translation of π-calculus variants into Petri nets. We survey the most significant of such works and classify them according to the following criteria:

1. Modelling approach: Open systems vs. closed systems.

2. Nature of the translation: Syntactic vs. semantic.

3. Expressivity: supported π-calculus features.

4. Tool support: no support, ad-hoc tools or reuse of existing tools.

2.3.1 Open systems vs. closed systems

The existing translations of π-calculus into Petri nets are almost equally divided between the ones that support the open systems approach of modelling and the ones that only support the closed systems approach.

2.3.1.1 Translation only for closed systems

There are two main works concerning the translation of variants of the π-calculus with reduction semantics into Petri nets:

In paper [38], the author presents the semantic translation of a π-calculus variant into P/T nets. The translation is proved isomorphic and the π-calculus states can be retrieved from the Petri net up to an adapted notion of structural congruence. This translation may under some conditions – either syntactic and decidable (infinite handler process) or semantic and undecidable (structural stationarity) – provide finite representations for dynamically reconfigurable systems with potentially an unbounded number of components and connections. This way behavioural properties of some infinite-state systems can be verified automatically. To that end, a modelling and verification tool PETRUCHIO [5] has been implemented. For the verification part, the model checker LoLA [2, 56] can be reused thanks to the target language of the P/T nets.

In paper [39], the author presents a translation of finite control processes (FCP) to safe low-level Petri nets with read arcs. The translation can be performed in three main steps:

1. model the substitution σ that maps the bound names and formal parameters occurring in the FCP and active at the current π-calculus state to their value, as a set of Petri nets places (called the substitution net).

2. translate the control of each thread by adding transitions into the substitution net. Each subterm t of the thread, which may be a stop process 0, a call $K\lfloor\tilde{a}\rfloor$, a restriction $\nu r.S$ or a sum, is translated into a subnet with a unique entry place.

3. synchronize the subnets corresponding to different threads on communication actions. If t and t' are two transitions labelled by $\overline{a}\langle b\rangle$ and $x(y)$, respectively, between two different threads, a set of transitions implementing the communication is added.

Based on this translation, an ad-hoc verification tool was developed.

2.3.1.2 Translation for open systems

There are three main works related to open systems:

In [15], the authors present the translation of early labelled semantics of a π-calculus variant into place/transition Petri nets with inhibitor arcs. They consider a special feature of conflict names to ensure the mutual exclusion of actions. Each place of the translated net corresponds to either a sequential process with its conflict names, or a set of conflict names shared by parallel processes or finally a set of restricted names. Transitions are constructed using axioms which describe the behaviours of silent action, input, output, bound output and synchronization. The authors present also some decidability results (such as the satisfaction of linear time μ-calculus formulae or reachability) for a subset of the π-calculus generating finite nets.

In [21], the authors propose a structural translation of possibly recursive π-calculus terms for late transition semantics into high-level Petri nets. First, the term is translated into context-based representation in which restrictions are removed and the context is the interpretation of names. Then, the context-based representation is used for constructing the Petri net. Moreover, authors informally explain how to extend the translation for match in relatively straightforward way (but not for the mismatch). Notice that even for simple finite π-calculus systems the state-space of the translation is infinite.

Paper [48] presents a syntax-driven translation of the π-graphs with iterators based on late semantics into finite, one-safe coloured Petri nets. The translation is performed in 2 steps: translating the control flow (which gives the net structure) and translating the context (which gives the corresponding initial marking). First, the translation of control flow is synthesized from monadic nets, polyadic nets and iterator nets using operators of relabelling, disjoint union and merge.

- A *monadic net* is used to encode directly the constructs input/output and match/mismatch (separate match/mismatch from input/output), each of them is translated into a unique place with corresponding type. The place is connected to a set of context transitions corresponding to semantics rules that are potentially enabled when these constructs are in redex position, and is connected to a single predecessor transition and a successor transition.

- A *polyadic net* is used to encode directly the constructs such as sum, parallel and silent. Each of them is translated into a place which is connected to a single corresponding context transition. The place has potentially multiple predecessors.

- An *iterator net* is used to encode directly iterator constructs.

Next, the context translation is performed as follows:

- The *global context* Γ is translated into a unique context place with a single token Γ, it is connected to all context transitions.
- A unique *observation place* Ω is connected to all context transitions, it contains information about the current observation.
- Finally, a reset transition is connected to the observation place allowing to discharge the previous observation, and the guards are defined for all context transitions.

2.3.1.3 Discussion

It is clear that the translation of closed systems is simpler than the one of open systems. For example in [48], in the same framework both approaches are compared and quoting the authors:

> "... considering the reduction semantics only, our translation can be greatly simplified ..."

Moreover, we can also observe that the only verification tools based on Petri net translations are for closed systems only. However, the translation of open systems is more faithful to the traditional π-calculus semantics. It is even required when reduction semantics is not enough. For example, with reduction semantics we cannot observe an input action. Let us reproduce once more the quote of Robin Milner[9]:

> *You can't do behavioural analysis with the chemical semantics [...] I think the strength of labels is that you get the chance of congruential behaviours.*

We can also observe that late transitions require higher-level Petri nets if compared to early transitions. This can be explained because the bound names appearing in the transition labels involve a notion of data structure and related computations in the translated nets. This is especially obvious in paper [48].

2.3.2 Syntactic vs. semantic translation

2.3.2.1 Syntactic translation

In a syntactic translation, the structure of the translated Petri net corresponds directly to the syntactic structure of the initial π-calculus term.

In [21] the structural translation is for late semantics and results in high-level Petri nets with a finite net structure but possibly having infinite state-space even for some finite control processes.

In [39] the translation is for reduction semantics and results in low-level Petri nets with read arcs which has finite net structure and finite state-space. The language supports only finite control processes.

In [48], the translation is for late semantics and results in high-level Petri nets with a finite structure and finite state-space. The language supports finite control processes using iterators. In all three approaches, the size of the translated net is polynomial.

2.3.2.2 Semantic translation

The semantic translation of a π-calculus term is performed by building the abstract state-space of the term, then translating this state-space into Petri nets.

In [38], the translation is for reduction semantics and results in P/T nets. The obtained net is finite for finite behaviours and potentially infinite for some infinite control processes. However, infinite handler processes and the more general class of structural stationary processes – which has infinite state-space– are also translated to finite P/T nets.

In [15], the translation is for early labelled causal semantics and results in low-level Petri nets with inhibitor arcs. The obtained net is potentially infinite.

In both cases, the size of the translated net, when finite, is potentially exponential.

2.3.2.3 Discussion

Semantic translations require simpler classes of Petri nets and are thus more easily supported by verification tools. However, the structural translations are conceptually simpler because they are closer to the input language. Moreover, it is easier to be modular. If the source language is modular then we can apply compositional translation of π-calculus term, *e.g.*, in the case of [21], the authors introduce a compositional translation where the construction of a resulting net is driven by the syntactic structure of a π-calculus term, with net composition operators corresponding to the process algebra operators. Finally, the translated nets are of a polynomial size while it is potentially exponential in the case of semantics translations. It seems likely that for the quite high-level late semantics the only reasonable approach is a structural translation. The finite translation of infinite control processes seems on the other hand restricted to semantic approaches.

2.3.3 The π-calculus variants

Almost all the π-calculus variants considered in the translations have common features, such as a prefix $\alpha.P$, where α can be an input $c(x)$, an output $\overline{c}\langle a\rangle$ or a silent action τ, parallel composition $P \mid Q$, a non-deterministic choice $P + Q$. The variation mostly concerns the expression of repetitive behaviours and the support for name comparison through match and mismatch.

2.3.3.1 Repetitive behaviour

General recursion In languages that support general recursion we can define processes with parameters, like $D(x_1, \ldots, x_n) \stackrel{\text{def}}{=} P$, where x_i are formal parameters and P is a π-calculus process possibly containing calls of the form $\alpha.D(v_1, \ldots, v_n)$. In particular, one may define processes having a parallel composition inside the recursion, like

$$E \stackrel{\text{def}}{=} \overline{a}\langle b\rangle.E \mid a(x).E.$$

Languages with general recursion are the most general languages in the sense of expressivity, however, most non-trivial properties are undecidable for them, for example state-space computation, reachability. The languages translated in [15, 38, 21] support general recursion.

Limited recursion Languages with limited recursion do not support parallel operators within recursion (it is equivalent to finite control processes). For example, they accept the process definition like $D \stackrel{\text{def}}{=} \overline{a}\langle b\rangle.D$, but not $E \stackrel{\text{def}}{=} \overline{a}\langle b\rangle.E \mid a(x).D$. The language in [39] is such a case.

Iterators Repetitive behaviours can be also specified using iterators, like in [48]. For example, $I : (\nu k) * a(c).\overline{c}\langle k\rangle.0$ is the definition of an iterator with label I. It is structurally equivalent to finite control processes.

2.3.3.2 Name comparison

Usually, in π-calculus languages, name comparison can be a match or a mismatch. A match between two names a and b, denoted by $[a = b]$, is a comparison between these two names, which is evaluated to true if they are equal. In contrast, a mismatch between a and b, denoted by $[a \neq b]$, is evaluated to true if they are distinct.

Match and mismatch have non-trivial labelled semantics. Match is essential because synchronization requires an implicit match (at least in late transition semantics). It is sometimes argued (*e.g.*, in [44] or [55]) that mismatch is less important. But works

about testing equivalence for π-calculus [13] argue the contrary. In most cases, mismatch semantics is complex. The languages in [38, 15] do not provide match and mismatch, while in [21], the authors informally explain how to extend the translation for match in a relatively straightforward way (but not for mismatch), and in [39] the the authors discuss only informally the integration of match and mismatch. The main argument is that mismatch cannot be added in a simple way because to test if $[x \neq y]$ we need to test $x = a$ and $y \neq a$, or $[x \neq a]$ and $[y = a]$, where $[x \neq a]$ complements $[x = a]$. In recent paper [48] both match and mismatch are supported thanks to the introduction of an explicit equivalence of names and distinctions.

2.3.3.3 Discussion

Match and mismatch are complex and formally taken into account only in [48] despite their importance, especially for match. Although mismatch is less fundamental outside late transition semantics (not in the cases of [39, 38, 15]). The challenge is how to allow full expressivity similar to general recursion but with simple criteria for decidability. For example, in [38], the criterion of structural stationarity is complex and only semi-decidable.

2.3.4 Verification and tool support

There are many existing support tools for Petri nets, which can be classified along the classes of Petri nets that they support: from low-level P/T nets to higher-level nets.

In the related work, the translation of [38] uses LoLA [2, 56], a model checker for P/T nets. The properties are specified using computational tree logic (CTL). Moreover, this tool uses reduction techniques and supports simple properties such as deadlocks or reachability checking. The tool MPSat [3, 33] used in [39] is an ad-hoc tool.

For high-level nets, there is no existing approach with tool support.

Discussion

In general, the motivation of translating π-calculus into Petri nets is to reuse existing verification tools for Petri nets. For example, [38] translates π-calculus into P/T net so it can reuse tools. However, in [39], the translated net is beyond P/T net, the author must develop an ad-hoc tool. Other translations face both theoretical and practical problems. The work [15] translates π-calculus into infinite net structure, [21] translates into infinite state space, thus they cannot reuse existing tools (theoretical problem). Moreover, the class of translated nets does not match an existing tool (practical problem).

Paper	Open/ Closed	Syntactic/ Semantic	Petri nets class	Expr.	Match	Mismatch	Tool existing
[39]	Closed	Syntactic	Low-level + read-arcs	FCP	Informal	Informal	Ad-hoc
[48]	Open	Syntactic	High-level	FCP	Yes	Yes	No
[38]	Closed	Semantic	P/T	Full	No	No	Existing
[15]	Open	Semantic	Low-level + inhibitors	Full	No	No	No
[21]	Open	Syntactic	High-level	Full	Informal	No	No

Table 2.1: Summary of translation from π-calculus into Petri nets

2.4 Synthesis

The main objective of the thesis is to automate the verification of reconfigurable systems. Our approach is to translate the π-calculus, in which the systems are modelled, into Petri nets and then use existing tools for Petri nets to analyze the translated net. The summary of our investigation about translations of π-calculus variants into Petri nets is provided in Table 2.1.

In the case of closed systems, the semantic translation of [38] can target place/transition Petri nets, and there are existing efficient tool in this cases, LoLA [2, 56] in this case is such an example. The problem of semantic translations is that the obtained Petri nets are potentially (and in all but the simpler cases) of an exponential size. It seems that for this reason only syntactic translations are more appropriate. The translation of [39] supports finite control processes and requires only slightly higher-level (but still low-level) Petri nets. Thus, an ad-hoc tool was developed in this case. In summary, we can say that for closed systems, the problem we address in this thesis is already solved.

However, the situation is less ideal in the case of open systems. As the matter of fact, there is no existing approach with any form of tool support, without talking about the reuse of an existing tool. This is somewhat unsatisfying since efficient verification tools exist even for (not too) high-level Petri nets. The challenge is to find a class of high-level Petri nets that is expressive enough for the translation and has a direct support in an existing tool. One problem we face is the very large gap between the very abstract π-calculus formalisms on the one side, and the much more concrete Petri nets on the other side. Instead aiming for a direct translation, that we think is very hard, we rely on an intermediate model – the π-graphs [46, 45, 47] – that proves very useful to reduce this abstraction gap. The π-graphs variant we consider in this thesis provides all the features of the π-calculus (plus some more such as an explicit representation of threads). Unlike the variant that is translated to Petri nets in [47], it supports a more general form of repetitive behaviors through the notion of replicators. Meanwhile, the π-graphs are much

closer – both in spirit and technically – to the Petri nets than the more classical π-calculi.

Chapter 3

Modelling open reconfigurable systems using π-graphs

This chapter introduces the π-graphs formalism from a modelling perspective. The discussion remains mostly informal. First, we describe a π-graphs model for a small but non-trivial example of an open client/server system. Based on this example, we can illustrate most of the modelling features of the proposed formalism. Then we also provide some example of interactions with the prototype tool we developed to support the modelling and verification activities using π-graphs.

3.1 An example of open reconfigurable system

We mostly follow the same notion of open reconfigurable system as the one of the π-calculus and related formalisms, especially [37]. However, there are several specificities in the π-graphs that we intend to illustrate based on a simple although non-trivial example. This is a model of a simple client/server system that exhibits the main properties of *reconfigurable systems*:

- they involve multiple concurrent activities,
- these activities can communicate with each others, and
- most importantly, they have dynamic connections.

The fundamental characteristic of such systems is thus that their structure evolves dynamically. In the example, this is illustrated by the connections between the clients and the server that are created and destroyed dynamically.

Moreover, the example is a model of an *open system* in that the number and exact behavior of the clients are abstracted away, only the interface between the server part

and the clients is modelled. Put in other terms, the clients are part of the environment and we focus on the server part only.

Last but not least, the example also illustrates one of the most important specificity in compared to most of other π-calculus variants: the visual nature of the formalism.

3.1.1 Overview of the system

The server part of the system comprises three components: A receptionist, a dispatcher and a processor, which are represented by three circles with labels *Recp*, *Disp*, and *Proc*, respectively. The relationships between components are illustrated in Figure 3.1. From now on, we call this system a *RDP system* for short.

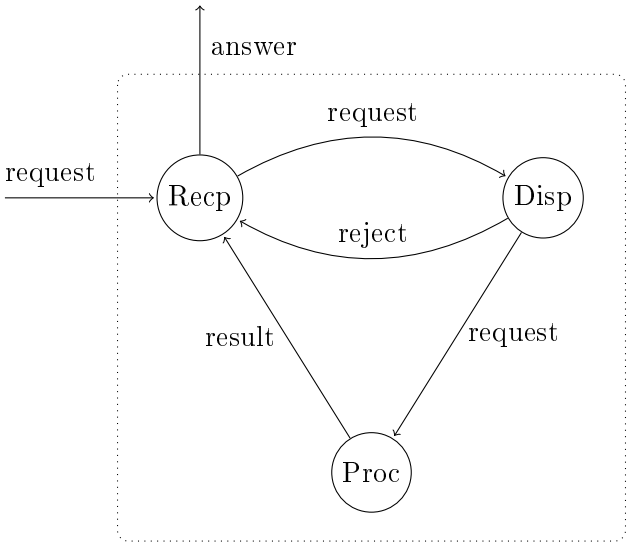


Figure 3.1: An overview of a RDP system

The components are connected by labelled directed arcs representing the communications between them. The label of an arc represents the information that is sent out or received from a component. The direction of an arcs determines which component is the sender and which one is the receiver. The receptionist can also communicate with the clients outside the system boundary which is represented by a dotted rectangle.

3.1.2 Functionality of components

In general, each component can perform two kinds of actions: *sending* (resp. *receiving*) information to (resp. from) the clients or the other components. The functionality of each component is described as follows:

- The receptionist receives requests from clients, transfers them to the dispatcher,

gets feedback (*i.e.*, rejects in case the request is not correct) from the dispatcher or results form the processor, and answers the clients.

- The dispatcher receives requests from the receptionist, determines the kind of requests and dispatches them to the corresponding sub-component in the processor, or sends a reject back to the receptionist.

- The processor receives requests from the dispatcher, processes them and sends the result to the receptionist.

Moreover, we assume that each component can perform several actions, and even *concurrently* which means that each component, e.g the receptionist, may be thought as a department in an office which has responsibility for receiving and answering requests from clients. Suppose that the department has two officers. At a moment, there may be a situation in which one person is waiting for requests and another one is answering clients. These two persons can work concurrently.

3.1.3 Describing the communication using ports

We present the communication between components, and between the receptionist and clients in more details in Figure 3.2. The components of the system communicate with the others via *ports*. We denote by $\overline{a}$ a sending port and by a itself a receiving port. If one component sends information via port $\overline{a}$ and the other one is waiting for receiving information via port a, then the two components can communicate. Ports can be classified into two main groups:

- Public ports: used for communicating with both clients and components, *i.e.*, both outside and inside the system. For example, port a in the figure is a public one. The communication with clients must be performed via public ports.

- Restricted ports: used for communication between components inside the system in order to deny clients sending information to the system via these ports. However, a restricted port will become public as it is sent outside.

In Figure 3.2, ports are illustrated by small coloured circles with labels, the arc labelled r from outside to the system is the request of a client. Port a is public used for communication with clients, and ports b, c, d, e, f, g are restricted ones used for communication only between components. Moreover, the port k is a restricted one of the receptionist used for communication with a specific client after it is sent out to him. Port r is public, that is created after request r is received from a client. The functionality of components are given in more details as follows.

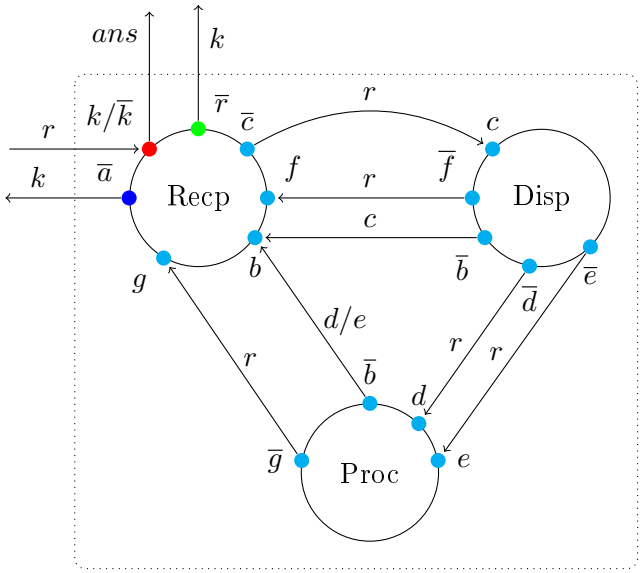


Figure 3.2: Modelling the communication of the RDP system

The receptionist: To receive requests from clients (from outside the system) via port k, the receptionist first let the clients know this port by sending k to the clients via the public port a. At this moment, k becomes a public port, thus clients use it to send requests to the receptionist. The receptionist then waits for requests from the clients via port k. When a request r is received, it is forwarded to the dispatcher via port c.

Similarly, to send an answer to a client whose request is r, the receptionist sends out its restricted port k to the client first via port r, and then it sends out the answer via port k. The client receives the answer via k. After each session of communication with clients, port k is reset to ensure that it is fresh for potential next clients.

The dispatcher: First, the dispatcher receives a request from the receptionist via port c. Then, it checks if the request is correct. If this is the case, then it dispatches the request to the processor. Otherwise, the request is refused, it is sent back to the receptionist via port f. Before sending back a reject, the dispatcher informs the receptionist about this by sending a signal c via the port b. Moreover, depending on the kinds of request r, it may be dispatched to the processor via either port d or e.

The processor: First, the processor receives requests from the dispatcher via ports d and e. Then, it processes the requests and sends the results to the receptionist via port g. Similarly to the dispatcher, it sends a signal d or e to the receptionist before sending the result of the request to inform the receptionist about the kinds of the corresponding one.

Summing up, the considered system has the following properties (which are those of open reconfigurable system):

- Components of the system can communicate together, and communicate with the

environment outside the system,

- The actions of the system can perform concurrently,
- The connections can be created and destroyed dynamically,
- Finally, the environment is open, in the sense that the system does not know in advance what it will receive.

3.2 Modelling the system using π-graphs

Based on the description of the system as so far, we now model it using π-graphs. Moreover, we also give an illustration for the evolution of the system.

3.2.1 Modelling the system

Suppose that each component can perform actions concurrently in two threads (can be thought of as there are two persons in each corresponding department as explained above); we call this number the capacity of the component. A π-graph model of the system is given in Figure 3.3.

Each component is modelled by a *replicator*. A replicator is represented by an identifier together with its capacity, and all actions that it performs. In this case, the receptionist is represented by a replicator R with capacity of two threads, denoted by $R[2]$, the dispatcher is represented by replicator D with capacity two, and the processor is represented by replicator P with capacity two. The actions of a replicator are organized in a hierarchical structure in the form of tree, in which each node represents an action. Each action may have some successors which are connected to their parent node by a directed dashed arc representing the precedence. If a node has multiple children, then after the action at parent node, there are multiple possibilities for the next action. The choice is non-deterministic, and it depends on the condition (called the guard) of the actions. A guard can be a *match*, such as $[x = c]$, which is true if two names x and c are compatible, or a *mismatch*, such as $[x \neq c]$, which is true if the two names are not compatible.

Besides two kinds of actions that a component can perform, *i.e.*, sending and receiving information via a given port, we use silent action, to model internal actions which are abstracted from the observation. From now on, we use channel names (or just names) to indicate ports and information that is sent or received via ports. In general, actions of a replicator can be classified into three groups:

- Output $\overline{c}\langle a\rangle$: a name a is sent out to the environment (to clients) via a channel c,

Restricted: b, c, d, e, f, g

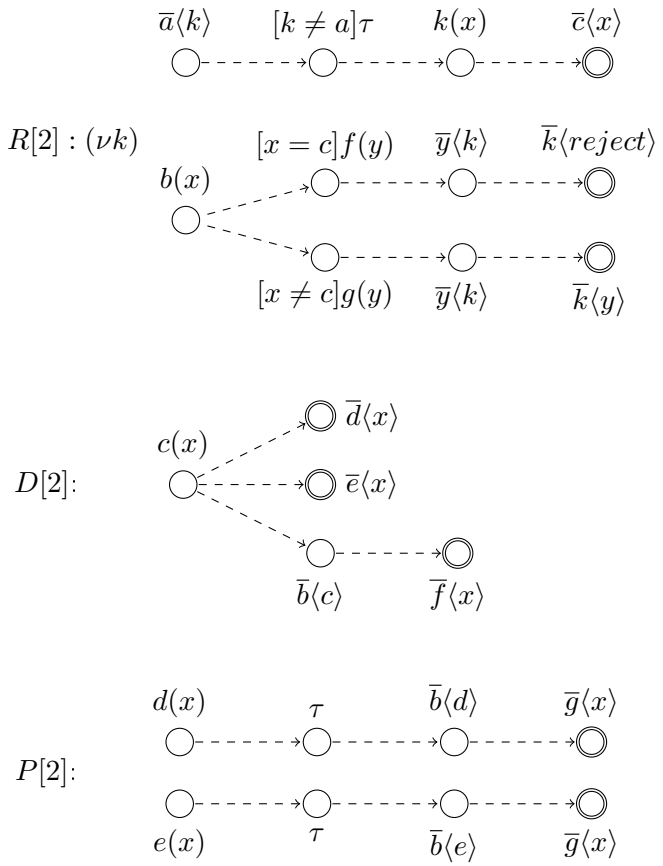


Figure 3.3: The π-graphs model of the RDP system

- Input $c(x)$: receives information from clients via channel c and puts it in place of x,
- Silent action τ: the internal action.

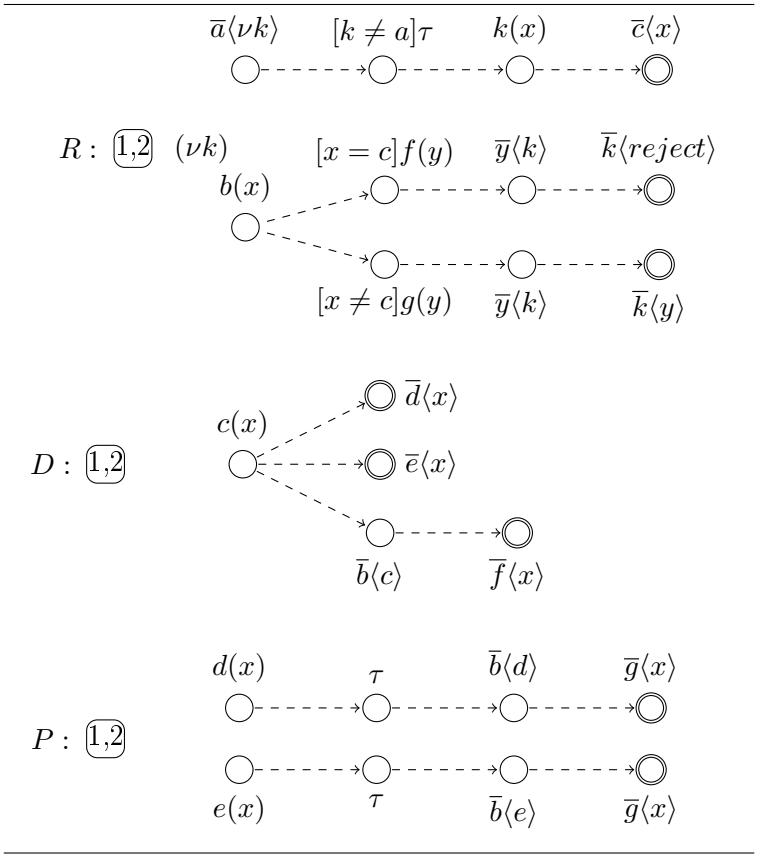
Moreover, the scope of names used in the system must be specified. Names b, c, d, e, f, g which are restricted channels, are indicated by a keyword **Restricted**. These restricted channels disallow external interferences. The name k is a restricted channel used for the receptionist replicator only, it is indicated by νk, meaning local to replicator R. Each replicator has a set of variable names used for input actions, like x. Other names are public.

3.2.2 A scenario of the π-graphs evolution

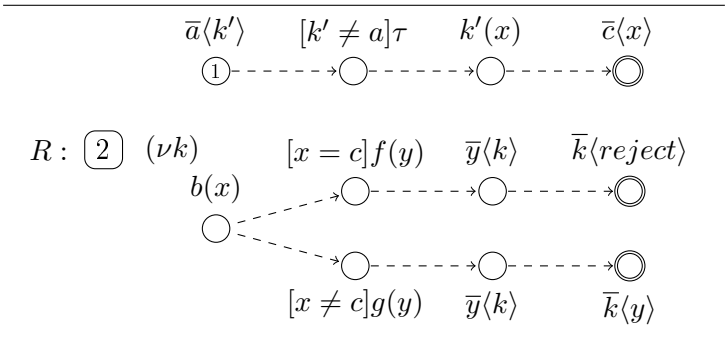
We consider the evolution of the π-graph in a scenario step by step.

Initially, each replicator has two threads that are available for performing actions. The receptionist is ready for sending its restricted channel k to clients via the public

channel a. The dispatcher is ready for receiving requests from the receptionist, and the processor is ready for receiving requests from the dispatcher. The only one action can be performed is sending k to a client.



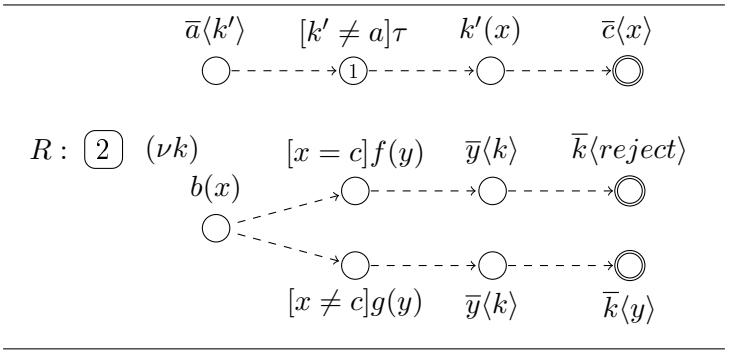
Step 1: The receptionist sends channel k to clients using thread 1. This illustrates the feature of channel passing that is the main way of describing dynamic connections in the π-calculus and its variants.



In terms of control, this is indicated by moving thread 1 at action $\overline{a}\langle k \rangle$ in replicator R. This means that from now on thread 1 is running. Remark that thread 2 is still available in replicator R. So, the Private channel k is sent to the client through the public channel a. In the π-graph, we say that channel k *escapes* the system. On the example, we use identifier k' to represent the new shared identity of the escaped channel (that cannot

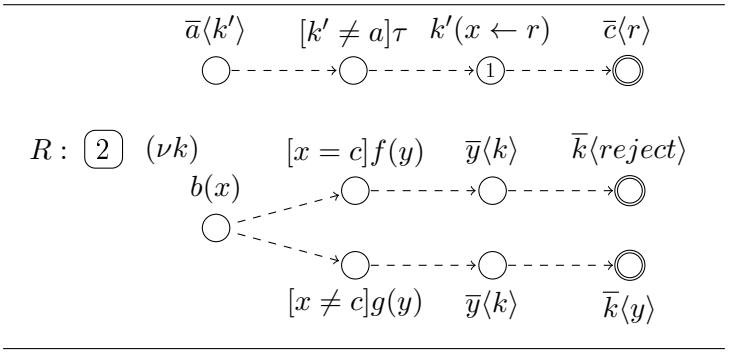
be considered private anymore). We call such dynamically created name a *fresh output name*. We will see in the next chapter the precise nature of fresh output names.

Step 2: The replicator R continues performing the next action which is a silent action with a mismatch condition $[k' \neq a]$.



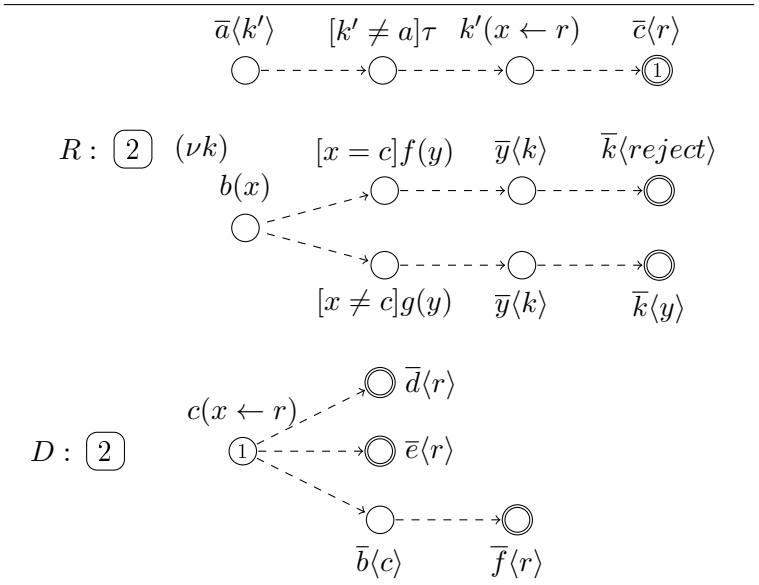
The objective of the match is to make sure that the two channels k' and a different. This is for example to avoid the communication between the next input $k'(x)$ and the output $\overline{a}\langle k \rangle$. In terms of control, the thread 1 is moved to action τ.

Step 3: Replicator R waits for requests from clients via the new channel k' by performing an input $k'(x)$.



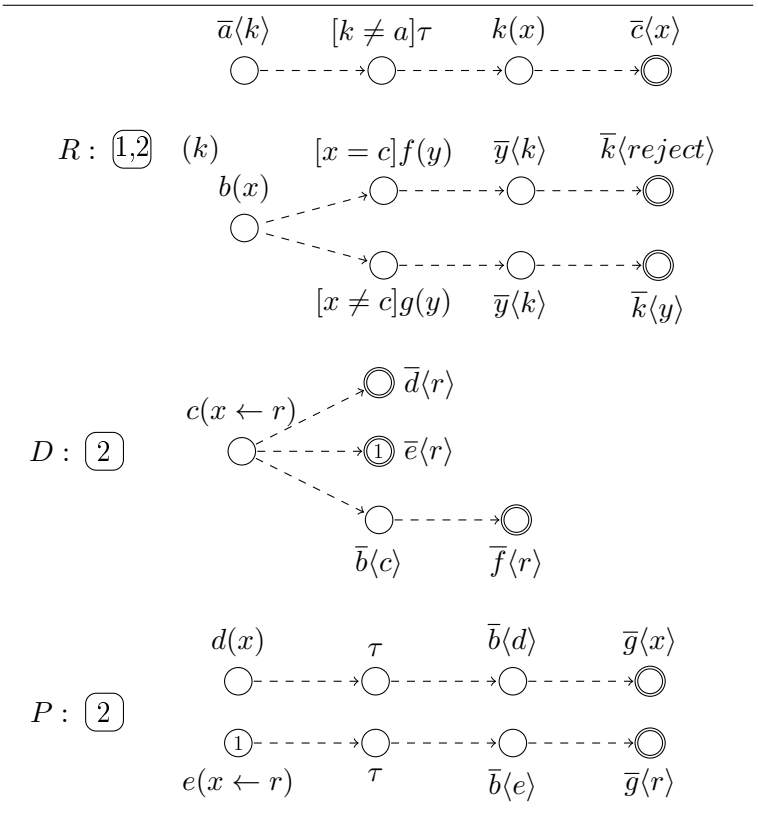
Suppose that a client sends a request r to R, r will substitute x, as the action is performed, which is denoted by $x \leftarrow r$. Thereby, the next action will become $\overline{c}\langle r \rangle$, *i.e.*, sending r via channel c. Notice that r is also a new identifier that is created dynamically. This is a *fresh input name* whose precise nature will be described in the next chapter.

Step 4: Replicator R sends out the request via channel c.



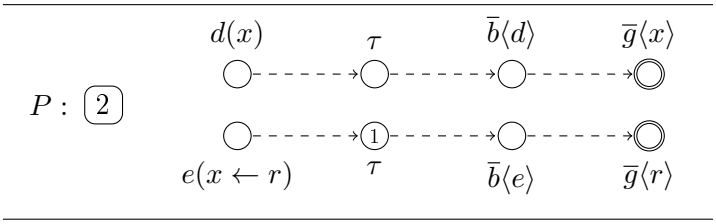
Because c is a restricted channel, it is used for communicating inside the system, thus only replicator D can receive r by performing an input $c(x)$. Similarly, thread 1 in replicator D is used for performing the action, and the variable name x is substituted by r after the communication. The next action that D can perform is either sending r via channel d or e, or sending it back to R if the request r is not correct. In this scenario we suppose that r is correct and it will be sent to replicator P via channel e.

Step 5: Because action $\overline{c}\langle r \rangle$ is final in R, the session of thread 1 is completed, and the thread is released.

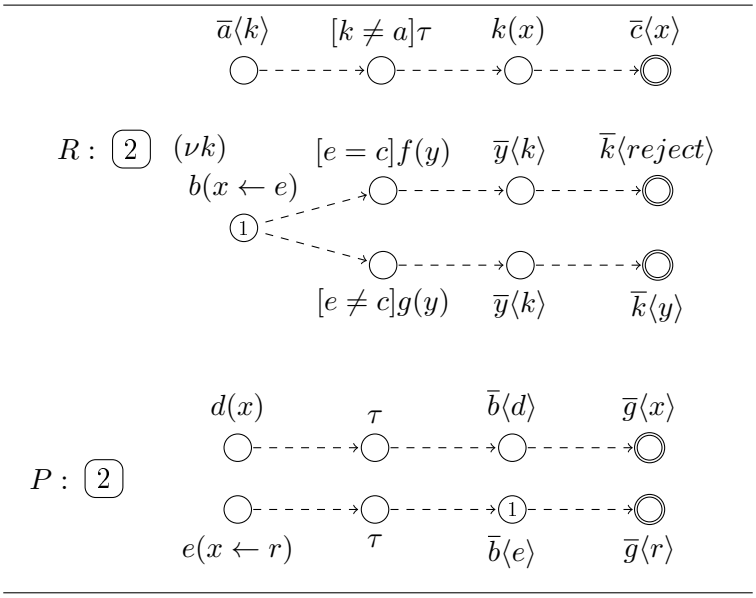


The replicator R goes to the initial state. Similarly to what happened in the previous step, there is a communication between replicators D and P by performing an output $\overline{e}\langle r\rangle$ in D and an input $e(x)$ in P.

Step 6: Replicator P continues with a silent action τ. Doing this action means that P processes the request r.

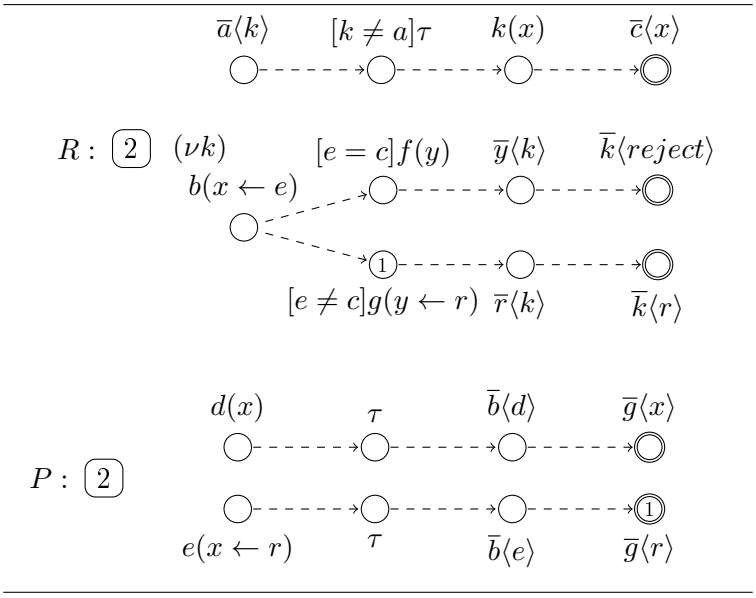


Step 7: The replicator P informs R that it will send the result. P sends a signal e via channel b, and R is ready for receiving on channel b.



Action $b(x)$ has two successors: $[x = c]f(y)$ and $[x \neq c]g(y)$. However, after the communication, only the action $[x \neq c]g(y)$ can be performed as the condition $[e = c]$ does not hold in this case.

Step 8: The replicator P does a communication with R via channel g. After the communication, R receives the result r.



Finally, R sends the result r to the client by sending first its restricted channel k to the client, then sending the result via the new channel.

3.3 A prototype tool for π-graphs

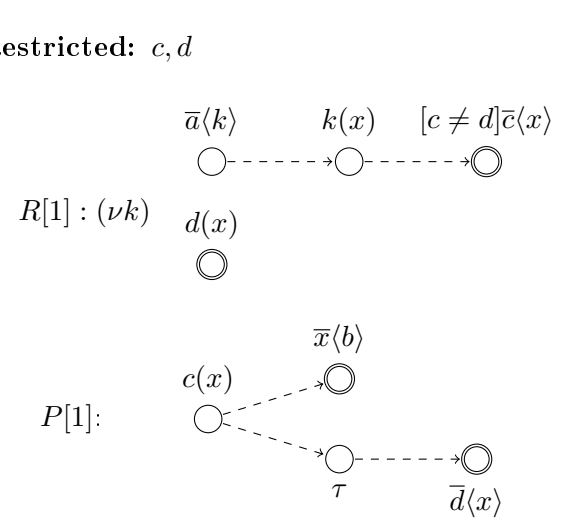
To study the semantics of π-graphs, we developed a π-graphs simulator PiSimilator, which supports the following main functionalities:

- it encodes a π-graphs model in a textual form,
- it explores the state-space of the model and simulate transitions, and
- it exports the transitions of a system in the form of a labelled transitions system into a textual representation, which can be used by external tools such as GRAPHVIZ [1] in order to visualize the evolution of the system.

PiSimilator's input language supports all the π-graphs constructs. As an illustration, we show in Figure 3.4b the textual representation of the π-graphs model represented in Figure 3.4a, which is a fragment of our running example from Figure 3.3 on page 42. The keywords *restricted* and *new* are used for indicating global restricted and local restricted names, respectively. As expected, symbols ‖ and + are used for the operators of parallel composition and non-deterministic choice.

Figure 3.5 shows the labelled transition system (LTS) of the model from Figure 3.4. It comprises 16 states represented by numbered circles with 0 being the initial one, and transitions represented by arcs with the following labels:

- *tau* meaning that the transition is a silent action or a communication,



(a) A simplified RDP system

```
Restricted c,d
[
    R[1] :! new(k) {
        a<k>.{k(x).{[c<>d]c<x>.{}}} +
        d(x).{}
    } ||

    P[1] :! {
        c(x).{
            tau.{d<x>.{}} +
            x<b>.{}
        }
    }
]
```

(b) Encoding of the simplified RDP system

Figure 3.4: A simplified version of the RDP system given in Figure 3.3

- $a(b)$ meaning that the transition is an input b on channel a, or
- $a < b >$ meaning that the transition is an output b on channel a.

In both last cases, a and b may be of the form of $?k$ or $!k$, with $k \in \mathbb{N}^+$, representing fresh names (*i.e.*, generated during the evolution of the system).

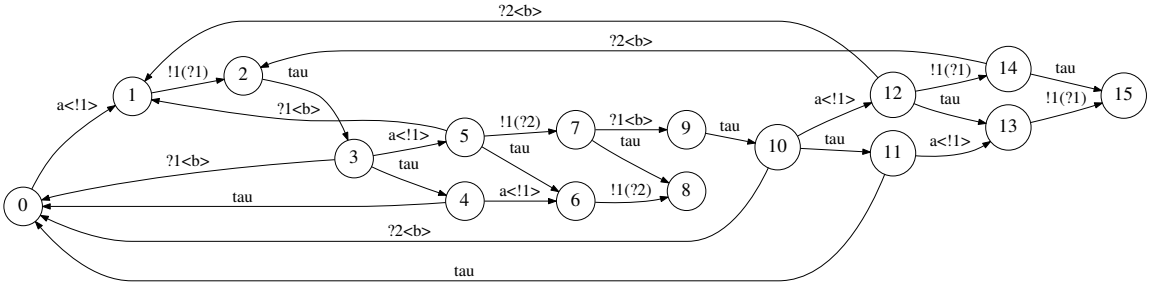


Figure 3.5: Semantics of the π-graphs of the simplified model in Figure 3.4

From the LTS in Figure 3.5 we can assert some properties of the system, such as for example that states 8 and 15 are deadlocks, while all the other states are not. Indeed, there is always a possible successor state for them. We may also observe that if we make a small modification on the model, such as changing the action $[c \neq d]\overline{c}\langle x\rangle$ in replicator R to $[c = d]\overline{c}\langle x\rangle$ and increasing the bound of R to 3, then the LTS of the obtained model (depicted in Figure 3.6) becomes much different: first, the number of states of the LTS increases from 16 to 55, and second, every path leads to a deadlock.

PiSimulator computes the LTS of a (textual) π-graphs specification in two steps:

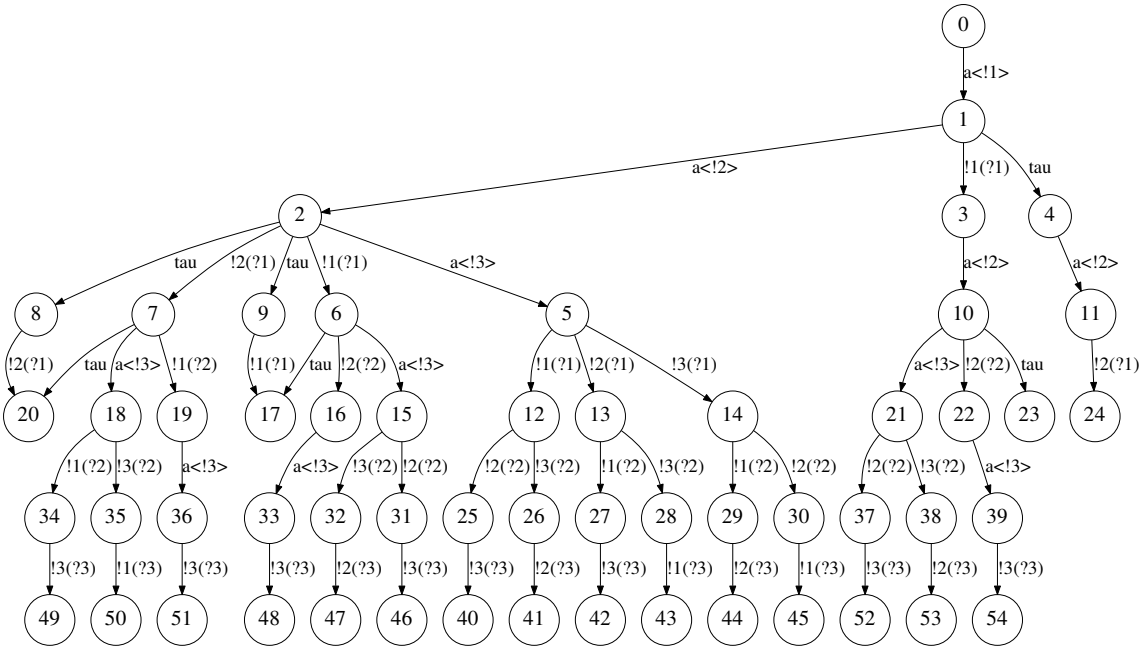


Figure 3.6: The LTS of the model in Figure 3.4 with two modifications: changing action $[c \neq d]\overline{c}\langle x\rangle$ in replicator R to $[c = d]\overline{c}\langle x\rangle$ and increasing the bound of R to 3

.

1. First, it inputs the specification by calling PiSimulator.input(spec),

2. Second, it builds the LTS by calling PiSimulator.build().

Moreover, PiSimulator provides the following additional functionalities:

- PiSimulator.reachable(C): checks if a context C (*i.e.*, C contains all information about a state) is reachable;

- PiSimulator.context(s): gets the corresponding context of a reachable state s. For example, it may be useful for determining why state 8 in Figure 3.5 is a deadlock by exploring its corresponding context;

- PiSimulator.enabled(s1,s2,l): checks if the system can evolve from state s1 to state s2 by performing an action with label l.

3.4 Synthesis

This chapter introduces the π-graphs formalism in an informal way and from a modelling perspective.

First, we describe a π-graphs model using a small example of an open client/server system. Each component is modelled by a replicator which has a thread bound representing the maximum number of actions can be performed concurrently in it. Replicators may use some names that are restricted to threads, and they may become public after be sent out of the environment. These components (replicators) are assumed to work in parallel. They can communicate with each other or perform internal communications between threads inside themselves. Especially, the communication topology of the system may change during the execution. In order to show the evolution of the system, a scenario is illustrated step by step. Based on this example, we can demonstrate most of the modelling features of the proposed formalism, such as interactions within open environments, concurrency, name restrictions and reconfigurations.

Then, we present the prototype tool that provides a simulator for π-graphs models. The tool, called PiSimulator, allows to encode π-graphs, explore the state space and simulate transitions of the system. It also supports exporting the labelled transition system into the input of a graph visualization tool, such as GRAPHVIZ, to visualize the evolution of the system. Thus, using the PiSimulator, we also can illustrate some simple properties of π-graphs models, such as deadlock or reversibility. Moreover, this tool may be useful for demonstrating the isomorphism of the translation of π-graphs into Petri nets (which will be done in Chapter 5) and interpreting the path counter-examples (which will be done in Chapter 6).

Chapter 4

The π-graphs formalism

This chapter describes the formal syntax and operational semantics of the π-graphs formalism.

4.1 Syntax of π-graphs

In this section we define the syntax of the process algebra of π-graphs. It is defined in top-down style. The top level is a diagram which represents the system we want to specify. Next levels are replicators, processes, and guarded actions. The syntax is defined in both graphical and textual forms.

4.1.1 Diagrams *Dia*

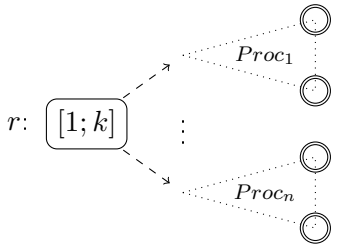
An open reconfigurable system is specified by a diagram. It specifies which replicators are part of the specification, and which names in the system are global restricted.

$$Dia ::= \widetilde{\nu A}\ [Rep_1 \parallel \ldots \parallel Rep_m] \quad \text{with } Rep_1, \ldots, Rep_m \text{ replicators}$$

All replicators that the system comprises are put in parallel using $\parallel$. The order of replicators in the specification of a diagram is not important. However, these replicators must be inside a scope that is indicated by a pair of square brackets [and] which are similar to keywords `begin` and `end` in some programming language. The global restricted names are indicated by a sequence $\widetilde{\nu A}$. In Figure 3.3 on page 42, the restricted names are b, c, d, e, f, g, this formally gives a list $\widetilde{\nu A} = \nu b, \ldots, \nu g$.

4.1.2 Replicators *Rep*

Each component in a system is specified by a replicator. It has a name, called replicator identifier, and a bound number of threads which is the maximum number of threads that can run simultaneously. A replicator specifies also processes in which threads run, and all names that are local to the replicator. Consider the following graphical and textual definition of a replicator,



$$Rep ::= r[k] :! \widetilde{\nu a}\ \{Proc_1 + \ldots + Proc_n\}$$

where

- r is the replicator identifier,
- k is its number of threads,
- $\widetilde{\nu a}$ is the list of its local restricted names, and
- $Proc_1, Proc_2, \ldots$ are processes.

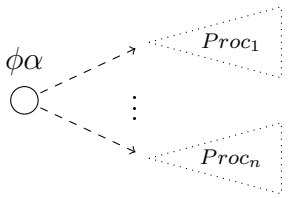
For example, replicator *Recp* in Figure 3.3 has only one local restricted name k, so $\widetilde{\nu a} = \nu k$. Moreover, a replicator may have more than one processes. At a moment, a thread can run on only one process. The choice of process on which a thread run is non-deterministic, which is specified by operator "+" as in π-calculus.

Intuitively, a replicator consists of a "*pool of threads*" from which threads are spawned for running processes. A replicator with maximum k threads has a set of threads with ids in $[1..k]$, which is denoted by $[1; k]$. A thread will disappear from the "pool of threads" as it is spawned, and it appears again as it ends a process. A process cannot start if all threads are spawned, *i.e.*, if there is no thread in the "pool of threads".

4.1.3 Processes *Proc*

In Def. 4.1, each replicator has a corresponding static graph whose nodes are guarded actions. In fact, the static graph of a replicator is a forest which consists of several possible trees. The root of each such tree represents the guarded action that will be performed at first for a given thread. After the guarded action is performed, if it is not final, one of its successors may be chosen for continuation performing. Each successor becomes the

root of a sub-tree, and the next guarded action is chosen in a same way up to a final one. The performance of a final action stops the process and the thread identifier returns to the "pool of threads". A tree can be regarded as a process which specifies how actions of a replicator are performed, *i.e.*, in which order of precedence the actions are performed. It is suitable to be defined recursively, as in the following:



$$Proc ::= \quad \phi\alpha.\{Proc_1 + \ldots + Proc_n\} \mid \qquad \phi\alpha.\{\}$$

In the above, operator + denotes the non-deterministic choice, while operator . denotes the precedence. By convention, if a process has only one successor, *e.g.*, $\phi\alpha.\{P\}$, we can omit brackets, $\phi\alpha.P$. Similarly, if a process has no successor, *e.g.*, $\phi\alpha.\{\}$, we write only the guarded action, $\phi\alpha$.

For example, the replicator D (for Dispatcher) in Figure 3.3 has only one process. It can defined as follows:

$$c(x).\{\overline{d}\langle x\rangle + \overline{e}\langle x\rangle + \overline{b}\langle c\rangle.\{\overline{f}\langle x\rangle\}\}$$

4.1.4 Guarded actions $\phi\alpha$

A guarded action is a conditional one. It is a combination between a guard and an action. An action can be a silent action, an output, or an input. A guard is a sequence of comparisons of names (match or mismatch).

(guard)	ϕ	::=	`true` $\mid m\phi$
(match/mismatch)	m	::=	$[a = b] \mid [a \neq b]$
(action)	α	::=	(silent) τ $\mid$ (output) $\overline{c}\langle a\rangle$ $\mid$ (input) $c(x)$

where a, b, c are names and x is a variable. A match is a comparison of two names that is true if these two names are compatible, otherwise it is false. Similarly, a mismatch is true if these names are not compatible, otherwise it is false. If a guard comprises many comparisons, then it is interpreted as a conjunction of all comparisons. For example, the guard $[a = b][b \neq c]$`true` is interpreted as a conjunction $[a = b] \wedge [b \neq c]$. We can omit `true` if the sequence of name comparisons in a guard is not empty.

Using the syntax of π-graphs, the textual specification of the illustrative example in Figure 3.3 is provided in Listing 4.1, where *restricted* and *new* are keywords used for indicating global restricted and local restricted names, respectively. The symbols ‖ and + denote the parallel composition and non-deterministic choice operators.

Listing 4.1: The π-graphs specification of the system RDP in Figure 3.3

```
Restricted b,c,d,e,f,g
[
        R[2] :! new(k) {
             a<k>.{[a<>k]tau.{k(x).{c<x>.{}}}} +
             b(x).{
                 [x=c]f(y).{y<k>.{k<reject>.{}}} +
                 [x<>c]g(y).{y<k>.{k<y>.{}}}
             }
        } ||

        D[2] :! {
             c(x).{
                 d<x>.{} +
                 e<x>.{} +
                 b<c>.{f<x>.{}}
             }
        } ||

        P[2] :! {
             d(x).{tau.{b<d>.{g<x>.{}}}} +
             e(x).{tau.{b<e>.{g<x>.{}}}} +
        }
]
```

4.2 Operational semantics of π-graphs

The semantics of a π-graphs diagram is defined as a labelled transition system in which the states are configurations the system may reach and the labelled transitions indicate under which conditions the change of the states may occur. Unlike classical process algebras, like π-calculus where the states correspond to reached terms, the states in π-graphs are composed of two parts:

- A static part, which corresponds to the structure of the π-graphs diagram that does not change during the evolution, and
- A dynamic part, which is composed of threads and name contexts, called together a global context, which are interpreted on the diagram structure.

These two parts are similar to what happens in models like Petri-nets where there are also a static part (a Petri-net structure) and a dynamic part (tokens). This division will make easier the translation from π-graphs to Petri-nets that we will present in the next chapter.

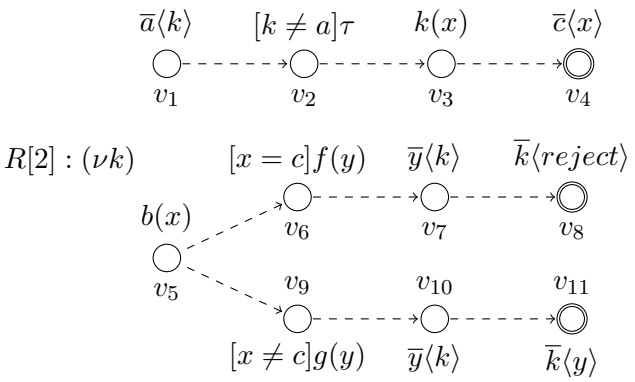
4.2.1 Static part: Graph model of a π-graph

In this section we formalize the concept of a π-graph. A π-graph comprises some replicators. Each replicator owns a set of threads that can run simultaneously, and a set of actions that the replicator can perform. As the actions of a replicator are performed in order of precedence, they can be represented as a directed graph whose nodes are guarded actions, *e.g.*, input $[x = c]f(y)$ or output $\overline{a}\langle k\rangle$. A replicator also owns names that are used only locally, called local restricted names. We also have global restricted names at the π-graph level. All the other names are considered as free. Formally, a π-graph is defined as follows.

Definition 4.1. *A* π-graph π *is a tuple* $\langle \mathcal{R}, \mathcal{K}, \mathcal{G}, \mathcal{N}\rangle$ *with*

- $\mathcal{R}$ *a finite set of replicator identifiers,*
- $\mathcal{K} \in \mathcal{R} \to \mathbb{N}$ *the thread bounds (maximum number of simultaneously running threads),*
- $\mathcal{G}$ *is a function from* $\mathcal{R}$ *such that each* $\mathcal{G}(r)$ *is a static graph* $\langle V_r, L_r, E_r\rangle$ *with*
  - V_r *the set of vertices of the graph,*
  - $L_r \in V_r \to \{\phi\alpha$ *guarded action*$\}$ *the labelling of each vertex with a guarded action,*
  - $E_r \subset V_r \times V_r$ *the set of directed edges of the graph*
- $\mathcal{N} = \langle \mathcal{N}_{\mathsf{gres}}, \mathcal{N}_{\mathsf{free}}, \mathcal{N}_{\mathsf{tres}}, \mathcal{N}_{\mathsf{var}}\rangle$ *are syntactic names, where*
  - $\mathcal{N}_{\mathsf{gres}} \in \Sigma \to \mathbb{N}$ *a set of graph-restricted names,*
  - $\mathcal{N}_{\mathsf{free}} \subset \Sigma$ *a set of free names.*
  - $\mathcal{N}_{\mathsf{tres}} \stackrel{\text{def}}{=} \{r \mapsto \overrightarrow{N} \mid r \in \mathcal{R} \wedge \overrightarrow{N} \in \Sigma \to \mathbb{N}\}$ *a function from* $\mathcal{R}$ *to vectors of thread-restricted names,*
  - $\mathcal{N}_{\mathsf{var}} \in \mathcal{R} \to \Sigma$ *a function from* $\mathcal{R}$ *to sets of bound variables,*

For example, with the π-graph model in Figure 3.3, $\mathcal{R} = \{R, D, P\}$, and $\mathcal{K}(R) = \mathcal{K}(D) = \mathcal{K}(P) = 2$. The set of global restricted names $\mathcal{N}_{\mathsf{gres}} = \{b, c, d, e, f, g\}$, and the set of free names $\mathcal{N}_{\mathsf{free}} = \{a, reject\}$. We illustrate the static graphs of replicators for one of them, such as $\mathcal{G}(R)$. Suppose that vertices of $\mathcal{G}(R)$ are $V_R = \{v_1, v_2, \ldots, v_{11}\}$, as follows:



The static graph of replicator R and its local restricted names are as follows:

- Labelling map of vertices: $L_R(v_1) = \overline{a}\langle k\rangle, L_R(v_2) = [k \neq a]\tau, L_R(v_3) = k(x)$, $L_R(v_4) = \overline{c}\langle x\rangle, L_R(v_5) = b(x), L_R(v_6) = [x = c]f(y), L_R(v_7) = \overline{y}\langle k\rangle$, $L_R(v_8) = \overline{k}\langle reject\rangle, L_R(v_9) = [x \neq c]g(y), L_R(v_{10}) = \overline{y}\langle k\rangle, L_R(v_{11}) = \overline{k}\langle y\rangle$,
- Set of directed edges: $E_R = \{(v_1, v_2), (v_2, v_3), (v_3, v_4), (v_5, v_6), (v_6, v_7), (v_7, v_8), (v_5, v_9), (v_9, v_{10}), (v_{10}, v_{11})\}$,
- Local restricted names $\mathcal{N}_{\mathsf{tres}}(R) = \{k\}$, and bound variables $\mathcal{N}_{\mathsf{var}}(R) = \{x\}$.

4.2.2 Dynamic part: Global context of a π-graph

The global context of a π-graph diagram represents all information about the current configuration of the system, which consists of information about threads and about names. The information about threads, called *control flow context*, indicates which threads are already used and for which actions, and which threads are still in the pool of threads. The information about names, called *name context*, indicates which ones are received and which ones are sent out to the environment. Moreover, it indicates also the scope of names and the relation between them, *i.e.*, which names are known outside and which ones are known only inside the system, which names are compatible and which ones are not.

4.2.2.1 Control flow context

The *control flow context* of a π-graphs diagram, denoted by Δ, can be regarded as a distribution of threads to vertices. It determines which threads are used (spawned) and which ones are not, *i.e.*, the availability of threads for performing actions. Let π be a π-graphs diagram. We recall that, by Def. 4.1, each replicator r in π has a set of vertices V_r and a set of threads that can run concurrently. Thus, the control flow context Δ of a diagram can be defined as the control flow context of each replicator r, denoted by Δ_r.

Definition 4.2 (Control flow context)**.** *The* control flow context Δ_r *for a replicator* r *is a mapping* $V_r \to \mathbb{P}(\mathbb{N})$ *associating (possibly empty) sets of thread identifiers to some* π*-graph vertices. The set of threads that are present at a given vertex* v *in replicator* r *is denoted by* $\Delta_r(v)$*.*

Threads of a replicator are represented by thread identifiers, called thread ids for short, which are positive integer numbers. A vertex v in replicator r is said to *have control for a thread* i if i is present at vertex v, *i.e.*, $i \in \Delta^r(v)$. The vertex v is said to *have control* if there are some threads such that v has control for them, *i.e.*, $\Delta_r(v) \neq \emptyset$. Initially, no vertex of replicator r has control. The initial control flow context of r is $\Delta_r^0 = \{v \to \emptyset \mid v \in V_r\}$.

We consider an example of control flow context of replicator R which represents component Receptionist in Figure 3.3. Suppose that at some stage replicator R already received a request from some client on thread 1, and already sent back a result to some client on thread 2, as illustrated in Figure 4.1. The control flow context of R at this stage is:

$$\Delta_R = \{v_3 \to \{1\}; v_{11} \to \{2\}; v_j \to \emptyset \mid j \in \{1, 2, 4..10\}\}$$

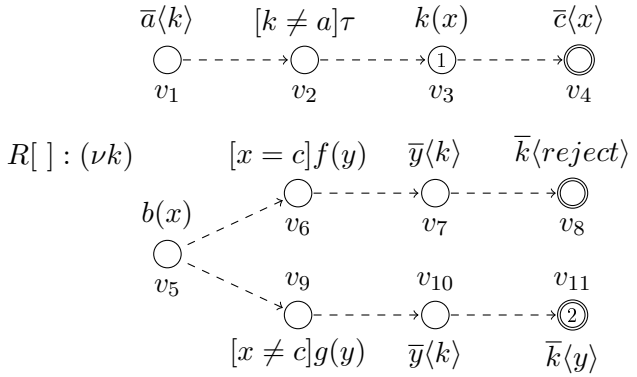


Figure 4.1: A threads distribution of replicator R

Suppose that at a next stage replicator R can send the received request r to replicator D, then thread 1 will move to vertex v_4 to perform the output $\overline{c}\langle r\rangle$. Moreover, thread 2 will be released as the action at vertex v_{11} is final. Thus, the next control flow context of R is:

$$\Delta'_R = \{v_4 \to \{1\}; v_j \to \emptyset \mid j \in \{1..3, 5..11\}\}$$

The control flow context of a replicator evolves with the evolution of the system. As a thread is spawned for performing a process of the replicator, it will appear at the first action of the process. When an action at vertex v has control for a thread i, if v has successors then thread i can perform an action at one of the successors, otherwise, i is terminated, it moves back to the "pool of threads" and becomes available for the next execution of a process. Threads can be spawned, can be terminated, or can move between

vertices of a replicator, but the total number of threads on all vertices of r can not exceed the capacity $\mathcal{K}(r)$.

Invariant 4.1. $\forall r \in \mathcal{R},\ \sum_{v \in V_r} |\Delta_r(v)| \leq \mathcal{K}(r)$ *where* $\mathcal{R}$ *is a set of all replicators of* π.

Definition 4.3 (Inactive threads)**.** *For a given configuration, an* inactive thread *is a thread that is not used for performing any action. Let* $\pi = \langle \mathcal{R}, \mathcal{K}, \mathcal{G}, \mathcal{N} \rangle$ *be a* π*-graph, and* Δ_r *the control-flow context for* $r \in \mathcal{R}$*. The set of inactive threads ids of replicator* r *is denoted by* $\mathsf{inact}(\Delta_r)$*, defined as follows:*

$$\mathsf{inact}(\Delta_r) \stackrel{\text{def}}{=} \{1, \ldots, \mathcal{K}(r)\} \setminus \bigcup_{v \in V_r} \Delta_r(v)$$

Proposition 4.1 (Thread location)**.** *For any* r *in* $\mathcal{R}$ *and* i *in* $[1..\mathcal{K}(r)]$*, either* $i \in \mathsf{inact}(\Delta_r)$ *or there exists a unique vertex* $v \in V_r$ *such that* $i \in \Delta_r(v)$.

Proof. At each stage, a thread i can either be used by only one action or be in the "pool of threads". If it is in the pool, recall that the control-flow context Δ_r is a mapping $V_r \to \mathbb{P}(\mathbb{N})$ (cf. Def. 4.2), there is no vertex that has control for i, thus $i \notin \mathsf{cod}(\Delta_r)$. By Def. 4.3, i is inactive in r, i.e. $i \in \mathsf{inact}(\Delta_r)$. Otherwise, there exists a vertex $v \in V_r$ that has control for thread i, i.e. $i \in \Delta_r(v)$. Moreover, v is unique in V_r. Thus, the property holds. □

4.2.2.2 Name context

A name context of a π-graph keeps information about what was already received and sent out of the system and about the scopes of names and the relations between them. This information may change during the evolution of the system. To represent these changes, we associate *syntactic names* to their *instances*, which can be regarded as their values, and all information about syntactic names is encoded in the instances and the relations between them. At each stage, the instances of syntactic names may be updated.

Definition 4.4. *Let* $\pi = \langle \mathcal{R}, \mathcal{K}, \mathcal{G}, \mathcal{N} \rangle$ *be a* π*-graph. The set* $\mathcal{S}$ *of* ***syntactic names*** *of* π *is the (disjoint) union of its global restricted names and its thread restricted names (private and variable names) which are indexed by thread ids:*

$$\mathcal{S} \stackrel{\text{def}}{=} \mathcal{N}_{\mathsf{gres}} \cup \mathcal{N}_{\mathsf{free}} \cup \bigcup_{r \in R, i \in [1..\mathcal{K}(r)]} (\{\nu_i^r a \mid \nu^r a \in \mathcal{N}_{\mathsf{tres}}(r)\} \cup \{y_i^r \mid y^r \in \mathcal{N}_{\mathsf{var}}(r)\})$$

where $\mathcal{N}_{\mathsf{gres}}$ *and* $\mathcal{N}_{\mathsf{free}}$ *are sets of global restricted names and free names,* $\mathcal{N}_{\mathsf{tres}}(r)$ *and* $\mathcal{N}_{\mathsf{var}}(r)$ *are sets of thread restricted names and variable names of replicator* r.

Global restricted and free names do not depend on threads, so each global restricted name itself is a syntactic one. However, a thread restricted name may correspond to

several syntactic ones depending on the number of threads of the replicator to which the name belongs. For example, $\mathcal{S} = \mathcal{S}_1 \cup \mathcal{S}_2$ is the set of syntactic names of the π-graph that is illustrated in Figure 3.3, where

$$\mathcal{S}_1 = \mathcal{N}_{\mathsf{gres}} \cup \mathcal{N}_{\mathsf{free}} = \{b, c, d, e, f, g\} \cup \{a, reject\}.$$

Moreover, notice that replicator R has a private name k, thus it gives the corresponding syntactic names νk_1^R and νk_2^R. Similarly, each variable of each replicator has its corresponding syntactic version for each thread. Thus

$$\mathcal{S}_2 = \{\nu k_1^R, \nu k_2^R, x_1^R, x_2^R, x_1^D, x_2^D, x_1^P, x_2^P\}.$$

The instances of syntactic names may be free names or *fresh names*. The last one are classified into three kinds: fresh input names for input information, output names for output information, and private names for private information. We introduce the following notations that allow us to specify these kinds of fresh names:

Definition 4.5. *The set of* ***fresh names*** $\mathcal{F}$ *is the disjoint union of:*

$$\begin{cases} \mathcal{F}_! \subseteq \{!n \mid n \in \mathbb{N}\} \text{ the set of output names} \\ \mathcal{F}_? \subseteq \{?n \mid n \in \mathbb{N}\} \text{ the set of input names} \\ \mathcal{F}_\nu \subseteq \{\nu n \mid n \in \mathbb{N}\} \text{ the set of private names} \end{cases}$$

Since the system that we model is open, it can potentially receive anything from the environment and we do not know in advance what will be received. Moreover, if the system sends out private information, then the sent information is new for the environment. We need notations for describing these situations. In Def. 4.5, our approach is to use integer numbers together with different prefixes ($!, ?$, and ν) to represent the different kinds of fresh names. For example, three names $!1, ?1, \nu 2$ are respectively input, output, and private names. The fresh name $!1$ can appear, for example, when the system sends out private information. Similarly, $?1$ can appear when the system receives something.

In order to express configurations, the name context should provide the information about:

- what is the current instantiation of the syntactic names: this will be ensured by the mapping β, called *name environment*, which associates a value with each syntactic name;
- which instances have already been matched and should be interpreted as equal: this will be provided by a *dynamic partition* γ of instances. Each set in γ (in fact an equivalent class for equality) gathers names (instances) which have already been matched;

- which instances have already been mismatched and should be interpreted as different: this will be provided by a set of pairs of sets in δ, called *distinctions*. A distinction $C_1 \leftrightarrow C_2$ records the fact that instances in C_1 are different from those in C_2.

These notations are illustrated in Figure 4.2. The name context is represented on the left. The dynamic partition γ and the set of distinctions δ form a graph where the nodes are the sets in γ and the edges are the pairs in δ. In particular, one can see that a and ?1 have already been matched (they are both in the same class), as well as ?2 and !1. Also, the name $\nu 1$ is different from all the others (the class containing it has a distinction with all the other classes).

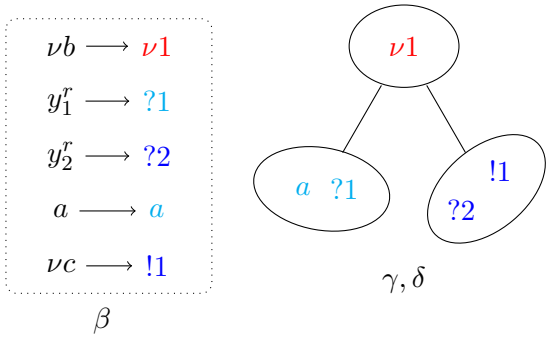


Figure 4.2: An example of name context

In the context in Figure 4.2, syntactic names are in black and the instances are in other colours. Based on this context, we have assertions of scopes of syntactic names and the relations between these as follows:

- νb is private to the system, but other syntactic names are not private;
- $(\nu b \neq a) \wedge (\nu b \neq \nu c) \wedge (\nu b \neq x_1^r) \wedge (\nu b \neq x_2^r)$;
- $(a = x_1^r) \wedge (\nu c = x_2^r)$.

In the following, we present more formally and in more details three parts (β, γ, δ) of a name context and we give the definition of π-graphs name context. Moreover, abstract devices, called logical clocks, are also introduced. They are used for generating fresh names in a name context.

Name environment The name environment β relates the syntactic occurrences of names in the π-graphs diagram to their instances, which can be either a syntactic name, a fresh name or $\perp$. The symbol $\perp$ represents an unknown instance that is similar to the symbol nil in some programming language. The association $x \rightarrow \perp$ indicates that the instance of syntactic name x is undefined (or unknown).

Given a π-graph $\pi = \langle \mathcal{R}, \mathcal{K}, \mathcal{G}, \mathcal{N} \rangle$ and a name x in π, *i.e.*, $x \in \mathcal{N}_{\mathsf{gres}} \cup \mathcal{N}_{\mathsf{free}} \cup \bigcup_{r \in \mathcal{R}} \mathcal{N}_{\mathsf{tres}}(r) \cup \bigcup_{r \in \mathcal{R}} \mathcal{N}_{\mathsf{var}}(r)$, the notation $\beta_i^r(x)$ gives the instance of x for replicator r and thread i, as follows:

$$\beta_i^r(x) \stackrel{\text{def}}{=} \begin{cases} \beta(x) \text{ if } x \in \mathsf{dom}(\beta) \wedge (\mathsf{gres}(x) \text{ or } \mathsf{free}(x)) \\ \beta(x_i^r) \text{ if } x_i^r \in \mathsf{dom}(\beta) \wedge (\mathsf{tres}(x) \text{ or } \mathsf{var}(x)) \end{cases} \tag{4.1}$$

where $\mathsf{gres}(x)$, $\mathsf{tres}(x)$, $\mathsf{free}(x)$, and $\mathsf{var}(x)$ are predicates asserting that x is a global restricted name, a thread restricted name, a free name, or a variable name, respectively. They are defined as follows:

- $\mathsf{gres}(x)$ if $x \in \mathcal{N}_{\mathsf{gres}}$
- $\mathsf{tres}(x)$ if $x = \nu_i^r a$ with $r \in \mathcal{R}$, $\nu^r a \in \mathcal{N}_{\mathsf{tres}}(r)$ and $i \in [1..\mathcal{K}(r)]$
- $\mathsf{var}(x)$ if $x = y_i^r$ with $r \in \mathcal{R}$, $y^r \in \mathcal{N}_{\mathsf{var}}(r)$ and $i \in [1..\mathcal{K}(r)]$
- $\mathsf{free}(x)$ otherwise.

If x is a global restricted or free name, then it does not depend on threads nor on replicators, thus the corresponding syntactic name of x is x itself. In this case, the instance of x in r and thread i is just $\beta(x)$. Similarly, if x is a thread restricted or a variable name, then it depends on replicators and threads, thus the corresponding syntactic name in replicator r and thread i is x_i^r. Therefore, the instance of x is $\beta(x_i^r)$. In both cases, it must be ensured that the corresponding syntactic name of x is present in the domain of β which is asserted by predicate $x \in \mathsf{dom}(\beta)$ or $x_i^r \in \mathsf{dom}(\beta)$. For example, in the name context in Figure 4.2 we have: $\beta_1^r(y) = ?1, \beta_1^r(a) = \beta_2^r(a) = a$, and $\beta_1^r(b) = \beta_2^r(b) = \nu 1$.

Dynamic partition The dynamic partition γ gives a contextual representation of name equality, that may evolve over time. The partition applies on the range of the name environment β. For two names $a, b \in \mathsf{ran}(\beta)$, if we have $[a]_\gamma = [b]_\gamma$ (or there exist an equivalence class C in γ such that $a, b \in C$), then a and b are considered equal and in particular, the guard $[a = b]$ will be evaluated to `true`.

A peculiarity of π-graphs (and thus the π-calculus) is that we cannot consider a and b as unequal solely based on the fact that they are in distinct equivalent classes of γ, we need explicit *distinctions*.

Distinctions A distinction in δ relates two equivalence classes of γ when they are asserted unequal. Graphically, the pair γ, δ can be seen as an undirected graph with the equivalences classes of γ as vertices, and the distinctions as edges (see the right-hand side part of Figure 4.2). Since edges are undirected, we work up to symmetry. Let C_1

and C_2 be two distinct equivalent classes of γ, then we denote $C_1 \leftrightarrow_\gamma C_2 \in \delta$ to express $C_1, C_2 \in \gamma$ and both $(C_1, C_2) \in \delta$ and $(C_2, C_1) \in \delta$.

To understand the relation between the dynamic partition and the distinctions, consider x_1, x_2 two names occurring in γ. Then:

- if there exists a class in γ which contains both x_1 and x_2 then $x_1 = x_2$ is true
- if x_1 and x_2 are in different classes C_1 and C_2 then:
  - if there exists a distinction between C_1 and C_2 then $x_1 \neq x_2$ is true
  - otherwise, we do not know whether $x_1 = x_2$ or $x_1 \neq x_2$ is true or false. For example, in Figure 4.2, we do not know whether ?1 =?2 or ?1 $\neq$?2.

Definition 4.6. *A π-graphs* name context Γ *is a triple* $\langle \beta, \gamma, \delta \rangle$ *with:*

- $\beta \in \mathcal{S} \to \mathcal{S} \cup \mathcal{F} \cup \{\bot\}$ *a* ***name environment***
- $\gamma \subset \mathbb{P}(\mathcal{S} \cup \mathcal{F})$ *a* ***dynamic partition*** *of* $\mathsf{ran}(\beta)$
- $\delta \subseteq \gamma \times \gamma$ *a set of* ***distinctions***

where $\mathcal{S}$ is a set of syntactic names, $\mathcal{F}$ is a set of fresh names, and $\bot$ is a special name representing an undefined instance.

4.2.2.3 Logical clocks

During the evolution of the system, fresh names may be created or removed but it must be ensured that the created names are not used in the range of the name environment yet and the removed names can be reused later. To do that, we define three *logical clocks* used to generate fresh values based on the current range of β, as follows:

- Output clock $\mathsf{clk}_!(\beta, \gamma, \delta) \stackrel{\mathsf{def}}{=} \mathsf{min}(\mathbb{N} \setminus \{n \mid !n \in \mathsf{ran}(\beta)\})$
- Input clock $\mathsf{clk}_?(\beta, \gamma, \delta) \stackrel{\mathsf{def}}{=} \mathsf{min}(\mathbb{N} \setminus \{n \mid ?n \in \mathsf{ran}(\beta)\})$
- Private clock $\mathsf{clk}_\nu(\beta, \gamma, \delta) \stackrel{\mathsf{def}}{=} \mathsf{min}(\mathbb{N} \setminus \{n \mid \nu n \in \mathsf{ran}(\beta)\})$

As in Def. 4.5, fresh names are classified into three types: input names, output names, and private names. Each type of fresh names is generated by a corresponding logical clock, *i.e.*, the output clock generates output names, input clock generates input names, and private clock generates private names. We describe below how an output clock generates fresh names, the other types of clocks work similarly. This can be done in two steps as follows:

1. Compute all input names in the range of the current name environment,
2. Determine what is the smallest unused id (minimum value in $\{n \mid !n \in \mathsf{ran}(\beta)\}$). It is used for generating the output name.

For example, suppose that the current name environment β of a name context Γ is as in Figure 4.3 that has a set of fresh input names $\mathcal{F}_\nu = \{\nu 1\}$, a set of fresh input names $\mathcal{F}_? = \{?1\}$ and a set of fresh output names $\mathcal{F}_! = \{!1\}$. The performance of an input action $d(e)$ updates the name environment β to β'. In β', input name ?2 that is generated by the input clock is assigned to e.

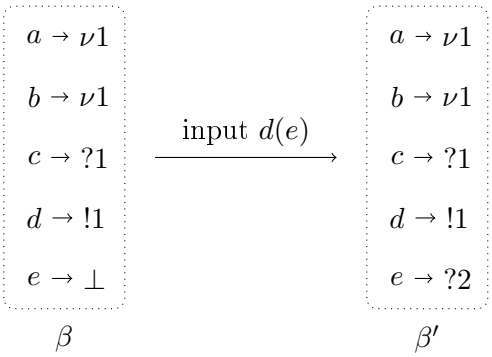


Figure 4.3: Generating input name ?2 (and a new name environment)

4.2.3 Operators on a name context

There are many events that may give rise to change a name context of a π-graph. For example, spawning a thread or terminating a thread after performing a terminal action, etc. We define four basic operators allowing to update the name context as follows:

1. Removing instantiations,
2. Instantiating a name,
3. Exchanging an instantiation, and
4. Refining the dynamic partition.

4.2.3.1 Removing instantiations

In the name context, an instantiation can be thought of as a value which is assigned to one or more syntactic names. During the evolution of the context, a syntactic name can be updated, for example, when an input action is activated. To do this, it first releases the old value and then takes the new one. If the released value is no longer used by

another syntactic name then it will be removed from the name context. The operator of removing instantiations of syntactic names allows us to release the instantiations from the syntactic names (setting it to $\perp$).

Let $\Gamma = (\beta, \gamma, \delta)$ be a name context of a π-graphs diagram π, X a set of syntactic names whose values will be released from a thread i in replicator r. The operator removing instantiations of X is defined as follows:

$$\Gamma_i^r - X = \beta', \gamma', \delta' \text{ with } \left[\begin{array}{l} \beta' \stackrel{\text{def}}{=} \beta \setminus \left(\begin{array}{l} \{x \mapsto \beta_i^r(x) \mid x \in X \cap \mathsf{dom}(\beta) \wedge \\ (\mathsf{gres}(x) \vee \mathsf{free}(x))\} \\ \cup \{x_i^r \mapsto \beta_i^r(x) \mid x \in X \wedge x_i^r \in \mathsf{dom}(\beta) \wedge \\ (\mathsf{tres}(x) \vee \mathsf{var}(x))\} \end{array}\right) \\ \gamma' \stackrel{\text{def}}{=} \{C' \mid C' \neq \emptyset \wedge C' = C \setminus Z \wedge C \in \gamma\} \\ \delta' \stackrel{\text{def}}{=} \{C_1 \setminus Z \leftrightarrow_{\gamma'} C_2 \setminus Z \mid C_1 \leftrightarrow_{\gamma} C_2 \in \delta\} \\ \quad \text{with } Z \stackrel{\text{def}}{=} \{z \mid \exists x \in X,\ z = \beta_i^r(x) \wedge z \notin \mathsf{ran}(\beta')\} \end{array}\right.$$

This operator updates all three parts of the name context. In the name environment β, all syntactic names $a \in X$ will get a new value $\perp$ which indicates that their instantiations become undefined. In the dynamic partition γ and the distinctions δ, all instantiations that are no longer used in β will be removed, empty classes and their connections to others will be also removed.

For example, Figure 4.4 illustrates the operator of removing instantiations of two syntactic names a and d from a name context $\Gamma = (\beta, \gamma, \delta)$. Before removing, $\nu 1$ is assigned to both a and b, !1 is assigned to d. After removing, both a and d take $\perp$ as their instantiations. The instantiation !1 of d is removed from γ but $\nu 1$ is maintained because it is also assigned to b. Because !1 is alone so after removing it, the corresponding class and the connection to the class of $\nu 1$ is also removed. The name context Γ is updated to $\Gamma' = (\beta', \delta', \gamma')$.

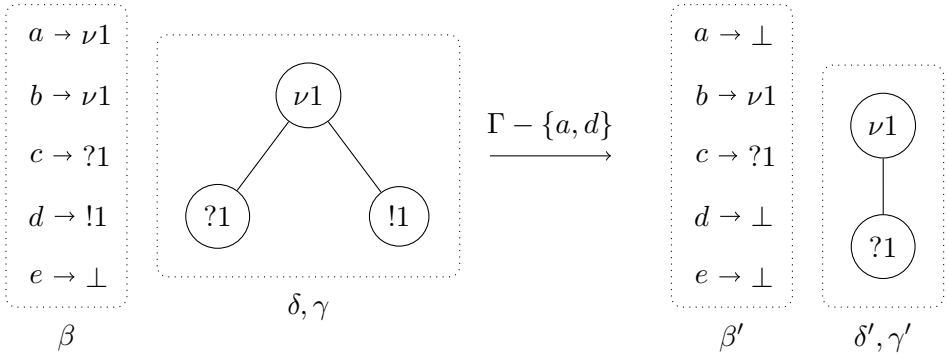


Figure 4.4: Removing instantiations of two syntactic names a and d

4.2.3.2 Instantiating a name

The instance of a syntactic name can be updated to different values during the evolution of the name context. For example, after performing an input action $c(x)$, variable x is substituted by the received information and the old value of x may be removed if it is no longer assigned to any other syntactic name. The update is performed by applying the operator of name instantiation, which is done in two steps:

- remove the instantiation of the syntactic name, then
- insert the pair of the syntactic name and its new instantiation into the name environment and update the dynamic partition and distinctions.

More formally, let Γ be the name context of a π-graphs diagram π, r a replicator in π and i a given thread identity of r. We instantiate a syntactic name x in r with a fresh name z by performing two steps: first, we remove the instantiation of x using the operator defined above to obtain a new name context, $\Gamma_i^r - \{x\}$, and then insert the association $x \to z$ into the new name context. The operator of inserting an association into a name context is defined as follows:

$$\Gamma_i^r + (x \mapsto z) = \beta', \gamma', \delta' \text{ with } \begin{cases} \beta' \stackrel{\text{def}}{=} \begin{cases} \beta \cup \{x \mapsto z\} & \text{if } \mathsf{gres}(x) \vee \mathsf{free}(x) \\ \beta \cup \{x_i^r \mapsto z\} & \text{otherwise} \end{cases} \\ \text{if } \mathsf{fresh}_\Gamma(z) \text{ then :} \\ \begin{cases} \gamma' \stackrel{\text{def}}{=} \gamma \cup \{\{z\}\} \\ \delta' \stackrel{\text{def}}{=} \begin{cases} \delta \cup \{\{z\} \leftrightarrow_{\gamma'} C \mid C \neq \{z\}\} \text{ if } \mathsf{priv}(z) \\ \delta \cup \{\{z\} \leftrightarrow_{\gamma'} \{y\} \mid \mathsf{priv}(y)\} \text{ otherwise} \end{cases} \end{cases} \\ \text{otherwise } \gamma' = \gamma \wedge \delta' = \delta \end{cases}$$

where

- $\mathsf{fresh}_\Gamma(z)$ is a predicate asserting that a fresh name z has been used or not in the name context Γ. It is defined as follows:

$$\mathsf{fresh}_{\beta,\gamma,\delta}(z) \stackrel{\text{def}}{=} z \notin \mathsf{dom}(\beta) \cup \mathsf{ran}(\beta) \cup \{\bot\}$$

- $\mathsf{priv}(z)$ is a predicate asserting that a fresh name z is a private fresh name; it is true iff $\exists n \in \mathbb{N}^*,\ z = \nu n$. Similarly, to check if a given name z is not a private fresh name, we define a predicate $\mathsf{pub}(z)$ which is true iff $\neg\mathsf{priv}(z)$.

The idea of inserting an association $x \to z$ into the name context $\Gamma = (\beta, \delta, \gamma)$ is that if z is an unused fresh name, *i.e.*, $\mathsf{fresh}_\Gamma(z)$ is true, then we add into γ a new singleton class C that contains only z and update the distinctions in δ between C and the other classes. If z is a private fresh name then we add distinctions between C and all others

to ensure that z is different from all the other names, otherwise, we add a distinction between C and classes of other private fresh names to ensure that z is different from all the other fresh private names. Moreover, an association between a and z is added into the name environment. If z is already existing in the name context, then both γ and δ do not change.

Figure 4.5 illustrates the operator of instantiating a syntactic name a with a fresh output name !2, denoted by $\Gamma[a \leftarrow !2]$. This operator may affect all three components of the name context. Let $(\beta', \delta', \gamma')$ be the new name context after performing this operator. For the name environment, since $\nu 1$ is the instantiation of both a and b, it still appears in β'. However, in β', !2 is associated with a. For the dynamic partition and distinctions, since !2 does not appear in β, a new class for !2 is added to δ and γ is also updated. Because !2 is not a private fresh name, only one distinction between the class of !2 and the class of $\nu 1$ is added to γ.

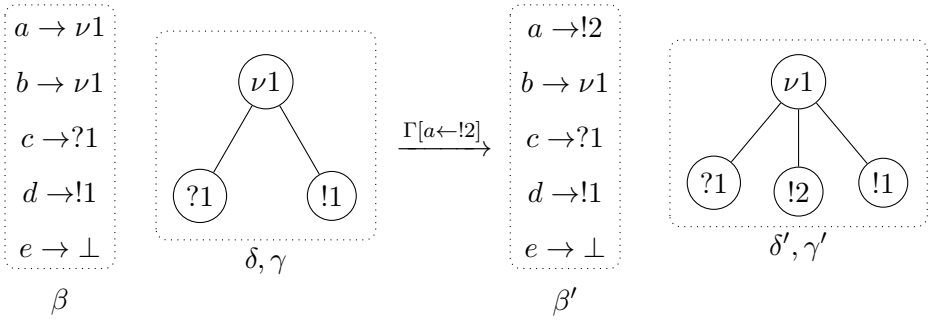


Figure 4.5: Instantiating a name a with !2

4.2.3.3 Exchanging instantiations

This operator allows to exchange a private fresh name, which is alone in a class of the dynamic partition, with a new fresh output name.

Given a private fresh name νn in the name context Γ and a fresh output name $!m$, the exchange between νn and $!m$ is defined as follows:

$$\Gamma[\nu n \leftrightarrows !m] = \beta', \gamma', \delta' \text{ with } \begin{cases} \beta' \stackrel{\text{def}}{=} (\ \beta \setminus \{x \mapsto \nu n \mid x \in \mathsf{dom}(\beta) \wedge \beta(x) = \nu n\}) \\ \qquad\qquad \cup \{x \mapsto !m \mid x \in \mathsf{dom}(\beta) \wedge \beta(x) = \nu n\} \\ \gamma' \stackrel{\text{def}}{=} (\gamma \setminus \{\{\nu n\}\}) \cup \{\{!m\}\} \\ \delta' \stackrel{\text{def}}{=} \ \{C_1 \leftrightarrow_\gamma C_2 \mid C_1 \leftrightarrow_\gamma C_2 \in \delta \wedge \nu n \notin C_1 \cup C_2\} \\ \qquad\qquad \cup \{\{!m\} \leftrightarrow_{\gamma'} C \mid \exists z \in C,\ z \in \mathcal{F} \setminus \{!m\}\} \end{cases}$$

The exchange operator updates each of the three elements of the name context. In the name environment, each association to the private fresh name νn is replaced by an association to the new fresh output name $!m$. In the dynamic partition, the class that contains νn, which is a singleton one, is replaced by a new one that also contains only $!m$.

All the distinctions between classes that do not contain νn are maintained. Moreover, because $!m$ is a new fresh output name, it must be different from any other fresh output names in the name environment. This is represented by adding distinctions between the class of $!m$ and the classes of other fresh names.

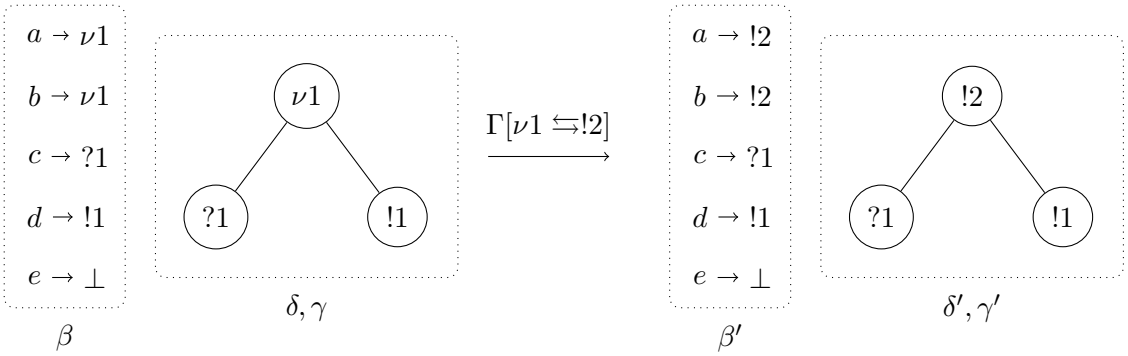


Figure 4.6: Exchanging private fresh name $\nu 1$ with new fresh output name !2

For example, Figure 4.6 illustrates the exchanging fresh private name $\nu 1$ with a fresh output name !2. Before the exchange, the fresh private name $\nu 1$ is alone in a class and it is assigned to both a and b. After the exchange, the new fresh output name !2 is assigned to both a and b. Moreover, all the distinctions between the class that contains $\nu 1$ and classes of other fresh names do not change, meaning that the relation between !2 and other fresh names is the same as that of $\nu 1$ with the others.

This operator may be useful when a private name is sent out at the first time, which will be called a bound output in Section 4.2.4.2. Suppose that initially, a restricted name a is associated to a fresh private name νk. This private name is different from all the others, it is alone in a class that has distinctions with all the other classes in the dynamic partition. When a is sent out, its instantiation is replaced by a new output fresh name, which is generated by the output clock, and is still different from the other fresh names in the current context. In the example in Figure 4.6, the operator $\Gamma[\nu 1 \leftrightarrows !2]$ may be used for the performance of action $\overline{c}\langle a \rangle$.

4.2.3.4 Refining dynamic partition

Refining dynamic partition allows us to make two given names become equal or unequal in the name context.

Refining with an equality We define the refining with an equality $z = z'$. We assume that $z, z' \in \mathsf{ran}(\beta) \setminus \{\bot\}$, $[z]_\gamma \neq [z']_\gamma$ and $[z]_\gamma \leftrightarrow [z']_\gamma \notin \delta$ to ensure these two names are not distinct before refining because two distinct names can not become equal. The

operator of refining with an equality is defined as follows:

$$\Gamma\lhd_{z=z'} = \beta', \gamma', \delta' \text{ with } \left[\begin{array}{l} \beta' \stackrel{\text{def}}{=} \beta \\ \gamma' \stackrel{\text{def}}{=} (\gamma \setminus \{[z]_\gamma, [z']_\gamma\}) \cup \{[z]_\gamma \cup [z']_\gamma\} \\ \delta' \stackrel{\text{def}}{=} \delta \setminus \{C_1 \leftrightarrow_\gamma C_2 \in \delta \mid C_1 = [z]_\gamma \vee C_1 = [z']_\gamma\} \\ \qquad \cup \{C \leftrightarrow_\gamma ([z]_\gamma \cup [z']_\gamma) \mid (C \leftrightarrow_\gamma [z]_\gamma \in \delta \wedge [z']_\gamma \notin C) \vee \\ \qquad (C \leftrightarrow_\gamma [z']_\gamma \in \delta \wedge [z]_\gamma \notin C)\} \end{array}\right.$$

This operator updates the dynamic partition and the distinctions of the name context, but leaves the name environment unchanged. First, two classes that contain the instances z and z' are replaced by a new class containing all their members. Second, the relations between classes are maintained by replacing each distinction between a class to one of two removed classes by a distinction to the new class. This operator is used, for example, to evaluate a *match* in a guarded action.

Figure 4.7 illustrates the changing of the name context after refining with an equality. Before refining, the two names !1 and ?2 are in two different classes and both are different from the fresh private name $\nu 1$. After the refining, these two names are grouped in the same class and this class has a distinction with the class of $\nu 1$. This distinction maintains the relation between these two names to the name $\nu 1$.

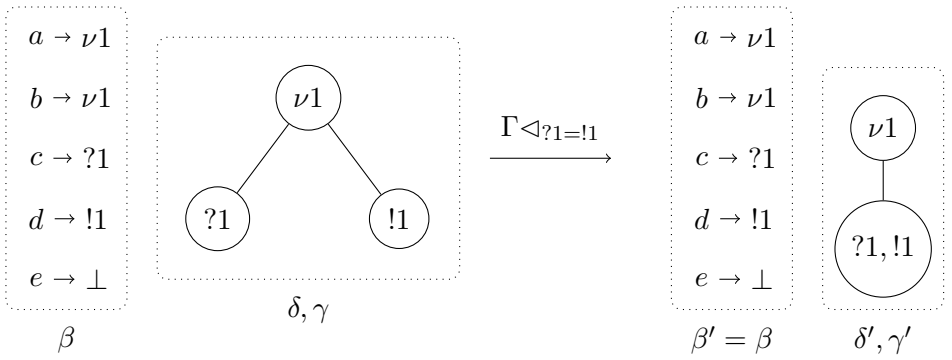


Figure 4.7: Refining with an equality $\Gamma\lhd_{?1=!1}$.

Refining with an inequality We define the refining of a dynamic partition with an inequality $z \neq z'$, assuming that $z, z' \in \mathsf{ran}(\beta) \setminus \{\bot\}$ and $[z]_\gamma \leftrightarrow_\gamma [z']_\gamma \notin \delta$, as follows:

$$\Gamma\lhd_{z\neq z'} = \beta', \gamma', \delta' \text{ with } \left[\begin{array}{l} \beta' \stackrel{\text{def}}{=} \beta \\ \gamma' \stackrel{\text{def}}{=} \gamma \\ \delta' \stackrel{\text{def}}{=} \delta \cup \{[z]_{\gamma'} \leftrightarrow_{\gamma'} [z']_{\gamma'}\} \end{array}\right.$$

Like for equality, this operator does not change the name environment and the dynamic partition, it only adds a distinction between two classes that contain the two unequal names. We use this operator to evaluate a *mismatch* in a guarded action.

Figure 4.8 illustrates the change of the distinctions after a refining with an inequality ?1 $\neq$!1. These names are in two different classes, so they may be equal or unequal. After the refining, a distinction between two corresponding classes is added to indicate that these names are unequal.

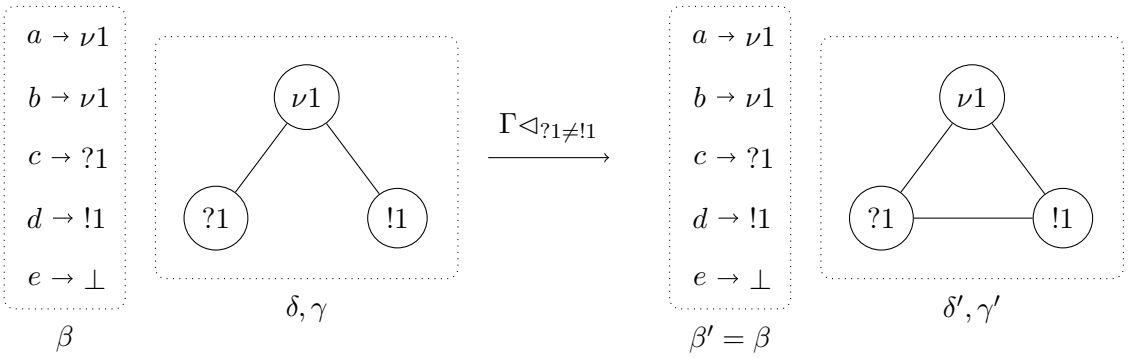


Figure 4.8: Refining with an inequality $\Gamma \lhd_{?1 \neq !1}$.

4.2.3.5 Initial name context

The initial name context is the name context in which no action has been activated and no thread has been used. Some syntactic names have to be initialized, but the names for which the initialization is not necessary are left uninitialized. For example,

- global restricted names and free names may be potentially used by everyone so they must be available from the beginning;
- thread restricted names and variable names depend on threads, so they will be initialized when a thread is spawned.

Hence, in the initial name context, all global restricted names are instantiated by fresh private names and all free names are instantiated by themselves. Each fresh name owns a unique class because we do not know if they are equal. Each class of a fresh private name, which is different from the others, has a distinction to all the other classes excluding itself. Formally, the initial name environment is defined as follows:

$$\begin{aligned} \beta_0 \stackrel{\text{def}}{=} \; & \{x \mapsto \nu n \mid x \in \mathsf{dom}(\overrightarrow{\mathcal{N}_{\mathsf{gres}}}) \wedge n = \overrightarrow{\mathcal{N}_{\mathsf{gres}}}(x)\} \\ & \cup \{x \mapsto x \mid x \in \mathcal{N}_{\mathsf{free}}\} \end{aligned}$$

In this definition, we consider all *global restricted names* $\mathcal{N}_{\mathsf{gres}}$ as a vector $\overrightarrow{\mathcal{N}_{\mathsf{gres}}}$ instead of a set as in Def. 4.1. If we considered $\mathcal{N}_{\mathsf{gres}}$ as a set of global restricted names as in Def. 4.1, then there would be many choices for mapping an instance to a syntactic name, depending on the order of generating of the fresh names. For example, suppose that x and y are global restricted names. If x is instantiated first, then $x \to \nu 1$, and then y is

instantiated, $y \to \nu 2$. But if the order is reversed, then $y \to \nu 1$ and $x \to \nu 2$. Thus, to make the instantiation deterministic, we consider $\mathcal{N}_{\mathsf{gres}}$ as a vector $\overrightarrow{\mathcal{N}_{\mathsf{gres}}} \in \Sigma \to \mathbb{N}$, and each syntactic name x with index n is instantiated to a private name νn. This technique is also applied for the *thread restricted names* $\mathcal{N}_{\mathsf{tres}}$.

The initial dynamic partition is:

$$\gamma_0 \stackrel{\text{def}}{=} \{\{x\} \mid x \in \mathsf{ran}(\beta_0) \setminus \{\bot\}\}$$

and the initial distinctions are:

$$\begin{aligned} \delta_0 \stackrel{\text{def}}{=} \; & \{\{\nu n\} \leftrightarrow_{\gamma_0} \{\nu m\} \mid \nu n, \nu m \in \mathsf{ran}(\beta_0) \wedge n \neq m\} \\ & \cup \{\{x\} \leftrightarrow_{\gamma_0} \{\nu n\} \mid x \in \mathcal{N}_{\mathsf{free}} \wedge \nu n \in \mathsf{ran}(\beta_0)\} \end{aligned}$$

It may be observed that, given a fresh private name z in the name context $\Gamma = (\beta, \delta, \gamma)$, z must always be different from the other names during the evolution of Γ. It is alone in a class and this class has a distinction with each other class. This invariant is defined as follows:

Invariant 4.2. *Let β, γ, δ be a name context. Then we have :*

$$\begin{aligned} \forall z \in \mathsf{ran}(\beta),\ \mathsf{priv}(z) \implies & [z]_\gamma = \{z\} \\ & \wedge\ \forall C \in \gamma \setminus \{z\},\ \{z\} \leftrightarrow_\gamma C \in \delta \end{aligned}$$

4.2.4 The evolution of a global context

The condition for activating an action and the evolution of the context depend on the types of actions. For example, an input action $\phi c(x)$ at vertex v can be activated if there exist a thread i that can be moved to v, the guard ϕ has to be true in the name context Γ. After activating, the thread i is moved to vertex v, the name context is updated to Γ' so that the guard ϕ is really true in Γ' and x is the new input value.

Similarly, if the label of v is an output action $\overline{c}\langle a \rangle$ in which the instantiation of a is a fresh private name, then after sending, the instantiation of a is *escaped*. That means the instantiation of a becomes a free name and a may be matched to other names later on.

4.2.4.1 Evaluating guards

The first condition for activating an action is that the guard of the action must be true in the current name context. Given a name context $\Gamma = (\beta, \gamma, \delta)$ and a guard ϕ, we have to check if ϕ may be made true in Γ. Then, if it is true, we update the context related to the guard (*i.e.*, update the dynamic partition and distinctions such as, for example, put two instantiations in the same class or add a distinction between two classes). This is defined by two functions check and eval respectively.

First of all, all the syntactic names in the guard ϕ are substituted by their instantiations belonging to a given thread i of a replicator r, denoted by $\beta_i^r(\phi)$, as follows:

$$\left[\begin{array}{l} \beta_i^r(\texttt{true}) \stackrel{\text{def}}{=} \texttt{true} \\ \beta_i^r(m\phi) \stackrel{\text{def}}{=} \beta_i^r(m)\beta_i^r(\phi) \\ \beta_i^r([a=b]) \stackrel{\text{def}}{=} [\beta_i^r(a) = \beta_i^r(b)] \\ \beta_i^r([a\neq b]) \stackrel{\text{def}}{=} [\beta_i^r(a) \neq \beta_i^r(b)] \end{array}\right.$$

Function check checks if a given guard ϕ may be made true in the name context Γ. It is denoted by $\mathsf{check}_{\beta,\gamma,\delta}(\phi)$ and defined as follows:

$$\left[\begin{array}{l} \mathsf{check}_{\beta,\gamma,\delta}(\texttt{true}) \text{ is true} \\ \mathsf{check}_{\beta,\gamma,\delta}(m\phi) \text{ iff } \mathsf{check}_{\beta,\gamma,\delta}(m) \wedge \mathsf{check}_{\beta',\gamma',\delta'}(\phi) \\ \qquad\qquad \text{where } \beta',\gamma',\delta' = \mathsf{eval}_m(\beta,\gamma,\delta) \\ \mathsf{check}_{\beta,\gamma,\delta}([a=b]) \text{ iff } [a]_\gamma = [b]_\gamma \vee [a]_\gamma \leftrightarrow_\gamma [b]_\gamma \notin \delta \\ \mathsf{check}_{\beta,\gamma,\delta}([a\neq b]) \text{ iff } [a]_\gamma \neq [b]_\gamma \end{array}\right.$$

We check from left to right if each name comparison may be true and the final result is a conjunction of these results. If a comparison is a match, then it may be true if the classes of two names in the match have no distinction; otherwise it is false. A mismatch may be true if the two names are not in the same class. If a name comparison may be true then it is evaluated, changing the name context so that the comparison becomes effectively true.

The evaluation of a guard ϕ in the context Γ is denoted by function $\mathsf{eval}_\Gamma(\phi)$. It may produce a new dynamic partition and new distinctions. The definition is as follows:

$$\left[\begin{array}{l} \mathsf{eval}_{\texttt{true}}(\beta,\gamma,\delta) \stackrel{\text{def}}{=} \beta,\gamma,\delta \\ \mathsf{eval}_{m\phi}(\beta,\gamma,\delta) \stackrel{\text{def}}{=} \mathsf{eval}_\phi(\beta',\gamma',\delta') \text{ where } \beta',\gamma',\delta' = \mathsf{eval}_m(\beta,\gamma,\delta) \\ \mathsf{eval}_{[a=b]}(\beta,\gamma,\delta) \stackrel{\text{def}}{=} \left\{\begin{array}{l} \beta,\gamma,\delta \text{ if } [a]_\gamma = [b]_\gamma \\ (\beta,\gamma,\delta) \lhd_{a=b} \text{ otherwise} \end{array}\right. \\ \mathsf{eval}_{[a\neq b]}(\beta,\gamma,\delta) \stackrel{\text{def}}{=} \left\{\begin{array}{l} \beta,\gamma,\delta \text{ if } [a]_\gamma \leftrightarrow_\gamma [b]_\gamma \in \delta \\ (\beta,\gamma,\delta) \lhd_{a\neq b} \text{ otherwise} \end{array}\right. \end{array}\right.$$

As above, the list of name comparisons is evaluated from left to right. For a match, the two names are grouped into the same class by applying the operator of refining with equality. For a mismatch, a distinction between two classes is added by applying the operator of refining with inequality.

Graphically, Figure 4.9 illustrates the evaluation of the guard $[c \neq a][c = d]$ in a name context $\Gamma = (\beta,\gamma,\delta)$. First of all, all syntactic names are substituted in the guard to $[?1 \neq \nu 1][?1 =!1]$. Next, mismatch $[?1 \neq \nu 1]$ is checked in the current name context (β,γ,δ). Because the instantiations $?1$ and $\nu 1$ are not in the same class, $\mathsf{check}_{\beta,\gamma,\delta}([?1 \neq \nu 1]) = \texttt{true}$, and we can update the context by calling function $\mathsf{eval}_{[?1\neq\nu 1]}(\beta,\gamma,\delta)$. Because both

instantiations are already in different classes, which are connected by distinctions, the name context is not changed, and we have $\beta' = \beta$.

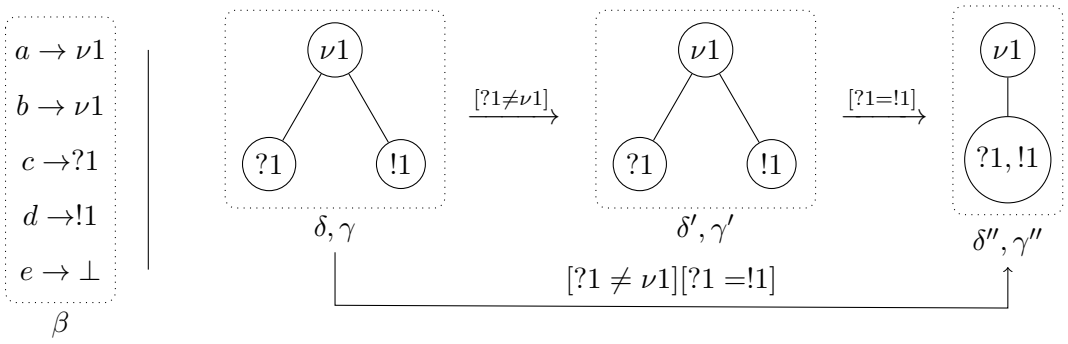


Figure 4.9: Evaluation of the guard $[c \neq a][c = d]$

Next, match [?1 =!1] is checked. Similarly, because both ?1 and !1 are in two classes which have no distinction, $\mathsf{check}_{\beta',\gamma',\delta'}([?1 =!1]) = \texttt{true}$ and we can evaluate this match by calling function $\mathsf{eval}_{[?1=!1]}(\beta', \gamma', \delta')$. This evaluation refines the dynamic partition and distinctions with equality [?1 =!1], it updates the name context to $(\beta'', \gamma'', \delta'')$ with $\beta'' = \beta$.

In summary, the evaluation of guard $[c \neq a][c = d]$ does not change the name environment, it combines two classes that contain the instances of c and d and maintains the distinction between the combined class and the class of the instance of a.

4.2.4.2 Commitment of actions

Let $\Gamma = (\beta, \delta, \gamma)$ be a name context of a diagram π, r a replicator in π, and v a vertex in r with an action $\phi\alpha$ having control with thread i. The commitment of a silent, an input and an output action, and the communication between an input and an output action are defined in the following.

Function check is used to check if a guard of the action can be made true in a name context. If true, the action is activated and the context is updated by function eval. The checking of guards is performed before activating the action and the evaluation of guards is performed afterwards. These two operations can be specified by two functions pre and post as follows:

- $\mathsf{pre}_\phi(\beta, \gamma, \delta, r, i) \stackrel{\text{def}}{=} \mathsf{check}_{\beta,\gamma,\delta}(\beta_i^r(\phi))$
- $\mathsf{post}_\phi(\beta, \gamma, \delta, r, i) \stackrel{\text{def}}{=} \mathsf{eval}_{\beta_i^r(\phi)}(\beta, \gamma, \delta)$

Silent step The silent action $\phi\tau$ at vertex v can be activated if the guard ϕ may be true in the name context Γ. After activating, the name context is updated by evaluating the guard ϕ. The commitment is defined as follows:

$$[\text{tau}] \quad \Gamma \vdash r : \overset{\phi\tau}{\textcircled{i}} \overset{\tau}{\Rightarrow} \mathsf{post}_{\text{tau}}(\Gamma, r, i, \phi) \qquad \text{if } \mathsf{pre}_{\text{tau}}(\Gamma, r, i, \phi)$$

$$\text{with} \quad \mathsf{pre}_{\text{tau}}(\Gamma, r, i, \phi) = \mathsf{pre}_{\phi}(\Gamma, r, i)$$
$$\mathsf{post}_{\text{tau}}(\Gamma, r, i, \phi) = \mathsf{post}_{\phi}(\Gamma, r, i)$$

Input An input action $\phi c(x)$ at vertex v can be activated if the guard ϕ can be made true in the name context Γ and the channel c is public after the evaluation of ϕ. After the activation, a new fresh input name is assigned to name x. The commitment is defined as follows:

$$[\text{in}] \quad \Gamma \vdash r : \overset{\phi c(x)}{\textcircled{i}} \overset{c'(x')}{\Longrightarrow} \mathsf{post}_{\text{in}}(\Gamma, r, i, \phi c(x)) \qquad \text{if } \mathsf{pre}_{\text{in}}(\Gamma, r, i, \phi c(x))$$

$$\text{with} \quad \mathsf{pre}_{\text{in}}(\beta, \gamma, \delta, r, i, \phi c(x)) = \begin{cases} \mathsf{pre}_{\phi}(\beta, \gamma, \delta, r, i) \\ \wedge\ \beta', \gamma', \delta' = \mathsf{post}_{\phi}(\beta, \gamma, \delta, r, i) \\ \wedge\ \mathsf{pub}(\beta_i'^{r}(c)) \end{cases}$$
$$\mathsf{post}_{\text{in}}(\Gamma, r, i, \phi c(x)) = \Gamma'[x \leftarrow \mathsf{clk}_?(\beta')] \quad \text{where } \Gamma' = \mathsf{post}_{\phi}(\Gamma, r, i)$$
$$c' = \beta_i'^{r}(c), x' = \mathsf{clk}_?(\beta')$$

The new fresh input name is generated by the input logical clock based on the name context after the evaluation of the guard ϕ.

Output An output action $\phi\overline{c}\langle a\rangle$ can be activated at vertex v if the guard ϕ can be made true in the name context Γ and both c and a are public after the evaluation of ϕ. After the activation, the name context is the evaluated one. The commitment is defined as follows:

$$[\text{out}] \quad \Gamma \vdash r : \overset{\phi\overline{c}\langle a\rangle}{\textcircled{i}} \overset{\overline{c'}\langle a'\rangle}{\Longrightarrow} \mathsf{post}_{\text{out}}(\Gamma, r, i, \phi) \qquad \text{if } \mathsf{pre}_{\text{out}}(\Gamma, r, i, \phi\overline{c}\langle a\rangle)$$

$$\text{with} \quad \mathsf{pre}_{\text{out}}(\beta, \gamma, \delta, r, i, \phi\overline{c}\langle a\rangle) = \begin{cases} \mathsf{pre}_{\phi}(\beta, \gamma, \delta, r, i) \\ \wedge\ \beta', \gamma', \delta' = \mathsf{post}_{\phi}(\beta, \gamma, \delta, r, i) \\ \wedge\ \mathsf{pub}(\beta_i'^{r}(c)) \wedge\ \mathsf{pub}(\beta_i'^{r}(a)) \end{cases}$$
$$\mathsf{post}_{\text{out}}(\Gamma, r, i, \phi) = \mathsf{post}_{\phi}(\Gamma, r, i)$$
$$c' = \beta_i'^{r}(c), a' = \beta_i'^{r}(a)$$

Bound output A bound output action $\phi\overline{c}\langle a\rangle$ sends out a private name. The commitment is defined as follows:

$$[\nu\text{out}] \quad \Gamma \vdash r : \overset{\phi\overline{c}\langle a\rangle}{(i)} \xRightarrow{\overline{c'}\langle a'\rangle} \mathsf{post}_{\nu\text{out}}(\Gamma, r, i, \phi\overline{c}\langle a\rangle) \qquad \text{if } \mathsf{pre}_{\nu\text{out}}(\Gamma, r, i, \phi\overline{c}\langle a\rangle)$$

$$\text{with} \quad \mathsf{pre}_{\nu\text{out}}(\beta, \gamma, \delta, r, i, \phi\overline{c}\langle a\rangle) = \begin{cases} \mathsf{pre}_{\phi}(\beta, \gamma, \delta, r, i) \\ \wedge\ \beta', \gamma', \delta' = \mathsf{post}_{\phi}(\beta, \gamma, \delta, r, i) \\ \wedge\ \mathsf{pub}(\beta_i'^{r}(c)) \wedge\ \mathsf{priv}(\beta_i'^{r}(a)) \end{cases}$$

$$\mathsf{post}_{\nu\text{out}}(\Gamma, r, i, \phi\overline{c}\langle a\rangle) = \ \Gamma'[\beta_i^r(a) \leftrightarrows \mathsf{clk}_!(\beta')], \text{ where } \Gamma' = \mathsf{post}_{\phi}(\Gamma, r, i)$$

$$c' = \beta_i'^{r}(c), a' = \nu\mathsf{clk}_!(\beta')$$

Similarly to the output action, it can be activated if the guard can be made true in the name context and the channel is public after the evaluation of the guard. After the activation, the output name is instantiated with a fresh output name.

After sending out a fresh private name, it is no longer private. It is replaced by a new fresh output name, which is generated by the output clock based on the evaluated name context. Consider an example of the bound output action as in Figure 4.10.

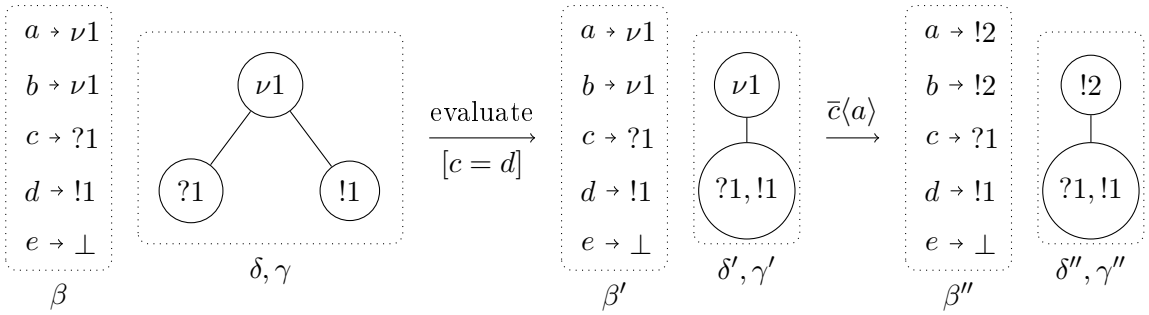


Figure 4.10: A bound output action $[c = d]\overline{c}\langle a\rangle$

First, the guard $[c = d]$ is checked if it can be made true in the name context Γ. Because ?1 and !1, the instantiations of c and d, respectively, are in two distinct classes that are not connected by a distinction, the guard is true and the output action $\overline{c}\langle a\rangle$ may be activated. After the evaluation of the guard, Γ becomes Γ', in which the instantiation $\nu 1$ of the output name a is a fresh private name and the instantiation ?1 of the channel c is public. After the activation of the output action, the private fresh name is exchanged by a new output fresh name !2.

Proposition 4.2. *Let Γ be a name context, r a replicator, and i a thread in r. For any output action $\phi\overline{c}\langle a\rangle$, we always have:*

$$\neg(\mathsf{pre}_{\text{out}}(\Gamma, r, i, \phi\overline{c}\langle a\rangle) \wedge\ \mathsf{pre}_{\nu\text{out}}(\Gamma, r, i, \phi\overline{c}\langle a\rangle)) = \mathtt{true}$$

Proof. Let $P = \mathsf{pre}_{\text{out}}(\Gamma, r, i, \phi\overline{c}\langle a\rangle) \ \wedge\ \mathsf{pre}_{\nu\text{out}}(\Gamma, r, i, \phi\overline{c}\langle a\rangle)$. By the definition of the precondition of an output $\mathsf{pre}_{\text{out}}(\dots)$ (cf. p. 73), and of a bound-output $\mathsf{pre}_{\nu\text{out}}(\dots)$ (cf.

p. 73), we have:

$$P = \begin{cases} \mathsf{pre}_\phi(\beta, \gamma, \delta, r, i) \\ \wedge\ \beta', \gamma', \delta' = \mathsf{post}_\phi(\beta, \gamma, \delta, r, i) \\ \wedge\ \mathsf{pub}(\beta_i'^r(c)) \ \wedge\ \mathsf{priv}(\beta_i'^r(a)) \ \wedge\ \mathsf{pub}(\beta_i'^r(a)). \end{cases}$$

Moreover, by the definition of function $\mathsf{pub}(a)$ and $\mathsf{priv}(a)$ (cf. p. 65), we always have: $\mathsf{priv}(\beta_i'^r(a)) \wedge\ \mathsf{pub}(\beta_i'^r(a)) = \mathsf{false}$. Thus P always being false, the property holds. □

Communication If there is a pair of input and output actions that send and receive information on the same channel then they can potentially communicate together.

Let $\phi_1 \overline{c}\langle a \rangle$ be an output action in replicator r_o at a vertex v_o, which has control with thread i_o, and let $\phi_2 d(x)$ be an input action in replicator r_i at a vertex v_i, which has control with thread i_i. The communication between these actions can be activated if both the guards ϕ_1 and ϕ_2 can be made true in the current name context and the two channels are compatible. The last meaning that the instances of c and d are not in two different classes of γ connected with a distinction, *i.e.*, $[c]_\gamma \neq [d]_\gamma$ and $([c]_\gamma, [d]_\gamma) \notin \delta$. After the activation, two channels are asserted equal and the instantiation of the output name is assigned to the input name. The commitment is defined as follows:

$$\text{[com]} \quad \Gamma \vdash r_\mathrm{o} : \overset{\phi_1 \overline{c}\langle a \rangle}{\bigcirc\!\!\!\!\bigcirc\ i_\mathrm{o}} \quad r_\mathrm{i} : \overset{\phi_2 d(x)}{\bigcirc\!\!\!\!\bigcirc\ i_\mathrm{i}} \quad \overset{\tau}{\Rightarrow} \mathsf{post}_\mathrm{com}(\Gamma, r_\mathrm{o}, i_\mathrm{o}, \phi_1 \overline{c}\langle a \rangle, r_\mathrm{i}, i_\mathrm{i}, \phi_2 d(x))$$

$$\text{if } \mathsf{pre}_\mathrm{com}(\Gamma, r_\mathrm{o}, i_\mathrm{o}, \phi_1 \overline{c}\langle a \rangle, r_\mathrm{i}, i_\mathrm{i}, \phi_2 d(x))$$

$$\text{with } \mathsf{pre}_\mathrm{com}(\beta, \gamma, \delta, r_\mathrm{o}, i_\mathrm{o}, \phi_1 \overline{c}\langle a \rangle, r_\mathrm{i}, i_\mathrm{i}, \phi_2 d(x)) = \begin{cases} \wedge\ \mathsf{pre}_{\phi_1}(\beta, \gamma, \delta, r_\mathrm{o}, i_\mathrm{o}) \wedge\ \beta', \gamma', \delta' = \mathsf{post}_{\phi_1}(\beta, \gamma, \delta, r_\mathrm{o}, i_\mathrm{o}) \\ \wedge\ \mathsf{pre}_{\phi_2}(\beta', \gamma', \delta', r_\mathrm{i}, i_\mathrm{i}) \wedge\ \beta'', \gamma'', \delta'' = \mathsf{post}_{\phi_2}(\beta', \gamma', \delta', r_\mathrm{i}, i_\mathrm{i}) \\ \wedge\ c' = \beta''^{r_\mathrm{o}}_{i_\mathrm{o}}(c) \wedge d' = \beta''^{r_\mathrm{i}}_{i_\mathrm{i}}(d) \wedge a' = \beta''^{r_\mathrm{o}}_{i_\mathrm{o}}(a) \\ \wedge\ [c']_{\gamma''} \leftrightarrow_{\gamma''} [d']_{\gamma''} \notin \delta'' \end{cases}$$

$$\mathsf{post}_\mathrm{com}(\Gamma, r_\mathrm{o}, i_\mathrm{o}, \phi_1 \overline{c}\langle a \rangle, r_\mathrm{i}, i_\mathrm{i}, \phi_2 d(x)) = \Gamma'''^{r_\mathrm{i}}_{i_\mathrm{i}}[x \leftarrow a']$$

$$\text{with } \Gamma''' = \Gamma'' \lhd_{c'=d'}$$

$$\text{and } \Gamma'' = \mathsf{post}_{\phi_2}(\Gamma', r_\mathrm{i}, i_\mathrm{i})$$

$$\text{and } \Gamma' = \mathsf{post}_{\phi_1}(\Gamma, r_\mathrm{o}, i_\mathrm{o})$$

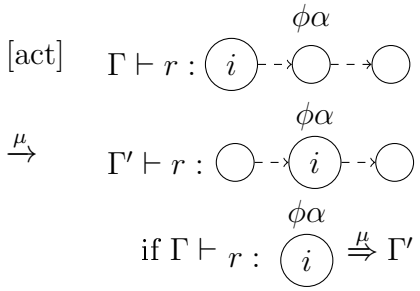
4.2.4.3 Graph rewrite rules

Graph rewrite rules describe the evolution of threads in the π-graph.

Atomic action This is a simplest case in which the considered action has both predecessor and successor, and the predecessor has control for at least one thread.

Let $\phi\alpha$ be an action at vertex v in replicator r, i a thread for which the predecessor of

v has control and Γ the current name context. The atomic action rule, denoted by [act], is defined as follows:



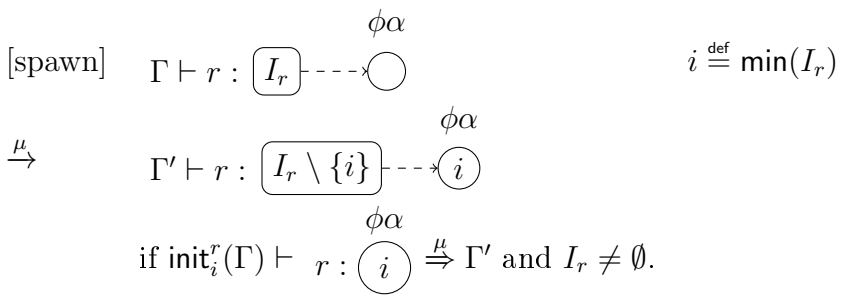
If the action $\phi\alpha$ can be activated on thread i and its performance updates name context Γ to Γ' then thread i will be moved to v. The thread will be available for performing the action at a successor of v.

Spawning a thread To activate an action that has no predecessor, the total number of used threads must be less than the capacity of the replicator. In such a case, a thread may be spawned activating the action.

When a new thread i of replicator r is spawned, all the names that depend on this thread must be instantiated. The thread-restricted names are initialized to fresh values. The variable names are associated to $\perp$ because they are undefined at this stage. Formally, the initialization of name context Γ for a thread i in replicator r is defined as follows:

$$\mathsf{init}_i^r(\Gamma) \stackrel{\mathsf{def}}{=} \{x_i^r \mapsto \perp \mid x \in \mathcal{N}_{\mathsf{var}}^r\} \cup \mathsf{new}(\overrightarrow{\mathcal{N}_{\mathsf{tres}}^r}, r, i, \Gamma)$$
$$\text{with } \mathsf{new}(\langle x_1, \ldots, x_{n-1}, x_n\rangle, r, i, \Gamma) \stackrel{\mathsf{def}}{=} \mathsf{new}(x_n, r, i, \mathsf{new}(x_{n-1}, r, i, \ldots \mathsf{new}(x_1, r, i, \Gamma)\ldots))$$
$$\text{and } \mathsf{new}(x, r, i, \Gamma) \stackrel{\mathsf{def}}{=} \Gamma_i^r[x \leftarrow \mathsf{clk}_\nu(\beta)]$$

Let $\phi\alpha$ be an action at vertex v in replicator r. We assume that action $\phi\alpha$ is not a terminal one and the vertex v has no predecessor. The rule of spawning a thread, denoted by [spawn], is defined as follows:

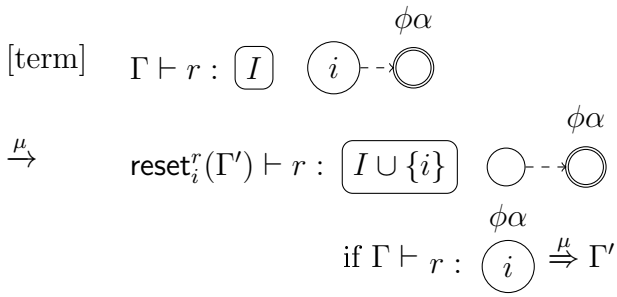


Suppose that Γ is the current name context and i is a spawned thread. The spawning of the thread updates the name context Γ to $\mathsf{init}_i^r(\Gamma)$. If the performance of the action $\phi\alpha$ updates the initialized context to Γ', then the thread can be used by the action. Because the action is not terminal, the thread is available for activating the successor action.

Terminating a thread After performing a terminal action, the thread that is used for this performance is terminated and it is available for next session. The termination of a thread i in replicator r requires an update of the name context. All the private names and the variables attached to this thread must be reset, their instantiations have to disappear from the name context. We define a function reset for this purpose as follows:

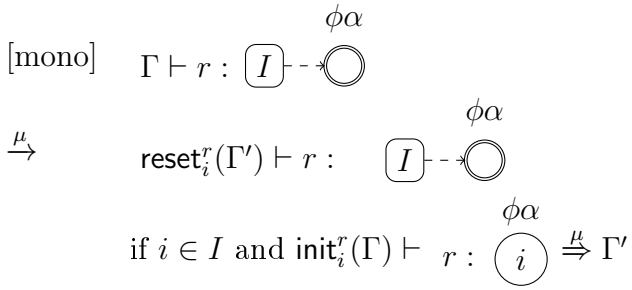
$$\mathsf{reset}_i^r(\Gamma) \stackrel{\mathsf{def}}{=} \Gamma_i^r - (\mathsf{dom}(\overrightarrow{\mathcal{N}_{\mathsf{tres}}^r}) \cup \mathcal{N}_{\mathsf{var}}^r)$$

Let $\phi\alpha$ be an action at vertex v of replicator r. We assume that action $\phi\alpha$ is a terminal one and the predecessor of v has control for a thread i. The rule of terminating a thread, denoted as [term] is defined as follows:



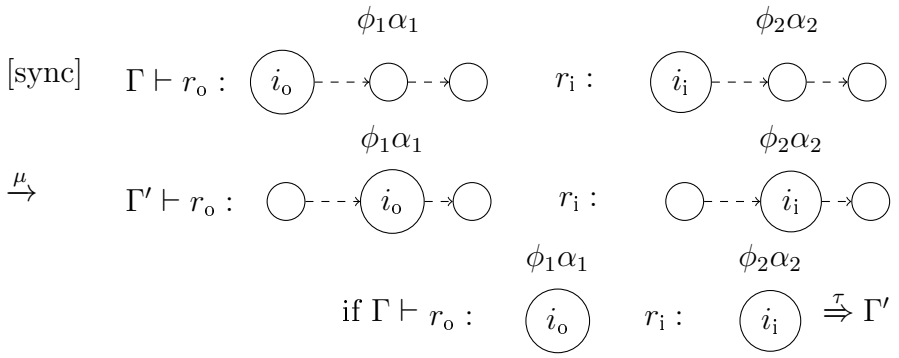
If the performing of action $\phi\alpha$ on thread i updates the name context Γ to Γ' then after performing this action, Γ will be reset and the thread i will be available in the pool of threads.

Special case: monadic action Monadic action is a terminal one and it has no predecessor. Its activating requires a thread i is spawned and after performing this action, the name context is reset. Let $\phi\alpha$ be a monadic action in replicator r. The rewrite rule for this action, denoted by [mono], is defined as follows:



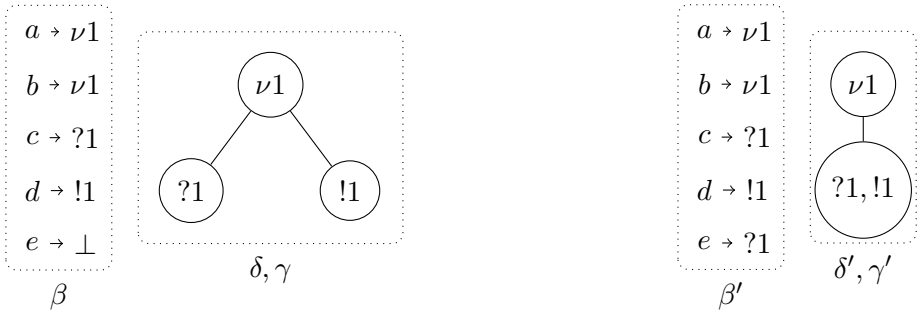
If the activating of action $\phi\alpha$ with thread i updates the initialized name context to Γ', then after performing this action, the name context will be reset and thread i will also be available in the pool of threads.

Synchronization Let $\phi_1\alpha_1$ be an output action at vertex v_{out} in replicator r_{out}, $\phi_2\alpha_2$ an input action at vertex v_{in} in replicator r_{in}, Γ the name context before performing the synchronization. The predecessor of v_{out} and v_{in} have control for a thread i_{out} and i_{in} respectively. The rule of synchronization between two actions is defined as follows:



Remark In the communication rule, there may be a communication within a single replicator (*i.e.*, $r_{in} = r_{out}$), but in this case, it must be within two distinct threads (*i.e.*, $i_{in} \neq i_{out}$).

Consider an example of the synchronization between an output $[c = d]\overline{a}\langle c\rangle$ and an input $b(e)$ as in Figure 4.11. Before the communication (depicted in Figure 4.11a), the instantiations of c and d (*i.e.*, ?1 and !1, respectively) are in two classes which are not connected by a distinction, meaning that the guard $[c = d]$ is evaluated to true. Moreover, the instantiations of both a and b are $\nu 1$, thus the communication between these two actions may be activated. After the communication (depicted in Figure 4.11b), the dynamic partition is updated, in which the instantiations of c and d are in the same class, and the distinctions are also updated (remaining only one distinction). Moreover, name e (which is associated to $\perp$ before the communication) is associated to ?1 (the instantiation of c).



(a) Before the communication

(b) After the communication

Figure 4.11: The communication between $[c = d]\overline{a}\langle c\rangle$ and $b(e)$

Derived synchronization rules A synchronization is a coordination between an output and an input action. Each of them can be one of four kinds of action: atomic, spawning a thread, terminating a thread or monadic. So, there are in total 16 rules for synchronization, consisting of the rule [sync] and 15 derived rules that are combinations with the [spawn], [term] and [mono] rules.

The main issue is that there are initializations to perform before one or both synchronizing threads are spawned, and also resets to perform after one or both synchronized threads are terminated. The initializations on one side, and the resets on the other side trivially commute because they operate only on distinct pairs (replicator r, thread i). Compared to the [sync] rule above, the only notable difference for the other 15 rules is the computation of the resulting name context, called Γ' in the Table 4.1.

From the semantics rules above, one may observe that if two classes has a distinction then they must be different in the dynamic partition γ. Moreover, γ is a partition of all the instantiations of the name environment β. These properties are invariants in π-graphs formalism, they are defined as follows:

Invariant 4.3. $\forall C_1, C_2 \in \gamma.\ (C_1, C_2) \in \delta \rightarrow C_1 \neq C_2$.

Invariant 4.4. *γ is a partition of* $\mathsf{ran}(\beta)$*, i.e.,*

- $\emptyset \notin \gamma$
- $\bigcup_{C \in \gamma} C = \mathsf{ran}(\beta)$
- $\forall C_1, C_2 \in \gamma.\ C_1 \neq C_2 \rightarrow C_1 \cap C_2 = \emptyset$

Besides the classification of actions in π-graphs into input, output, bound output and silent, they can be classified based on their relation to the other actions in the π-graphs structure, called structural type of actions. In the following, we present the formal definition of structural types and several related properties.

Definition 4.7 (Structural type)**.** *Let $\pi = \langle \mathcal{R}, \mathcal{K}, \mathcal{G}, \mathcal{N} \rangle$ be a π-graph, $\mathcal{G}(r) = \langle V_r, L_r, E_r \rangle$ the static graph of a replicator r in π. The* structural type *of an action at vertex v in V_r, denoted by* $\mathsf{stype}(r, v)$*, is either:*

- `mono`*: if* $\mathsf{deg}_r^-(v) = 0$ *and* $\mathsf{deg}_r^+(v) = 0$*;*
- `spwn`*: if* $\mathsf{deg}_r^-(v) = 0$ *and* $\mathsf{deg}_r^+(v) > 0$*;*
- `term`*: if* $\mathsf{deg}_r^-(v) = 1$ *and* $\mathsf{deg}_r^+(v) = 0$*;*
- `norm`*: if* $\mathsf{deg}_r^-(v) = 1$ *and* $\mathsf{deg}_r^+(v) > 0$.

where $\mathsf{deg}_r^-(v)$ *(resp.* $\mathsf{deg}_r^+(v)$*) means the input (resp. output) degree of vertex v in the static graph $\mathcal{G}(r)$ of replicator r.*

Proposition 4.3 (Disjoint structural types)**.** *The four structural types of atomic actions are* disjoint.

Derived rule from a [sync] between two threads in context Γ $(r_\mathrm{o}, i_\mathrm{o})$ is output thread, $(r_\mathrm{i}, i_\mathrm{i})$ is input thread	**Initializations**	**Resets**
[**sync&spawn/out**] ([spawn] for output thread)	$\mathsf{init}_{i_\mathrm{o}}^{r_\mathrm{o}}(\Gamma)$	
[**sync&spawn/in**] ([spawn] for input thread)	$\mathsf{init}_{i_\mathrm{i}}^{r_\mathrm{i}}(\Gamma)$	
[**sync&spawn – spawn**] ([spawn] for both threads)	$\mathsf{init}_{i_\mathrm{i}}^{r_\mathrm{i}}(\mathsf{init}_{i_\mathrm{o}}^{r_\mathrm{o}}(\Gamma))$	
[**sync&term/out**] ([term] for output thread)		$\mathsf{reset}_{i_\mathrm{o}}^{r_\mathrm{o}}(\Gamma')$
[**sync&term/in**] ([term] for input thread)		$\mathsf{reset}_{i_\mathrm{i}}^{r_\mathrm{i}}(\Gamma')$
[**sync&term – term**] ([term] for both threads)		$\mathsf{reset}_{i_\mathrm{i}}^{r_\mathrm{i}}(\mathsf{reset}_{i_\mathrm{o}}^{r_\mathrm{o}}(\Gamma'))$
[**sync&spawn/out – term/in**] ([spawn] for output thread, [term] for input)	$\mathsf{init}_{i_\mathrm{o}}^{r_\mathrm{o}}(\Gamma)$	$\mathsf{reset}_{i_\mathrm{i}}^{r_\mathrm{i}}(\Gamma')$
[**sync&spawn/in – term/out**] ([spawn] for input thread, [term] for output)	$\mathsf{init}_{i_\mathrm{i}}^{r_\mathrm{i}}(\Gamma)$	$\mathsf{reset}_{i_\mathrm{o}}^{r_\mathrm{o}}(\Gamma')$
[**sync&mono/out**] ([mono] for output thread)	$\mathsf{init}_{i_\mathrm{o}}^{r_\mathrm{o}}(\Gamma)$	$\mathsf{reset}_{i_\mathrm{o}}^{r_\mathrm{o}}(\Gamma')$
[**sync&mono/in**] ([mono] for input thread)	$\mathsf{init}_{i_\mathrm{i}}^{r_\mathrm{i}}(\Gamma)$	$\mathsf{reset}_{i_\mathrm{i}}^{r_\mathrm{i}}(\Gamma')$
[**sync&mono – mono**] ([mono] for both threads)	$\mathsf{init}_{i_\mathrm{i}}^{r_\mathrm{i}}(\mathsf{init}_{i_\mathrm{o}}^{r_\mathrm{o}}(\Gamma))$	$\mathsf{reset}_{i_\mathrm{i}}^{r_\mathrm{i}}(\mathsf{reset}_{i_\mathrm{o}}^{r_\mathrm{o}}(\Gamma'))$
[**sync&mono/out – spawn/in**] ([mono] for output thread, [spawn] for input)	$\mathsf{init}_{i_\mathrm{i}}^{r_\mathrm{i}}(\mathsf{init}_{i_\mathrm{o}}^{r_\mathrm{o}}(\Gamma))$	$\mathsf{reset}_{i_\mathrm{o}}^{r_\mathrm{o}}(\Gamma')$
[**sync&mono/in – spawn/out**] ([mono] for input thread, [spawn] for output)	$\mathsf{init}_{i_\mathrm{i}}^{r_\mathrm{i}}(\mathsf{init}_{i_\mathrm{o}}^{r_\mathrm{o}}(\Gamma))$	$\mathsf{reset}_{i_\mathrm{i}}^{r_\mathrm{i}}(\Gamma')$
[**sync&mono/out – term/in**] ([mono] for output thread, [term] for input)	$\mathsf{init}_{i_\mathrm{o}}^{r_\mathrm{o}}(\Gamma)$	$\mathsf{reset}_{i_\mathrm{i}}^{r_\mathrm{i}}(\mathsf{reset}_{i_\mathrm{o}}^{r_\mathrm{o}}(\Gamma'))$
[**sync&mono/in – term/out**] ([mono] for input thread, [term] for output)	$\mathsf{init}_{i_\mathrm{i}}^{r_\mathrm{i}}(\Gamma)$	$\mathsf{reset}_{i_\mathrm{i}}^{r_\mathrm{i}}(\mathsf{reset}_{i_\mathrm{o}}^{r_\mathrm{o}}(\Gamma'))$

Table 4.1: Derived synchronization rules

Proof. Let $\pi = \langle \mathcal{R}, \mathcal{K}, \mathcal{G}, \mathcal{N} \rangle$ be a π-graphs, $\mathcal{G}(r) = \langle V_r, L_r, E_r \rangle$ the static graph of a replicator r in π. For any vertex v in V_r, its input degree $\mathsf{deg}_r^-(v)$ is either 0 or 1, and its output degree $\mathsf{deg}_r^+(v)$ is either 0 or more. Thus, there are four totally disjoint cases for a combination between $\mathsf{deg}_r^-(v)$ and $\mathsf{deg}_r^+(v)$, which is the structural type of the action at (r, v). So, the four structural types in Def. 4.7 are *disjoint.* □

Proposition 4.4. *Let π be a π-graph and an action at (r, v) such that its structural type is either* term *or* mono. *For any context $\mathcal{C}_\pi$ of π, we always have $\Delta_r(v) = \emptyset$.*

Proof. This property is derived directly from rule term and mono (cf. p. 76). □

Definition 4.8 (Global rewrite of π-graphs)**.** *Let $\mathcal{C}_\pi$ and $\mathcal{C}'_\pi$ be the contexts of a π-graphs π. The* global rewrite of π *with label μ, denoted by $\mathcal{C}_\pi \xrightarrow{\mu} \mathcal{C}'_\pi$, is triggered by the performance of either an* atomic action *or a* synchronization, *and the applied rule is:*

- *Single rewrite rule* [act], [spawn], [term], [mono] *if the structural type of the atomic action is* `norm`, `spwn`, `term` *or* `mono`, *respectively;*
- *Rule* [sync] *in case of the synchronization.*

Each rule matches a sub-component of π and updates only this component.

Finally, as mentioned above, the evolution of a π-graph is described by a labelled transition system whose states are contexts that the system may reach, and labelled transitions indicate under which conditions the change of states may occur. The formal definition of the labelled transition system of a π-graphs is presented as follows:

Definition 4.9. *Let Λ be a set of labels of transitions. A* labelled transition system *(LTS) of a π-graph π over Λ is a tuple $(\mathscr{C}, \mathcal{C}_\pi^0, \mathscr{R})$, where:*

- *$\mathscr{C}$ is a finite set of contexts $\mathcal{C}_\pi$,*
- *$\mathcal{C}_\pi^0$ is an initial context in $\mathscr{C}$,*
- *$\mathscr{R} \subseteq \mathscr{C} \times \Lambda \times \mathscr{C}$ is a transition relation.*

A transition t in $\mathscr{R}$ with label λ is often written as $\mathcal{C}_\pi \xrightarrow{\lambda} \mathcal{C}'_\pi$.

All these definitions and propositions will be used in the translation of π-graphs into Petri nets and in the proof of the conformance of the translation in Chapter 5.

4.3 Synthesis

This chapter presents the π-graphs formalism formally. It provides a formal syntax and operational semantics of π-graphs.

The syntax is presented in a top-down manner, which starts from a diagram and ends with guarded actions. A diagram models the whole system, which consists of a finite set of replicators modelling components of the system, and are supposed to work in parallel. Each replicator specifies a process in which threads may run in the form of a tree whose nodes are guarded actions and directed edges represent their precedence. A replicator also defines a thread bound, which is the maximum number of threads that can run simultaneously in it. Replicators make the encoding of π-graphs more compact because multiple threads can share a sub-graph.

Names used in π-graphs can be classified into free names, variable names, and restricted names. There are two kinds of restricted names: the global ones, which are at the diagram level, and local ones, which are at the replicator level. Actions can be classified into silent, input, output and bound output. The difference between output and bound output is that the bound output is the output of restricted names at the first time, after that the name is not longer restricted and becomes public.

The operational semantics of a π-graphs diagram is defined as a labelled transition system (LTS) in which the states are global contexts (configurations) that the system may reach and the labelled transitions indicate under which conditions the change of the states may occur. A global context is composed of a control flow and a name context. The former indicates which actions are active and which threads are available for each replicator. The later records the instantiations of names (through a name environment) and the relations between names, such as the equality (through a dynamic partition) and the inequality (through a set of distinctions). Moreover, a set of operators on a context, a set of commitment and rewrite rules and logical clocks are provided to describe the evolution of the global context. Commitment rules describe the evolution of names and rewrite rules describes the evolution of threads on the π-graphs structure. Logical clocks are used to generate fresh names. Especially, a garbage collector, which is implemented as function reset, is used to remove automatically all the occurrences of inactive names in the context each time a thread terminates. Both the clocks and the garbage collector make the LTS become finite.

Chapter 5

Translating a π-graph into Petri nets

Petri nets are a modelling formalism that was invented by Carl Adam Petri [42]. They are suitable for modelling concurrent and distributed systems, and they have many support tools that allow us to analyze these systems. In the previous chapter, we modelled open reconfigurable systems using π-graphs. As a variant of π-calculus formalism, π-graphs has an expressive power for modelling the communication and the dynamic changing of the structure of the systems. However, there are not many available support tools and verification techniques for π-calculus. Thus, we translate the model in π-graphs into Petri nets to use available tools and techniques for Petri nets to analyze the systems. Section 5.1 presents the translation from π-graphs into high-level Petri nets presented in Section 2.2. In Section 5.2 we prove that the translated Petri nets behave dynamically as the original π-graphs. In other words, the Petri nets translation of a π-graph has the conformance property.

5.1 Translation of π-graphs into Petri nets

The translation of a π-graph π in a context (Γ, Δ) into high-level Petri nets is structural, which means that we rely only on the structure of π and the information in (Γ, Δ) to construct the corresponding Petri net. The translation is obtained in two parts:

1. Obtaining the Petri net structure from the π-graph,

2. Obtaining the marking for the translated net structure from (Γ, Δ).

Before introducing the formal definition of the translation, these two parts will be illustrated by a simple example.

5.1.1 An example of the translation

Suppose that we have a π-graph shown in Figure 5.1 with an arbitrary but well defined context (Γ_1, Δ_1).

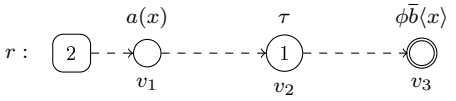


Figure 5.1: A simple π-graph with one replicator

The two parts of the translation are described in detail as follows:

5.1.1.1 Part I: Obtaining the Petri net structure

This part will produce a Petri net structure $S = (P, T, U, G)$ (it means an unmarked high-level Petri net) in 5 steps:

1. Creating the places,
2. Translating the atomic π-graphs actions into atomic transitions,
3. Translating the synchronizations into synchronization transitions,
4. Creating the name context place and connecting it to the transitions,
5. Computing the guards for all transitions.

1. Create the places For each replicator (here, we have only one replicator r) we create one initial place (here, p_r^0). For each inner vertex (here, v_1 and v_2) we create a place (here, p_1 and p_2, respectively). The terminal vertex v_3 does not give any place in the translation. These three places of the translated net structure are represented in Figure 5.2.

Figure 5.2: Places of the translated net structure S.

All places but initial one (p_r^0) may carry tokens which are pairs of constants of the form $r.i$ where r is a replicator and i is a thread identifier. Initial place p_r^0 is intended to carry the set of such tokens (actually, only one token of the form $X \subseteq \{r.i \mid i \in [1..\mathcal{K}(r)]\}$).

Step 2: Translate atomic actions Each atomic action of the π-graph gives rise to an atomic transition which is labelled by the action. The arcs of the obtained translation depend on the connections of the vertex in the π-graph.

For vertex v_2, the corresponding transition t_2, with label τ, is connected to p_1 and p_2, where p_1 is the input place and p_2 is the output one (as v_2 is preceded by v_1 and v_2 is not a terminal vertex in the π-graph). The arcs around t_2 are labelled $r.i$ as the action at v_2 is a silent action τ. If the action was an input (resp. output) then these arcs would be labelled $r_{\mathrm{i}}.i_{\mathrm{i}}$ (resp. $r_{\mathrm{o}}.i_{\mathrm{o}}$).

For vertex v_1, the corresponding transition t_1 with label $a(x)$ is connected to p_r^0 and p_1, where p_r^0 is the input place and p_1 is the output one (as v_1 is an initial vertex and it is not a terminal one in the π-graph). Because t_1 is connected to the initial place p_r^0, its connectivity is particular as shown in Figure 5.3, where X is a variable. It means that the firing of transition t_1 will take the set X from place p_r^0, will find the thread $r.i$ in X with i being the smallest thread identifier in X, will put into p_r^0 the set (one token) $X \setminus \{\mathsf{min}(X)\}$ and into p_1 the pair $r.i$. In other words, this firing will spawn a new thread $r.i$.

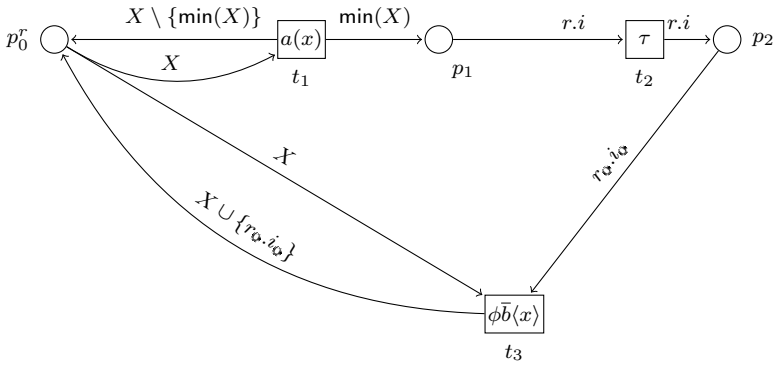


Figure 5.3: Adding atomic transitions

Finally, for vertex v_3, the corresponding transition t_3 with label $\phi\bar{b}\langle x\rangle$ has p_2 as the input place and p_r^0 as output place (as v_3 is preceded by v_2 and v_3 is a terminal vertex in the π-graph). The input arc from p_2 to t_3 is labelled $r_{\mathrm{o}}.i_{\mathrm{o}}$ depending on the type of the action at v_3 as in the case of t_1. Similarly to transition t_2, the connection between t_3 and p_r^0 is also particular because of the type of p_r^0. The input arc from p_r^0 to t_3 is labelled X and the output arc is labelled $X \cup \{r_{\mathrm{o}}.i_{\mathrm{o}}\}$ meaning that token $r_{\mathrm{o}}.i_{\mathrm{o}}$ is added to the set (token) in p_r^0. These three atomic transitions of the translated net are represented in Figure 5.3.

Step 3: Translate synchronizations In the π-graph, any pair of input and output actions can potentially lead to a synchronization, so we create a new synchronization

transition for each such pair, called a sync-transition. In the example, the transition t_4 labelled τ with double border is such a sync-transition. The connectivity of this sync-transition is inherited from the input and output transitions. All transitions of the translated net are represented in Figure 5.4.

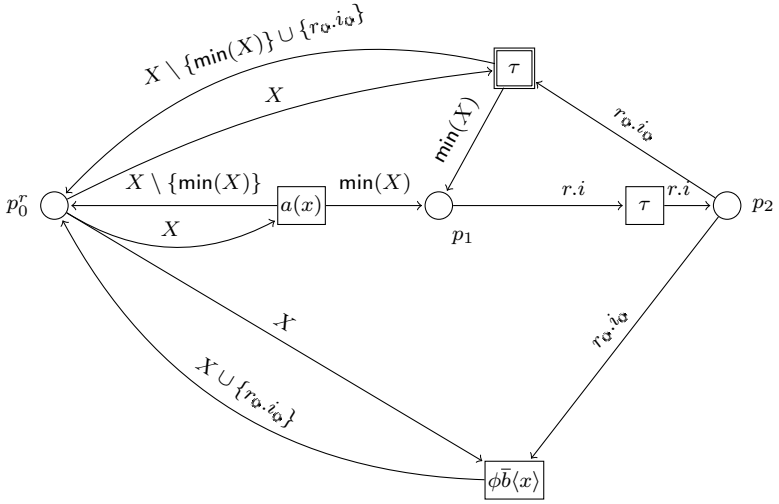


Figure 5.4: Adding sync-transitions

Step 4: Create a name context place At this stage, we add a context place p_Γ and connect it to each transition. The input arc is labelled Γ and the output one is labelled Γ', where Γ and Γ' are net variables. This place will store the current name context of the π-graph. Figure 5.5 illustrates the connectivity of the name context place p_Γ.

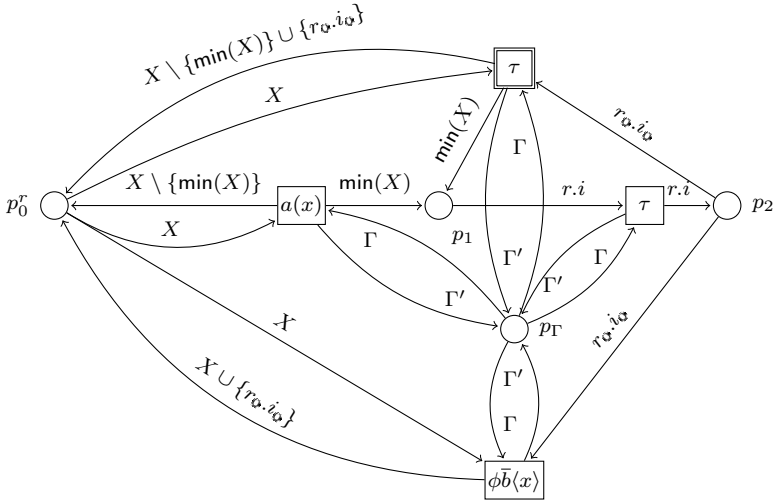


Figure 5.5: Adding the name context place p_Γ

Step 5: Make guards The transition guards depend on the type of the action (input, output, internal or synchronization) and on the situation of the corresponding vertex

(or two vertices in the case of a sync-transition) in the π-graph. In particular, in our example, the transition t_3 corresponds to the action at vertex v_3 which is an output on a terminal vertex. The guard of t_3 is defined as follows:

$$\mathsf{G}(t_3) = \left[\begin{array}{l} \mathsf{pre}_{\text{out}}(\Gamma, r, i_{\text{o}}, \phi\bar{b}\langle x\rangle) \quad \wedge \\ \Gamma' \leftarrow \mathsf{reset}^{r}_{i_{\text{o}}}(\mathsf{post}_{\text{out}}(\Gamma, r, i_{\text{o}}, \phi\bar{b}\langle x\rangle)) \end{array} \right.$$

where the first line is a firing condition and the second line may be seen as an always true Boolean expression used to update the name context information.

The sync-transition with label τ depends on the actions at vertices v_1 and v_3. Its guard is defined as follows:

$$\mathsf{G}(t_4) = \left[\begin{array}{l} \mathsf{pre}_{\text{com}}(\mathsf{init}^{r_{\text{o}}}_{i_{\text{o}}}(\Gamma), r_{\text{o}}, i_{\text{o}}, \phi\bar{b}\langle x\rangle, r_{\text{i}}, i_{\text{i}}, a(x)) \quad \wedge \\ \Gamma' \leftarrow \mathsf{reset}^{r_{\text{i}}}_{i_{\text{i}}}(\mathsf{post}_{\text{com}}(\mathsf{init}^{r_{\text{o}}}_{i_{\text{o}}}(\Gamma), r_{\text{o}}, i_{\text{o}}, \phi\bar{b}\langle x\rangle, r_{\text{i}}, i_{\text{i}}, a(x))). \end{array} \right.$$

Notice that in these guards, $\mathsf{pre}_{\text{out}}(), \mathsf{post}_{\text{out}}, \mathsf{pre}_{\text{com}}, \mathsf{post}_{\text{com}}, \mathsf{reset}()$ and $\mathsf{init}()$ are functions that are used to define the precondition and post-condition of an action in π-graphs.

5.1.1.2 Part II: Translating the context

For each π-graph we obtain a corresponding Petri net structure which does not depend on the context of the π-graph and is static. The context of π-graphs is translated into a marking of the translated net structure which is obtained as follows:

- The name context is put into the name context place p_Γ as a unique structured token.
- For each replicator, threads at an inner vertex in the π-graph are put into the corresponding places (there is never threads in terminal vertices); and
- The set of all threads in the pool of threads is put into the initial place as a unique token;

For the example in Figure 5.1, the translated marking is shown in Figure 5.6. One may observe that there is a correspondence between the π-graph in the context (Γ_1, Δ_1) and the translated Petri net. In the π-graph, the atomic actions at vertices v_1 and v_3, and the synchronization between these two actions can be activated if their preconditions are satisfied. Similarly, in the translated net, the corresponding transitions t_1, t_3 and the sync-transition can be performed if their guards are satisfied. Because the guard of transitions are essentially the same as the preconditions of the corresponding actions in the π-graph, if an action in the π-graph may be activated then the corresponding transition can be performed in the translated net.

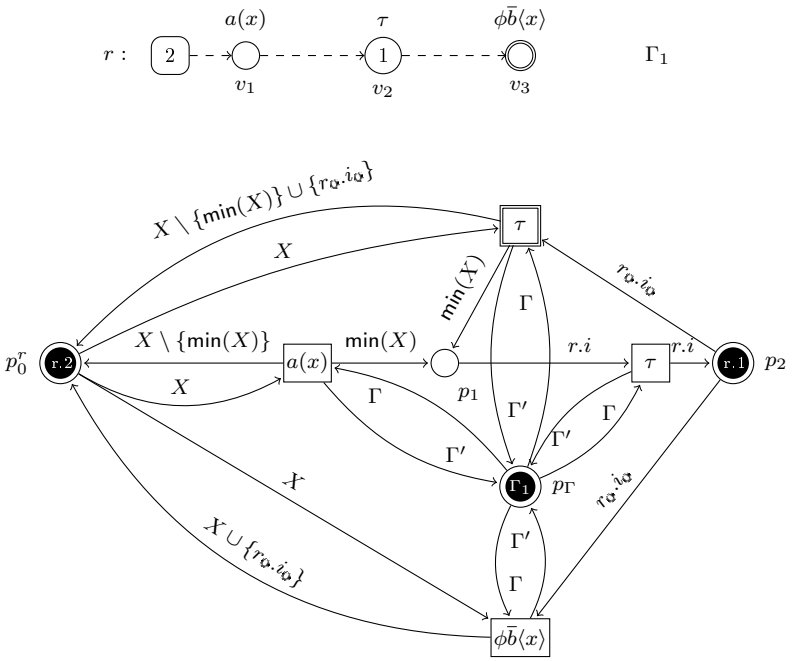


Figure 5.6: Translation of the context producing a marking.

5.1.2 Formal definition of the translation

As mentioned above, in the translation of the context, in order to optimize the size of the state space, we need to determine the minimum value in the set of available threads of a given replicator. We define a function that allows us to compute this minimum as follows.

Definition 5.1 (Minimum token)**.** *Let $I_r \subseteq [1..\mathcal{K}(r)]$ be subset of threads of replicator r and $T_r = \{r.i \mid i \in I_r\}$ be the set of tokens corresponding to I_r. The* minimum token *of T_r, denoted by $\mathsf{min}(T_r)$, is token $r.i$ having the minimal i.*

We start the translation by producing the Petri net structure, the corresponding marking will be introduced at the end.

Definition 5.2 (Translated net structure)**.** *Let $\pi = \langle \mathcal{R}, \mathcal{K}, \mathcal{G}, \mathcal{N} \rangle$ be a π-graph, and χ_π the type of the name context of π. The* translated net structure *of π, denoted by $\mathsf{net}(\pi)$, is a tuple $(\mathsf{P}, \mathsf{T}, \mathsf{U}, \mathsf{G})$ which is obtained as follows:*

The set of places P

$$\begin{aligned} \mathsf{P} &= \{p_\Gamma\} \cup \mathsf{P}_0 \cup \mathsf{P}_v \quad \textit{with} \\ \mathsf{P}_0 &= \{p_0^r \mid r \in \mathcal{R}\} \quad \textit{initial places} \\ \mathsf{P}_v &= \bigcup_{r \in \mathcal{R}} \{p_v^r \mid v \in V_r, \mathsf{stype}(r, v) \in \{\mathsf{norm}, \mathsf{spwn}\}\} \end{aligned} \tag{5.1}$$

Type of places:

$$\forall p \in \mathsf{P}, \mathsf{U}(p) = \begin{cases} \chi_\pi & \text{if } p = p_\Gamma, \\ \mathbb{P}(\{r.i \mid i \in [1..\mathcal{K}(r)]\}) & \text{if } p \in \mathsf{P}_0, p = p_0^r, \\ \{r.i \mid i \in [1..\mathcal{K}(r)]\} & \text{if } p \in \mathsf{P}_v, p = p_v^r. \end{cases} \tag{5.2}$$

The set of transitions $\mathsf{T} = \mathsf{T}_1 \cup \mathsf{T}_2$, *where* T_1 *is the set of atomic transitions and* T_2 *is the set of synchronization transitions.*

$\mathsf{T}_1 = \mathfrak{N} \cup \mathfrak{S} \cup \mathfrak{T} \cup \mathfrak{M}$ *is the set of atomic transitions, with*

$$\mathfrak{N} = \{t_v^r \mid r \in \mathcal{R}, v \in V_r, \mathsf{stype}(r, v) = \mathsf{norm}\}, \tag{5.3a}$$

$$\mathfrak{S} = \{t_v^r \mid r \in \mathcal{R}, v \in V_r, \mathsf{stype}(r, v) = \mathsf{spwn}\}, \tag{5.3b}$$

$$\mathfrak{T} = \{t_v^r \mid r \in \mathcal{R}, v \in V_r, \mathsf{stype}(r, v) = \mathsf{term}\}, \tag{5.3c}$$

$$\mathfrak{M} = \{t_v^r \mid r \in \mathcal{R}, v \in V_r, \mathsf{stype}(r, v) = \mathsf{mono}\}. \tag{5.3d}$$

$$\mathsf{T}_2 = \{t_{v_o v_i}^{r_o r_i} \mid t_u^{r_o}, t_v^{r_i} \in \mathsf{T}_1, L_{r_o}(v_o) = \phi\bar{c}\langle a\rangle, L_{r_i}(v_i) = \psi d(x)\} \tag{5.4}$$

For each transition $t \in \mathsf{T}$, *its label* $\mathsf{U}(t)$ *is defined as follows:*

$$\mathsf{U}(t) = \begin{cases} L_r(v) & \text{if } t = t_v^r \in \mathsf{T}_1 \\ \boldsymbol{\tau} & \text{if } t \in \mathsf{T}_2 \end{cases} \tag{5.5}$$

The arcs *For each transition* t *in* T, *the arcs connected to* t *are defined by its input and output places.*

The input places of t, *denoted by* ${}^\bullet t$, *are:*

$${}^\bullet t = \begin{cases} \{p_0^r, p_\Gamma\} & \text{if} \quad t = t_v^r \in \mathfrak{S} \cup \mathfrak{M} \\ \{p_u^r, p_\Gamma\} & \text{if} \quad t = t_v^r \in \mathfrak{N} \quad \text{with } (u, v) \in E_r \\ \{p_u^r, p_0^r, p_\Gamma\} & \text{if} \quad t = t_v^r \in \mathfrak{T} \quad \text{with } (u, v) \in E_r \\ {}^\bullet t_{v_o}^{r_o} \cup {}^\bullet t_{v_i}^{r_i} & \text{if} \quad t = t_{v_o v_i}^{r_o r_i} \in \mathsf{T}_2 \end{cases} \tag{5.6}$$

The output places of t, *denoted by* $t^\bullet$, *are:*

$$t^\bullet = \begin{cases} \{p_0^r, p_\Gamma\} & \text{if} \quad t = t_v^r \in \mathfrak{T} \cup \mathfrak{M} \\ \{p_v^r, p_\Gamma\} & \text{if} \quad t = t_v^r \in \mathfrak{N} \\ \{p_v^r, p_0^r, p_\Gamma\} & \text{if} \quad t = t_v^r \in \mathfrak{S} \\ t_{v_o}^{r_o \bullet} \cup t_{v_i}^{r_i \bullet} & \text{if} \quad t = t_{v_o v_i}^{r_o r_i} \in \mathsf{T}_2 \end{cases} \tag{5.7}$$

The arc labels *assigning to each arc an expression (possibly with variables) compatible with its input/output place, where* $X, \Gamma, r, r_o, r_i, i, i_o, i_i$ *are variables, are defined as follows:*

- *Input arcs labels for all atomic transitions t in T and all places p in ${}^\bullet t$ are:*

$$\mathsf{U}((p,t)) = \begin{cases} \Gamma & \text{if } p = p_\Gamma \\ X & \text{if } p = p_0^r \\ r.i & \text{if } (p = p_v^r) \wedge (\mathsf{U}(t) = \phi\tau) \\ r_o.i_o & \text{if } (p = p_v^r) \wedge (\mathsf{U}(t) = \phi\overline{c}\langle a\rangle) \\ r_i.i_i & \text{if } (p = p_v^r) \wedge (\mathsf{U}(t) = \phi c(x)) \end{cases} \quad \textit{(cf. Tab. 5.3)} \tag{5.8}$$

- *Input arcs labels for all sync-transitions $t = t^{r_o r_i}_{v_o v_i} \in \mathsf{T}_2$ and all place p in ${}^\bullet t = \{p_1, p_2, p_\Gamma \mid p_1 \in {}^\bullet t^{r_o}_{v_o}, p_2 \in {}^\bullet t^{r_i}_{v_i}\}$, are:*

$$\mathsf{U}((p,t)) = \begin{cases} \Gamma & \text{if} \quad p = p_\Gamma \\ X & \text{if} \quad (p_1 = p_2 = p_0^r) \wedge (p = p_1) \\ r_o.i_o, r_i.i_i & \text{if} \quad (p_1 = p_2 = p_v^r) \wedge (p = p_1) \\ X_o & \text{if} \quad (p_1 \neq p_2) \wedge (p = p_1 = p_0^r) \\ r_o.i_o & \text{if} \quad (p_1 \neq p_2) \wedge (p = p_1 = p_v^r) \\ X_i & \text{if} \quad (p_1 \neq p_2) \wedge (p = p_2 = p_0^r) \\ r_i.i_i & \text{if} \quad (p_1 \neq p_2) \wedge (p = p_2 = p_v^r) \end{cases} \quad \textit{(cf. Tab. 5.4)} \tag{5.9}$$

- *Output arcs labels for all atomic-transitions t in T_1 and all places p in $t^\bullet$ are:*

$$\mathsf{U}((t,p)) = \begin{cases} \Gamma' & \text{if } p = p_\Gamma \\ X' & \text{if } p = p_0^r \\ r.i & \text{if } (p = p_v^r) \wedge (\mathsf{U}(t) = \phi\tau) \\ r_o.i_o & \text{if } (p = p_v^r) \wedge (\mathsf{U}(t) = \phi\overline{c}\langle a\rangle) \\ r_i.i_i & \text{if } (p = p_v^r) \wedge (\mathsf{U}(t) = \phi c(x)) \end{cases} \quad \textit{(cf. Tab. 5.3)} \tag{5.10}$$

- *Output arcs labels for all sync-transitions $t = t^{r_o r_i}_{v_o v_i} \in \mathsf{T}_2$ and all places p in $t^\bullet = \{p_1, p_2, p_\Gamma \mid p_1 \in t^{r_o}_{v_o}{}^\bullet, p_2 \in t^{r_i}_{v_i}{}^\bullet\}$ are:*

$$\mathsf{U}((t,p)) = \begin{cases} \Gamma' & \text{if} \quad p = p_\Gamma \\ X' & \text{if} \quad (p_1 = p_2) \wedge (p = p_1 = p_0^r) \\ X'_o & \text{if} \quad (p_1 \neq p_2) \wedge (p = p_1 = p_0^r) \\ X'_i & \text{if} \quad (p_1 \neq p_2) \wedge (p = p_2 = p_0^r) \\ r_o.i_o & \text{if} \quad (p_1 \neq p_2) \wedge (p = p_1 = p_v^r) \\ r_i.i_i & \text{if} \quad (p_1 \neq p_2) \wedge (p = p_2 = p_v^r) \end{cases} \quad \textit{(cf. Tab. 5.4)} \tag{5.11}$$

The transition guard *for each transition t in T, denoted by $\mathsf{G}(t)$, is given in Tab. 5.1 if $t \in \mathsf{T}_1$ and in Tab. 5.2 if $t \in \mathsf{T}_2$.*

t_v^r	$\mathsf{U}(t_v^r)$	$\mathsf{G}(t_v^r)(\Gamma, r, i)$
$\mathfrak{N}$	$\phi\tau$	$\mathsf{pre}_{\mathrm{tau}}(\Gamma, r, i, \phi); \Gamma' \leftarrow \mathsf{post}_{\mathrm{tau}}(\Gamma, r, i, \phi)$
	$\phi c(x)$	$\mathsf{pre}_{\mathrm{in}}(\Gamma, r_{\mathrm{i}}, i_{\mathrm{i}}, \phi c(x)); \Gamma' \leftarrow \mathsf{post}_{\mathrm{in}}(\Gamma, r_{\mathrm{i}}, i_{\mathrm{i}}, \phi c(x))$
	$\phi\overline{c}\langle a\rangle$	$(\mathsf{pre}_{\mathrm{out}}(\Gamma, r_{\mathrm{o}}, i_{\mathrm{o}}, \phi\overline{c}\langle a\rangle'); \Gamma' \leftarrow \mathsf{post}_{\mathrm{out}}(\Gamma, r_{\mathrm{o}}, i_{\mathrm{o}}, \phi))$ or $(\mathsf{pre}_{\nu\mathrm{out}}(\Gamma, r_{\mathrm{o}}, i_{\mathrm{o}}, \phi\overline{c}\langle a\rangle); \Gamma' \leftarrow \mathsf{post}_{\nu\mathrm{out}}(\Gamma, r_{\mathrm{o}}, i_{\mathrm{o}}, \phi\overline{c}\langle a\rangle))$
$\mathfrak{S}$	$\phi\tau$	$\mathsf{pre}_{\mathrm{tau}}(\mathsf{init}_i^r(\Gamma), r, i, \phi); \Gamma' \leftarrow \mathsf{post}_{\mathrm{tau}}(\mathsf{init}_i^r(\Gamma), r, i, \phi)$
	$\phi c(x)$	$\mathsf{pre}_{\mathrm{in}}(\mathsf{init}_{i_{\mathrm{i}}}^{r_{\mathrm{i}}}(\Gamma), r_{\mathrm{i}}, i_{\mathrm{i}}, \phi c(x)); \Gamma' \leftarrow \mathsf{post}_{\mathrm{in}}(\Gamma, r_{\mathrm{i}}, i_{\mathrm{i}}, \phi c(x))$
	$\phi\overline{c}\langle a\rangle$	$(\mathsf{pre}_{\mathrm{out}}(\mathsf{init}_{i_{\mathrm{o}}}^{r_{\mathrm{o}}}(\Gamma), r_{\mathrm{o}}, i_{\mathrm{o}}, \phi\overline{c}\langle a\rangle); \Gamma' \leftarrow \mathsf{post}_{\mathrm{out}}(\mathsf{init}_{i_{\mathrm{o}}}^{r_{\mathrm{o}}}(\Gamma), r_{\mathrm{o}}, i_{\mathrm{o}}, \phi))$ or $(\mathsf{pre}_{\nu\mathrm{out}}(\mathsf{init}_{i_{\mathrm{o}}}^{r_{\mathrm{o}}}(\Gamma), r_{\mathrm{o}}, i_{\mathrm{o}}, \phi\overline{c}\langle a\rangle); \Gamma' \leftarrow \mathsf{post}_{\nu\mathrm{out}}(\mathsf{init}_{i_{\mathrm{o}}}^{r_{\mathrm{o}}}(\Gamma), r_{\mathrm{o}}, i_{\mathrm{o}}, \phi\overline{c}\langle a\rangle))$
$\mathfrak{T}$	$\phi\tau$	$\mathsf{pre}_{\mathrm{tau}}(\Gamma, r, i, \phi); \Gamma' \leftarrow \mathsf{reset}_i^r(\mathsf{post}_{\mathrm{tau}}(\Gamma, r, i, \phi))$
	$\phi c(x)$	$\mathsf{pre}_{\mathrm{in}}(\Gamma, r_{\mathrm{i}}, i_{\mathrm{i}}, \phi c(x)); \Gamma' \leftarrow \mathsf{reset}_{i_{\mathrm{i}}}^{r_{\mathrm{i}}}(\mathsf{post}_{\mathrm{in}}(\Gamma, r_{\mathrm{i}}, i_{\mathrm{i}}, \phi c(x)))$
	$\phi\overline{c}\langle a\rangle$	$(\mathsf{pre}_{\mathrm{out}}(\Gamma, r_{\mathrm{o}}, i_{\mathrm{o}}, \phi\overline{c}\langle a\rangle); \Gamma' \leftarrow \mathsf{reset}_{i_{\mathrm{o}}}^{r_{\mathrm{o}}}(\mathsf{post}_{\mathrm{out}}(\Gamma, r_{\mathrm{o}}, i_{\mathrm{o}}, \phi)))$ or $(\mathsf{pre}_{\nu\mathrm{out}}(\Gamma, r_{\mathrm{o}}, i_{\mathrm{o}}, \phi\overline{c}\langle a\rangle); \Gamma' \leftarrow \mathsf{reset}_{i_{\mathrm{o}}}^{r_{\mathrm{o}}}(\mathsf{post}_{\nu\mathrm{out}}(\Gamma, r_{\mathrm{o}}, i_{\mathrm{o}}, \phi\overline{c}\langle a\rangle)))$
$\mathfrak{M}$	$\phi\tau$	$\mathsf{pre}_{\mathrm{tau}}(\mathsf{init}_i^r(\Gamma), r, i, \phi); \Gamma' \leftarrow \mathsf{reset}_i^r(\mathsf{post}_{\mathrm{tau}}(\mathsf{init}_i^r(\Gamma), r, i, \phi))$
	$\phi c(x)$	$\mathsf{pre}_{\mathrm{in}}(\mathsf{init}_{i_{\mathrm{i}}}^{r_{\mathrm{i}}}(\Gamma), r_{\mathrm{i}}, i_{\mathrm{i}}, \phi c(x)); \Gamma' \leftarrow \mathsf{reset}_{i_{\mathrm{i}}}^{r_{\mathrm{i}}}(\mathsf{post}_{\mathrm{in}}(\mathsf{init}_{i_{\mathrm{i}}}^{r_{\mathrm{i}}}(\Gamma), r_{\mathrm{i}}, i_{\mathrm{i}}, \phi c(x)))$
	$\phi\overline{c}\langle a\rangle$	$(\mathsf{pre}_{\mathrm{out}}(\mathsf{init}_{i_{\mathrm{o}}}^{r_{\mathrm{o}}}(\Gamma), r_{\mathrm{o}}, i_{\mathrm{o}}, \phi\overline{c}\langle a\rangle);$ $\Gamma' \leftarrow \mathsf{reset}_{i_{\mathrm{o}}}^{r_{\mathrm{o}}}(\mathsf{post}_{\mathrm{out}}(\mathsf{init}_{i_{\mathrm{o}}}^{r_{\mathrm{o}}}(\Gamma), r_{\mathrm{o}}, i_{\mathrm{o}}, \phi)))$ or $(\mathsf{pre}_{\nu\mathrm{out}}(\mathsf{init}_{i_{\mathrm{o}}}^{r_{\mathrm{o}}}(\Gamma), r_{\mathrm{o}}, i_{\mathrm{o}}, \phi\overline{c}\langle a\rangle);$ $\Gamma' \leftarrow \mathsf{reset}_{i_{\mathrm{o}}}^{r_{\mathrm{o}}}(\mathsf{post}_{\nu\mathrm{out}}(\mathsf{init}_{i_{\mathrm{o}}}^{r_{\mathrm{o}}}(\Gamma), r_{\mathrm{o}}, i_{\mathrm{o}}, \phi\overline{c}\langle a\rangle)))$

Table 5.1: Guards of atomic-transitions

In the definition, the unique place $p_\Gamma \in \mathsf{P}$ is called a *context place* of the translated net, other places are called *control places*. Similarly, a token in a control place is called a *control token*, and a token in a context place is called a *context token*. Note that, an initial place contains a special token, which is a set of control tokens.

Tab. 5.1 shows the definition of the guards of atomic transitions t_v^r, where $\mathsf{G}(t_v^r)(\Gamma, r, i)$ denotes the evaluation of the guard of t_v^r with the name context token Γ in the context place p_Γ and with thread i of replicator r. Each guard is composed of a precondition, which is the precondition of the corresponding action at vertex v in replicator r, denoted (r, v), in the π-graph. It also contains a computation of the name context token Γ' after firing the transition, which is the post-condition of the corresponding action. Obviously, the evaluation of a guard has the same value as the precondition.

Similarly, Tab. 5.2 on the next page shows the definition of the guards of synchronization transitions. We denote by $\mathsf{G}(t_{v_{\mathrm{o}}v_{\mathrm{i}}}^{r_{\mathrm{o}}r_{\mathrm{i}}})(\Gamma, r_{\mathrm{o}}, i_{\mathrm{o}}, r_{\mathrm{i}}, i_{\mathrm{i}})$ the evaluation of the guard of a sync-transition $t_{v_{\mathrm{o}}v_{\mathrm{i}}}^{r_{\mathrm{o}}r_{\mathrm{i}}}$ with the name context token Γ, thread i_{o} of replicator r_{o} performing an output and thread i_{i} of r_{i} performing an input. The synchronization transitions are grouped with respect to the types of their atomic transitions (*i.e.*, $\mathfrak{N}, \mathfrak{M}, \mathfrak{S}, \mathfrak{T}$). Moreover, depending on these types, the name context Γ may be initialized or reset in the same way as given in Tab. 4.1 on page 80.

No.	$t^{r_o r_i}_{v_o v_i}$	$\mathsf{G}(t^{r_o r_i}_{v_o v_i})(\Gamma, r_o, i_o, r_i, i_i)$
1	$\mathfrak{N} - \mathfrak{N}$	$\mathsf{pre}_{com}(\Gamma, r_o, i_o, \phi_o \overline{c}\langle a\rangle, r_i, i_i, \phi_i d(x))$; $\Gamma' \leftarrow \mathsf{post}_{com}(\Gamma, r_o, i_o, \phi_o \overline{c}\langle a\rangle, r_i, i_i, \phi_i d(x))$
2	$\mathfrak{N} - \mathfrak{S}$	$\mathsf{pre}_{com}(\mathsf{init}^{r_i}_{i_i}(\Gamma), r_o, i_o, \phi_o \overline{c}\langle a\rangle, r_i, i_i, \phi_i d(x))$; $\Gamma' \leftarrow \mathsf{post}_{com}(\mathsf{init}^{r_i}_{i_i}(\Gamma), r_o, i_o, \phi_o \overline{c}\langle a\rangle, r_i, i_i, \phi_i d(x))$
3	$\mathfrak{N} - \mathfrak{T}$	$\mathsf{pre}_{com}(\Gamma, r_o, i_o, \phi_o \overline{c}\langle a\rangle, r_i, i_i, \phi_i d(x))$; $\Gamma' \leftarrow \mathsf{reset}^{r_i}_{i_i}(\mathsf{post}_{com}(\Gamma, r_o, i_o, \phi_o \overline{c}\langle a\rangle, r_i, i_i, \phi_i d(x)))$
4	$\mathfrak{N} - \mathfrak{M}$	$\mathsf{pre}_{com}(\mathsf{init}^{r_i}_{i_i}(\Gamma), r_o, i_o, \phi_o \overline{c}\langle a\rangle, r_i, i_i, \phi_i d(x))$; $\Gamma' \leftarrow \mathsf{reset}^{r_i}_{i_i}(\mathsf{post}_{com}(\mathsf{init}^{r_i}_{i_i}(\Gamma), r_o, i_o, \phi_o \overline{c}\langle a\rangle, r_i, i_i, \phi_i d(x)))$
5	$\mathfrak{S} - \mathfrak{S}$	$\mathsf{pre}_{com}(\mathsf{init}^{r_i}_{i_i}(\mathsf{init}^{r_o}_{i_o}(\Gamma)), r_o, i_o, \phi_o \overline{c}\langle a\rangle, r_i, i_i, \phi_i d(x))$; $\Gamma' \leftarrow \mathsf{post}_{com}(\mathsf{init}^{r_i}_{i_i}(\mathsf{init}^{r_o}_{i_o}(\Gamma)), r_o, i_o, \phi_o \overline{c}\langle a\rangle, r_i, i_i, \phi_i d(x))$
6	$\mathfrak{S} - \mathfrak{T}$	$\mathsf{pre}_{com}(\mathsf{init}^{r_o}_{i_o}(\Gamma), r_o, i_o, \phi_o \overline{c}\langle a\rangle, r_i, i_i, \phi_i d(x))$; $\Gamma' \leftarrow \mathsf{reset}^{r_i}_{i_i}(\mathsf{post}_{com}(\mathsf{init}^{r_o}_{i_o}(\Gamma), r_o, i_o, \phi_o \overline{c}\langle a\rangle, r_i, i_i, \phi_i d(x)))$
7	$\mathfrak{S} - \mathfrak{M}$	$\mathsf{pre}_{com}(\mathsf{init}^{r_i}_{i_i}(\mathsf{init}^{r_o}_{i_o}(\Gamma)), r_o, i_o, \phi_o \overline{c}\langle a\rangle, r_i, i_i, \phi_i d(x))$; $\Gamma' \leftarrow \mathsf{reset}^{r_i}_{i_i}(\mathsf{post}_{com}(\mathsf{init}^{r_i}_{i_i}(\mathsf{init}^{r_o}_{i_o}(\Gamma)), r_o, i_o, \phi_o \overline{c}\langle a\rangle, r_i, i_i, \phi_i d(x)))$
8	$\mathfrak{T} - \mathfrak{T}$	$\mathsf{pre}_{com}(\Gamma, r_o, i_o, \phi_{out} \overline{c}\langle a\rangle, r_i, i_i, \phi_i d(x))$; $\Gamma' \leftarrow \mathsf{reset}^{r_i}_{i_i}(\mathsf{reset}^{r_o}_{i_o}(\mathsf{post}_{com}(\Gamma, r_o, i_o, \phi_o \overline{c}\langle a\rangle, r_i, i_i, \phi_i d(x))))$
9	$\mathfrak{T} - \mathfrak{M}$	$\mathsf{pre}_{com}(\mathsf{init}^{r_i}_{i_i}(\Gamma), r_o, i_o, \phi_o \overline{c}\langle a\rangle, r_i, i_i, \phi_i d(x))$; $\Gamma' \leftarrow \mathsf{reset}^{r_o}_{i_o}(\mathsf{post}_{com}(\mathsf{init}^{r_i}_{i_i}(\Gamma), r_o, i_o, \phi_o \overline{c}\langle a\rangle, r_i, i_i, \phi_i d(x)))$
10	$\mathfrak{M} - \mathfrak{M}$	$\mathsf{pre}_{com}(\mathsf{init}^{r_i}_{i_i}(\mathsf{init}^{r_o}_{i_o}(\Gamma)), r_o, i_o, \phi_o \overline{c}\langle a\rangle, r_i, i_i, \phi_i d(x))$; $\Gamma' \leftarrow \mathsf{reset}^{r_i}_{i_i}(\mathsf{reset}^{r_o}_{i_o}(\mathsf{post}_{com}(\mathsf{init}^{r_i}_{i_i}(\mathsf{init}^{r_o}_{i_o}(\Gamma)),$ $r_o, i_o, \phi_o \overline{c}\langle a\rangle, r_i, i_i, \phi_i d(x))))$

Table 5.2: Guards of sync-transitions

Tab. 5.3 on the next page shows rules of assigning arc labels for atomic transitions depending the types of transitions and types of the corresponding actions in π-graphs. In this table, we do not present the label of input and output arcs between the transition and the context place p_Γ for simplicity. In fact, p_Γ is connected to all transitions with Γ and Γ' as input and output arc labels, respectively. Thus, the arc label of an atomic transition may be

- a variable Γ, representing the context token that the transition consumes from p_Γ,
- a variable Γ' (instantiated in the guards in Tab. 5.1) represents the context token produced to p_Γ after firing the transition,
- a structured expression $r.i$ (resp. $r_o.i_o$ or $r_i.i_i$) representing a control token (thread) used by a silent action (resp. by an output or an input),
- a variable X, representing a special token (*i.e.*, contains a set of control tokens in the form of $r.i$) in an initial place that the transition consume or produce,

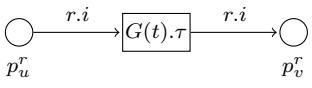
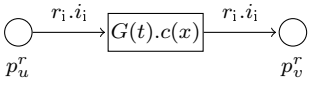
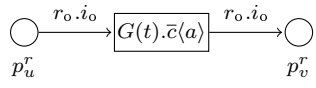
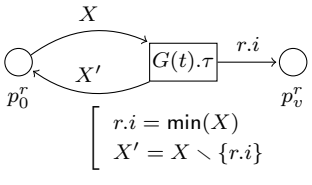
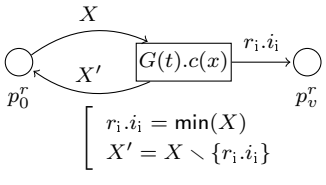
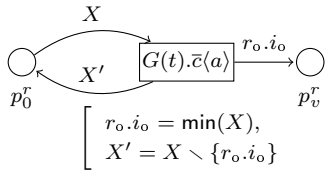
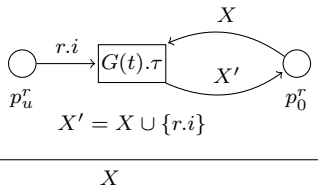
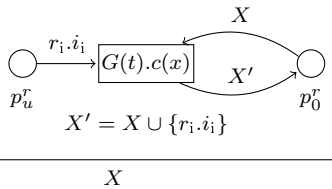
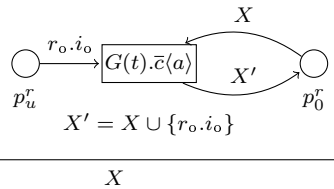
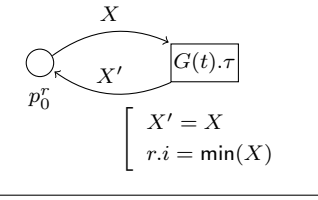
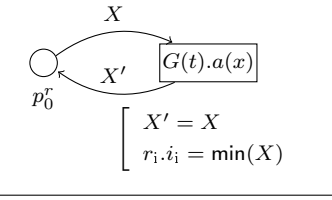
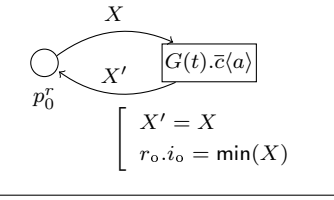
t_v^r	$\phi\tau$	$\phi c(x)$	$\phi\overline{c}\langle a\rangle$
$\mathfrak{N}$			
$\mathfrak{S}$			
$\mathfrak{T}$			
$\mathfrak{M}$			

Table 5.3: Arc label of an atomic-transition t_v^r

- an expression $X \cup \{r.i\}$ (resp. $X \setminus \{r.i\}$) meaning that a control token $r.i$ is added to X (resp. removed from X) and then X is put back to the initial place, or
- an expression $\mathsf{min}(X)$ representing the minimum control token in X, computed using Def 5.1.

We can observe that if the type of an atomic transition t_v^r is $\mathfrak{T}$, then p_0^r is both input and output place of t_v^r. However, place p_0^r does not provide control token for firing the transition but it receives control token that t_v^r consumes from place p_u^r. In this case, p_u^r is a *control input place* of t_v^r, but not p_0^r. Similarly, if the type of t_v^r is $\mathfrak{S}$, then place p_v^r is a *control output place* of t_v^r. In this case, p_0^r is a control input place of t_v^r.

Tab. 5.4 shows rules of assigning arc labels for sync-transitions. Similarly to Tab. 5.3, we do not present the arcs between the transition and context place p_Γ. Since a sync-transition is constructed by fusing an output and an input atomic transitions, it needs two control tokens $r_\mathrm{o}.i_\mathrm{o}$ and $r_\mathrm{i}.i_\mathrm{i}$ for firing, where r_o and r_i may be the same. Arc labels of a sync-transition may be

- a variable Γ or Γ', where Γ' is computed using definitions from Tab. 5.2
- a structured expression $r_\mathrm{o}.i_\mathrm{o}$ or $r_\mathrm{i}.i_\mathrm{i}$, as before
- a variable X_o, representing a special token of the initial place, which was connected before the synchronization to the output,

- a variable X_i, the same as above for input
- a variable X, the same as above, but when the same place was both an input and an output,
- variables X'_o, X'_i, or X', corresponding to updated tokens X_o, X_i and X, respectively,
- a pair of variables $r_{\mathrm{o}}.i_{\mathrm{o}}, r_{\mathrm{i}}.i_{\mathrm{i}}$, representing a pair of output and input control tokens that are consumed from the same place

We denote by $\mathsf{cip}(t^r_v)$ and $\mathsf{cop}(t^r_v)$ the control input place and control output place of an atomic transition t^r_v, and have the following related properties.

Proposition 5.1 (Unique control input place)**.** *Let π be a π-graph. If t^r_v is an atomic transition in the translated net structure of π, then it has exactly one control input place,*

$$\mathsf{cip}(t^r_v) = \begin{cases} p^r_0 & \text{if} \quad t^r_v \in \mathfrak{S} \cup \mathfrak{M} \\ p^r_u & \text{if} \quad t^r_v \in \mathfrak{N} \cup \mathfrak{T}, \quad \text{where } (u,v) \in E_r. \end{cases}$$

Proof. By Def. 5.2 (Eq. 5.6). □

Proposition 5.2 (Unique control output place)**.** *Let π be a π-graph. If t^r_v is an atomic-transition in the translated net structure of π, then it has exactly one control output place,*

$$\mathsf{cop}(t^r_v) = \begin{cases} p^r_0 & \text{if} \quad t^r_v \in \mathfrak{M} \cup \mathfrak{T} \\ p^r_v & \text{if} \quad t^r_v \in \mathfrak{N} \cup \mathfrak{S}. \end{cases}$$

Proof. By Def. 5.2 (Eq. 5.7). □

Proposition 5.3 (Control places of sync transitions)**.** *Let π be a π-graph and t a sync-transition in* $\mathsf{net}(\pi)$*. Then t has* at most two *control input places and* at most two *control output places.*

Proof. Let $t = t^{r_1 r_2}_{v_1 v_2}$ be a synchronization transition. By Def. 5.2 (Eq. 5.6, Eq. 5.7), the control input (resp. output) places of t are the union of control input (resp. output) places of atomic transitions $t^{r_1}_{v_1}$ and $t^{r_2}_{v_2}$. Moreover, by Prop. 5.1 and Prop. 5.2, each of $t^{r_1}_{v_1}$ and $t^{r_2}_{v_2}$ has exactly one control input place and one control output place (which may be the same place). Thus, the union of them contains *at most two* places. Therefore, the property holds. □

Definition 5.3 (Translated marking)**.** *Let $\pi = \langle \mathcal{R}, \mathcal{K}, \mathcal{G}, \mathcal{N} \rangle$ be a π-graph, $\mathcal{C}_\pi = (\Gamma, \Delta)$ a context of π, and $\mathsf{N} = \mathsf{net}(\pi)$ the* translated net structure*. The* translated marking *of $\mathcal{C}_\pi$ in N, denoted by $\mathsf{mark}(\mathcal{C}_\pi, \mathsf{N})$, is a marking M_π, where:*

1. $\mathsf{M}_\pi(p_\Gamma) = \{\Gamma\}$

2. $\forall p_v^r \in \mathsf{P}_v,\ \mathsf{M}_\pi(p_v^r) = \{r.i \mid i \in \Delta_r(v)\}$

3. $\forall p_0^r \in \mathsf{P}_0,\ \mathsf{M}_\pi(p_0^r) = \{X\} \quad with\ X = \{r.i \mid i \in \mathsf{inact}(\Delta_r)\}.$

According to Def. 5.3, given an initial context $(\Gamma_\pi^0, \Delta_\pi^0)$, the corresponding initial marking M_π^0 is:

$$\mathsf{M}_\pi^0(p) = \begin{cases} \{\Gamma_\pi^0\} & \text{if } p = p_\Gamma \\ \emptyset & \text{if } p = p_v^r, \forall p_v^r \in \mathsf{P}_v \\ \{X\} & \text{if } p = p_0^r, \forall p_0^r \in \mathsf{P}_0 \quad \text{with } X = \{r.i \mid i \in \{1, \ldots, \mathcal{K}(r)\}\} \end{cases}$$

Definition 5.4 (Translation)**.** *Let $\mathcal{C}_\pi$ be a context of a π-graph π. The translation of π in context $\mathcal{C}_\pi$, denoted by $\mathsf{trans}(\pi, \mathcal{C}_\pi)$, is a high-level Petri net $\langle \mathsf{P}, \mathsf{T}, \mathsf{U}, \mathsf{G}; \mathsf{M} \rangle$ in which the net structure $\langle \mathsf{P}, \mathsf{T}, \mathsf{U}, \mathsf{G} \rangle = \mathsf{net}(\pi)$ and the marking $\mathsf{M}_\pi = \mathsf{mark}(\mathcal{C}_\pi, \mathsf{net}(\pi))$.*

Proposition 5.4 (Global occurrence of transitions)**.** *Let π be a π-graph. The occurrence of a transition t in $\mathsf{net}(\pi)$ matches a subset of places related to t and updates only them, the rest does not change.*

Proof. By Def. 5.2 (Eq. 5.6), the context place p_Γ is the input place of all transitions in $\mathsf{net}(\pi)$. Moreover, by Def. 5.3, the place p_Γ contains only one token, which is the name context token. Thus, all the transitions are mutual exclusive, and at most one of them may fire at a time. Therefore, only the markings of places which are input and output places of this transition are updated, the rest does not change. Property is proved. □

5.2 Conformance of the translation

Let $\pi = \langle \mathcal{R}, \mathcal{K}, \mathcal{G}, \mathcal{N} \rangle$ be a π-graph. The translated net structure of π is $\mathsf{net}(\pi) = \langle \mathsf{P}, \mathsf{T}, \mathsf{U}, \mathsf{G} \rangle$. We denote by $\mathcal{C}_\pi = \langle \Gamma, \Delta \rangle$ a context of π. The marking corresponding to $\mathcal{C}_\pi$ in the translated net is $\mathsf{mark}(\mathcal{C}_\pi, \mathsf{net}(\pi))$ (cf. Def. 5.3). In the conformance proof, since π and $\mathsf{net}(\pi)$ are static, we introduce the abbreviation $\mathsf{mark}(\mathcal{C}_\pi)$. From now on, we omit π and $\mathsf{net}(\pi)$.

The proof is decomposed into two levels: local and global conformance. The former concerns the correspondence between each kind of actions in π-graphs, which is an atomic action or a synchronization, and a transition in the translated Petri net. The later concerns the correspondence between any kind of actions and the transitions, *i.e.*, the conformance of the whole translation.

In order to make easier the reading of the proof, which is complex because using many definitions, propositions and lemmas, we present (before going to technical details) the structure of each important part of the proof by a directed graph whose nodes are:

- definitions, which are denoted by D1, D2, ... (if the definition appears in Chapter 5) or D3.1, D4.2 (in case it appears in another chapter, *e.g.*, chapter 3, 4)
- tables, which are denoted by tb1, tb2, ...
- rules, which are denoted by Rc in the case of commitment rules or Rr in the case of rewrite rules
- propositions, which are denoted by P1, P2, ...
- lemmas, which are denoted by L1, L2, ..., or
- theorems, which are denoted by T1, T2, ...

The arcs represent the dependence.

5.2.1 Local conformance

We prove the soundness and the completeness of atomic actions and synchronizations. Informally, the soundness property asserts that if there is a rewrite in the π-graphs with label μ then there exists a corresponding transition with the same label in the translated Petri net. The completeness property asserts that if there is a transition in the Petri net with label μ then there exists a corresponding rewrite with the same label in the π-graphs. In the following, we prove each property for an atomic action and for a synchronization. For each proof, we begin with a dependence graph and then its detail.

5.2.1.1 Soundness for atomic actions

The proof of the soundness for an atomic action (Theorem 5.1, denoted by T1) is structured as shown in a dependence graph in Figure 5.7.

Proposition 5.5 (Existence of atomic transitions)**.** *For any atomic action in a π-graph π, there exists a transition in* $\mathsf{net}(\pi)$ *such that it has the same label as the action.*

Proof. Suppose that $\pi = \langle \mathcal{R}, \mathcal{K}, \mathcal{G}, \mathcal{N} \rangle$. By Def. 4.7, for any atomic action at (r, v), with $r \in \mathcal{R}$, $\mathcal{G}(r) = \langle V_r, L_r, E_r \rangle$, and $v \in V_r$, its structural type is either: norm, spwn, term, or mono. By Def. 5.2 (eq. 5.3), for each case of the action, there exists a corresponding atomic-transition t_v^r in T_1. Moreover, by Def. 5.2 (Eq. 5.5), the syntactical label of the action at (r, v) and the transition t_v^r are the same. Thus, the property holds. □

Definition 5.5 (Available thread)**.** *Let $\mathcal{C}_\pi = (\Gamma, \Delta)$ be the context of a π-graph π and r a replicator in π. A thread i is* available *to the action at (r, v) if and only if*

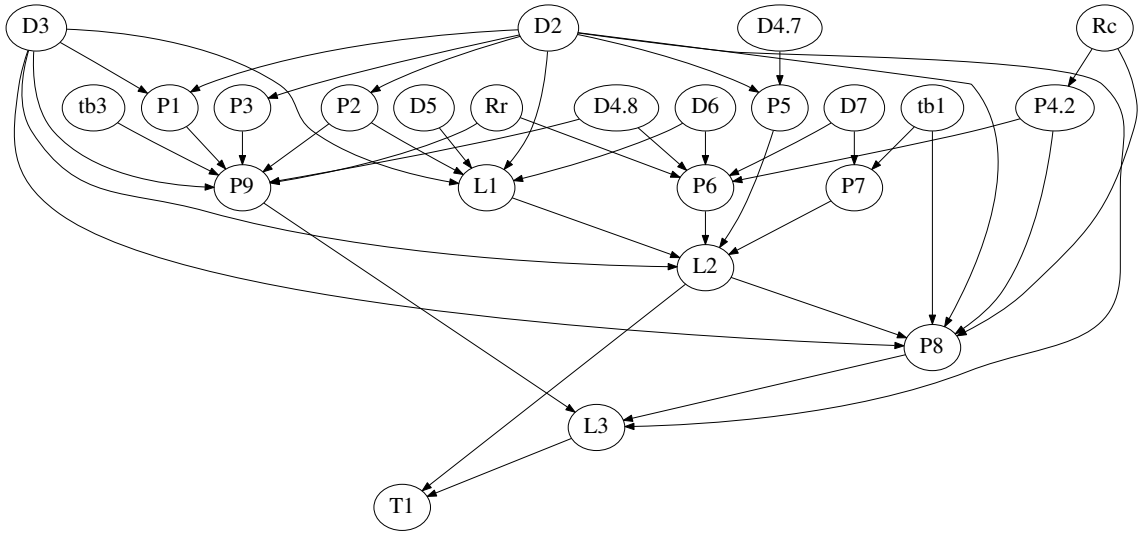


Figure 5.7: Dependence graph of soundness for an atomic action

$\mathsf{avail}_\pi(i, r, v, \mathcal{C}_\pi) = \mathsf{true}$, *with:*

$$\mathsf{avail}_\pi(i, r, v, \mathcal{C}_\pi) \stackrel{\text{def}}{=} \begin{cases} i \in \mathsf{inact}(\Delta_r) & \textit{if} \quad \mathsf{stype}(r, v) \in \{\mathsf{spwn}, \mathsf{mono}\} \\ i \in \Delta_r(u) & \textit{if} \quad \mathsf{stype}(r, v) \in \{\mathsf{norm}, \mathsf{term}\} \textit{ with } (u, v) \in E_r \end{cases}$$

Definition 5.6 (Available control token)**.** *Let π be a π-graph, M_π a marking of $\mathsf{net}(\pi)$ and t a transition in $\mathsf{net}(\pi)$. A control token $r.i$ is* available *to t in the marking M_π, denoted by $\mathsf{avail}_n(r.i, t, \mathsf{M}_\pi)$, if and only if the token is present in the marking of a* control input place *of t.*

Definition 5.7 (Atomic precondition)**.** *Let Γ_π be the name context of a π-graph π and i a thread that is available to the action at (r, v). The precondition of the atomic action in Γ_π with thread i, denoted by $\mathsf{pre}_{\text{atom}}(\Gamma_\pi, r, v, i)$, is defined as follows:*

$$\mathsf{pre}_{\text{atom}}(\Gamma_\pi, r, v, i) \stackrel{\text{def}}{=} \begin{cases} \mathsf{pre}_{\text{tau}}(\Gamma_\pi, r, i, \phi) & \textit{if} \quad L_r(v) = \phi\tau \\ \mathsf{pre}_{\text{in}}(\Gamma_\pi, r, i, \phi c(x)) & \textit{if} \quad L_r(v) = \phi c(x) \\ (\mathsf{pre}_{\text{out}}(\Gamma_\pi, r, i, \phi\overline{c}\langle a\rangle) & \\ \quad \vee\ \mathsf{pre}_{\nu\text{out}}(\Gamma_\pi, r, i, \phi\overline{c}\langle a\rangle)) & \textit{if} \quad L_r(v) = \phi\overline{c}\langle a\rangle \end{cases}$$

If the structural type of the action at vertex v in replicator r $\mathsf{stype}(r, v)$ is mono *or* spwn*, then Γ_π is substituted by $\mathsf{init}_i^r(\Gamma_\pi)$ on the right hand side.*

Proposition 5.6 (Thread existence for atomic actions)**.** *Let $\mathcal{C}_\pi = (\Gamma, \Delta)$ be the context of a π-graph π. If there exists $\mathcal{C}'_\pi$ such that $\mathcal{C}_\pi \xrightarrow[(r,v)]{\mu} \mathcal{C}'_\pi$, then there exists a thread $i \in [1..\mathcal{K}(r)]$ such that $\mathsf{avail}_\pi(i, r, v, \mathcal{C}_\pi) = \mathsf{true}$ and $\mathsf{pre}_{\text{atom}}(\Gamma, r, v, i) = \mathsf{true}$.*

Proof. First, we prove that if $\mathcal{C}_\pi \xrightarrow[(r,v)]{\mu} \mathcal{C}'_\pi$, then there exists a thread i such that $\mathsf{avail}_\pi(i, r, v, \mathcal{C}_\pi) = \mathsf{true}$, by analyzing all cases of *structural type* of the action at (r, v).

If the structural type of the action is norm or term, then by Def. 4.8 on page 81, the rewrite is triggered by applying rule [act] or [term] (cf. p. 75). By these rules, there exists a thread i in $[1..\mathcal{K}(r)]$ such that $i \in \Delta_r(u)$, where $(u, v) \in E_r$. So, by Def. 5.6, we have: $\mathsf{avail}_\pi(i, r, v, \mathcal{C}_\pi) = \mathsf{true}$.

Otherwise, if the structural type of the action is spwn or mono, then by Def. 4.8, the rewrite is triggered by applying rule [spawn] or [mono] (cf. p. 76). By these rules, there exists a thread i in $[1..\mathcal{K}(r)]$ such that $i = \mathsf{min}(\mathsf{inact}(\Delta_r))$. Thus, $i \in \mathsf{inact}(\Delta_r)$. So, by Def. 5.6, $\mathsf{avail}_\pi(i, r, v, \mathcal{C}_\pi) = \mathsf{true}$. In all cases of the action, there always exists a thread i such that $\mathsf{avail}_\pi(i, r, v, \mathcal{C}_\pi) = \mathsf{true}$.

Second, we prove that with such a thread, we also have $\mathsf{pre}_{\mathrm{atom}}(\Gamma, r, v, i) = \mathsf{true}$. By all four rules [act], [spawn], [term] and [mono], in order to trigger a rewrite for an action $\phi\alpha$ at (r, v), we must have a commitment as follows:

$$\Gamma \vdash r : \overset{\phi\alpha}{\textcircled{i}} \overset{\mu}{\Rightarrow} \Gamma'$$

in which Γ is substituted by $\mathsf{init}_i^r(\Gamma)$ in case of applying rule [spawn], or [mono]. We analyze by cases of the action and show that $\mathsf{pre}_{\mathrm{atom}}(\Gamma, r, v, i) = \mathsf{true}$.

If α is a silent action, $\alpha = \tau$, then by rule [tau], we have $\mathsf{pre}_{\mathrm{tau}}(\Gamma, r, i, \phi) = \mathsf{true}$. Thus, by Def. 5.7, $\mathsf{pre}_{\mathrm{atom}}(\Gamma, r, v, i) = \mathsf{true}$. Similarly, if α is an input action, $\alpha = c(x)$, then by rule [in], we have $\mathsf{pre}_{\mathrm{in}}(\Gamma, r, i, \phi) = \mathsf{true}$. Thus, by Def. 5.7, $\mathsf{pre}_{\mathrm{atom}}(\Gamma, r, v, i) = \mathsf{true}$.

Otherwise, α is an output action, $\alpha = \overline{c}\langle a \rangle$ then by Prop. 4.2, two exclusive rules can be applied [out] or [νout]. If rule [out] is applied, then we have $\mathsf{pre}_{\mathrm{out}}(\Gamma, r, i, \phi) = \mathsf{true}$; otherwise, $\mathsf{pre}_{\nu\mathrm{out}}(\Gamma, r, i, \phi) = \mathsf{true}$. By Def. 5.7, $\mathsf{pre}_{\mathrm{atom}}(\Gamma, r, v, i) = \mathsf{true}$. In all cases of action, we also have $\mathsf{pre}_{\mathrm{atom}}(\Gamma, r, v, i) = \mathsf{true}$. Thus, the property holds. □

Lemma 5.1 (Correspondance between available threads and control tokens)**.** *Let $\mathcal{C}_\pi$ be the context of a π-graph π, M_π the corresponding marking $\mathcal{C}_\pi$ of the translated net structure, and t_v^r an atomic-transition. If there exists a thread i such that $\mathsf{avail}_\pi(i, r, v, \mathcal{C}_\pi) = \mathsf{true}$, then $\mathsf{avail}_n(r.i, t_v^r, \mathsf{M}_\pi) = \mathsf{true}$.*

Proof. We prove by analyzing all the cases of the *structural type* of the action at (r, v).

If the action is spwn or mono, then by Def. 5.2 (Eq. 5.3b and Eq. 5.3d), t_v^r belongs to $\mathfrak{S}$ or $\mathfrak{M}$. By Prop. 5.1, place p_0^r is the unique control input place of t_v^r. Moreover, by Def. 5.5, in these two cases, we have $i \in \mathsf{inact}(\Delta_r)$. By Def. 5.3, the control token $r.i$ is present in place p_0^r. Thus, we have $r.i \in \mathsf{M}_\pi(p_0^r)$. By Def. 5.6, we have $\mathsf{avail}_n(r.i, t_v^r, \mathsf{M}_\pi) = \mathsf{true}$.

Otherwise, the action is norm or term then by Def. 5.2 (Eq. 5.3a and Eq. 5.3c), t_v^r belongs to $\mathfrak{N}$ or $\mathfrak{T}$. By Prop. 5.1, p_u^r is the unique control input place of t_v^r, where $(u, v) \in E_r$. Moreover, by Def. 5.5, in these two cases, we have $i \in \Delta_r(u)$. By Def. 5.3, the control token $r.i$ is present in place p_u^r, so, $r.i \in \mathsf{M}_\pi(p_u^r)$. Thus, by Def. 5.6, we have $\mathsf{avail}_n(r.i, t_v^r, \mathsf{M}_\pi) = \mathsf{true}$. In all cases of the action, we always have $\mathsf{avail}_n(r.i, t_v^r, \mathsf{M}_\pi) = \mathsf{true}$. Therefore, the property holds. □

Proposition 5.7 (Equivalence between preconditions and guards for atomic action)**.** *Let Γ_π be the name context of a π-graph π, t_v^r an atomic transition in the translated net such that it is the corresponding transition of the action at (r, v), and i the id of the thread that is available to t_v^r. Then $\mathsf{pre}_{\mathrm{atom}}(\Gamma, r, v, i) = \mathsf{true}$ if and only if $\mathsf{G}(t_v^r)(\Gamma, r, i) = \mathsf{true}$.*

Proof. We show that in any case of the action at (r, v) (input, output, silent), its precondition $\mathsf{pre}_{\mathrm{atom}}(\Gamma, r, v, i)$ is exactly the same as the evaluation of the guard of the corresponding transition, $\mathsf{G}(t_v^r)(\Gamma, r, i)$.

First, we prove the property for two structural types of the action, norm and term. If $L_r(v) = \phi\tau$, then by Def. 5.7, $\mathsf{pre}_{\mathrm{atom}}(\Gamma, r, v, i) = \mathsf{pre}_{\mathrm{tau}}(\Gamma, r, i, \phi)$. By Tab. 5.1 (case $\phi\tau$ for $\mathfrak{N}, \mathfrak{T}$), the evaluation of the guard of t_v^r is $\mathsf{G}(t_v^r)(\Gamma, r, i) = \mathsf{pre}_{\mathrm{tau}}(\Gamma, r, i, \phi)$. Similarly, if $L_r(v) = \phi c(x)$, then by Def. 5.7, $\mathsf{pre}_{\mathrm{atom}}(\Gamma, r, v, i) = \mathsf{pre}_{\mathrm{in}}(\Gamma, r, i, \phi c(x))$. By Tab. 5.1 (case $\phi c(x)$ for $\mathfrak{N}, \mathfrak{T}$), the evaluation of the guard of t_v^r is $\mathsf{pre}_{\mathrm{in}}(\Gamma, r, i, \phi c(x))$. Otherwise, $L_r(v) = \phi\overline{c}\langle a\rangle$, by Def. 5.7, $\mathsf{pre}_{\mathrm{atom}}(\Gamma, r, v, i) = \mathsf{pre}_{\mathrm{out}}(\Gamma, r, i, \phi\overline{c}\langle a\rangle) \vee \mathsf{pre}_{\nu\mathrm{out}}(\Gamma, r, i, \phi\overline{c}\langle a\rangle)$. By Tab. 5.1 (case $\phi\overline{c}\langle a\rangle$ for $\mathfrak{N}, \mathfrak{T}$), the evaluation of the guard of t_v^r is $\mathsf{pre}_{\mathrm{out}}(\Gamma, r, i, \phi\overline{c}\langle a\rangle)$ $\vee\, \mathsf{pre}_{\nu\mathrm{out}}(\Gamma, r, i, \phi\overline{c}\langle a\rangle)$. From the analysis, we always have $\mathsf{pre}_{\mathrm{atom}}(\Gamma, r, v, i) = \mathsf{G}(t_v^r)(\Gamma, r, i)$. Thus, the property holds in the case of the *structural type* of the action at (r, v) norm and term.

Second, we prove for two other structural types of the action, mono and spwn. By Def. 5.7, the precondition $\mathsf{pre}_{\mathrm{atom}}(\Gamma, r, v, i)$ is the same as above, in which Γ is substituted by $\mathsf{init}_i^r(\Gamma)$. Moreover, by Tab. 5.1, the evaluation of the guard is also the same as given in two structural types above, with Γ substituted by $\mathsf{init}_i^r(\Gamma)$. So, $\mathsf{pre}_{\mathrm{atom}}(\Gamma, r, v, i) = \mathsf{G}(t_v^r)(\Gamma, r, i)$. Thus, the property holds in the case of the *structural type* of the action mono and spwn.

From the analysis above, in all the cases of the action and in all the structural types, the evaluation of the precondition of the action at (r, v) is exactly the same as the evaluation of the guard of t_v^r. Thus, the property holds. □

Lemma 5.2 (Existence of the occurrence of an atomic transition)**.** *Let $\mathcal{C}_\pi = (\Gamma, \Delta)$ be a context of a π-graph π. If there exists a context $\mathcal{C}'_\pi$ such that $\mathcal{C}_\pi \xrightarrow[(r,v)]{\mu} \mathcal{C}'_\pi$, then there exists a marking M'_π such that $\mathsf{mark}(\mathcal{C}_\pi)[t_v^r{:}\mu\rangle\mathsf{M}'_\pi$.*

Proof. Let $\mathcal{C}'_\pi$ be a context such that $\mathcal{C}_\pi \xrightarrow[(r,v)]{\mu} \mathcal{C}'_\pi$. By Prop. 5.6, there exists a thread i such that $\mathsf{avail}_\pi(i, r, v, \mathcal{C}_\pi) = \mathsf{true}$ and $\mathsf{pre}_{\mathrm{atom}}(\Gamma, r, v, i) = \mathsf{true}$. Let $\mathsf{M}_\pi = \mathsf{mark}(\mathcal{C}_\pi)$. By

Lem. 5.1, from $\mathsf{avail}_\pi(i, r, v, \mathcal{C}_\pi) = \mathsf{true}$, we have $\mathsf{avail}_n(r.i, t_v^r, \mathsf{M}_\pi) = \mathsf{true}$. Moreover, by Prop. 5.7, from $\mathsf{pre}_{\mathrm{atom}}(\Gamma, r, v, i) = \mathsf{true}$, we have $\mathsf{G}(t_v^r)(\Gamma, r, i) = \mathsf{true}$, and by Def. 5.3, $\mathsf{M}_\pi(p_\Gamma) = \{\Gamma\}$. Thus, $\mathsf{G}(t_v^r)(\mathsf{M}_\pi(p_\Gamma), r, i) = \mathsf{true}$. Furthermore, by Prop. 5.5, both the transition t_v^r and the action at (r, v) have the same label. Thus, there exists a marking M'_π such that we have an occurrence of transition $\mathsf{mark}(\mathcal{C}_\pi)[t_v^r{:}\mu\rangle\mathsf{M}'_\pi$. □

Proposition 5.8 (Marking of the context place)**.** *Let* $\mathcal{C}_\pi = (\Gamma_\pi, \Delta_\pi)$ *and* $\mathcal{C}'_\pi = (\Gamma'_\pi, \Delta'_\pi)$. *If* $\mathcal{C}_\pi \xrightarrow[(r,v)]{\mu} \mathcal{C}'_\pi$ *and there exists* M'_π *such that* $\mathsf{mark}(\mathcal{C}_\pi)[t_v^r{:}\mu\rangle\mathsf{M}'_\pi$, *then* $\mathsf{mark}(\mathcal{C}'_\pi)(p_\Gamma) = \mathsf{M}'_\pi(p_\Gamma)$.

Proof. Remind that if $\mathcal{C}_\pi \to \mathcal{C}'_\pi$ then there exists a marking M'_π such that $\mathsf{mark}(\mathcal{C}_\pi)[t_v^r{:}\mu\rangle\mathsf{M}'_\pi$ (according to Lem. 5.2). We prove that for any *structural type* of an action at (r, v), Γ'_π is exactly the token Γ' that is produced to the context place p_Γ after firing the transition t_v^r.

First, we prove for the structural type norm by analyzing all the cases of the action. If the action is a silent one, then by rule [tau] (cf. p. 72), $\Gamma'_\pi = \mathsf{post}_{\mathrm{tau}}(\Gamma_\pi, r, i, \phi)$. By Def. 5.3, $\mathsf{mark}(\mathcal{C}_\pi)(p_\Gamma) = \{\Gamma'_\pi\}$. Thus, by Tab. 5.1 (case $\mathfrak{N}$, $\phi\tau$), we have $\Gamma' = \mathsf{post}_{\mathrm{tau}}(\Gamma_\pi, r, i, \phi)$. If it is an input action, then by rule [in] (cf. p. 73), $\Gamma'_\pi = \mathsf{post}_{\mathrm{in}}(\Gamma_\pi, r, i, \phi c(x))$. By Def. 5.3, $\mathsf{mark}(\mathcal{C}_\pi)(p_\Gamma) = \{\Gamma'_\pi\}$. Thus, by Tab. 5.1 (case $\mathfrak{N}$, $\phi c(x)$), we have $\Gamma' = \mathsf{post}_{\mathrm{in}}(\Gamma_\pi, r, i, \phi c(x))$. Otherwise, it is an output action, we have two possibilities for Γ'_π:

1. $\Gamma'_\pi = \mathsf{post}_{\mathrm{out}}(\Gamma_\pi, r, i, \dots)$ if $\mathsf{pre}_{\mathrm{out}}(\Gamma_\pi, r, i, \phi\overline{c}\langle a\rangle) = \mathsf{true}$ (by rule [out]);
2. $\Gamma'_\pi = \mathsf{post}_{\nu\mathrm{out}}(\Gamma_\pi, r, i, \phi\overline{c}\langle a\rangle)$ if $\mathsf{pre}_{\nu\mathrm{out}}(\Gamma_\pi, r, i, \phi\overline{c}\langle a\rangle) = \mathsf{true}$ (by rule [νout]).

Moreover, by Def. 5.3, we have $\mathsf{mark}(\mathcal{C}_\pi)(p_\Gamma) = \{\Gamma'_\pi\}$. Thus, by Tab. 5.1 (case $\mathfrak{N}$, $\phi\overline{c}\langle a\rangle$), $\Gamma' = \mathsf{post}_{\mathrm{out}}(\Gamma_\pi, r, i, \phi\overline{c}\langle a\rangle)$ if $\mathsf{pre}_{\mathrm{out}}(\Gamma_\pi, r, i, \phi\overline{c}\langle a\rangle) = \mathsf{true}$; and $\Gamma' = \mathsf{post}_{\nu\mathrm{out}}(\Gamma_\pi, r, i, \phi\overline{c}\langle a\rangle)$ if $\mathsf{pre}_{\nu\mathrm{out}}(\Gamma_\pi, r, i, \phi\overline{c}\langle a\rangle) = \mathsf{true}$.

From the analysis above, and because $\mathsf{pre}_{\mathrm{out}}(\dots)$ and $\mathsf{pre}_{\nu\mathrm{out}}(\dots)$ are exclusive (by Prop. 4.2), thus we have $\Gamma'_\pi = \Gamma'$. Moreover, by Def. 5.3, we have: $\mathsf{mark}(\mathcal{C}'_\pi)(p_\Gamma) = \{\Gamma'_\pi\}$; and by Def. 5.2, we have $\mathsf{M}'_\pi(p_\Gamma) = \Gamma'$. Thus, $\mathsf{mark}(\mathcal{C}'_\pi)(p_\Gamma) = \mathsf{M}'_\pi(p_\Gamma)$.

Second, we prove the property for the other structural types (*i.e.*, spwn, term, and mono) based on the proof for the type norm. In the case of spwn, Γ_π is substituted by $\mathsf{init}_i^r(\Gamma_\pi)$. In the case of term, Γ' is substituted by $\mathsf{reset}_i^r(\Gamma')$. In the case of mono, Γ_π is substituted by $\mathsf{init}_i^r(\Gamma_\pi)$, and Γ' is substituted by $\mathsf{reset}_i^r(\Gamma')$. For all the cases, we always have $\Gamma'_\pi = \Gamma'$. Thus, similarly, we have $\mathsf{M}'_\pi(p_\Gamma) = \Gamma'$. Therefore, $\mathsf{mark}(\mathcal{C}'_\pi)(p_\Gamma) = \mathsf{M}'_\pi(p_\Gamma)$. The property holds. □

Proposition 5.9 (Marking of control places)**.** *Let* $\mathcal{C}_\pi = (\Gamma_\pi, \Delta_\pi)$ *and* $\mathcal{C}'_\pi = (\Gamma'_\pi, \Delta'_\pi)$. *If* $\mathcal{C}_\pi \xrightarrow[r,v]{\mu} \mathcal{C}'_\pi$ *and there exists a marking* M'_π *of* $\mathsf{net}(\pi)$ *such that* $\mathsf{mark}(\mathcal{C}_\pi)[t_v^r{:}\mu\rangle\mathsf{M}'_\pi$, *then for any control place* p *in* $\mathsf{net}(\pi)$, $\mathsf{M}'_\pi(p) = \mathsf{mark}(\mathcal{C}'_\pi)(p)$.

Proof. Let i be the thread that performs the action at (r, v). We prove the property by analyzing all the cases of the *structural types* of the action.

norm: By Def. 4.8, rule [act] is applied for the rewrite and only controls of vertices u, v, where $(u, v) \in E_r$, are updated. We have $\Delta'_r(u) = \Delta_r(u) \setminus \{i\}$, $\Delta'_r(v) = \Delta_r(v) \cup \{i\}$, and $\forall w \notin \{u, v\}, \Delta'_r(w) = \Delta_r(w)$. By Def. 5.3, the corresponding marking $\mathsf{mark}(\mathcal{C}'_\pi)$ is determined as follows:

$$\begin{cases} \mathsf{mark}(\mathcal{C}'_\pi)(p^r_u) = \mathsf{mark}(\mathcal{C}_\pi)(p^r_u) \setminus \{r.i\} \\ \mathsf{mark}(\mathcal{C}'_\pi)(p^r_v) = \mathsf{mark}(\mathcal{C}_\pi)(p^r_v) \cup \{r.i\} \\ \forall p \notin \{p^r_u, p^r_v, p_\Gamma\}, \mathsf{mark}(\mathcal{C}'_\pi)(p) = \mathsf{mark}(\mathcal{C}_\pi)(p) \end{cases} \tag{$\mathfrak{N}1$}$$

Moreover, by Prop. 5.1 and Prop. 5.2, place p^r_u is the unique control input place and p^r_v is the unique control output place of the transition t^r_v. By Prop. 5.4, only the markings of places p^r_u, p^r_v, and p_Γ are updated by the occurrence of the transition. Thus, by Tab. 5.3 (case $\mathfrak{N}$), the marking M'_π is determined as follows:

$$\begin{cases} \mathsf{M}'_\pi(p^r_u) = \mathsf{mark}(\mathcal{C}_\pi)(p^r_u) \setminus \{r.i\} \\ \mathsf{M}'_\pi(p^r_v) = \mathsf{mark}(\mathcal{C}_\pi)(p^r_v) \cup \{r.i\} \\ \forall p \notin \{p^r_u, p^r_v, p_\Gamma\}, \mathsf{M}'_\pi(p) = \mathsf{mark}(\mathcal{C}_\pi)(p) \end{cases} \tag{$\mathfrak{N}2$}$$

From ($\mathfrak{N}1$) and ($\mathfrak{N}2$), we have: $\forall p \neq p_\Gamma, \mathsf{M}'_\pi(p) = \mathsf{mark}(\mathcal{C}'_\pi)(p)$.

spwn: By Def. 4.8, rule [spawn] is applied for the rewrite and only controls of vertex v and set of inactive threads are updated. Let $i = \mathsf{min}(\mathsf{inact}(\Delta_r))$. We have: $\mathsf{inact}(\Delta'_r) = \mathsf{inact}(\Delta_r) \setminus \{i\}$, $\Delta'_r(v) = \Delta_r(v) \cup \{i\}$, and $\forall u \neq v, \Delta'_r(u) = \Delta_r(u)$. Thus, by Def. 5.3, the corresponding marking $\mathsf{mark}(\mathcal{C}'_\pi)$ is determined as follows:

$$\begin{cases} \mathsf{mark}(\mathcal{C}'_\pi)(p^r_0) = \mathsf{mark}(\mathcal{C}_\pi)(p^r_0) \setminus \{r.i\} \\ \mathsf{mark}(\mathcal{C}'_\pi)(p^r_v) = \mathsf{mark}(\mathcal{C}_\pi)(p^r_v) \cup \{r.i\} \\ \forall p \notin \{p^r_0, p^r_v, p_\Gamma\}, \mathsf{mark}(\mathcal{C}'_\pi)(p) = \mathsf{mark}(\mathcal{C}_\pi)(p) \end{cases} \tag{$\mathfrak{S}1$}$$

Moreover, by Prop. 5.1 and Prop. 5.2, p^r_0 is the unique control input place and p^r_v is the unique control output place of the transition t^r_v. By Prop. 5.4, only the markings of places p^r_0, p^r_v, and p_Γ are updated by the occurrence of the transition. Thus, by Tab. 5.3 (case $\mathfrak{S}$), the marking M'_π is determined as follows:

$$\begin{cases} \mathsf{M}'_\pi(p^r_0) = \mathsf{mark}(\mathcal{C}_\pi)(p^r_0) \setminus \{r.i\} \\ \mathsf{M}'_\pi(p^r_v) = \mathsf{mark}(\mathcal{C}_\pi)(p^r_v) \cup \{r.i\} \\ \forall p \notin \{p^r_0, p^r_v, p_\Gamma\}, \mathsf{M}'_\pi(p) = \mathsf{mark}(\mathcal{C}_\pi)(p) \end{cases} \tag{$\mathfrak{S}2$}$$

From ($\mathfrak{S}1$) and ($\mathfrak{S}2$), we have: $\forall p \neq p_\Gamma, \mathsf{M}'_\pi(p) = \mathsf{mark}(\mathcal{C}'_\pi)(p)$.

term: By Def. 4.8, rule [term] is applied for the rewrite and only controls of vertex u, where $(u, v) \in E_r$, and set of inactive threads are updated. We have: $\Delta'_r(u) = \Delta_r(u) \setminus \{i\}, \mathsf{inact}(\Delta'_r) = \mathsf{inact}(\Delta_r) \cup \{i\}$, and $\forall w \neq u, \Delta'_r(w) = \Delta_r(w)$. Thus, by Def. 5.3, the corresponding marking $\mathsf{mark}(\mathcal{C}'_\pi)$ is determined as follows:

$$\begin{cases} \mathsf{mark}(\mathcal{C}'_\pi)(p^r_u) = \mathsf{mark}(\mathcal{C}_\pi)(p^r_u) \setminus \{r.i\} \\ \mathsf{mark}(\mathcal{C}'_\pi)(p^r_0) = \mathsf{mark}(\mathcal{C}_\pi)(p^r_0) \cup \{r.i\} \\ \forall p \notin \{p^r_0, p^r_u, p_\Gamma\}, \mathsf{mark}(\mathcal{C}'_\pi)(p) = \mathsf{mark}(\mathcal{C}_\pi)(p) \end{cases} \quad (\mathfrak{T}1)$$

Moreover, by Prop. 5.1 and Prop. 5.2, p^r_u is the unique control input place and p^r_0 is the unique control output place of t^r_v. By Prop. 5.4, only the markings of places p^r_0, p^r_u, and p_Γ are updated by the occurrence of the transition. Thus, by Tab. 5.3 (case $\mathfrak{T}$), the marking M'_π is determined as follows:

$$\begin{cases} \mathsf{M}'_\pi(p^r_u) = \mathsf{mark}(\mathcal{C}_\pi)(p^r_u) \setminus \{r.i\} \\ \mathsf{M}'_\pi(p^r_0) = \mathsf{mark}(\mathcal{C}_\pi)(p^r_0) \cup \{r.i\} \\ \forall p \notin \{p^r_0, p^r_u, p_\Gamma\}, \mathsf{M}'_\pi(p) = \mathsf{mark}(\mathcal{C}_\pi)(p) \end{cases} \quad (\mathfrak{T}2)$$

From ($\mathfrak{T}1$) and ($\mathfrak{T}2$), we have: $\forall p \neq p_\Gamma, \mathsf{M}'_\pi(p) = \mathsf{mark}(\mathcal{C}'_\pi)(p)$.

mono: By Def. 4.8, rule [mono] is applied for the rewrite and the control context of r does not change, we have: $\mathsf{inact}(\Delta'_r) = \mathsf{inact}(\Delta_r)$, and $\forall v \in V_r, \Delta'_r(v) = \Delta_r(v)$. Thus, by Def. 5.3, the corresponding marking $\mathsf{mark}(\mathcal{C}'_\pi)$ is determined as follows:

$$\forall p \neq p_\Gamma, \mathsf{mark}(\mathcal{C}'_\pi)(p) = \mathsf{mark}(\mathcal{C}_\pi)(p) \quad (\mathfrak{M}1)$$

Moreover, by Prop. 5.1 and Prop. 5.2, p^r_0 is the unique control input place and control output place of t^r_v. By Prop. 5.4, only the markings of p^r_0 and p_Γ are updated by the occurrence of the transition. Thus, by Tab. 5.3 (case $\mathfrak{M}$), the marking M'_π is determined as follows:

$$\forall p \neq p_\Gamma, \mathsf{M}'_\pi(p) = \mathsf{mark}(\mathcal{C}_\pi)(p) \quad (\mathfrak{M}2)$$

From ($\mathfrak{M}1$) and ($\mathfrak{M}2$), we have: $\forall p \neq p_\Gamma, \mathsf{M}'_\pi(p) = \mathsf{mark}(\mathcal{C}'_\pi)(p)$.

From the analysis above, for any structural type of the action and for any control place p, we always have $\mathsf{M}'_\pi(p) = \mathsf{mark}(\mathcal{C}'_\pi)(p)$. The property is proved. □

Lemma 5.3 (Equal markings)**.** *Let $\mathcal{C}_\pi$ and $\mathcal{C}'_\pi$ be contexts of a π-graph π. If $\mathcal{C}_\pi \xrightarrow[r,v]{\mu} \mathcal{C}'_\pi$ and there exists a marking M'_π of $\mathsf{net}(\pi)$ such that $\mathsf{mark}(\mathcal{C}_\pi)[t^r_v{:}\mu\rangle\mathsf{M}'_\pi$, then $\mathsf{M}'_\pi = \mathsf{mark}(\mathcal{C}'_\pi)$.*

Proof. We prove that for any place p in the translated net, the marking $\mathsf{M}'_\pi(p)$ is exactly the same as the marking $\mathsf{mark}(\mathcal{C}'_\pi)(p)$.

By Def. 5.2 (Eq. 5.1), the place p is either a context place (i.e. $p = p_\Gamma$) or a control place. If $p = p_\Gamma$, then by Prop. 5.8, we have $\mathsf{mark}(\mathcal{C}'_\pi)(p) = \mathsf{M}'_\pi(p)$. Otherwise, by Prop. 5.9, we also have $\mathsf{mark}(\mathcal{C}'_\pi)(p) = \mathsf{M}'_\pi(p)$. In both cases of the place p, we have $\mathsf{mark}(\mathcal{C}'_\pi)(p) = \mathsf{M}'_\pi(p)$. Thus, the property is proved. □

Theorem 5.1 (Soundness for an atomic action)**.** *Let $\mathcal{C}_\pi$ be a context of π. For any $\mu, \mathcal{C}'_\pi, r, v$ such that $\mathcal{C}_\pi \xrightarrow[(r,v)]{\mu} \mathcal{C}'_\pi$, there exists a marking M'_π of $\mathsf{net}(\pi)$ such that $\mathsf{mark}(\mathcal{C}_\pi)[t^r_v{:}\mu\rangle\mathsf{M}'_\pi$ and $\mathsf{M}'_\pi = \mathsf{mark}(\mathcal{C}'_\pi)$.*

Proof. By Lem. 5.2, if there exists a rewrite $\mathcal{C}_\pi \xrightarrow[r,v]{\mu} \mathcal{C}'_\pi$ then there exists an occurrence $\mathsf{mark}(\mathcal{C}_\pi)[t^r_v{:}\mu\rangle\mathsf{M}'_\pi$. Moreover, by Lem. 5.3, we have $\mathsf{M}'_\pi = \mathsf{mark}(\mathcal{C}'_\pi)$. Thus, the property holds. □

5.2.1.2 Soundness for synchronizations

Similarly to the soundness for an atomic action, the proof of the soundness for a synchronization (Theorem 5.2, denoted by T2) is structured as shown in Figure 5.8.

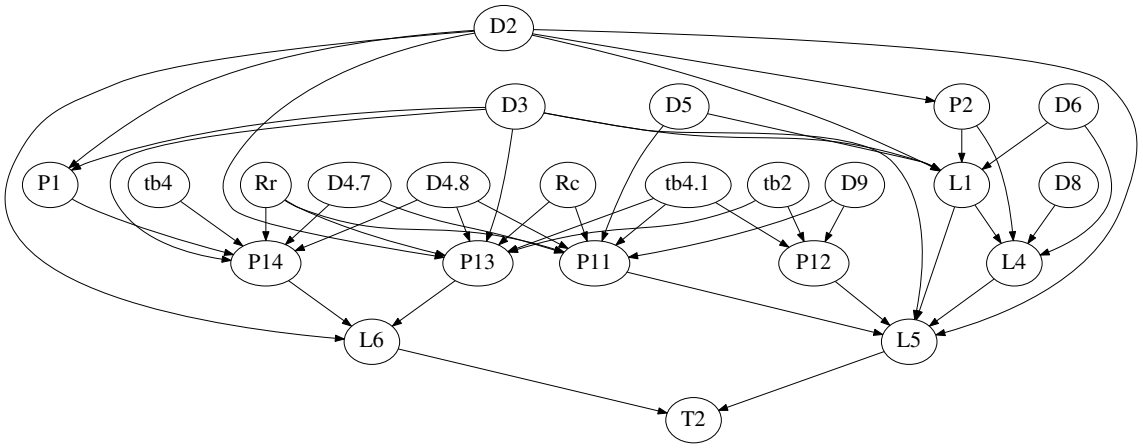


Figure 5.8: Dependence graph of the soundness for a synchronization

Proposition 5.10 (Correspondance between syncs and sync-transitions)**.** *For any pair of an output at (r_o, v_o) and an input at (r_i, v_i) in a π-graph π, there exists a corresponding sync-transition $t^{r_o r_i}_{v_o v_i}$ in $\mathsf{net}(\pi)$ with label τ.*

Proof. By Def. 5.2 (Eq. 5.3), for any structural type of the actions at $(r_\mathrm{o}, v_\mathrm{o})$ and at $(r_\mathrm{i}, v_\mathrm{i})$ in π, there exist corresponding transitions $t^{r_\mathrm{o}}_{v_\mathrm{o}}$ and $t^{r_\mathrm{i}}_{v_\mathrm{i}}$ in $\mathsf{net}(\pi)$. Moreover, by Def. 5.2 (Eq. 5.4), for any two such transitions which correspond to an output and an

input action, there exists a corresponding sync-transition. In this case, there exists a sync-transition $t^{r_o r_i}_{v_o v_i} \in \mathsf{T}_2$. In the other hand, since $t^{r_o r_i}_{v_o v_i} \in \mathsf{T}_2$, thus, by Def. 5.2 (Eq. 5.5), its label is $\boldsymbol{\tau}$. □

Definition 5.8 (Available control tokens for a sync transition)**.** *Let* M_π *be the marking of a translated net* $\mathsf{net}(\pi)$ *of a* π*-graph* π*. Two control tokens* $r_1.i_1$ *and* $r_2.i_2$ *are available for a sync-transition* $t^{r_1 r_2}_{v_1 v_2}$ *if* $\mathsf{avail}^{\oplus}_n(t^{r_1 r_2}_{v_1 v_2}, r_1.i_1, r_2.i_2, \mathsf{M}_\pi)$ *is* true*, with:*

$$\mathsf{avail}^{\oplus}_n(t^{r_1 r_2}_{v_1 v_2}, r_1.i_1, r_2.i_2, \mathsf{M}_\pi) \stackrel{\mathsf{def}}{=} r_1.i_1 \in \mathsf{M}_\pi(\mathsf{cip}(t^{r_1}_{v_1})) \ \wedge \ r_2.i_2 \in \mathsf{M}_\pi(\mathsf{cip}(t^{r_2}_{v_2})),$$

where $\mathsf{cip}(t^r_v)$ *(resp.* $\mathsf{cop}(t^r_v)$*) is the control input (resp. output) place of the atomic transition* t^r_v*.*

Definition 5.9 (Precondition of a sync)**.** *Let* i_o *be a thread that performs an output* $\phi\overline{c}\langle a\rangle$ *at* (r_o, v_o)*, and* i_i *a thread that performs an input* $\psi d(x)$ *at* (r_i, v_i)*. The precondition of a synchronization between these two actions is defined as follows:*

$$\mathsf{pre}_{\mathrm{sync}}(\Gamma, r_o, v_o, i_o, r_i, v_i, i_i) \stackrel{\mathsf{def}}{=} \mathsf{pre}_{\mathrm{com}}(\Gamma, r_o, i_o, \phi\overline{c}\langle a\rangle, r_i, i_i, \psi d(x)).$$

If the structural type *of the actions at* (r_o, v_o) *or* (r_i, v_i) *is* mono *or* spwn*, then* Γ *is replaced as in column* Initialization *of Tab. 4.1 on page 80.*

Proposition 5.11 (Available threads for a sync)**.** *For any sync rewrite* $\mathcal{C}_\pi \xrightarrow[(r_o,v_o),(r_i,v_i)]{\mu} \mathcal{C}'_\pi$*, there exist threads* i_o *in* $[1..\mathcal{K}(r_o)]$ *and* i_i *in* $[1..\mathcal{K}(r_i)]$ *such that:*

$$\mathsf{avail}_\pi(i_o, r_o, v_o, \mathcal{C}_\pi) \wedge \mathsf{avail}_\pi(i_i, r_i, v_i, \mathcal{C}_\pi) \wedge \mathsf{pre}_{\mathrm{sync}}(\Gamma, r_o, v_o, i_o, r_i, v_i, i_i) = \mathsf{true}$$

Proof. Because the rewrite is triggered by a synchronization, by Def. 4.8 on page 81, rule [sync] is applied.

First, we prove that $\mathsf{avail}_\pi(i_\mathrm{o}, r_\mathrm{o}, v_\mathrm{o}, \mathcal{C}_\pi) = \mathsf{true}$ and $\mathsf{avail}_\pi(i_\mathrm{i}, r_\mathrm{i}, v_\mathrm{i}, \mathcal{C}_\pi) = \mathsf{true}$. By the rule [sync] (cf. p. 77), there exist $i_\mathrm{o} \in [1..\mathcal{K}(r_\mathrm{o})]$ and $i_\mathrm{i} \in [1..\mathcal{K}(r_\mathrm{i})]$ such that the thread i_o (resp. i_i) is used for performing the action at $(r_\mathrm{o}, v_\mathrm{o})$ (resp. $(r_\mathrm{i}, v_\mathrm{i})$). If the structural type of the action at $(r_\mathrm{o}, v_\mathrm{o})$ is mono or spwn (*i.e.*, $\mathsf{deg}^-_{r_\mathrm{o}}(v_\mathrm{o}) = 0$, by Def. 4.7), then $i_\mathrm{o} \in \mathsf{inact}(\Delta_{r_\mathrm{o}})$; otherwise, *i.e.*, there exists u_o such that $(u_\mathrm{o}, v_\mathrm{o}) \in E_{r_\mathrm{o}}$ (by def. 4.7), then $i_\mathrm{o} \in \Delta_{r_\mathrm{o}}(u_\mathrm{o})$. Thus, by Def. 5.5, we have $\mathsf{avail}_\pi(i_\mathrm{o}, r_\mathrm{o}, v_\mathrm{o}, \mathcal{C}_\pi) = \mathsf{true}$. Similarly, we also have $\mathsf{avail}_\pi(i_\mathrm{i}, r_\mathrm{i}, v_\mathrm{i}, \mathcal{C}_\pi) = \mathsf{true}$.

Second, we prove that $\mathsf{pre}_{\mathrm{sync}}(\Gamma, r_\mathrm{o}, v_\mathrm{o}, i_\mathrm{o}, r_\mathrm{i}, v_\mathrm{i}, i_\mathrm{i}) = \mathsf{true}$. By rule [sync], we also have a commitment for a communication between these two atomic actions. If both of them are norm actions, then by rule [com] (cf. p. 75), we have $\mathsf{pre}_{\mathrm{com}}(\Gamma, r_\mathrm{o}, i_\mathrm{o}, \phi\overline{c}\langle a\rangle, r_\mathrm{i}, i_\mathrm{i}, \psi d(x))$ is true, where $\phi\overline{c}\langle a\rangle$ is the output at $(r_\mathrm{o}, v_\mathrm{o})$ and $\psi d(x)$ is the input at $(r_\mathrm{i}, v_\mathrm{i})$. Otherwise, Γ is replaced as in column *Initialization* of Tab. 4.1 on page 80. Thus, by Def. 5.9, we have $\mathsf{pre}_{\mathrm{sync}}(\Gamma, r_\mathrm{o}, v_\mathrm{o}, i_\mathrm{o}, r_\mathrm{i}, v_\mathrm{i}, i_\mathrm{i}) = \mathsf{true}$. □

Lemma 5.4. *Let $\mathcal{C}_\pi$ be a context of a π-graph π and M_π the corresponding marking of $\mathcal{C}_\pi$ in $\mathsf{net}(\pi)$. If $L_{r_1}(v_1) = \phi\overline{c}\langle a\rangle$ and $L_{r_2}(v_2) = \psi c(x)$, then*

$$\mathsf{avail}_\pi(i_1, r_1, v_1, \mathcal{C}_\pi) \ \wedge \ \mathsf{avail}_\pi(i_2, r_2, v_2, \mathcal{C}_\pi) \Rightarrow \mathsf{avail}_n^{\oplus}(t_{v_1v_2}^{r_1r_2}, r_1.i_1, r_2.i_2, \mathsf{M}_\pi).$$

Proof. Suppose that $\mathsf{avail}_\pi(i_1, r_1, v_1, \mathcal{C}_\pi) = \mathsf{true}$ and $\mathsf{avail}_\pi(i_2, r_2, v_2, \mathcal{C}_\pi) = \mathsf{true}$, we prove that $\mathsf{avail}_n^{\oplus}(t_{v_1v_2}^{r_1r_2}, r_1.i_1, r_2.i_2, \mathsf{M}_\pi) = \mathsf{true}$.

By Lem. 5.1, $\mathsf{avail}_\pi(i_1, r_1, v_1, \mathcal{C}_\pi) \Rightarrow \mathsf{avail}_n(r_1.i_1, t_{v_1}^{r_1}, \mathsf{M}_\pi)$. Thus, $\mathsf{avail}_n(r_1.i_1, t_{v_1}^{r_1}, \mathsf{M}_\pi) = \mathsf{true}$. By Prop. 5.1, the atomic-transition $t_{v_1}^{r_1}$ has a unique control input place, $p_1 = \mathsf{cip}(t_{v_1}^{r_1})$. By Def. 5.6, we have: $r_1.i_1 \in \mathsf{M}_\pi(p_1)$. Similarly, let $p_2 = \mathsf{cip}(t_{v_2}^{r_2})$, we also have: $r_2.i_2 \in \mathsf{M}_\pi(p_2)$. Thus, by Def. 5.8, $\mathsf{avail}_n^{\oplus}(t_{v_1v_2}^{r_1r_2}, r_1.i_1, r_2.i_2, \mathsf{M}_\pi) = \mathsf{true}$. □

Proposition 5.12. *We have:*

$$\mathsf{pre}_{\mathrm{sync}}(\Gamma, r_o, v_o, i_o, r_i, v_i, i_i) \Leftrightarrow \mathsf{G}(t_{v_ov_i}^{r_or_i})(\Gamma, r_o, i_o, r_i, i_i)$$

Proof. We prove that for all the types of actions at $(r_{\mathrm{o}}, v_{\mathrm{o}})$ and $(r_{\mathrm{i}}, v_{\mathrm{i}})$, the precondition $\mathsf{pre}_{\mathrm{sync}}(\Gamma, r_{\mathrm{o}}, v_{\mathrm{o}}, i_{\mathrm{o}}, r_{\mathrm{i}}, v_{\mathrm{i}}, i_{\mathrm{i}})$ is exactly the same as the evaluation of the corresponding guard $\mathsf{G}(t_{v_{\mathrm{o}}v_{\mathrm{i}}}^{r_{\mathrm{o}}r_{\mathrm{i}}})(\Gamma, r_{\mathrm{o}}, i_{\mathrm{o}}, r_{\mathrm{i}}, i_{\mathrm{i}})$.

First, we prove the property for the case in which the structural type of both actions is norm. By Def. 5.9,

$$\mathsf{pre}_{\mathrm{sync}}(\Gamma, r_{\mathrm{o}}, v_{\mathrm{o}}, i_{\mathrm{o}}, r_{\mathrm{i}}, v_{\mathrm{i}}, i_{\mathrm{i}}) = \mathsf{pre}_{\mathrm{com}}(\Gamma, r_{\mathrm{o}}, i_{\mathrm{o}}, \phi\overline{c}\langle a\rangle, r_{\mathrm{i}}, i_{\mathrm{i}}, \psi d(x)).$$

Moreover, by Tab. 5.2 on page 92 (case $\mathfrak{N} - \mathfrak{N}$),

$$\mathsf{G}(t_{v_{\mathrm{o}}v_{\mathrm{i}}}^{r_{\mathrm{o}}r_{\mathrm{i}}})(\Gamma, r_{\mathrm{o}}, i_{\mathrm{o}}, r_{\mathrm{i}}, i_{\mathrm{i}}) = \mathsf{pre}_{\mathrm{com}}(\Gamma, r_{\mathrm{o}}, i_{\mathrm{o}}, \phi\overline{c}\langle a\rangle, r_{\mathrm{i}}, i_{\mathrm{i}}, \psi d(x)).$$

Thus, we have $\mathsf{pre}_{\mathrm{sync}}(\Gamma, r_{\mathrm{o}}, v_{\mathrm{o}}, i_{\mathrm{o}}, r_{\mathrm{i}}, v_{\mathrm{i}}, i_{\mathrm{i}}) = \mathsf{G}(t_{v_{\mathrm{o}}v_{\mathrm{i}}}^{r_{\mathrm{o}}r_{\mathrm{i}}})(\Gamma, r_{\mathrm{o}}, i_{\mathrm{o}}, r_{\mathrm{i}}, i_{\mathrm{i}})$.

Second, we prove the proposition for all the other structural types of these actions. By Def. 5.9, the substitution for Γ is given in column Initialization of Tab. 4.1 on page 80, which is exactly the substitution for the guard in Tab. 5.2. Thus, the precondition and the evaluation of the corresponding guard are the same. □

Lemma 5.5 (Existence of an occurrence of a sync-transition)**.** *Let $\mathcal{C}_\pi$ be a context of a π-graph π. If $\mathcal{C}_\pi \xrightarrow[(r_1,v_1),(r_2,v_2)]{\tau} \mathcal{C}'_\pi$, then there exists M'_π such that $\mathsf{mark}(\mathcal{C}_\pi)[t_{v_1v_2}^{r_1r_2}{:}\boldsymbol{\tau}\rangle\mathsf{M}'_\pi$.*

Proof. By Prop. 5.11 on the facing page, there exist $i_1 \in [1..\mathcal{K}(r_1)]$ and $i_2 \in [1..\mathcal{K}(r_2)]$ such that

$$\mathsf{avail}_\pi(i_1, r_1, v_1, \mathcal{C}_\pi) \wedge \mathsf{avail}_\pi(i_2, r_2, v_2, \mathcal{C}_\pi) \wedge \mathsf{pre}_{\mathrm{sync}}(\Gamma, r_1, v_1, i_1, r_2, v_2, i_2) = \mathsf{true}.$$

Let $\mathsf{M}_\pi = \mathsf{mark}(\mathcal{C}_\pi)$. By Lem. 5.1 on page 98, $\mathsf{avail}_\pi(i_1, r_1, v_1, \mathcal{C}_\pi) \Rightarrow \mathsf{avail}_n(r_1.i_1, t^{r_1}_{v_1}, \mathsf{M}_\pi)$ and $\mathsf{avail}_\pi(i_2, r_2, v_2, \mathcal{C}_\pi) \Rightarrow \mathsf{avail}_n(r_2.i_2, t^{r_2}_{v_2}, \mathsf{M}_\pi)$, thus, we have $\mathsf{avail}_n(r_1.i_1, t^{r_1}_{v_1}, \mathsf{M}_\pi) = \mathsf{true}$ and $\mathsf{avail}_n(r_2.i_2, t^{r_2}_{v_2}, \mathsf{M}_\pi) = \mathsf{true}$. By Lem. 5.4 on the preceding page, we have:

$$\mathsf{avail}^{\oplus}_n(t^{r_1 r_2}_{v_1 v_2}, r_1.i_1, r_2.i_2, \mathsf{M}_\pi) = \mathsf{true}. \tag{1}$$

Moreover, by Def. 5.3 on page 94, $\mathsf{M}_\pi(p_\Gamma) = \{\Gamma\}$. Thus, by Prop. 5.12 on the preceding page, from the precondition $\mathsf{pre}_{\mathrm{sync}}(\Gamma, r_1, v_1, i_1, r_2, v_2, i_2) = \mathsf{true}$, we have:

$$\mathsf{G}(t^{r_1 r_2}_{v_1 v_2})(\mathsf{M}_\pi(p_\Gamma), r_1, i_1, r_2, i_2) = \mathsf{true}. \tag{2}$$

Thus, from (1), (2) and $\mathsf{U}(t^{r_1 r_2}_{v_1 v_2}) = \tau$ (by Def. 5.2 on page 88, as $t^{r_1 r_2}_{v_1 v_2} \in \mathsf{T}_2$), there exists a marking M'_π such that $\mathsf{mark}(\mathcal{C}_\pi)[t^{r_1 r_2}_{v_1 v_2}{:}\boldsymbol{\tau}\rangle\mathsf{M}'_\pi$. □

Proposition 5.13. *Let $\mathcal{C}_\pi = (\Gamma_\pi, \Delta_\pi)$ and $\mathcal{C}'_\pi = (\Gamma'_\pi, \Delta'_\pi)$. If $\mathcal{C}_\pi \xrightarrow[(r_1,v_1),(r_2,v_2)]{\tau} \mathcal{C}'_\pi$ and there exists M'_π such that $\mathsf{mark}(\mathcal{C}_\pi)[t^{r_1 r_2}_{v_1 v_2}{:}\tau\rangle\mathsf{M}'_\pi$, then $\mathsf{M}'_\pi(p_\Gamma) = \mathsf{mark}(\mathcal{C}'_\pi)(p_\Gamma)$.*

Proof. Let $\phi\overline{c}\langle a\rangle$ be the output at (r_1, v_1), $\psi d(x)$ the input at (r_2, v_2), and i_1 (resp. i_2) the thread that performs the output (resp. input). By Def. 5.3, $\mathsf{mark}(\mathcal{C}'_\pi)(p_\Gamma) = \{\Gamma'_\pi\}$. Moreover, by Def. 5.2, $\mathsf{M}'_\pi(p_\Gamma) = \{\Gamma'\}$ (Γ' is the token that is produced to the context place p_Γ after firing the transition $t^{r_1 r_2}_{v_1 v_2}$). Thus, we prove that for any structural type of the input and output, $\Gamma'_\pi = \Gamma'$.

First, we prove the property for the case $\mathsf{norm} - \mathsf{norm}$, in which the structural types of both input and output are norm. Because the rewrite is triggered by a synchronization, thus, by Def. 4.8, rule [sync] (cf. p. 77) is applied. By this rule, the context Γ'_π is given by rule [com] (cf. p. 75), $\Gamma'_\pi = \mathsf{post}_{\mathrm{com}}(\Gamma_\pi, r_1, i_1, \phi\overline{c}\langle a\rangle, r_2, i_2, \psi d(x))$. By Def. 5.3, $\mathsf{mark}(\mathcal{C}_\pi)(p_\Gamma) = \{\Gamma_\pi\}$, thus, by Tab. 5.2 on page 92 (case 1), we have: $\Gamma' = \mathsf{post}_{\mathrm{com}}(\Gamma_\pi, r_1, i_1, \phi\overline{c}\langle a\rangle, r_2, i_2, \psi d(x))$. So, $\Gamma'_\pi = \Gamma'$, the property holds.

Second, we prove the property for all the other cases based on the case $\mathsf{norm} - \mathsf{norm}$. In these cases, Γ_π (resp. Γ'_π) is substituted as in column *Initialization* (resp. *Resets*) of Tab. 4.1 on page 80, which is the same way for computing Γ' in Tab. 5.2 on page 92. Thus, $\Gamma'_\pi = \Gamma'$, and the property holds. □

Proposition 5.14. *Let $\mathcal{C}_\pi = (\Gamma_\pi, \Delta_\pi)$ and $\mathcal{C}'_\pi = (\Gamma'_\pi, \Delta'_\pi)$ be two contexts of a π-graph π. If $\mathcal{C}_\pi \xrightarrow[(r_1,v_1),(r_2,v_2)]{\tau} \mathcal{C}'_\pi$ and there exists M'_π such that $\mathsf{mark}(\mathcal{C}_\pi)[t^{r_1 r_2}_{v_1 v_2}{:}\boldsymbol{\tau}\rangle\mathsf{M}'_\pi$, then for any* control place *p in $\mathsf{net}(\pi)$, we have $\mathsf{M}'_\pi(p) = \mathsf{mark}(\mathcal{C}'_\pi)(p)$.*

Proof. By Def. 4.7 on page 79, each atomic action can be one of four kinds: norm, spwn, term, or mono. Because of the symmetry and the similarity between the cases, we develop the 4 basic cases as follows. Suppose that i_1 (resp. i_2) is the thread that performs the action at (r_1, v_1) (resp. (r_2, v_2)).

$\mathsf{norm} - *$: First, we prove for the case $\mathsf{norm} - \mathsf{norm}$, where both the output an the input are norm actions. By Def. 4.7 on page 79, there exist vertices u_1, w_1, u_2, and w_2 such that:

$$\begin{cases} (u_1, v_1) \in E_{r_1}, (v_1, w_1) \in E_{r_1} \\ (u_2, v_2) \in E_{r_2}, (v_2, w_2) \in E_{r_2} \end{cases}$$

1. Case $u_1 \neq u_2$: By rule [sync] (cf. p. 77) and Def. 4.8 on page 81,

$$\begin{cases} \Delta'_{r_1}(u_1) = \Delta_{r_1}(u_1) \setminus \{i_1\}, \Delta'_{r_1}(v_1) = \Delta_{r_1}(v_1) \cup \{i_1\} \\ \Delta'_{r_2}(u_2) = \Delta_{r_2}(u_2) \setminus \{i_2\}, \Delta'_{r_2}(v_2) = \Delta_{r_2}(v_2) \cup \{i_2\} \\ \forall v \notin \{u_1, v_1, u_2, v_2\} : \Delta'_{r_1}(v) = \Delta_{r_1}(v), \Delta'_{r_2}(v) = \Delta_{r_2}(v) \end{cases}$$

   By Def. 5.3 on page 94,

$$\begin{cases} \mathsf{mark}(\mathcal{C}'_\pi)(p^{r_1}_{u_1}) = \mathsf{mark}(\mathcal{C}_\pi)(p^{r_1}_{u_1}) \setminus \{r_1.i_1\} \\ \mathsf{mark}(\mathcal{C}'_\pi)(p^{r_1}_{v_1}) = \mathsf{mark}(\mathcal{C}_\pi)(p^{r_1}_{v_1}) \cup \{r_1.i_1\} \\ \mathsf{mark}(\mathcal{C}'_\pi)(p^{r_2}_{u_2}) = \mathsf{mark}(\mathcal{C}_\pi)(p^{r_2}_{u_2}) \setminus \{r_2.i_2\} \\ \mathsf{mark}(\mathcal{C}'_\pi)(p^{r_2}_{v_2}) = \mathsf{mark}(\mathcal{C}_\pi)(p^{r_2}_{v_2}) \cup \{r_2.i_2\} \\ \forall p \notin \{p^{r_1}_{u_1}, p^{r_1}_{v_1}, p^{r_2}_{u_2}, p^{r_2}_{v_2}, p_\Gamma\} : \mathsf{mark}(\mathcal{C}'_\pi)(p) = \mathsf{mark}(\mathcal{C}_\pi)(p) \end{cases} \quad (\mathfrak{MM}1.1)$$

   Moreover, by Tab. 5.4 on page 118 (case 1) and Prop. 5.4 on page 95 we have:

$$\begin{cases} \mathsf{M}'_\pi(p^{r_1}_{u_1}) = \mathsf{mark}(\mathcal{C}_\pi)(p^{r_1}_{u_1}) \setminus \{r_1.i_1\} \\ \mathsf{M}'_\pi(p^{r_1}_{v_1}) = \mathsf{mark}(\mathcal{C}_\pi)(p^{r_1}_{v_1}) \cup \{r_1.i_1\} \\ \mathsf{M}'_\pi(p^{r_2}_{u_2}) = \mathsf{mark}(\mathcal{C}_\pi)(p^{r_2}_{u_2}) \setminus \{r_2.i_2\} \\ \mathsf{M}'_\pi(p^{r_2}_{v_2}) = \mathsf{mark}(\mathcal{C}_\pi)(p^{r_2}_{v_2}) \cup \{r_2.i_2\} \\ \forall p \notin \{p^{r_1}_{u_1}, p^{r_1}_{v_1}, p^{r_2}_{u_2}, p^{r_2}_{v_2}, p_\Gamma\} : \mathsf{M}'_\pi(p) = \mathsf{mark}(\mathcal{C}_\pi)(p) \end{cases} \quad (\mathfrak{MM}1.2)$$

   From $(\mathfrak{MM}1.1)$ and $(\mathfrak{MM}1.2)$, we have: $\forall p \neq p_\Gamma, \mathsf{M}'_\pi(p) = \mathsf{mark}(\mathcal{C}'_\pi)(p)$.

2. Case $u_1 = u_2 = u$: In this case, $r_1 = r_2 = r$. By rule [sync] and Def. 4.8 on page 81,

$$\begin{cases} \Delta'_r(u) = \Delta_r(u) \setminus \{i_1, i_2\} \\ \Delta'_r(v_1) = \Delta_r(v_1) \cup \{i_1\}, \Delta'_r(v_2) = \Delta_r(v_2) \cup \{i_2\} \\ \forall v \notin \{u, v_1, v_2\} : \Delta'_r(v) = \Delta_r(v) \end{cases}$$

   By Def. 5.3 on page 94,

$$\begin{cases} \mathsf{mark}(\mathcal{C}'_\pi)(p^r_u) = \mathsf{mark}(\mathcal{C}_\pi)(p^r_u) \setminus \{r.i_1, r.i_2\} \\ \mathsf{mark}(\mathcal{C}'_\pi)(p^r_{v_1}) = \mathsf{mark}(\mathcal{C}_\pi)(p^r_{v_1}) \cup \{r.i_1\} \\ \mathsf{mark}(\mathcal{C}'_\pi)(p^r_{v_2}) = \mathsf{mark}(\mathcal{C}_\pi)(p^r_{v_2}) \cup \{r.i_2\} \\ \forall p \notin \{p^r_u, p^r_{v_1}, p^r_{v_2}, p_\Gamma\} : \mathsf{mark}(\mathcal{C}'_\pi)(p) = \mathsf{mark}(\mathcal{C}_\pi)(p) \end{cases} \quad (\mathfrak{MM}2.1)$$

Moreover, by Tab. 5.4 on page 118 (case 2) and Prop. 5.4 on page 95 we have:

$$\begin{cases} \mathsf{M}'_\pi(p^r_u) = \mathsf{mark}(\mathcal{C}_\pi)(p^r_u) \setminus \{r.i_1, r.i_2\} \\ \mathsf{M}'_\pi(p^r_{v_1}) = \mathsf{mark}(\mathcal{C}_\pi)(p^r_{v_1}) \cup \{r.i_1\} \\ \mathsf{M}'_\pi(p^r_{v_2}) = \mathsf{mark}(\mathcal{C}_\pi)(p^r_{v_2}) \cup \{r.i_2\} \\ \forall p \notin \{p^r_u, p^r_{v_1}, p^r_{v_2}, p_\Gamma\} : \mathsf{M}'_\pi(p) = \mathsf{mark}(\mathcal{C}_\pi)(p) \end{cases} \quad (\mathfrak{MM}2.2)$$

From ($\mathfrak{MM}2.1$) and ($\mathfrak{MM}2.2$), we have: $\forall p \neq p_\Gamma, \mathsf{M}'_\pi(p) = \mathsf{mark}(\mathcal{C}'_\pi)(p)$.

Second, all the other cases, *i.e.*, $\mathsf{norm} - \mathsf{spwn}$, $\mathsf{norm} - \mathsf{term}$ and $\mathsf{norm} - \mathsf{mono}$, are similar to the case $\mathsf{norm} - \mathsf{norm}$. In these cases, the move of the thread used for the output is the same as the case $\mathsf{norm} - \mathsf{norm}$, the thread used for the input is similar.

$\mathsf{spwn} - *$: First, we prove the property for the case $\mathsf{spwn} - \mathsf{spwn}$ by analyzing two subcases, in which two replicators are either different or not.

1. $r_1 \neq r_2$: By rule [sync] (cf. p. 77) and Def. 4.8, we have:

$$\begin{cases} \mathsf{inact}(\Delta'_{r_1}) = \mathsf{inact}(\Delta_{r_1}) \setminus \{i_1\}; \Delta'_{r_1}(v_1) = \Delta_{r_1}(v_1) \cup \{i_1\} \\ \mathsf{inact}(\Delta'_{r_2}) = \mathsf{inact}(\Delta_{r_2}) \setminus \{i_2\}; \Delta'_{r_2}(v_2) = \Delta_{r_2}(v_2) \cup \{i_2\} \\ \forall v \notin \{v_1, v_2\} : \Delta'_{r_1}(v) = \Delta_{r_1}(v), \Delta'_{r_2}(v) = \Delta_{r_2}(v) \end{cases} \quad (\mathfrak{SS}1)$$

2. $r_1 = r_2 = r$: Similarly, we have:

$$\begin{cases} \mathsf{inact}(\Delta'_r) = \mathsf{inact}(\Delta_r) \setminus \{i_1, i_2\} \\ \Delta'_r(v_1) = \Delta_r(v_1) \cup \{i_1\}; \Delta'_r(v_2) = \Delta_r(v_2) \cup \{i_2\} \\ \forall v \notin \{v_1, v_2\} : \Delta'_r(v) = \Delta_r(v) \end{cases} \quad (\mathfrak{SS}2)$$

From ($\mathfrak{SS}1$) and ($\mathfrak{SS}2$), similarly to the proof for the case $\mathsf{norm} - \mathsf{norm}$, we also have: $\forall p \neq p_\Gamma, \mathsf{M}'_\pi(p) = \mathsf{mark}(\mathcal{C}'_\pi)(p)$. The property holds.

The other cases, *i.e.*, $\mathsf{spwn} - \mathsf{term}$ and $\mathsf{spwn} - \mathsf{mono}$, are similar to the case $\mathsf{spwn} - \mathsf{spwn}$.

$\mathsf{term} - *$: First, we prove the property for the case $\mathsf{term} - \mathsf{term}$. By Def. 4.7 on page 79, there exist vertices u_1 and u_2 such that $(u_1, v_1) \in E_{r_1}$ and $(u_2, v_2) \in E_{r_2}$. We analyze two cases concerning the replicators.

1. $r_1 \neq r_2$: We have:

$$\begin{cases} \mathsf{inact}(\Delta'_{r_1}) = \mathsf{inact}(\Delta_{r_1}) \cup \{i_1\}; \Delta'_{r_1}(u_1) = \Delta_{r_1}(u_1) \setminus \{i_1\} \\ \mathsf{inact}(\Delta'_{r_2}) = \mathsf{inact}(\Delta_{r_2}) \cup \{i_2\}; \Delta'_{r_2}(u_2) = \Delta_{r_2}(u_2) \setminus \{i_2\} \\ \forall v \notin \{u_1, u_2\} : \Delta'_{r_1}(v) = \Delta_{r_1}(v), \Delta'_{r_2}(v) = \Delta_{r_2}(v) \end{cases} \quad (\mathfrak{TT}1)$$

2. $r_1 = r_2 = r$: We have:

$$\begin{cases} \mathsf{inact}(\Delta'_r) = \mathsf{inact}(\Delta_r) \cup \{i_1, i_2\} \\ \Delta'_{r_1}(u_1) = \Delta_{r_1}(u_1) \setminus \{i_1\}; \Delta'_{r_2}(u_2) = \Delta_{r_2}(u_2) \setminus \{i_2\} \\ \forall v \notin \{u_1, u_2\} : \Delta'_{r_1}(v) = \Delta_{r_1}(v), \Delta'_{r_2}(v) = \Delta_{r_2}(v) \end{cases} \quad (\mathfrak{TT}2)$$

From ($\mathfrak{TT}1$) and ($\mathfrak{TT}2$), similarly, the property holds. For the remaining case, i.e. $\mathsf{term} - \mathsf{mono}$, the proof is similar to the case $\mathsf{term} - \mathsf{term}$.

$\mathsf{mono} - *$: For the case $\mathsf{mono} - \mathsf{mono}$, the threads distribution does not change before and after applying the rewrite rule, so the marking does not change. Thus, the property holds. For the other cases, the proof is based on the cases above.

□

Lemma 5.6. *Let $\mathcal{C}_\pi$ and $\mathcal{C}'_\pi$ be contexts of a π-graph π. If $\mathcal{C}_\pi \xrightarrow[(r_1,v_1),(r_2,v_2)]{\tau} \mathcal{C}'_\pi$ and there exists M'_π such that $\mathsf{mark}(\mathcal{C}_\pi)[t^{r_1r_2}_{v_1v_2}{:}\boldsymbol{\tau}\rangle\mathsf{M}'_\pi$, then we have: $\mathsf{M}'_\pi = \mathsf{mark}(\mathcal{C}'_\pi)$.*

Proof. By Def. 5.2 on page 88, a place p in the translated net structure is either a context place p_Γ or a control place. If $p = p_\Gamma$, by Prop. 5.13 on page 106, we have: $\mathsf{mark}(\mathcal{C}'_\pi)(p) = \mathsf{M}'_\pi(p)$. Otherwise, by Prop. 5.14 on page 106, we also have: $\mathsf{mark}(\mathcal{C}'_\pi)(p) = \mathsf{M}'_\pi(p)$. Thus, the property holds. □

Theorem 5.2. *Let $\mathcal{C}_\pi$ be a context of a π-graph π. For any $\mu, \mathcal{C}'_\pi, r_o, v_o, r_i, v_i$ such that $\mathcal{C}_\pi \xrightarrow[(r_o,v_o),(r_i,v_i)]{\tau} \mathcal{C}'_\pi$, there exists M'_π s.t $\mathsf{mark}(\mathcal{C}_\pi)[t^{r_or_i}_{v_ov_i}{:}\tau\rangle\mathsf{M}'_\pi$ and $\mathsf{M}'_\pi = \mathsf{mark}(\mathcal{C}'_\pi)$.*

Proof. By Lem. 5.5 on page 105, if there exists a rewrite $\mathcal{C}_\pi \xrightarrow[(r,v)]{\mu} \mathcal{C}'_\pi$ then there exists an occurrence $\mathsf{mark}(\mathcal{C}_\pi)[t^r_v{:}\mu\rangle\mathsf{M}'_\pi$. Moreover, by Lem. 5.6, we have $\mathsf{M}'_\pi = \mathsf{mark}(\mathcal{C}'_\pi)$. Thus, the property holds. □

5.2.1.3 Completeness for atomic actions

The proof of the completeness for an atomic action (Theorem 5.3, denoted by T3) is structured as shown in Figure 5.9. In this dependence graph, theorem T1 is used for proving T3 but we do not show its dependence graph as it was already shown before.

Definition 5.10 (Converse of a marking)**.** *Let $\pi = \langle \mathcal{R}, \mathcal{K}, \mathcal{G}, \mathcal{N} \rangle$ be a π-graph and M_π a marking of $\mathsf{net}(\pi)$. A* converse of marking M_π*, denoted by $\mathsf{convmark}(\mathsf{M}_\pi)$, is a context (Γ, Δ), in which:*

1. $\Gamma = \mathsf{M}_\pi(p_\Gamma)$

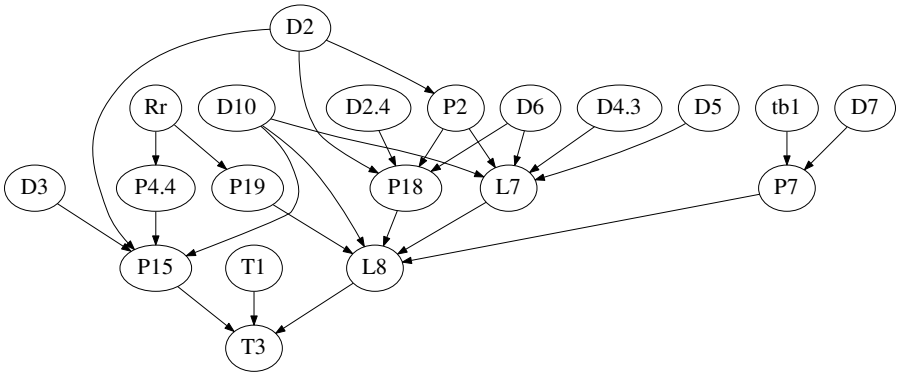


Figure 5.9: Dependence graph of the completeness for an atomic action

2. $\forall r \in \mathcal{R}, \forall v \in V_r$*, with* V_r *is the set of vertices of* $\mathcal{G}(r)$*,*

$$\Delta_r(v) = \begin{cases} \{i \mid r.i \in \mathsf{M}_\pi(p_v^r)\} & \textit{if} \quad \mathsf{stype}(r, v) \in \{\mathsf{norm}, \mathsf{spwn}\} \\ \emptyset & \textit{otherwise} \end{cases}$$

Proposition 5.15. *Let* $\mathcal{C}_\pi$ *be a context of a* π*-graph* π*. We have* $\mathsf{convmark}(\mathsf{mark}(\mathcal{C}_\pi)) = \mathcal{C}_\pi$*.*

Proof. Let $\mathcal{C}_\pi = (\Gamma, \Delta)$, $\mathsf{M}_\pi = \mathsf{mark}(\mathcal{C}_\pi)$, and $\mathcal{C}'_\pi = \mathsf{convmark}(\mathsf{M}_\pi) = (\Gamma', \Delta')$. We prove that $\mathcal{C}'_\pi = \mathcal{C}_\pi$ by showing that $\Gamma' = \Gamma$ and $\Delta'_r(v) = \Delta_r(v)$ for any replicator r in $\mathcal{R}$ and any vertex v in V_r.

On one hand, by Def. 5.3 on page 94 (case 1), we have: $\mathsf{M}_\pi(p_\Gamma) = \{\Gamma\}$. Moreover, by Def. 5.10 on the previous page (case 1), we have: $\Gamma' = \mathsf{M}_\pi(p_\Gamma)$. Thus, $\Gamma'_\pi = \Gamma_\pi$.

On the other hand, for any r in $\mathcal{R}$ and v in V_r, if $\mathsf{stype}(r, v) \in \{\mathsf{norm}, \mathsf{spwn}\}$, then by Def. 5.2 on page 88 (Eq. 5.1), there exists a place $p_v^r \in P_v$ such that $\mathsf{M}_\pi(p_v^r) = \{r.i \mid i \in \Delta_r(v)\}$. Moreover, by Def. 5.10 on the previous page (case 2), we have: $\Delta'_r(v) = \{i \mid r.i \in \mathsf{M}_\pi(p_v^r)\}$. Thus, $\Delta'_r(v) = \Delta_r(v)$. Otherwise, by Prop. 4.4 on page 81, $\Delta_r(v) = \emptyset$. Moreover, by Def. 5.10 on the previous page (case 2), we also have $\Delta'_r(v) = \emptyset$. Thus, $\Delta'_r(v) = \Delta_r(v)$. So, for any case of (r, v), we always have: $\Delta'_r(v) = \Delta_r(v)$. □

Proposition 5.16. *Let* π *be a* π*-graph* π *and* M_π *a marking of* $\mathsf{net}(\pi)$*. We have*

$$\mathsf{mark}(\mathsf{convmark}(\mathsf{M}_\pi)) = \mathsf{M}_\pi.$$

Proof. Let $\mathcal{C}_\pi = \mathsf{convmark}(\mathsf{M}_\pi) = (\Gamma, \Delta)$, $\mathsf{M}'_\pi = \mathsf{mark}(\mathcal{C}_\pi)$. We prove that $\mathsf{M}_\pi = \mathsf{M}'_\pi$ by showing that $\mathsf{M}_\pi(p) = \mathsf{M}'_\pi(p)$ for any place p in the translated net $\mathsf{net}(\pi)$. By Def. 5.2 on page 88 (Eq. 5.1), the place p is either a context place (*i.e.*, $p = p_\Gamma$) or a control place. We have two cases:

First, if $p = p_\Gamma$, then by Def. 5.10, $\mathsf{M}_\pi(p) = \Gamma$. Moreover, by Def. 5.3, $\mathsf{M}'_\pi(p) = \Gamma$. Thus, $\mathsf{M}_\pi(p) = \mathsf{M}'_\pi(p)$.

Second, if p is a control place, then it is either an initial place or not. If p is not an initial place, *i.e.*, $p = p^r_v$, then by Def. 5.3 on page 94, $\mathsf{M}'_\pi(p) = \{r.i \mid i \in \Delta_r(v)\}$. Moreover, by Def. 5.2 on page 88 (Eq. 5.1), $\mathsf{stype}(r, v) \in \{\mathsf{norm}, \mathsf{spwn}\}$. By Def. 5.10 on page 109, $\Delta_r(v) = \{i \mid r.i \in \mathsf{M}_\pi(p)\}$, thus, $\mathsf{M}_\pi(p) = \{r.i \mid i \in \Delta_r(v)\}$. Thus, $\mathsf{M}'_\pi(p) = \mathsf{M}_\pi(p)$. Otherwise, *i.e.*, $p = p^r_0$, because the set of control tokens in a replicator is constant and each place which is not initial has the same control tokens in both M_π and M'_π, thus, $\mathsf{M}_\pi(p^r_0) = \mathsf{M}_\pi(p^r_0)$.

For all the cases of place p, we always have $\mathsf{M}'_\pi(p^r_v) = \mathsf{M}_\pi(p^r_v)$. Thus, the property holds. □

Proposition 5.17 (Correspondance between atomic transitions and actions)**.** *Let* $\mathsf{net}(\pi) = (\mathsf{P}, \mathsf{T}, \mathsf{U}, \mathsf{G})$ *be the translated net structure of a* π*-graph* π*. For any atomic transition in* T*, there exists an action in* π *with the same label.*

Proof. For any atomic-transition t in T_1, by Def. 5.2 on page 88 (Eq. 5.3), $\mathsf{T}_1 = \mathfrak{N} \cup \mathfrak{S} \cup \mathfrak{T} \cup \mathfrak{M}$, thus t belongs to either $\mathfrak{N}$, $\mathfrak{S}$, $\mathfrak{T}$, or $\mathfrak{M}$. Suppose that $t = t^r_v$, we prove that in any case of t, there exists a corresponding action with the same label by analyzing all the cases of t.

- If t belongs to $\mathfrak{N}$, then by Def. 5.2 on page 88 (Eq. 5.3a), there exists a corresponding action at (r, v) such that $\mathsf{stype}(r, v) = \mathsf{norm}$.
- If t belongs to $\mathfrak{S}$, then by Def. 5.2 (Eq. 5.3b), there exists a corresponding action at (r, v) such that $\mathsf{stype}(r, v) = \mathsf{spwn}$.
- If t belongs to $\mathfrak{T}$, then by Def. 5.2 (Eq. 5.3c), there exists a corresponding action at (r, v) such that $\mathsf{stype}(r, v) = \mathsf{term}$.
- If t belongs to $\mathfrak{M}$, then by Def. 5.2 (Eq. 5.3d), there exists a corresponding action at (r, v) such that $\mathsf{stype}(r, v) = \mathsf{mono}$.

Moreover, by Def. 5.2 (Eq. 5.5), the label of action at (r, v) is the same as the label of transition $t = t^r_v$. Thus, the property holds. □

Proposition 5.18. *Let* $\mathsf{net}(\pi)$ *be the translated net of a* π*-graph* π*. If there exists* $\mathsf{M}_\pi, \mathsf{M}'_\pi$ *such that* $\mathsf{M}_\pi[t^r_v{:}\mu\rangle\mathsf{M}'_\pi$*, then there exists* $i \in [1..\mathcal{K}(r)]$ *such that* $\mathsf{avail}_n(r.i, t^r_v, \mathsf{M}_\pi) = \mathsf{true}$ *and* $\mathsf{G}(t^r_v)(\mathsf{M}_\pi(p_\Gamma), r, i) = \mathsf{true}$.

Proof. By Prop. 5.1, the transition t^r_v has a unique control input place, let $p = \mathsf{cip}(t^r_v)$. Moreover, by Def. 5.2 (Eq. 5.6 and Eq. 5.7), the transition t^r_v has a unique context place p_Γ. So, all input places of t^r_v is the set ${}^\bullet t^r_v = \{p, p_\Gamma\}$.

By Def. 5.2 on page 88 (Eq. 5.8), $\mathsf{U}(p_\Gamma, t_v^r) = \Gamma$ and the label of input arc $\mathsf{U}(p, t_v^r)$ is either X, $r_\mathrm{o}.i_\mathrm{o}$, $r_\mathrm{i}.i_\mathrm{i}$ or $r.i$. By the Petri nets transition rule (cf. p. 24), there exists a control token $r.i \in \mathsf{M}_\pi(p)$ and a context token $\Gamma_\pi \in \mathsf{M}_\pi(p_\Gamma)$ such that the evaluation of guard $\mathsf{G}(t_v^r)(\Gamma_\pi, r, i)$ is true. Because $\mathsf{M}_\pi(p_\Gamma) = \Gamma_\pi$, thus, $\mathsf{G}(t_v^r)(\mathsf{M}_\pi(p_\Gamma), r, i) = \mathsf{true}$. By Def. 5.6, from $r.i \in \mathsf{M}_\pi(p)$ we have $\mathsf{avail}_n(r.i, t_v^r, \mathsf{M}_\pi) = \mathsf{true}$. Moreover, by Def. 5.2, we also have $i \in [1..\mathcal{K}(r)]$. Thus, the property holds. □

Lemma 5.7. *Let π be a π-graph, M_π a marking of $\mathsf{net}(\pi)$, $\mathcal{C}_\pi$ the* converse marking *of M_π, t_v^r an atomic-transition and i a thread in π. Then, if $\mathsf{avail}_n(r.i, t_v^r, \mathsf{M}_\pi) = \mathsf{true}$, then $\mathsf{avail}_\pi(i, r, v, \mathcal{C}_\pi) = \mathsf{true}$.*

Proof. By Def. 5.6 on page 97, from $\mathsf{avail}_n(r.i, t_v^r, \mathsf{M}_\pi) = \mathsf{true}$, there exists a control input place p of t_v^r such that $r.i$ is present in the marking $\mathsf{M}_\pi(p)$. By Prop. 5.1 on page 94, t_v^r has only one control input place, either p_0^r or p_u^r. We have two cases:

- If $p = p_0^r$, then $r.i \in \mathsf{M}_\pi(p_0^r)$. By Def. 5.10 on page 109 (case 2), for any vertex v in V_r, $i \notin \Delta_r(v)$, thus, by Def. 4.3 on page 58, $i \in \mathsf{inact}(\Delta_r)$. Moreover, by Prop. 5.1 on page 94, the action at (r, v) must be mono or spwn. Thus, we have: $i \in \mathsf{inact}(\Delta_r) \wedge \mathsf{stype}(r, v) \in \{\mathsf{mono}, \mathsf{spwn}\}$. By Def. 5.5 on page 96, $\mathsf{avail}_\pi(i, r, v, \mathcal{C}_\pi) = \mathsf{true}$.

- If $p = p_u^r$, then $r.i \in \mathsf{M}_\pi(p_u^r)$. By Def. 5.10 (case 2), we have $i \in \Delta_r(u)$, with $(u, v) \in E_r$. Moreover, by Prop. 5.1, the action at (r, v) must be norm or term. Thus, we have: $i \in \Delta_r(u) \wedge \mathsf{stype}(r, v) \in \{\mathsf{norm}, \mathsf{term}\}$, where $(u, v) \in E_r$. By Def. 5.5, we have: $\mathsf{avail}_\pi(i, r, v, \mathcal{C}_\pi) = \mathsf{true}$.

□

Proposition 5.19. *Let r be a replicator in a π-graph π. For any context $\mathcal{C}_\pi = (\Gamma, \Delta)$ of π, if there exists a thread $i \in [1..\mathcal{K}(r)]$ such that $\mathsf{avail}_\pi(i, r, v, \mathcal{C}_\pi) = \mathsf{true}$ and $\mathsf{pre}_{\mathrm{atom}}(\Gamma, r, v, i) = \mathsf{true}$, then there exists $\mathcal{C}'_\pi$ such that $\mathcal{C}_\pi \xrightarrow[(r,v)]{\mu} \mathcal{C}'_\pi$.*

Proof. Let $\mathcal{C}_\pi$ be a context of π. For any structural type of the action at (r, v), the first condition $\mathsf{avail}_\pi(i, r, v, \mathcal{C}_\pi)$ is that for controls and the second $\mathsf{pre}_{\mathrm{atom}}(\Gamma, r, v, i) = \mathsf{true}$ is that for the precondition for a rewrite. Thus, by the corresponding rewrite rule (cf. Sec. 4.2.4.3 on page 75), there exists a context $\mathcal{C}'_\pi$ such that we have a rewrite $\mathcal{C}_\pi \xrightarrow[(r,v)]{\mu} \mathcal{C}'_\pi$. The property holds. □

Lemma 5.8 (Existence of an atomic rewrite)**.** *Let π be a π-graph, M_π a marking of the translated net. Then, if there exists an atomic transition t_v^r and a marking M'_π such that $\mathsf{M}_\pi[t_v^r{:}\mu\rangle\mathsf{M}'_\pi$, then there exists a context $\mathcal{C}'_\pi$ such that $\mathsf{convmark}(\mathsf{M}_\pi) \xrightarrow[(r,v)]{\mu} \mathcal{C}'_\pi$.*

Proof. By Prop. 5.18 on page 111, there exists a thread i in $[1..\mathcal{K}(r)]$ such that $\mathsf{avail}_n(r.i, t_v^r, \mathsf{M}_\pi) = \mathsf{true}$ and $\mathsf{G}(t_v^r)(\mathsf{M}_\pi(p_\Gamma), r, i) = \mathsf{true}$. Let $\mathcal{C}_\pi = \mathsf{convmark}(\mathsf{M}_\pi)$. By Lem. 5.7 on the preceding page, from $\mathsf{avail}_n(r.i, t_v^r, \mathsf{M}_\pi) = \mathsf{true}$, we have $\mathsf{avail}_\pi(i, r, v, \mathcal{C}_\pi) = \mathsf{true}$. By Prop. 5.7 on page 99, from $\mathsf{G}(t_v^r)(\mathsf{M}_\pi(p_\Gamma), r, i) = \mathsf{true}$, we have $\mathsf{pre}_{\mathrm{atom}}(\mathsf{M}_\pi(p_\Gamma), r, v, i) = \mathsf{true}$.

Moreover, by Def. 5.10 on page 109 (case 1), if $\mathcal{C}_\pi = (\Gamma, \Delta)$, then the corresponding marking of the context place is $\mathsf{M}_\pi(p_\Gamma) = \Gamma$. Thus,

$$\mathsf{avail}_\pi(i, r, v, \mathcal{C}_\pi) \wedge \mathsf{pre}_{\mathrm{atom}}(\Gamma, r, v, i) = \mathsf{true}$$

By Prop. 5.19 on the facing page, there exists $\mathcal{C}'_\pi$ such that $\mathcal{C}_\pi \xrightarrow[(r,v)]{\mu} \mathcal{C}'_\pi$. The property holds. □

Theorem 5.3. *Let π be a π-graph and M_π a marking of the translated net structure of π. If there exists a transition t_v^r and a marking M'_π such that $\mathsf{M}_\pi[t_v^r{:}\mu\rangle\mathsf{M}'_\pi$, then $\mathsf{convmark}(\mathsf{M}_\pi) \xrightarrow[(r,v)]{\mu} \mathsf{convmark}(\mathsf{M}'_\pi)$.*

Proof. By Lem. 5.8 on the preceding page, there exists $\mathcal{C}'_\pi$ such that $\mathsf{convmark}(\mathsf{M}_\pi) \xrightarrow[(r,v)]{\mu} \mathcal{C}'_\pi$. By Thrm. 5.1 on page 103, we have $\mathsf{mark}(\mathcal{C}'_\pi) = \mathsf{M}'_\pi$. Thus, $\mathsf{convmark}(\mathsf{mark}(\mathcal{C}'_\pi)) = \mathsf{convmark}(\mathsf{M}'_\pi)$. Moreover, by Prop. 5.15 on page 110 we have $\mathsf{convmark}(\mathsf{mark}(\mathcal{C}'_\pi)) = \mathcal{C}'_\pi$. Thus, $\mathcal{C}'_\pi = \mathsf{convmark}(\mathsf{M}'_\pi)$, so $\mathsf{convmark}(\mathsf{M}_\pi) \xrightarrow[(r,v)]{\mu} \mathsf{convmark}(\mathsf{M}'_\pi)$. The property holds. □

5.2.1.4 Completeness for synchronizations

The proof of the completeness for a synchronization (Theorem 5.4, denoted by T4) is structured as shown in Figure 5.10. Similarly, in this dependence graph, we omit the fragment concerning theorem T2.

Proposition 5.20. *Let π be a π-graph. For any sync-transition $t_{v_o v_i}^{r_o r_i}$ in $\mathsf{net}(\pi)$, there exists a pair of an output action at (r_o, v_o) and an input action at (r_i, v_i).*

Proof. For any sync-transition $t_{v_\mathrm{o} v_\mathrm{i}}^{r_\mathrm{o} r_\mathrm{i}}$, by Def. 5.2 (Eq. 5.4), there exist two atomic-transitions $t_{v_\mathrm{o}}^{r_\mathrm{o}}$ and $t_{v_\mathrm{i}}^{r_\mathrm{i}}$ with labels $\phi\overline{c}\langle a\rangle$ and $\psi d(x)$, respectively. Moreover, by Prop. 5.17 on page 111, for each atomic-transition, there exists a corresponding atomic action with the same label. Thus, there exists an output action at $(r_\mathrm{o}, v_\mathrm{o})$ and an input action at $(r_\mathrm{i}, v_\mathrm{i})$. The property holds. □

Proposition 5.21. *Let π be a π-graph, $t_{v_1 v_2}^{r_1 r_2}$ a sync-transition in $\mathsf{net}(\pi)$ and M_π a marking of $\mathsf{net}(\pi)$. If there exists a marking M'_π such that $\mathsf{M}_\pi[t_{v_1 v_2}^{r_1 r_2}{:}\mu\rangle\mathsf{M}'_\pi$, then there exist threads $i_1 \in [1..\mathcal{K}(r_1)]$ and $i_2 \in [1..\mathcal{K}(r_2)]$ such that:*

$$\mathsf{avail}_n^{\oplus}(t_{v_1 v_2}^{r_1 r_2}, r_1.i_1, r_2.i_2, \mathsf{M}_\pi) = \mathsf{true} \ \wedge \ \mathsf{G}(t_{v_1 v_2}^{r_1 r_2})(\mathsf{M}_\pi(p_\Gamma), r_1, i_1, r_2, i_2) = \mathsf{true}.$$

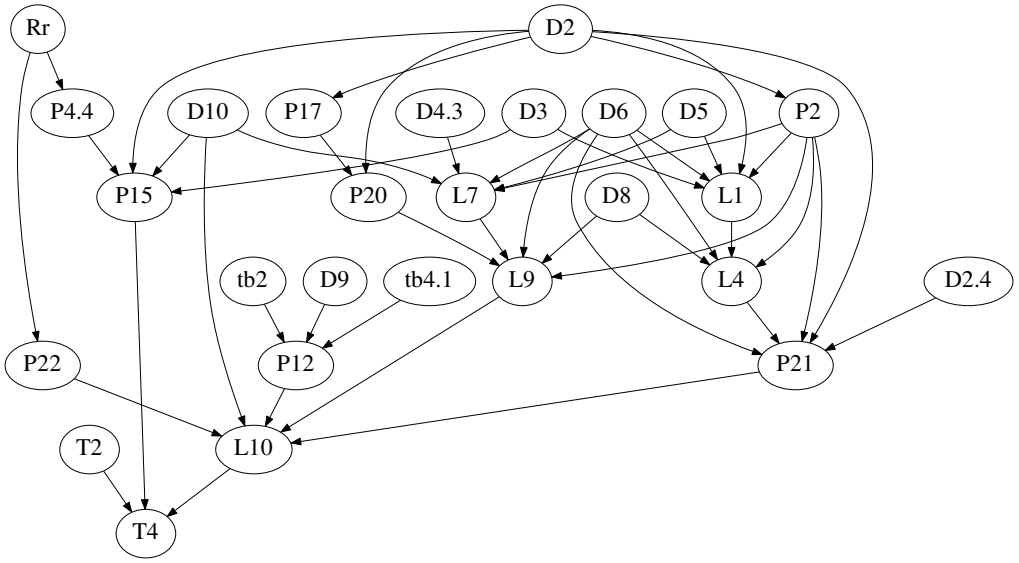


Figure 5.10: Dependence graph of the completeness for a synchronization

Proof. By Def. 5.2 on page 88 (Eq. 5.6), we have: ${}^\bullet t^{r_1 r_2}_{v_1 v_2} = {}^\bullet t^{r_1}_{v_1} \cup {}^\bullet t^{r_2}_{v_2}$. By Prop. 5.1 on page 94, each atomic-transition has a unique control input place. Let p_1 be the control input place of $t^{r_1}_{v_1}$ and p_2 be the control input place of $t^{r_2}_{v_2}$. All the control input places of the sync-transition are in the set ${}^\bullet t^{r_1 r_2}_{v_1 v_2} = \{p_\Gamma, p_1, p_2\}$. Thus, by Def. 2.4 on page 24, the transition is enabled if and only if there exists at least one control token in each control place p_1 and p_2, there exists a context token in p_Γ and if the guard is evaluated to true. Thus, there exists $i_1 \in [1..\mathcal{K}(r_1), i_2 \in [1..\mathcal{K}(r_2)]$ such that:

$$r_1.i_1 \in \mathsf{M}_\pi(p_1) \wedge r_2.i_2 \in \mathsf{M}_\pi(p_2) = \mathsf{true} \tag{1}$$

$$\text{and} \quad \mathsf{G}(t^{r_1 r_2}_{v_1 v_2})(\mathsf{M}_\pi(p_\Gamma), r_1, i_1, r_2, i_2) = \mathsf{true} \tag{2}$$

By Def. 5.6 on page 97, from (1), we have: $\mathsf{avail}_n(r_1.i_1, t^{r_1}_{v_1}, \mathsf{M}_\pi) \wedge \mathsf{avail}_n(r_1.i_1, t^{r_1}_{v_1}, \mathsf{M}_\pi) = \mathsf{true}$. Moreover, by Lem. 5.4 on page 105, from (2), we have $\mathsf{avail}^{\oplus}_n(t^{r_1 r_2}_{v_1 v_2}, r_1.i_1, r_2.i_2, \mathsf{M}_\pi) = \mathsf{true}$. Thus, the property holds. □

Lemma 5.9. *Let π be a π-graph, M_π a marking of $\mathsf{net}(\pi)$, $\mathcal{C}_\pi$ the* converse marking *of M_π, and $r_1.i_1$ and $r_2.i_2$ be two control tokens. Suppose $L_{r_1}(v_1) = \phi\bar{c}\langle a\rangle$ and $L_{r_2}(v_2) = \psi c(x)$. If $\mathsf{avail}^{\oplus}_n(t^{r_1 r_2}_{v_1 v_2}, r_1.i_1, r_2.i_2, \mathsf{M}_\pi) = \mathsf{true}$, then $\mathsf{avail}_\pi(i_1, r_1, v_1, \mathcal{C}_\pi) = \mathsf{true}$ and $\mathsf{avail}_\pi(i_2, r_2, v_2, \mathcal{C}_\pi) = \mathsf{true}$.*

Proof. By Prop. 5.20 on the preceding page, there exists an output action at (r_1, v_1) and an input action at (r_2, v_2) corresponding to atomic transitions $t^{r_1}_{v_1}$ and $t^{r_2}_{v_2}$.

By Prop. 5.1 on page 94, each atomic transition has a unique control input place. Let p_1 be the control input place of $t^{r_1}_{v_1}$ and p_2 the control input place of $t^{r_2}_{v_2}$. By Def. 5.8 on page 104, the control token $r_1.i_1$ is present in marking $\mathsf{M}_\pi(p_1)$ and $r_2.i_2$ is present

in $\mathsf{M}_\pi(p_2)$. Moreover, by Def. 5.6 on page 97, we have: $\mathsf{avail}_n(r_1.i_1, t^{r_1}_{v_1}, \mathsf{M}_\pi) = \mathsf{true}$ and $\mathsf{avail}_n(r_2.i_2, t^{r_2}_{v_2}, \mathsf{M}_\pi) = \mathsf{true}$. Thus, by Lem. 5.7, we have: $\mathsf{avail}_\pi(i_1, r_1, v_1, \mathcal{C}_\pi) = \mathsf{true}$ and $\mathsf{avail}_\pi(i_2, r_2, v_2, \mathcal{C}_\pi) = \mathsf{true}$. The property holds. □

Proposition 5.22. *For any context $\mathcal{C}_\pi = (\Gamma, \Delta)$ of a π-graph π, if there exists threads $i_1 \in [1..\mathcal{K}(r_1)]$ and $i_2 \in [1..\mathcal{K}(r_2)]$ such that:*

$$\mathsf{avail}_\pi(i_1, r_1, v_1, \mathcal{C}_\pi) \wedge \mathsf{avail}_\pi(i_2, r_2, v_2, \mathcal{C}_\pi) \wedge \mathsf{pre}_{\mathrm{sync}}(\Gamma, r_1, v_1, i_1, r_2, v_2, i_2) = \mathsf{true}$$

then there exists $\mathcal{C}'_\pi$ such that $\mathcal{C}_\pi \xrightarrow[(r,v)]{\mu} \mathcal{C}'_\pi$.

Proof. For any structural type of the action at (r, v), the first condition $\mathsf{avail}_\pi(i_1, r_1, v_1, \mathcal{C}_\pi) = \mathsf{true}$ and $\mathsf{avail}_\pi(i_2, r_2, v_2, \mathcal{C}_\pi) = \mathsf{true}$ is the condition for controls, and the second one $\mathsf{pre}_{\mathrm{sync}}(\Gamma, r_1, v_1, i_1, r_2, v_2, i_2)$ is the the precondition for sync rewrite. Thus, by rewrite rules of π-graph, there exists a rewrite $\mathcal{C}_\pi \xrightarrow[(r,v)]{\mu} \mathcal{C}'_\pi$. The property holds. □

Lemma 5.10. *Let M_π be a marking of a translated net of a π-graph π. If there exist a sync-transition $t^{r_1 r_2}_{v_1 v_2}$ and a marking M'_π such that $\mathsf{M}_\pi[t^{r_1 r_2}_{v_1 v_2}{:}\boldsymbol{\tau}\rangle\mathsf{M}'_\pi$, then there exists $\mathcal{C}'_\pi$ such that $\mathsf{convmark}(\mathsf{M}_\pi) \xrightarrow[(r_1,v_1),(r_2,v_2)]{\boldsymbol{\tau}} \mathcal{C}'_\pi$.*

Proof. By Prop. 5.21 on page 113, the occurrence of transition $\mathsf{M}_\pi[t^{r_1 r_2}_{v_1 v_2}{:}\boldsymbol{\tau}\rangle\mathsf{M}'_\pi$ leads to there exists threads $i_1 \in [1..\mathcal{K}(r_1)]$ and $i_2 \in [1..\mathcal{K}(r_2)]$ such that

$$\mathsf{avail}^{\oplus}_n(t^{r_1 r_2}_{v_1 v_2}, r_1.i_1, r_2.i_2, \mathsf{M}_\pi) \wedge \mathsf{G}(t^{r_1 r_2}_{v_1 v_2})(\mathsf{M}_\pi(p_\Gamma), r_1, i_1, r_2, i_2) = \mathsf{true}.$$

By Lem. 5.9 on the preceding page, from $\mathsf{avail}^{\oplus}_n(t^{r_1 r_2}_{v_1 v_2}, r_1.i_1, r_2.i_2, \mathsf{M}_\pi) = \mathsf{true}$, we have:

$$\mathsf{avail}_\pi(i_1, r_1, v_1, \mathcal{C}_\pi) = \mathsf{true} \wedge \mathsf{avail}_\pi(i_1, r_1, v_1, \mathcal{C}_\pi) = \mathsf{true}.$$

By Prop. 5.12 on page 105, from $\mathsf{G}(t^{r_1 r_2}_{v_1 v_2})(\mathsf{M}_\pi(p_\Gamma), r_1, i_1, r_2, i_2) = \mathsf{true}$, we also have: $\mathsf{pre}_{\mathrm{sync}}(\mathsf{M}_\pi(p_\Gamma), r_1, v_1, i_1, r_2, v_2, i_2) = \mathsf{true}$. Let $\mathcal{C}_\pi$ be the converse of marking M_π, suppose $\mathcal{C}_\pi = (\Gamma, \Delta)$. By Def. 5.10 on page 109 (case 1), we have $\mathsf{M}_\pi(p_\Gamma) = \Gamma$. Thus,

$$\mathsf{avail}_\pi(i_1, r_1, v_1, \mathcal{C}_\pi) \wedge \mathsf{avail}_\pi(i_2, r_2, v_2, \mathcal{C}_\pi) \wedge \mathsf{pre}_{\mathrm{sync}}(\Gamma, r_1, v_1, i_1, r_2, v_2, i_2) = \mathsf{true}$$

Thus, by Prop. 5.22, there exists $\mathcal{C}'_\pi$ such that $\mathcal{C}_\pi \xrightarrow[(r_1,v_1),(r_2,v_2)]{\boldsymbol{\tau}} \mathcal{C}'_\pi$. Because $\mathcal{C}_\pi = \mathsf{convmark}(\mathsf{M}_\pi)$, we have: $\mathsf{convmark}(\mathsf{M}_\pi) \xrightarrow[(r_1,v_1),(r_2,v_2)]{\boldsymbol{\tau}} \mathcal{C}'_\pi$. The property holds. □

Theorem 5.4. *Let π be a π-graph and M_π a marking of the translated net structure of π. If there exist a transition $t^{r_1 r_2}_{v_1 v_2}$ and a marking M'_π such that $\mathsf{M}_\pi[t^{r_1 r_2}_{v_1 v_2}{:}\mu\rangle\mathsf{M}'_\pi$, then $\mathsf{convmark}(\mathsf{M}_\pi) \xrightarrow[(r_1,v_1),(r_2,v_2)]{\mu} \mathsf{convmark}(\mathsf{M}'_\pi)$.*

Proof. By Lem. 5.10 on the previous page, from the occurrence of transition $\mathsf{M}_\pi[t^{r_1r_2}_{v_1v_2}{:}\mu\rangle\mathsf{M}'_\pi$, there exists a context $\mathcal{C}'_\pi$ such that $\mathsf{convmark}(\mathsf{M}_\pi) \xrightarrow[(r_1,v_1),(r_2,v_2)]{\mu} \mathcal{C}'_\pi$. By Thrm. 5.2 on page 109, we have: $\mathsf{mark}(\mathcal{C}'_\pi) = \mathsf{M}'_\pi$, thus, $\mathsf{convmark}(\mathsf{mark}(\mathcal{C}'_\pi)) = \mathsf{convmark}(\mathsf{M}'_\pi)$. Moreover, by Prop. 5.15 on page 110 we have: $\mathsf{convmark}(\mathsf{mark}(\mathcal{C}'_\pi)) = \mathcal{C}'_\pi$. Thus, $\mathcal{C}'_\pi = \mathsf{convmark}(\mathsf{M}'_\pi)$, so $\mathsf{convmark}(\mathsf{M}_\pi) \xrightarrow[(r_1,v_1),(r_2,v_2)]{\mu} \mathsf{convmark}(\mathsf{M}'_\pi)$. The property holds. □

5.2.2 Global conformance

Finally, the proof of global conformance is structured as shown in Figure 5.11. In these dependence graphs, we omit the fragment concerning theorems T1, T2, T3 and T4.

(a) Soundness (b) Completeness

Figure 5.11: Dependence graph of global conformance

Theorem 5.5 (Soundness of the translation)**.** *Let $\mathcal{C}_\pi$ be a context of a π-graph π. For any μ and for any $\mathcal{C}'_\pi$, if $\mathcal{C}_\pi \xrightarrow{\mu} \mathcal{C}'_\pi$, then there exists a unique transition t and a marking M'_π such that $\mathsf{mark}(\mathcal{C}_\pi)[t{:}\mu\rangle\mathsf{M}'_\pi$ and $\mathsf{M}'_\pi = \mathsf{mark}(\mathcal{C}'_\pi)$.*

Proof. By Def. 4.8 on page 81, the rewrite $\mathcal{C}_\pi \xrightarrow{\mu} \mathcal{C}'_\pi$ is triggered by the performance of either an atomic action, which is denoted by $\mathcal{C}_\pi \xrightarrow[(r,v)]{\mu} \mathcal{C}'_\pi$ or a synchronization, which is denoted by $\mathcal{C}_\pi \xrightarrow[(r_1,v_1),(r_2,v_2)]{\boldsymbol{\tau}} \mathcal{C}'_\pi$. We have two cases:

First, if the rewrite is $\mathcal{C}_\pi \xrightarrow[(r,v)]{\mu} \mathcal{C}'_\pi$, then, by Prop. 5.5 on page 96, there exists a unique corresponding transition t^r_v such that $\mathsf{U}(t^r_v) = L_r(v)$. By Thrm. 5.1 on page 103, there exists M'_π such that $\mathsf{mark}(\mathcal{C}_\pi)[t^r_v{:}\mu\rangle\mathsf{M}'_\pi$ and $\mathsf{M}'_\pi = \mathsf{mark}(\mathcal{C}'_\pi)$.

Second, if the rewrite is $\mathcal{C}_\pi \xrightarrow[(r_1,v_1),(r_2,v_2)]{\boldsymbol{\tau}} \mathcal{C}'_\pi$, then by Prop. 5.10 on page 103, there exists a unique corresponding transition $t^{r_1r_2}_{v_1v_2}$ such that $\mathsf{U}(t^{r_1r_2}_{v_1v_2}) = \boldsymbol{\tau}$. By Thrm. 5.2 on page 109, there exists M'_π such that $\mathsf{mark}(\mathcal{C}_\pi)[t^{r_1r_2}_{v_1v_2}{:}\boldsymbol{\tau}\rangle\mathsf{M}'_\pi$ and $\mathsf{M}'_\pi = \mathsf{mark}(\mathcal{C}'_\pi)$. □

Theorem 5.6 (Completeness of the translation)**.** *Let π be a π-graph and M_π a marking of the translated net structure of π. If there exist a transition t and a marking M'_π such that $\mathsf{M}_\pi[t{:}\mu\rangle\mathsf{M}'_\pi$, then $\mathsf{convmark}(\mathsf{M}_\pi) \xrightarrow{\mu} \mathsf{convmark}(\mathsf{M}'_\pi)$.*

Proof. By Def. 5.2, a transition in the translated net is either an atomic transition, which is denoted by t_v^r, or a sync transition, which is denoted by $t_{v_1v_2}^{r_1r_2}$. We have two cases:

First, if the transition is t_v^r, then, by Prop. 5.5, there exists a unique action at (r, v) such that $\mathsf{U}(t_v^r) = L_r(v)$. By Thrm. 5.3, we have: $\mathsf{convmark}(\mathsf{M}_\pi) \xrightarrow[(r,v)]{\mu} \mathsf{convmark}(\mathsf{M}'_\pi)$.

Second, if the transition is $t_{v_1v_2}^{r_1r_2}$, then, by Prop. 5.10, there exists a unique output at (r_1, v_1) and a unique input at (r_2, v_2). By Thrm. 5.4, we have: $\mathsf{convmark}(\mathsf{M}_\pi) \xrightarrow[(r_1,v_1),(r_2,v_2)]{\tau} \mathsf{convmark}(\mathsf{M}'_\pi)$. □

5.3 Synthesis

This chapter presents the translation of π-graphs into high-level Petri nets and proves a series of important properties of the translation.

The translation is illustrated first on a small example, step by step. It consists of creating places with the marking corresponding to the control flow context, translating atomic actions and synchronizations into transitions, translating the name context into a specific marking of the context place, and finally, computing the guards for the transitions. Then, a formal definition of the translation is provided. The translation can be performed for any configuration of π-graphs.

As the main result, we prove that the translation is an isomorphism. The proof is decomposed in two levels: local and global conformance. The local conformance concerns the soundness and completeness properties of atomic actions and synchronizations. The global conformance concerns these two properties of the whole translation. Thus, properties of systems modelled in π-graphs can be checked in the translated Petri nets. Moreover, the counter-examples that are provided by the model checkers for Petri nets can be easily constructed in terms of the original π-graphs.

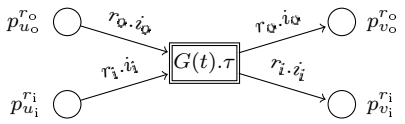
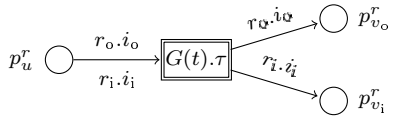
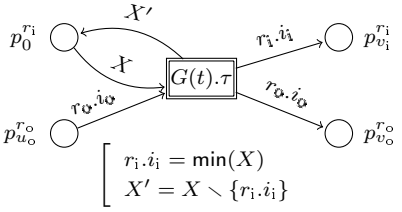
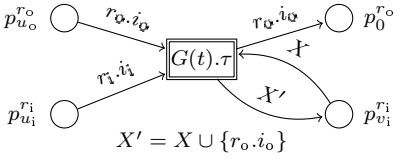
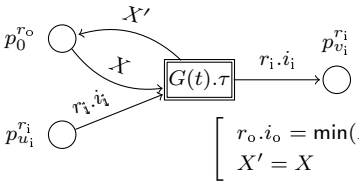
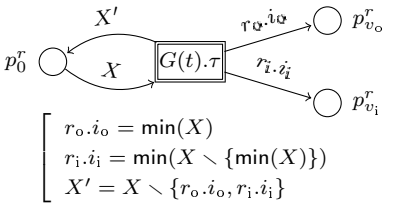
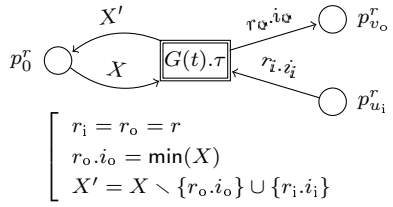
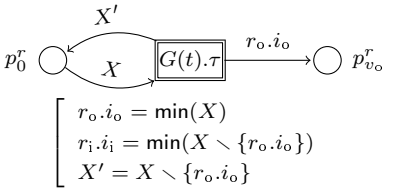
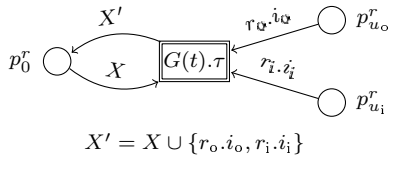
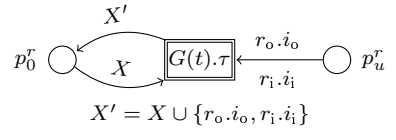
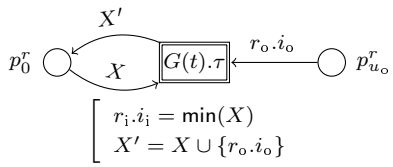
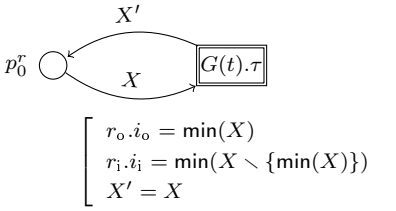
No.	Arc labels	No.	Arc labels
1		2	
3		4	
5		6	
7		8	
9		10	
11		12	

Table 5.4: Arc label of a sync-transition $t^{r_\mathrm{o} r_\mathrm{i}}_{v_\mathrm{o} v_\mathrm{i}}$

Chapter 6

Verification

After modelling reconfigurable systems using π-graphs and translating them into Petri nets, our next objective is to check properties of their behaviours. This chapter first describes a logic allowing to specify behavioural and temporal properties of π-graphs. The logical language is split into context and temporal operators. Context operators (in Section 6.1) allow to express properties of the name context and control context of a given π-graph state. For the temporal operators, we use Linear Temporal Logic (LTL) [8]. We saw in the previous chapter that there is an isomorphism between the π-graphs and the translated Petri net semantics. Hence, the logic is also a logic about the Petri nets. This means we can reuse existing Petri net analysis tools to automate the verification of properties. As an illustration, we show (in Section 6.3.2 on page 145) how to translate the properties into the logic of the NECO model checker [28].

6.1 Context properties of π-graphs

We remind the readers that a context consists of two parts: a name context, and a control context (cf. Section 4.2.2 on page 56). The name context represents the instances of names in the π-graph. The control context represents the current states of threads on vertices, *i.e.*, if a vertex has control for some threads.

6.1.1 Classification of context properties

Context properties of π-graphs can be classified along multiple dimensions:

1. Atomic vs. compound,
2. Unary vs. binary,
3. Signature, *i.e.*, the number and types of parameters.

Atomic vs. compound In general, a context property is a statement about some of the following four elements: replicator identifiers, threads, vertices and names. For example, "*A vertex v in replicator r has control for a thread i*" (cf. Section 4.2.2.1 on page 56). This property is denoted by $\texttt{active}(r, i, v)$ in the logic. Such a property cannot be derived from another one, thus it is called an *atomic property.*

Compound properties are higher-level assertions. They can be existential formulas, universal formulas, conjunctions, disjunctions, or the combination of these. For example, the property "*There exists a replicator such that all of its threads are active*" can be expressed as follows:

$$\exists r \in \mathcal{R}\ \forall i \in \mathcal{K}(r)\ \exists v \in V_r : \texttt{active}(r, i, v). \tag{6.1}$$

Such compound properties can be derived from atomic ones. For example, property (6.1) can be derived from $\texttt{active}(r, i, v)$ as follows:

$$\exists r \in \mathcal{R}\ \forall i \in \mathcal{K}(r)\ \exists v \in V_r : \texttt{active}(r, i, v) \equiv \bigvee_{r \in \mathcal{R}} \bigwedge_{i \in \mathcal{K}(r)} \bigvee_{v \in V_r} \texttt{active}(r, i, v).$$

This kind of derivation is specific to a given π-graph and can be cumbersome to write by hand. Thankfully, the derivation process can be generalized and systematized. For this, we use symbols * and ? to represent the universal and existential quantifiers, and typed parameters to distinguish quantifications over replicators, threads and vertices. For example, the property (6.1) can be derived as follows:

$$\underbrace{\exists r \in \mathcal{R}}_{?_{r:\texttt{Repl}}}\ \underbrace{\forall i \in \mathcal{K}(r)}_{*_{i:\texttt{Thrd}}}\ \underbrace{\exists v \in V_r}_{?_{v:\texttt{Vert}}} : \underbrace{\texttt{active}(r, i, v)}_{\texttt{active}(r:\texttt{Repl}, i:\texttt{Thrd}, v:\texttt{Vert})} \tag{6.2}$$

A property such as (6.2) is called a *compound property.* Note that it contains exactly one occurrence of atomic proposition.

It also possible to fix one or more parameters. For example, the compound property "*Thread* 1 *of replicator* r_1 *is active*" is written as follows:

$$?_{v:\texttt{Vert}}\ \texttt{active}(r_1, 1, v : \texttt{Vert}).$$

Unary vs. binary Most properties assert about a single replicator context. This is the case for atomic property $\texttt{active}(r, i, v)$ and also for the compound property (6.2). Assertions involving synchronization require two replicator contexts. For example, the atomic property *"Is there a synchronization between output thread i_1 at (r_1, v_1) and input thread i_2 at (r_2, v_2)?"* is denoted by $\texttt{sync}(r_1, i_1, v_1 \parallel r_2, i_2, v_2)$. The compound property *"Is there an internal synchronization in a given replicator r in the current state?"* can be written as follows:

$$?_{v_1:\texttt{Vert}}\ ?_{i_1:\texttt{Thrd}} \blacklozenge ?_{v_2:\texttt{Vert}}\ ?_{i_2:\texttt{Thrd}}\ \texttt{sync}(r, v_1 : \texttt{Vert}, i_1 : \texttt{Thrd} \parallel r, v_2 : \texttt{Vert}, i_2 : \texttt{Thrd}). \tag{6.3}$$

Signature The signature of an atomic property is its number of parameters, their relative order and their respective type. For example, the property $\texttt{active}(r, i, v)$ has signature RIV. The first parameter has type `Repl` for replicator, the second parameter has type `Thrd` for thread id and the last parameter has type `Vert` for vertex, hence the notation RIV. The signature of a compound property is the signature of its underlying atomic assertion. For example, the compound property $*_{i:\texttt{Thrd}}\,\texttt{fresh}(r, i : \texttt{Thrd}, n)$, meaning name n has not been instantiated in any thread of replicator r, has signature RIN. The first parameter has type `Repl` for replicator, the second parameter has type `Thrd` for thread id and the last parameter has type `Name` for names.

Not all possible combinations are allowed. The available signatures for unary properties are summarized in Figure 6.1 where r represents a replicator, i represents a thread id, v represents a vertex and n represents a name.

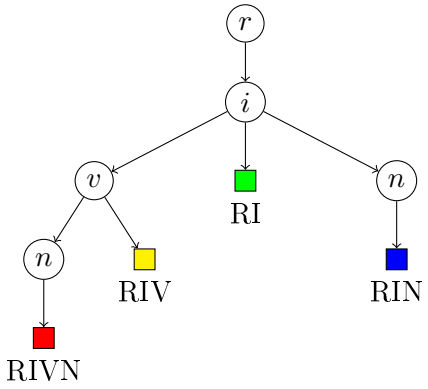


Figure 6.1: Possible signatures of unary context properties, where R means replicator, I means thread id, V means vertex and N means name

For binary properties, only two combinations are possible: RIN-RIN and RIV-RIV. For example, the property (6.3) has signature RIV-RIV. In both replicator contexts, the first parameter has type `Repl`, the second parameter has type `Vert` and the last parameter has type `Thrd`. Similarly, property $\texttt{equal}(r_1, i_1, n_1 \parallel r_2, i_2, n_2)$, meaning name n_1 in thread i_1 of replicator r_1 is equal to name n_2 in thread i_2 of replicator r_2, has signature RIN-RIN. In both replicator contexts, the parameters have type `Repl`, `Thrd` and `Name`, respectively.

6.1.2 Atomic context properties

According to the classification of the context properties defined above, all compound properties can be derived from atomic ones, thus we need a set of atomic properties as base. This section provides the formal propositions for all the atomic context properties by groups: RI, RIN, RIV, RIVN, RIN-RIN and RIV-RIV.

Recall that formally a context of a π-graph consists of two components $\mathcal{C} = (\Gamma, \Delta)$,

where $\Gamma = (\beta, \gamma, \delta)$ is the name context and Δ is the control context. An atomic property is defined based on these elements.

6.1.2.1 Atomic RI properties

Atomic RI properties are properties about a replicator r and one of its threads i, with the parameters list $(r : \texttt{Repl}, i : \texttt{Thrd})$.

1. $\mathsf{satisfy}_\phi(r : \texttt{Repl}, i : \texttt{Thrd})$: the guard ϕ holds for the thread i of replicator r in the current context (cf. p. 72).

$$\mathsf{satisfy}_\phi(r : \texttt{Repl}, i : \texttt{Thrd}) \stackrel{\text{def}}{=} \mathsf{pre}_\phi(\beta, \gamma, \delta, r, i)$$

2. $\mathsf{known}_z(r : \texttt{Repl}, i : \texttt{Thrd})$: a co-domain name z is known for thread i in replicator r, *i.e.*, there is some local name in r such that its instance for thread i is in the same class of the dynamic partition γ as z.

$$\mathsf{known}_z(r : \texttt{Repl}, i : \texttt{Thrd}) \stackrel{\text{def}}{=} \bigvee_{x \in X} y \in [\beta_i^r(x)]_\gamma, \text{where } X = \mathsf{tres}(r) \cup \mathsf{var}(r)$$

6.1.2.2 Atomic RIN properties

Properties in group RIN are mainly properties about π-graphs names. Note that if a name x is not instantiated, then an assertion about x is considered false.

1. $\mathsf{fresh}(r : \texttt{Repl}, i : \texttt{Thrd}, x : \texttt{Name})$: the name x in replicator r and thread i is a fresh name, *i.e.*, it does not appear in the name environment β, neither in the domain nor in the co-domain, (cf. p. 65).

$$\mathsf{fresh}(r : \texttt{Repl}, i : \texttt{Thrd}, x : \texttt{Name}) \stackrel{\text{def}}{=} \begin{cases} x \notin \mathsf{dom}(\beta) \cup \mathsf{cod}(\beta) \\ \wedge\ x_i^r \notin \mathsf{dom}(\beta) \end{cases}$$

2. $\mathsf{instance}_z(r : \texttt{Repl}, i : \texttt{Thrd}, x : \texttt{Name})$: the instance of name x in replicator r, thread i in the current name context (cf. p. 61) and the co-domain name z are in the same class of the dynamic partition, *i.e.*, they are considered equal in the current state.

$$\mathsf{instance}_z(r : \texttt{Repl}, i : \texttt{Thrd}, x : \texttt{Name}) \stackrel{\text{def}}{=} z \in [\beta_i^r(x)]_\gamma$$

3. $\mathsf{name}_\alpha(r : \texttt{Repl}, i : \texttt{Thrd}, x : \texttt{Name})$: the instance of name x in replicator r and thread i is a fresh input name in $\mathcal{F}_i$, a fresh output name in $\mathcal{F}_o$, or a fresh private name in $\mathcal{F}_\nu$, depending on the value of α.

$$\mathsf{name}_\alpha(r : \texttt{Repl}, i : \texttt{Thrd}, x : \texttt{Name}) \stackrel{\text{def}}{=} \beta_i^r(x) \in \mathcal{F}_\alpha \quad \text{with } \alpha \in \{\mathsf{o}, \mathsf{i}, \nu\}$$

6.1.2.3 Atomic RIV properties

Atomic RIV properties are properties of a replicator r and one of its threads i at a vertex v. We decompose the properties of this group into four subgroups:

- properties about the control,
- properties about the preconditions of atomic actions,
- properties about the activation of actions, and
- properties about the labels of transitions.

Properties about the control

1. $\mathsf{active}(r : \mathtt{Repl}, i : \mathtt{Thrd}, v : \mathtt{Vert})$: action at (r, v) is active, *i.e.*, it has control for thread i.

$$\mathsf{active}(r : \mathtt{Repl}, i : \mathtt{Thrd}, v : \mathtt{Vert}) \stackrel{\mathsf{def}}{=} i \in \Delta_r(v)$$

2. $\mathsf{avail}(r : \mathtt{Repl}, i : \mathtt{Thrd}, v : \mathtt{Vert})$: thread i is available for performing an action at (r, v), depending on the structural type of (r, v). If v is an initial vertex then thread i is in the pool of threads, otherwise, it is present in the predecessor of v.

$$\mathsf{avail}(r : \mathtt{Repl}, i : \mathtt{Thrd}, v : \mathtt{Vert}) \stackrel{\mathsf{def}}{=} \begin{cases} i \in \mathsf{inact}(\Delta_r) & \text{if } \mathsf{deg}_r^-(v) = 0 \\ \mathsf{active}(r, u, i) \wedge (u, v) \in E_r & \text{if } \mathsf{deg}_r^-(v) = 1 \end{cases}$$

Properties about preconditions of atomic actions

1. $\mathsf{pre}_{\mathrm{tau}}(r : \mathtt{Repl}, i : \mathtt{Thrd}, v : \mathtt{Vert})$: precondition for a silent action at (r, v) using thread i (cf. p. 72).

$$\mathsf{pre}_{\mathrm{tau}}(r : \mathtt{Repl}, i : \mathtt{Thrd}, v : \mathtt{Vert}) \stackrel{\mathsf{def}}{=} \begin{cases} L_r(v) = \phi\tau \wedge \mathsf{pre}_\phi(\beta, \gamma, \delta, r, i) & \text{if } \mathsf{deg}_r^-(v) = 1 \\ \left[\begin{array}{l} L_r(v) = \phi\tau \wedge \mathsf{pre}_\phi(\beta', \gamma', \delta', r, i) \\ \text{with } (\beta', \gamma', \delta') = \mathsf{init}_i^r(\Gamma) \end{array}\right. & \text{if } \mathsf{deg}_r^-(v) = 0 \end{cases}$$

2. $\mathsf{pre}_{\mathrm{out}}(r : \mathtt{Repl}, i : \mathtt{Thrd}, v : \mathtt{Vert})$: precondition for an output action at (r, v) using

thread i (cf. p. 73).

$$\mathsf{pre}_{\mathrm{out}}(r : \texttt{Repl}, i : \texttt{Thrd}, v : \texttt{Vert})$$
$$\stackrel{\mathsf{def}}{=} \begin{cases} \left[\begin{array}{l} L_r(v) = \phi\overline{c}\langle a\rangle \wedge \mathsf{pre}_\phi(\beta,\gamma,\delta,r,i) \\ \wedge\ \mathsf{pub}(\beta_i^r(c)) \wedge \mathsf{pub}(\beta_i^r(a)) \end{array}\right. & \text{if } \mathsf{deg}_r^-(v) = 1 \\ \left[\begin{array}{l} L_r(v) = \phi\overline{c}\langle a\rangle \wedge \mathsf{pre}_\phi(\beta',\gamma',\delta',r,i) \\ \wedge\ \mathsf{pub}(\beta_i'^r(c)) \wedge \mathsf{pub}(\beta_i'^r(c)) \\ \text{with } (\beta',\gamma',\delta') = \mathsf{init}_i^r(\Gamma) \end{array}\right. & \text{if } \mathsf{deg}_r^-(v) = 0 \end{cases}$$

3. $\mathsf{pre}_{\nu\mathrm{out}}(r : \texttt{Repl}, i : \texttt{Thrd}, v : \texttt{Vert})$: precondition for a bound output at (r, v) using thread i (cf. p. 73).

$$\mathsf{pre}_{\nu\mathrm{out}}(r : \texttt{Repl}, i : \texttt{Thrd}, v : \texttt{Vert})$$
$$\stackrel{\mathsf{def}}{=} \begin{cases} \left[\begin{array}{l} L_r(v) = \phi\overline{c}\langle a\rangle \wedge \mathsf{pre}_\phi(\beta,\gamma,\delta,r,i) \\ \wedge\ \mathsf{pub}(\beta_i^r(c)) \wedge \mathsf{priv}(\beta_i^r(a)) \end{array}\right. & \text{if } \mathsf{deg}_r^-(v) = 1 \\ \left[\begin{array}{l} L_r(v) = \phi\overline{c}\langle a\rangle \wedge \mathsf{pre}_\phi(\beta',\gamma',\delta',r,i) \\ \wedge\ \mathsf{pub}(\beta_i'^r(c)) \wedge \mathsf{priv}(\beta_i'^r(a)) \\ \text{with } (\beta',\gamma',\delta') = \mathsf{init}_i^r(\Gamma) \end{array}\right. & \text{if } \mathsf{deg}_r^-(v) = 0 \end{cases}$$

4. $\mathsf{pre}_{\mathrm{inp}}(r : \texttt{Repl}, i : \texttt{Thrd}, v : \texttt{Vert})$: precondition for an input at (r, v) using thread i (cf. p. 73).

$$\mathsf{pre}_{\mathrm{inp}}(r : \texttt{Repl}, i : \texttt{Thrd}, v : \texttt{Vert})$$
$$\stackrel{\mathsf{def}}{=} \begin{cases} L_r(v) = \phi c(x) \wedge \mathsf{pre}_\phi(\beta,\gamma,\delta,r,i) \wedge \mathsf{pub}(\beta_i^r(c)) & \text{if } \mathsf{deg}_r^-(v) = 1 \\ \left[\begin{array}{l} L_r(v) = \phi c(x) \wedge \mathsf{pre}_\phi(\beta',\gamma',\delta',r,i) \\ \wedge\ \mathsf{pub}(\beta_i'^r(c)) \text{ with } (\beta',\gamma',\delta') = \mathsf{init}_i^r(\Gamma) \end{array}\right. & \text{if } \mathsf{deg}_r^-(v) = 0 \end{cases}$$

5. $\mathsf{pre}_{\mathrm{atom}}(r : \texttt{Repl}, i : \texttt{Thrd}, v : \texttt{Vert})$: precondition for an atomic action at (r, v) using thread i.

$$\mathsf{pre}_{\mathrm{atom}}(r : \texttt{Repl}, i : \texttt{Thrd}, v : \texttt{Vert})$$
$$\stackrel{\mathsf{def}}{=} \mathsf{pre}_{\mathrm{tau}}(r,v,i) \vee \mathsf{pre}_{\mathrm{out}}(r,v,i) \vee \mathsf{pre}_{\nu\mathrm{out}}(r,v,i) \vee \mathsf{pre}_{\mathrm{inp}}(r,v,i)$$

Properties about the activation of actions

1. $\mathsf{atomic}_\alpha(r : \texttt{Repl}, i : \texttt{Thrd}, v : \texttt{Vert})$ checks if thread i is available for action α and the guard of the action can hold in the current context.

$$\mathsf{atomic}_\alpha(r : \texttt{Repl}, i : \texttt{Thrd}, v : \texttt{Vert})$$
$$\stackrel{\mathsf{def}}{=} \mathsf{avail}(r,i,v) \wedge \mathsf{pre}_\alpha(r,i,v) \quad \text{where } \alpha \in \{\mathsf{tau}, \mathsf{inp}, \mathsf{out}, \nu\mathsf{out}\}$$

For example, if α is tau, then we have $\mathsf{atomic}_{tau}(r,i,v) \stackrel{\mathsf{def}}{=} \mathsf{avail}(r,i,v) \wedge \mathsf{pre}_{\mathrm{tau}}(r,i,v)$, which corresponds to the performance of a silent action at (r,v) using thread i.

2. $\mathsf{atomic}(r : \mathtt{Repl}, i : \mathtt{Thrd}, v : \mathtt{Vert})$ checks if the action at (r,v) can be activated using thread i, whatever the type of the action, *i.e.*, silent action, input, output or bound output. The property is defined by applying property atomic_α above, as follows:

$$\begin{aligned}&\mathsf{atomic}(r : \mathtt{Repl}, i : \mathtt{Thrd}, v : \mathtt{Vert})\\ &\stackrel{\mathsf{def}}{=} \mathsf{atomic}_{tau}(r,i,v) \vee \mathsf{atomic}_{inp}(r,i,v) \vee \mathsf{atomic}_{out}(r,i,v) \vee \mathsf{atomic}_{\nu out}(r,i,v)\end{aligned}$$

Properties about the labels of transitions

- $\mathsf{label}_\mu(r : \mathtt{Repl}, i : \mathtt{Thrd}, v : \mathtt{Vert})$ asserts that the atomic action at (r,v) can perform with label μ using thread i. The parameter μ can be considered as a constant.

$$\begin{aligned}&\mathsf{label}_\mu(r : \mathtt{Repl}, i : \mathtt{Thrd}, v : \mathtt{Vert})\\ &\stackrel{\mathsf{def}}{=} \begin{cases} \mathsf{atomic}_{tau}(r,i,v) \wedge \mu = \tau & \text{if } L_r(v) = \phi\tau \\ \mathsf{atomic}_{inp}(r,i,v) \wedge \mu = \beta_i^r(c)(\mathsf{clk}_?(\beta)) & \text{if } L_r(v) = \phi c(x) \\ \left[\begin{array}{l} \mathsf{atomic}_{out}(r,i,v) \wedge \mu = \overline{\beta_i^r(c)}\langle\beta_i^r(a)\rangle \\ \vee\ \mathsf{atomic}_{\nu out}(r,i,v) \wedge \mu = \overline{\beta_i^r(c)}\langle\nu\mathsf{clk}_!(\beta)\rangle \end{array}\right. & \text{if } L_r(v) = \phi\overline{c}\langle a\rangle \end{cases}\end{aligned}$$

In the comparison between labels, we can use wild-cards. Symbol "$*$" is used to represent any channel name, sent-name, or received-name. For example, suppose that the action at (r,v) is an output. We want to check if the output action can send out a name "!6". The label we will check is $\overline{*}\langle !6\rangle$.

6.1.2.4 Atomic RIVN properties

Atomic RIVN properties assert the effect on a name after performing an action. A name n in replicator r may be updated when the action at (r,v) is activated. The performance of the action may update the instance of name n in the name environment β, or updates the relation between n and other names, which is represented by the distinctions δ of the name context.

1. $\mathsf{updated}_\beta(r : \mathtt{Repl}, i : \mathtt{Thrd}, v : \mathtt{Vert}, n : \mathtt{Name})$ checks if the action at (r,v) can be activated using thread i and the *instance* of name n in the thread will be updated after performing the action. Since a silent action and an output action do not

update the instance of any name in the name context, the property is defined as follows:

$$\begin{aligned}&\mathsf{updated}_{\beta}(r : \texttt{Repl}, i : \texttt{Thrd}, v : \texttt{Vert}, n : \texttt{Name})\\ &\stackrel{\text{def}}{=} \begin{cases}\beta_i^r(n) \neq \beta_i'^r(n) \quad \wedge\\ (\mathsf{atomic}_{inp}(r,i,v) \vee \mathsf{atomic}_{\nu out}(r,i,v))\end{cases}\end{aligned}$$

where β' is the name context after performing the atomic action, which is defined as the post-condition of an input and a bound output (cf. p. 72), as follows:

$$(\beta', \gamma', \delta') \stackrel{\text{def}}{=} \begin{cases}\mathsf{post}_{\text{in}}(\Gamma, r, i, \phi c(x)) & \text{if } L_r(v) = \phi c(x)\\ \mathsf{post}_{\nu\text{out}}(\Gamma, r, i, \phi\bar{c}\langle a\rangle) & \text{if } L_r(v) = \phi\bar{c}\langle a\rangle \wedge \mathsf{priv}(a)\end{cases}$$

2. $\mathsf{updated}_{\delta}(r : \texttt{Repl}, i : \texttt{Thrd}, v : \texttt{Vert}, n : \texttt{Name})$ checks if the action at (r, v) can be activated using thread i and name n will be updated after performing the action.

$$\begin{aligned}&\mathsf{updated}_{\delta}(r : \texttt{Repl}, i : \texttt{Thrd}, v : \texttt{Vert}, n : \texttt{Name})\\ &\stackrel{\text{def}}{=} \begin{cases}\exists n' \in \mathcal{N} \text{ s.t } ([n]_\gamma \leftrightarrow [n']_\gamma \in \delta \wedge [n]_{\gamma'} \leftrightarrow [n']_{\gamma'} \notin \delta') \quad \vee\\ \qquad\qquad ([n]_\gamma \leftrightarrow [n']_\gamma \notin \delta \wedge [n]_{\gamma'} \leftrightarrow [n']_{\gamma'} \in \delta')\\ \wedge \quad (\bigvee_\alpha \ \mathsf{atomic}_\alpha(r,i,v)) \text{ with } \alpha \in \{tau, inp, out, \nu out\}\end{cases}\end{aligned}$$

where γ' and δ' are the dynamic partition and the set of distinctions after performing the atomic action, it is defined as follows:

$$(\beta', \gamma', \delta') \stackrel{\text{def}}{=} \begin{cases}\mathsf{post}_{\text{tau}}(\Gamma, r, i, \phi) & \text{if } L_r(v) = \phi\tau\\ \mathsf{post}_{\text{in}}(\Gamma, r, i, \phi c(x)) & \text{if } L_r(v) = \phi c(x)\\ \mathsf{post}_{\text{out}}(\Gamma, r, i, \phi) & \text{if } L_r(v) = \phi\bar{c}\langle d\rangle \wedge \mathsf{pub}(d)\\ \mathsf{post}_{\nu\text{out}}(\Gamma, r, i, \phi\bar{c}\langle d\rangle) & \text{if } L_r(v) = \phi\bar{c}\langle d\rangle \wedge \mathsf{priv}(d)\end{cases}$$

3. $\mathsf{updated}(r : \texttt{Repl}, i : \texttt{Thrd}, v : \texttt{Vert}, n : \texttt{Name})$ checks if the activation of the action at (r, v) updated the instance of name n or updated the relationship of n with other names. It is defined as follows:

$$\begin{aligned}&\mathsf{updated}(r : \texttt{Repl}, i : \texttt{Thrd}, v : \texttt{Vert}, n : \texttt{Name})\\ &\stackrel{\text{def}}{=} \mathsf{updated}_{\beta}(r, i, v, n) \vee \mathsf{updated}_{\delta}(r, i, v, n)\end{aligned}$$

6.1.2.5 Atomic RIN-RIN properties

Atomic RIN-RIN properties assert about the relation between two names in two replicators (possibly in the same replicator). The two names may be equal, unequal or unknown (may be evaluated as equal or unequal), or they have the same instantiation.

1. $\mathsf{equal}(r : \mathtt{Repl}, i : \mathtt{Thrd}, x : \mathtt{Name} \parallel r' : \mathtt{Repl}, i' : \mathtt{Thrd}, x' : \mathtt{Name})$: the name x in replicator r, thread i and the name x' in replicator r', thread i' are equal, *i.e.*, their instantiations are in the same class of the dynamic partition. The definition of this property is as follows:

$$\begin{aligned}&\mathsf{equal}(r : \mathtt{Repl}, i : \mathtt{Thrd}, x : \mathtt{Name} \parallel r' : \mathtt{Repl}, i' : \mathtt{Thrd}, x' : \mathtt{Name})\\ &\stackrel{\mathsf{def}}{=} [\beta_i^r(x)]_\gamma = [\beta_{i'}^{r'}(x')]_\gamma\end{aligned}$$

2. $\mathsf{unequal}(r : \mathtt{Repl}, i : \mathtt{Thrd}, x : \mathtt{Name} \parallel r' : \mathtt{Repl}, i' : \mathtt{Thrd}, x' : \mathtt{Name})$: the name x in replicator r, thread i, and the name x' in replicator r', thread i' are unequal, *i.e.*, their instantiations are in two distinct classes of the dynamic partition and there is a distinction between these classes (cf. Invariant 4.3 on page 79).

$$\begin{aligned}&\mathsf{unequal}(r : \mathtt{Repl}, i : \mathtt{Thrd}, x : \mathtt{Name} \parallel r' : \mathtt{Repl}, i' : \mathtt{Thrd}, x' : \mathtt{Name})\\ &\stackrel{\mathsf{def}}{=} ([\beta_i^r(x)]_\gamma, [\beta_{i'}^{r'}(x')]_\gamma) \in \delta\end{aligned}$$

3. $\mathsf{unknown}(r : \mathtt{Repl}, i : \mathtt{Thrd}, x : \mathtt{Name} \parallel r' : \mathtt{Repl}, i' : \mathtt{Thrd}, x' : \mathtt{Name})$: we do not know if the two names x and x' *may be* equated or not, *i.e.*, their instantiations are in two distinct classes of the dynamic partition but there is no distinction between them.

$$\begin{aligned}&\mathsf{unknown}(r : \mathtt{Repl}, i : \mathtt{Thrd}, x : \mathtt{Name} \parallel r' : \mathtt{Repl}, i' : \mathtt{Thrd}, x' : \mathtt{Name})\\ &\stackrel{\mathsf{def}}{=} ([\beta_i^r(x)]_\gamma \neq [\beta_{i'}^{r'}(x')]_\gamma) \quad \wedge \quad ([\beta_i^r(x)]_\gamma, [\beta_{i'}^{r'}(x')]_\gamma) \notin \delta\end{aligned}$$

4. $\mathsf{shared}(r : \mathtt{Repl}, i : \mathtt{Thrd}, x : \mathtt{Name} \parallel r' : \mathtt{Repl}, i' : \mathtt{Thrd}, x' : \mathtt{Name})$: the instantiation of name x in replicator r, thread i, and the instantiation of name x' in replicator r', thread i' are the same, *i.e.*,

$$\begin{aligned}&\mathsf{shared}(r : \mathtt{Repl}, i : \mathtt{Thrd}, x : \mathtt{Name} \parallel r' : \mathtt{Repl}, i' : \mathtt{Thrd}, x' : \mathtt{Name})\\ &\stackrel{\mathsf{def}}{=} \beta_i^r(x) = \beta_{i'}^{r'}(x')\end{aligned}$$

6.1.2.6 Atomic RIV-RIV properties

Atomic RIV-RIV properties assert about the synchronization between two different replicators which are represented by two parameters lists of the form $(r_1 : \mathtt{Repl}, i_1 : \mathtt{Thrd}, v_1 : \mathtt{Vert} \parallel r_2 : \mathtt{Repl}, i_2 : \mathtt{Thrd}, v_2 : \mathtt{Vert})$. If the first parameters list is for an output then the second one is for an input, or *vice-versa*.

1. $\mathsf{pre}_{\mathrm{sync}}(r_1 : \mathtt{Repl}, i_1 : \mathtt{Thrd}, v_1 : \mathtt{Vert} \parallel r_2 : \mathtt{Repl}, i_2 : \mathtt{Thrd}, v_2 : \mathtt{Vert})$: checks the precondition for synchronization between an action at (r_1, v_1) and an action at

(r_2, v_2) using threads i_1 and i_2 respectively (cf. p. 75).

$$\mathsf{pre}_{\text{sync}}(r_1 : \texttt{Repl}, i_1 : \texttt{Thrd}, v_1 : \texttt{Vert} \parallel r_2 : \texttt{Repl}, i_2 : \texttt{Thrd}, v_2 : \texttt{Vert})$$
$$\stackrel{\text{def}}{=} \left[\begin{array}{l} \left\{ \begin{array}{l} L_{r_1}(v_1) = \phi_1 \overline{c}\langle a \rangle \wedge L_{r_2}(v_2) = \phi_2 d(x) \\ \wedge\ \mathsf{pre}_{\text{com}}(\beta', \gamma', \delta', r_1, i_1, \phi_1 \overline{c}\langle a \rangle, r_2, i_2, \phi_2 d(x)) \end{array} \right. \\ \vee \\ \left\{ \begin{array}{l} L_{r_1}(v_1) = \phi_1 d(x) \wedge L_{r_2}(v_2) = \phi_2 \overline{c}\langle a \rangle \\ \wedge\ \mathsf{pre}_{\text{com}}(\beta'', \gamma'', \delta'', r_1, i_1, \phi_1 d(x), r_2, i_2, \phi_2 \overline{c}\langle a \rangle) \end{array} \right. \end{array} \right.$$

where

- $(\beta', \gamma', \delta') = f_{init}(f_{init}(\Gamma, r_1, i_1, v_1), r_2, i_2, v_2)$,
- $(\beta'', \gamma'', \delta'') = f_{init}(f_{init}(\Gamma, r_2, i_2, v_2), r_1, i_1, v_1)$,
- $f_{init}(\Gamma, r, i, v) = \begin{cases} \mathsf{init}_i^r(\Gamma) & \text{if } \mathsf{deg}_r^-(v) = 0 \\ \Gamma & \text{otherwise.} \end{cases}$

2. $\mathsf{sync}(r_1 : \texttt{Repl}, i_1 : \texttt{Thrd}, v_1 : \texttt{Vert} \parallel r_2 : \texttt{Repl}, i_2 : \texttt{Thrd}, v_2 : \texttt{Vert})$ checks if the synchronization between atomic actions at (r_1, v_1) and at (r_2, v_2) using threads i_1 and i_2 can be performed. The property is defined as follows:

$$\mathsf{sync}(r_1 : \texttt{Repl}, i_1 : \texttt{Thrd}, v_1 : \texttt{Vert} \parallel r_2 : \texttt{Repl}, i_2 : \texttt{Thrd}, v_2 : \texttt{Vert})$$
$$\stackrel{\text{def}}{=} \left[\begin{array}{l} \mathsf{avail}(r_1, i_1, v_1) \wedge \mathsf{avail}(r_2, i_2, v_2) \\ \wedge\ \mathsf{pre}_{\text{sync}}(r_1, i_1, v_1, r_2, i_2, v_2) \end{array} \right.$$

6.1.3 The logic of context properties

This section presents the grammar, well-formedness constraints and semantics of context properties.

6.1.3.1 Syntax of context properties

Table 6.1 gives the syntax of context properties. A context property can be atomic or compound. The list of atomic context properties is given in Section (6.1.2) above. For compound properties, there are two cases: unary and binary ones. Unary properties are of the form $\delta\mathtt{P}(\sigma)$ with δ a derivation operator[1], $\mathtt{P}$ an atomic property and σ a list of parameters. A derivation operator δ is a possibly empty sequence of operators of the form $?_{p:\mathtt{T}}$ or $*_{p:\mathtt{T}}$. The first one is the existential quantification and the second one is the universal quantification. The parameters list σ is a comma-separated list of formal parameters $p : \mathtt{T}$ and constants $\underline{p}$, *e.g.*, a given thread or a replicator. An empty derivation

[1] We hope it will not be confusion between δ (derivation operators) and the set of distinctions used in the other context

(context property)	$\Phi_c ::=$	atomic property	
	$\mid$	compound property	
(atomic property)	$\varphi ::=$	$\mathtt{P}(\sigma)$	unary atomic
	$\mid$	$\mathtt{P}(\sigma_1 \parallel \sigma_2)$	binary atomic
(compound property)	$\vartheta ::=$	$\delta\ \mathtt{P}(\sigma)$	unary compound
	$\mid$	$\delta_1 \blacklozenge \delta_2\ \mathtt{P}(\sigma_1 \parallel \sigma_2)$	binary compound
	$\mid$	$\delta_1 \lozenge \delta_2\ \mathtt{P}(\sigma_1 \parallel \sigma_2)$	distinct binary compound
(derivation operators)	$\delta ::=$	$\emptyset$	no derivation operator
	$\mid$	$?_{p:\mathtt{T}}\ \delta$	existential quantification
	$\mid$	$*_{p:\mathtt{T}}\ \delta$	universal quantification
(type)	$\mathtt{T} ::=$	$\mathtt{Repl}$	replicator identifiers
	$\mid$	$\mathtt{Vert}$	vertex identifiers
	$\mid$	$\mathtt{Thrd}$	thread identifiers
	$\mid$	$\mathtt{Name}$	names
(parameters)	$\sigma ::=$	$\emptyset$	no parameter
	$\mid$	$\underline{p}, \sigma$	fixed parameter p
	$\mid$	$p : \mathtt{T}, \sigma$	formal parameter p

Table 6.1: Syntax of context properties

operator δ or parameter list σ is denoted by $\emptyset$. As a convenience we can omit trailing $\emptyset$'s, *i.e.*, $\delta\emptyset \equiv \delta$ and $\sigma, \emptyset \equiv \sigma$.

In a binary compound property $\delta_1 \blacklozenge \delta_2\ \mathtt{P}(\sigma_1 \parallel \sigma_2)$, $\mathtt{P}$ is a binary atomic property with left parameters σ_1 and right parameters σ_2. The operator δ_1 explains the derivation of the left parameters and δ_2 explains the derivation of the right parameters. By default, the $\blacklozenge$ operator does not restrict the pair of replicators involved in the property. If we want to assert a proposition about two distinct replicators then the operator $\lozenge$ must be used instead.

6.1.3.2 Well-formedness constraints

Observe that a property that has a correct syntax may still be not well-formed. For example, both properties $?_{r:\mathtt{Repl}}\,\mathtt{active}(r : \mathtt{Repl}, i : \mathtt{Thrd}, 1)$ and $?_{v:\mathtt{Vert}}\,?_{i:\mathtt{Thrd}}\,\mathtt{active}(r, i : \mathtt{Thrd}, v : \mathtt{Vert})$ are syntactically correct but they are not well-formed. In the first property, the number of derivation operators is different from the number of formal parameters. In the second one, the derivation operators do not match the formal parameters,

i.e., operator $?_{v:\texttt{Vert}}$ does not match parameter $i : \texttt{Thrd}$ and similarly, operator $?_{i:\texttt{Thrd}}$ does not match parameter $v : \texttt{Vert}$. We introduce well-formedness constraints in the form of *inference rules*. Each inference rule has the following form:

$$\frac{P_1 \quad \dots \quad P_n}{C}\ (\mathrm{N}) \qquad n \geq 0 \tag{6.4}$$

where N is the rule name, $P_1, \dots, P_n$ are premises, and C is the conclusion of the rule. If all premises $P_1, \dots, P_n$ are true, then we can conclude that the conclusion C is true.

We use boxes to enrich the syntax of context properties to state about well-formedness. A unary (resp. binary) compound property $\delta\boxed{\delta'}\mathtt{P}(\sigma, \boxed{\sigma'})$ (resp. $\delta_1\boxed{\delta'_1} \blacklozenge \delta_2\mathtt{P}(\sigma_1, \boxed{\sigma'_1} \parallel \sigma_2)$) is checked for well-formedness if and only if there is a finite tree (formal proof) with:

- the property is the root of the tree (conclusion of the proof)
- internal nodes are applications of inference rules
- the leaves are axioms (inference rules without hypothesis)

The purpose of the tree is to check well-formedness for the boxed components.

Unary constraints An unary compound property $\delta\mathtt{P}(\sigma)$ is well-formed if and only if there is a proof of $\boxed{\delta}\ \mathtt{P}(\boxed{\sigma})$ in the logical system of unary constraints given in Table 6.2.

$$\frac{}{\delta\boxed{\emptyset}\ \mathtt{P}(\sigma, \boxed{\emptyset})}\ (\mathrm{U-Correct})$$

$$\frac{\delta\boxed{\delta'}\ \mathtt{P}(\sigma, \underline{p}, \boxed{\sigma'}) \quad \underline{p} \notin \delta'}{\delta\boxed{\delta'}\ \mathtt{P}(\sigma, \boxed{\underline{p}, \sigma'})}\ (\mathrm{U-Fixed}) \qquad \frac{\delta \bullet_{p:\mathtt{T}} \boxed{\delta'}\ \mathtt{P}(\sigma, p : \mathtt{T}, \boxed{\sigma'})}{\delta\boxed{\bullet_{p:\mathtt{T}}\, \delta'}\ \mathtt{P}(\sigma, \boxed{p : \mathtt{T}, \sigma'})}\ (\mathrm{U-Derive})$$

Table 6.2: Well-formedness rules for unary compound properties

In these rules, we denote by "$\bullet$" a quantification operator which may be ? or *. Rule $(\mathrm{U-Correct})$ is the only axiom of the system *i.e.*, there is nothing left to be checked. Rule $(\mathrm{U-Derive})$ is applied to match a derivation $\bullet_{p:\mathtt{T}}$ with a parameter $p : \mathtt{T}$. The boxed property $\delta\boxed{\bullet_{p:\mathtt{T}}\, \delta'}\ \mathtt{P}(\sigma, \boxed{p : \mathtt{T}, \sigma'})$ is checked for well-formedness if and only if there is a tree such that the root is $\delta \bullet_{p:\mathtt{T}} \boxed{\delta'}\ \mathtt{P}(\sigma, p : \mathtt{T}, \boxed{\sigma'})$. The rule $(\mathrm{U-Fixed})$ can be used instead of $(\mathrm{U-Derive})$ in the case of the fixed parameter $\underline{p}$.

Binary constraints A binary compound property $\delta_1 \diamond \delta_2\mathtt{P}(\sigma_1 \parallel \sigma_2)$ is well-formed if and only if there is a proof of $\boxed{\delta_1} \diamond \delta_2\mathtt{P}(\boxed{\sigma_1} \parallel \sigma_2)$ in the logical system of binary constraints given in Table 6.3 and Table 6.4. Since the well-formedness of a binary compound property does not depend on the kind of combination operator $\diamond$ or $\blacklozenge$, if property

$\delta_1 \diamond \delta_2 \mathtt{P}(\sigma_1 \parallel \sigma_2)$ is well-formed then property $\delta_1 \blacklozenge \delta_2 \mathtt{P}(\sigma_1 \parallel \sigma_2)$ is well-formed, and vice-versa.

Well-formedness rules that are used to check well-formedness of left parameters and the corresponding derivation operators in binary compound properties are given in Table 6.3. Rule (B – LDerive) is applied to match a derivation $\bullet_{p:\mathtt{T}}$ with a parameter $p : \mathtt{T}$. The boxed property $\delta_1 \boxed{\bullet_{p:\mathtt{T}} \delta_1'} \blacklozenge \delta_2\ \mathtt{P}(\sigma_1, \boxed{p : \mathtt{T}, \sigma_1'} \parallel \sigma_2)$ is checked for well-formedness if and only if there is a tree with the root $\delta_1 \bullet_{p:\mathtt{T}} \boxed{\delta_1'} \blacklozenge \delta_2\ \mathtt{P}(\sigma_1, p : \mathtt{T}, \boxed{\sigma_1'} \parallel \sigma_2)$. Similarly as above, the rule (B – LFixed) can be used instead of (B – LDerive) in the case of the fixed parameter $\underline{p}$. The rule (Switch) is applied to switch checking well-formedness from left parameters to the right ones.

$$\frac{\delta_1 \blacklozenge \boxed{\delta_2}\ \mathtt{P}(\sigma_1 \parallel \boxed{\sigma_2})}{\delta_1 \boxed{\emptyset} \blacklozenge \delta_2\ \mathtt{P}(\sigma_1, \boxed{\emptyset} \parallel \sigma_2)}\ (\text{Switch})$$

$$\frac{\delta_1 \bullet_{p:\mathtt{T}} \boxed{\delta_1'} \blacklozenge \delta_2\ \mathtt{P}(\sigma_1, p : \mathtt{T}, \boxed{\sigma_1'} \parallel \sigma_2)}{\delta_1 \boxed{\bullet_{p:\mathtt{T}} \delta_1'} \blacklozenge \delta_2\ \mathtt{P}(\sigma_1, \boxed{p : \mathtt{T}, \sigma_1'} \parallel \sigma_2)}\ (\mathrm{B} - \text{LDerive})$$

$$\frac{\delta_1 \boxed{\delta_1'} \blacklozenge \delta_2\ \mathtt{P}(\sigma_1, \underline{p}, \boxed{\sigma_1'} \parallel \sigma_2) \quad \underline{p} \notin \delta_1'}{\delta_1 \boxed{\delta_1'} \blacklozenge \delta_2\ \mathtt{P}(\sigma_1, \boxed{\underline{p}, \sigma_1'} \parallel \sigma_2)}\ (\mathrm{B} - \text{LFixed})$$

Table 6.3: Well-formedness rules for *left parameters* of binary compound properties

Well-formedness rules that are used to check well-formedness of right parameters are given in Table 6.4. The rules (B – RDerive) and (B – RFixed) are similar to the rule (B – LDerive) and (B – LFixed), respectively. The rule (B – Correct) is the unique axiom of the system.

$$\frac{}{\delta_1 \blacklozenge \delta_2 \boxed{\emptyset}\ \mathtt{P}(\sigma_1 \parallel \sigma_2, \boxed{\emptyset})}\ (\mathrm{B} - \text{Correct}).$$

$$\frac{\delta_1 \blacklozenge \delta_2 \bullet_{q:\mathtt{T}} \boxed{\delta_2'}\ \mathtt{P}(\sigma_1 \parallel \sigma_2, q : \mathtt{T}, \boxed{\sigma_2'})}{\delta_1 \blacklozenge \delta_2 \boxed{\bullet_{q:\mathtt{T}} \delta_2'}\ \mathtt{P}(\sigma_1 \parallel \sigma_2, \boxed{q : \mathtt{T}, \sigma_2'})}\ (\mathrm{B} - \text{RDerive})$$

$$\frac{\delta_1 \blacklozenge \delta_2 \boxed{\delta_2'}\ \mathtt{P}(\sigma_1 \parallel \sigma_2, \underline{q}, \boxed{\sigma_2'}) \quad \underline{q} \notin \delta_2'}{\delta_1 \blacklozenge \delta_2 \boxed{\delta_2'}\ \mathtt{P}(\sigma_1 \parallel \sigma_2, \boxed{\underline{q}, \sigma_2'})}\ (\mathrm{B} - \text{RFixed})$$

Table 6.4: Well-formedness rules for *right parameters* of binary compound properties

Well-formedness checking A compound property is well-formed if it is derived by constraint rules. The derivation is a tree of rule applications with the given property as the root. To demonstrate the strategy of checking the well-formedness of a compound property, we consider an example for checking the well-formedness of a unary compound property

$$\mathtt{P} = ?_{r:\mathtt{Repl}}\, *_{i:\mathtt{Thrd}}\, \mathtt{active}(r:\mathtt{Repl}, i:\mathtt{Thrd}, 1).$$

Property P is well-formed because there is an inference for this property as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{?_{r:\mathtt{Repl}}\, *_{i:\mathtt{Thrd}}\, \boxed{\emptyset}\ \mathtt{active}(r, i, 1, \boxed{\emptyset})}(\mathsf{U}-\mathsf{Correct})}{?_{r:\mathtt{Repl}}\, *_{i:\mathtt{Thrd}}\, \boxed{\emptyset}\ \mathtt{active}(r:\mathtt{Repl}, i:\mathtt{Thrd}, \boxed{1})}(\mathsf{U}-\mathsf{Fixed})}{?_{r:\mathtt{Repl}}\, \boxed{*_{i:\mathtt{Thrd}}}\ \mathtt{active}(r:\mathtt{Repl}, \boxed{i:\mathtt{Thrd}}, 1)}(\mathsf{U}-\mathsf{Derive})}{\boxed{?_{r:\mathtt{Repl}}\, *_{i:\mathtt{Thrd}}}\ \mathtt{active}(\boxed{r:\mathtt{Repl}}, i:\mathtt{Thrd}, 1)}(\mathsf{U}-\mathsf{Derive})$$

Table 6.5: Well-formedness proof
of property $?_{r:\mathtt{Repl}}\, *_{i:\mathtt{Thrd}}\, \mathtt{active}(r:\mathtt{Repl}, i:\mathtt{Thrd}, 1)$

Property $?_{r:\mathtt{Repl}}\, ?_{v:\mathtt{Vert}}\, *_{i:\mathtt{Thrd}}\ \mathtt{active}(r:\mathtt{Repl}, i:\mathtt{Thrd}, v:\mathtt{Vert})$ is not well-formed because the derivation operators and parameters do not match. Operator $?_{v:\mathtt{Vert}}$ does not match parameter $i:\mathtt{Thrd}$, and operator $*_{i:\mathtt{Thrd}}$ does not match parameter $v:\mathtt{Vert}$.

6.1.3.3 Derivation of context properties

The derivation of an atomic context property φ, denoted by $\mathcal{D}_\pi[\![\varphi]\!]$, is itself, *i.e.*, $\mathcal{D}_\pi[\![\varphi]\!] = \varphi$. The derivation of a compound context property ϑ is a combination of atomic properties using logical conjunctions and disjunctions such that the combination has the same truth value as ϑ. We denote by $\mathcal{D}_\pi[\![\delta\ \mathtt{P}(\sigma)]\!]$ (resp. $\mathcal{D}_\pi[\![\delta_1 \blacklozenge \delta_2 \mathtt{P}(\sigma_1 \parallel \sigma_2)]\!]$) the derivation of property $\delta\ \mathtt{P}(\sigma)$ (resp. $\delta_1 \blacklozenge \delta_2 \mathtt{P}(\sigma_1 \parallel \sigma_2)$).

The derivation of a compound context property is obtained recursively through the derivation of compound properties with smaller sizes. As previously, we denote by $\bullet_{p:\mathtt{T}}$ a derivation operator of parameter p with type T, it may be $?_{p:\mathtt{T}}$ or $*_{p:\mathtt{T}}$. The derivation of the $\bullet$ itself is defined as follows:

$$\mathcal{D}_\pi[\![\bullet]\!] \stackrel{\text{def}}{=} \begin{cases} \wedge & \text{if } \bullet = * \\ \vee & \text{if } \bullet = ? \end{cases}$$

Derivation of unary properties We denote by $\mathcal{D}_\pi[\![\delta\mathtt{P}(\underline{\sigma_1}, \sigma_2)]\!]$ the derivation of the property for the remaining parameters σ_2 (σ_1 has already been derived, *i.e.*, fixed parameters). Recall that in an unary property, the first parameter is always a replicator identifier, after that may be a thread, a vertex, or a name.

First parameter If the first parameter (replicator identifier) of the property is not fixed, then it may be one of the replicators of the set $\mathcal{R}$ of the π-graph. In the derived property, the replicator parameter becomes fixed. Thus, we have the following rule:

$$\mathcal{D}_\pi[\![\bullet_{r:\texttt{Repl}}\ \delta\ \texttt{P}(r:\texttt{Repl},\sigma)]\!] \stackrel{\text{def}}{=} \mathcal{D}_\pi[\![\bullet]\!]_{\underline{r}\in\mathcal{R}}\ \mathcal{D}_\pi[\![\delta\ \texttt{P}(\underline{r},\sigma)]\!] \qquad \text{(USem-First)}$$

Next parameters If the first parameter, the replicator identifier r, is fixed and σ is a sequence of other instances of parameters, then the instance of parameter p depends on parameter type.

$$\mathcal{D}_\pi[\![\bullet_{p:\texttt{T}}\ \ \delta\ \texttt{P}(\underline{r,\sigma},p:\texttt{T},\sigma')]\!] \qquad \text{(USem-Param)}$$
$$\stackrel{\text{def}}{=} \begin{cases} \mathcal{D}_\pi[\![\bullet]\!]_{\underline{p}\in V_{\underline{r}}}\ \mathcal{D}_\pi[\![\delta\ \texttt{P}(\underline{r,\sigma,p},\sigma')]\!] & \text{if } \texttt{T}=\texttt{Vert} \\ \mathcal{D}_\pi[\![\bullet]\!]_{\underline{p}\in\mathcal{K}(\underline{r})}\ \mathcal{D}_\pi[\![\delta\ \texttt{P}(\underline{r,\sigma,p},\sigma')]\!] & \text{if } \texttt{T}=\texttt{Thrd} \\ \mathcal{D}_\pi[\![\bullet]\!]_{\underline{p}\in\mathcal{N}_{\underline{r}}}\ \mathcal{D}_\pi[\![\delta\ \texttt{P}(\underline{r,\sigma,p},\sigma')]\!] & \text{if } \texttt{T}=\texttt{Name} \end{cases}$$

where $\mathcal{N}_{\underline{r}}$ is the set of names in replicator $\underline{r}$.

If x is a fixed parameter, we mark it as that is derived and continue for the next parameter, as follows:

$$\mathcal{D}_\pi[\![\bullet_{p:\texttt{T}}\ \delta\ \texttt{P}(\underline{\sigma},x,\sigma')]\!] \quad \text{with } x\neq p \qquad \text{(USem-Fixed)}$$
$$\stackrel{\text{def}}{=} \mathcal{D}_\pi[\![\bullet_{p:\texttt{T}}\ \delta\ \texttt{P}(\underline{\sigma,x},\sigma')]\!]$$

End of derivation Finally, if all the parameters are instantiated, then

$$\mathcal{D}_\pi[\![\texttt{P}(\underline{\sigma})]\!] \stackrel{\text{def}}{=} \texttt{P}(\underline{\sigma}) \qquad \text{(USem-Final)}$$

Derivation of binary properties In binary properties, their parameters are composed of two parts and each of them always begin with a replicator identifier. The derivation of binary properties are determined recursively by instantiating parameters from left to right.

First parameters If the first parameter of two parts of parameters is not an instance, then the first replicator identifier parameter is instantiated first. Its instance may be one of replicator identifiers in set $\mathcal{R}$ of the π-graph. Thus, we have the following rule:

$$\mathcal{D}_\pi[\![\bullet_{r_1:\texttt{Repl}}\ \delta_1'\ \blacklozenge\ \bullet_{r_2:\texttt{Repl}}\ \delta_2'\ \texttt{P}(r_1:\texttt{Repl},\sigma_1\parallel r_2,\sigma_2)]\!] \qquad \text{(BSem-First)}$$
$$\stackrel{\text{def}}{=} \mathcal{D}_\pi[\![\bullet]\!]_{\underline{r_1}\in\mathcal{R}}\ \mathcal{D}_\pi[\![\delta_1'\ \blacklozenge\ \bullet_{r_2:\texttt{Repl}}\ \delta_2'\ \texttt{P}(\underline{r_1},\sigma_1\parallel r_2,\sigma_2)]\!]$$

If the first replicator parameter r_1 is not fixed and the second replicator parameter r_2 is fixed, then the possibilities of r_1 depend on the kind of combination operator. If it is

$\diamond$ operator, then r_2 must be excluded from all possibilities $\mathcal{R}$ of r_1.

$$\mathcal{D}_\pi[\![\bullet_{r_1:\mathtt{Repl}}\,\delta_1 \blacklozenge \delta_2\ \mathtt{P}(r_1,\sigma_1 \parallel \underline{r_2},\sigma_2)]\!] \qquad \text{(BSem-LFirst)}$$
$$\stackrel{\text{def}}{=} \mathcal{D}_\pi[\![\bullet]\!]_{\underline{r_1}\in\mathcal{R}}\ \mathcal{D}_\pi[\![\delta_1 \blacklozenge \delta_2\ \mathtt{P}(\underline{r_1},\sigma_1 \parallel \underline{r_2},\sigma_2)]\!]$$

$$\mathcal{D}_\pi[\![\bullet_{r_1:\mathtt{Repl}}\,\delta_1 \diamond \delta_2\ \mathtt{P}(r_1,\sigma_1 \parallel \underline{r_2},\sigma_2)]\!]$$
$$\stackrel{\text{def}}{=} \mathcal{D}_\pi[\![\bullet]\!]_{\underline{r_1}\in\mathcal{R}'}\ \mathcal{D}_\pi[\![\delta_1 \diamond \delta_2\ \mathtt{P}(\underline{r_1},\sigma_1 \parallel \underline{r_2},\sigma_2)]\!] \quad \text{with } \mathcal{R}' = \mathcal{R} \setminus \{\underline{r_2}\}$$

If all parameters in the first part are instantiated, then we continue with parameters in the second part. Similarly to rule (BSem-LFirst) we have:

$$\mathcal{D}_\pi[\![\blacklozenge \bullet_{r_2:\mathtt{Repl}}\,\delta_2\ \mathtt{P}(\underline{r_1,\sigma_1} \parallel r_2,\sigma_2)]\!] \qquad \text{(BSem-RFirst)}$$
$$\stackrel{\text{def}}{=} \mathcal{D}_\pi[\![\bullet]\!]_{\underline{r_2}\in\mathcal{R}}\ \mathcal{D}_\pi[\![\delta_2\ \mathtt{P}(\underline{r_1,\sigma_1} \parallel \underline{r_2},\sigma_2)]\!]$$

$$\mathcal{D}_\pi[\![\diamond \bullet_{r_2:\mathtt{Repl}}\,\delta_2\ \mathtt{P}(\underline{r_1,\sigma_1} \parallel r_2,\sigma_2)]\!]$$
$$\stackrel{\text{def}}{=} \mathcal{D}_\pi[\![\bullet]\!]_{\underline{r_2}\in\mathcal{R}'}\ \mathcal{D}_\pi[\![\delta_2\ \mathtt{P}(\underline{r_1,\sigma_1} \parallel \underline{r_2},\sigma_2)]\!] \quad \text{with } \mathcal{R}' = \mathcal{R} \setminus \{\underline{r_1}\}$$

Next parameters For other formal parameters:

$$\mathcal{D}_\pi[\![\bullet_{p:\mathtt{T}}\,\delta_1 \blacklozenge \delta_2\ \mathtt{P}(\underline{r_1,\sigma_1},p:\mathtt{T},\sigma_1' \parallel \sigma_2)]\!] \qquad \text{(BSem-LParam)}$$
$$\stackrel{\text{def}}{=} \begin{cases} \mathcal{D}_\pi[\![\bullet]\!]_{\underline{p}\in V_{r_1}}\ \mathcal{D}_\pi[\![\delta_1 \blacklozenge \delta_2\ \mathtt{P}(\underline{r_1,\sigma_1,p},\sigma_1' \parallel \sigma_2)]\!] & \text{if } \mathtt{T} = \mathtt{Vert} \\ \mathcal{D}_\pi[\![\bullet]\!]_{\underline{p}\in\mathcal{K}(r_1)}\ \mathcal{D}_\pi[\![\delta_1 \blacklozenge \delta_2\ \mathtt{P}(\underline{r_1,\sigma_1,p},\sigma_1' \parallel \sigma_2)]\!] & \text{if } \mathtt{T} = \mathtt{Thrd} \\ \mathcal{D}_\pi[\![\bullet]\!]_{\underline{p}\in\mathcal{N}_{r_1}}\ \mathcal{D}_\pi[\![\delta_1 \blacklozenge \delta_2\ \mathtt{P}(\underline{r_1,\sigma_1,p},\sigma_1' \parallel \sigma_2)]\!] & \text{if } \mathtt{T} = \mathtt{Name} \end{cases}$$

$$\mathcal{D}_\pi[\![\bullet_{q:\mathtt{T}}\,\delta_2\ \mathtt{P}(\underline{\sigma_1} \parallel \underline{r_2,\sigma_2},q:\mathtt{T},\sigma_2')]\!] \qquad \text{(BSem-RParam)}$$
$$\stackrel{\text{def}}{=} \begin{cases} \mathcal{D}_\pi[\![\bullet]\!]_{\underline{q}\in V_{\underline{r_2}}}\ \mathcal{D}_\pi[\![\delta_2\ \mathtt{P}(\underline{\sigma_1} \parallel \underline{r_2,\sigma_2,q},\sigma_2')]\!] & \text{if } \mathtt{T} = \mathtt{Vert} \\ \mathcal{D}_\pi[\![\bullet]\!]_{\underline{q}\in\mathcal{K}(\underline{r_2})}\ \mathcal{D}_\pi[\![\delta_2\ \mathtt{P}(\underline{\sigma_1} \parallel \underline{r_2,\sigma_2,q},\sigma_2')]\!] & \text{if } \mathtt{T} = \mathtt{Thrd} \\ \mathcal{D}_\pi[\![\bullet]\!]_{\underline{q}\in\mathcal{N}_{\underline{r_2}}}\ \mathcal{D}_\pi[\![\delta_2\ \mathtt{P}(\underline{\sigma_1} \parallel \underline{r_2,\sigma_2,q},\sigma_2')]\!] & \text{if } \mathtt{T} = \mathtt{Name} \end{cases}$$

For fixed parameters:

$$\mathcal{D}_\pi[\![\bullet_{p:\mathtt{T}}\,\delta_1' \blacklozenge \delta_2\ \mathtt{P}(\underline{\sigma_1'},x,\sigma_1'' \parallel \sigma_2)]\!] \quad \text{with } x \neq p \qquad \text{(BSem-LFixed)}$$
$$\stackrel{\text{def}}{=} \mathcal{D}_\pi[\![\bullet_{p:\mathtt{T}}\,\delta_1' \blacklozenge \delta_2\ \mathtt{P}(\underline{\sigma_1',x},\sigma_1'' \parallel \sigma_2)]\!]$$

$$\mathcal{D}_\pi[\![\bullet_{q:\mathtt{T}}\,\delta_2'\ \mathtt{P}(\underline{\sigma_1} \parallel \underline{\sigma_2'},x,\sigma_2'')]\!] \quad \text{with } x \neq q \qquad \text{(BSem-RFixed)}$$
$$\stackrel{\text{def}}{=} \mathcal{D}_\pi[\![\bullet_{q:\mathtt{T}}\,\delta_2'\ \mathtt{P}(\underline{\sigma_1} \parallel \underline{\sigma_2',x},\sigma_2'')]\!]$$

End of derivation When all parameters are fixed, the derivation of the property is defined as follows:

$$\mathcal{D}_\pi[\![\mathtt{P}(\underline{\sigma_1} \parallel \underline{\sigma_2})]\!] \stackrel{\text{def}}{=} \mathtt{P}(\underline{\sigma_1} \parallel \underline{\sigma_2}) \qquad \text{(BSem-Final)}$$

Remark: We assume the following structural equivalences:

i) $\underline{p, \sigma} \equiv \underline{p}, \underline{\sigma}$

ii) $\underline{\emptyset} \equiv \emptyset$

Illustration of derivation rules Consider a π-graph π in Figure 6.2, which represents a sub part of the RDP system that is described in Chapter 3. We determine the derivation of the following compound property:

$$\mathtt{P} = *_{r_1:\mathtt{Repl}}\, ?_{i_1:\mathtt{Thrd}}\, ?_{v_1:\mathtt{Vert}} \diamond ?_{i_2:\mathtt{Thrd}}\ \mathtt{sync}(r_1 : \mathtt{Repl}, i_1 : \mathtt{Thrd}, v_1 : \mathtt{Vert} \parallel R, i_2 : \mathtt{Thrd}, 11).$$

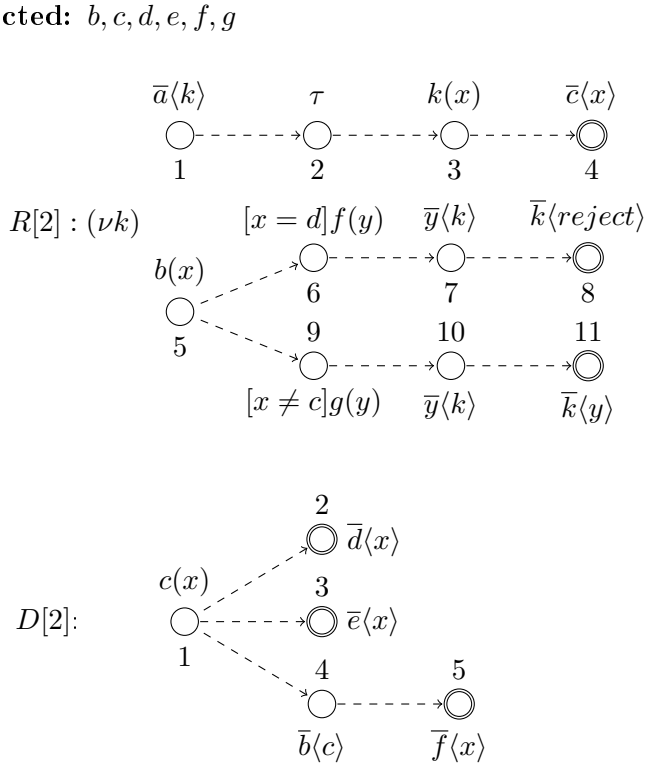


Figure 6.2: A sub part of the RDP system

Property P asserts that there exists the synchronization between the action at $(R, 11)$ and all the other replicators (in fact here there is only D because the second replicator is already fixed to R), which is indicated by the combination operator $\diamond$. The derivation

of P is determined as follows:

$$\begin{aligned}
\mathcal{D}_\pi[\![P]\!] &\stackrel{\text{def}}{=} \mathcal{D}_\pi[\![?_{i_1:\texttt{Thrd}}\, ?_{v_1:\texttt{Vert}}\, \diamond\, ?_{i_2:\texttt{Thrd}}\, \texttt{sync}(D, i_1 : \texttt{Thrd}, v_1 : \texttt{Vert} \parallel R, i_2 : \texttt{Thrd}, 11)]\!] \\
&\stackrel{\text{def}}{=} \left[\begin{array}{ll} \mathcal{D}_\pi[\![?_{v_1:\texttt{Vert}}\, \diamond\, ?_{i_2:\texttt{Thrd}}\, \texttt{sync}(D, 1, v_1 : \texttt{Vert} \parallel R, i_2 : \texttt{Thrd}, 11)]\!] & \vee \\ \mathcal{D}_\pi[\![?_{v_1:\texttt{Vert}}\, \diamond\, ?_{i_2:\texttt{Thrd}}\, \texttt{sync}(D, 2, v_1 : \texttt{Vert} \parallel R, i_2 : \texttt{Thrd}, 11)]\!] & \end{array}\right. \\
&\stackrel{\text{def}}{=} \left[\begin{array}{ll} \bigvee_{\underline{v_1} \in [1..5]} \mathcal{D}_\pi[\![\diamond\, ?_{i_2:\texttt{Thrd}}\, \texttt{sync}(D, 1, \underline{v_1} \parallel R, i_2 : \texttt{Thrd}, 11)]\!] & \vee \\ \bigvee_{\underline{v_1} \in [1..5]} \mathcal{D}_\pi[\![\diamond\, ?_{i_2:\texttt{Thrd}}\, \texttt{sync}(D, 2, \underline{v_1} \parallel R, i_2 : \texttt{Thrd}, 11)]\!] & \end{array}\right. \\
&\quad \ldots\ldots \\
&\stackrel{\text{def}}{=} \left[\begin{array}{ll}
\texttt{sync}(D,1,1 \parallel R,1,11) \vee \texttt{sync}(D,1,1 \parallel R,2,11) & \vee \\
\texttt{sync}(D,1,2 \parallel R,1,11) \vee \texttt{sync}(D,1,2 \parallel R,2,11) & \vee \\
\texttt{sync}(D,1,3 \parallel R,1,11) \vee \texttt{sync}(D,1,3 \parallel R,2,11) & \vee \\
\texttt{sync}(D,1,4 \parallel R,1,11) \vee \texttt{sync}(D,1,4 \parallel R,2,11) & \vee \\
\texttt{sync}(D,1,5 \parallel R,1,11) \vee \texttt{sync}(D,1,5 \parallel R,2,11) & \vee \\
\texttt{sync}(D,2,1 \parallel R,1,11) \vee \texttt{sync}(D,2,1 \parallel R,2,11) & \vee \\
\texttt{sync}(D,2,2 \parallel R,1,11) \vee \texttt{sync}(D,2,2 \parallel R,2,11) & \vee \\
\texttt{sync}(D,2,3 \parallel R,1,11) \vee \texttt{sync}(D,2,3 \parallel R,2,11) & \vee \\
\texttt{sync}(D,2,4 \parallel R,1,11) \vee \texttt{sync}(D,2,4 \parallel R,2,11) & \vee \\
\texttt{sync}(D,2,5 \parallel R,1,11) \vee \texttt{sync}(D,2,5 \parallel R,2,11) &
\end{array}\right.
\end{aligned}$$

6.1.3.4 Semantics of context properties

The interpretation of context properties is defined recursively as follows, using the definition of atomic properties:

Definition 6.1. *Let $\mathcal{C}_\pi$ be a π-graphs context and P a context property. P holds in context $\mathcal{C}_\pi$, denoted by $\mathcal{C}_\pi \models P$, iff $\mathsf{interp}_{\mathcal{C}_\pi}(\mathcal{D}_\pi[\![P]\!])$, with*

$$\mathsf{interp}_{\mathcal{C}_\pi}(p) \stackrel{\text{def}}{=} \begin{cases} \mathsf{true} & \textit{if the atomic property } p \textit{ holds in } \mathcal{C}_\pi \\ \mathsf{false} & \textit{otherwise} \end{cases}$$

$$\mathsf{interp}_{\mathcal{C}_\pi}(P_1 \vee P_2) \stackrel{\text{def}}{=} \mathsf{interp}_{\mathcal{C}_\pi}(P_1) \vee \mathsf{interp}_{\mathcal{C}_\pi}(P_2)$$

$$\mathsf{interp}_{\mathcal{C}_\pi}(P_1 \wedge P_2) \stackrel{\text{def}}{=} \mathsf{interp}_{\mathcal{C}_\pi}(P_1) \wedge \mathsf{interp}_{\mathcal{C}_\pi}(P_2)$$

Definition 6.2. *Let π be a π-graph and $\mathcal{C}_\pi$ a π-graphs context. The set of all atomic properties that hold in context $\mathcal{C}_\pi$, denoted by $\mathsf{atoms}(\mathcal{C}_\pi)$, is defined as follows:*

$$\mathsf{atoms}(\mathcal{C}_\pi) = \{p \textit{ an atomic context property} \mid \mathsf{interp}_{\mathcal{C}_\pi}(p)\}.$$

6.2 Temporal properties of π-graphs

Temporal properties of π-graphs are defined based on context properties.

6.2.1 Kripke structure for π-graphs

We introduce a Kripke structure for π-graphs based on their semantics as labelled transition system (LTS) (cf. Def. 4.9 on page 81). The main idea is to map each state in the LTS to the set of atomic context properties that hold in this state.

We begin with the classical definition of *Kripke structures*.

Definition 6.3. *A* Kripke structure *over a set of atomic properties LP is a transition system $KS = (\mathscr{S}, s^0, \mathscr{T}, \mathscr{P})$, where*

- $\mathscr{S}$ *a finite set of states*
- $s^0 \in \mathscr{S}$ *is an initial state*
- $\mathscr{T} \in \mathscr{S} \times \mathscr{S}$ *is a transition relation*
- $\mathscr{P} : \mathscr{S} \to 2^{LP}$ *is a property map (or interpretation function)*

Now we explain the mapping from π-graphs semantics to Kripke structures.

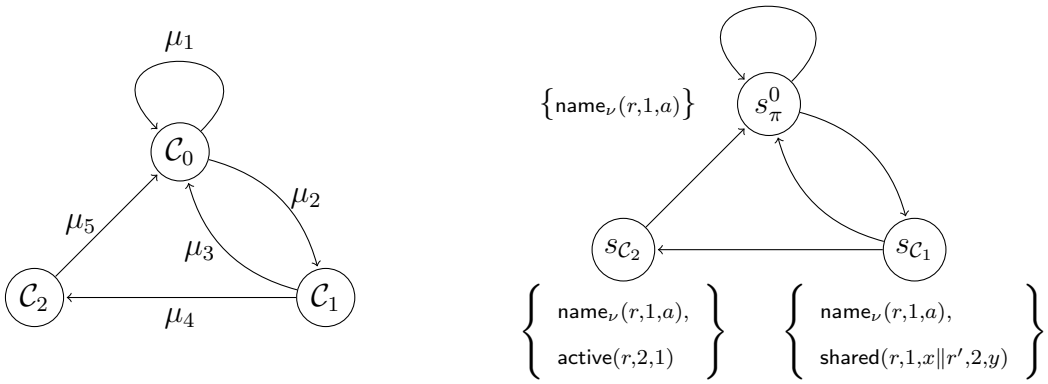
Definition 6.4. *Let $LTS_\pi = (\mathscr{C}_\pi, \mathcal{C}^0_\pi, \mathscr{R}_\pi)$ be a labelled transition system of a π-graph π, the corresponding Kripke structure is $KS_\pi = (S_\pi, s^0_\pi, T_\pi, P_\pi)$ such that:*

- $S_\pi = \{s_\mathcal{C} \mid \mathcal{C} \in \mathscr{C}_\pi\}$
- s^0_π *is* $s_{\mathcal{C}^0_\pi}$
- $T_\pi = \{(s_\mathcal{C}, s_{\mathcal{C}'}) \mid \mathcal{C} \xrightarrow{\mu} \mathcal{C}' \in \mathscr{R}_\pi\}$
- P_π *is such that for a state $s_\mathcal{C} \in S_\pi$ then $P_\pi(s_\mathcal{C})$ is the set of atomic context properties that are true in context $\mathcal{C}$. Formally, $P_\pi(s_\mathcal{C}) = \mathsf{atoms}(\mathcal{C})$.*

Consider a π-graph π with three reachable contexts $\mathcal{C}_0, \mathcal{C}_1$ and $\mathcal{C}_2$ where $\mathcal{C}_0$ is the initial context, and π consists of two replicators r and r'. The LTS of π is given in Figure 6.3a and the corresponding Kripke structure is given in Figure 6.3b with three states $s^0_\pi, s_{\mathcal{C}_1}$ and $s_{\mathcal{C}_2}$ corresponding to $\mathcal{C}_0, \mathcal{C}_1$ and $\mathcal{C}_2$, respectively. Each state is associated with a set of atomic properties that hold in this state, for example, $P_\pi(s_{\mathcal{C}_1}) = \{\mathsf{name}_\nu(r, 1, a), \mathsf{shared}(r, 1, x \parallel r', 2, y)\}$.

Definition 6.5. *Let $KS = (\mathscr{S}, s^0, \mathscr{T}, \mathscr{P})$ be a Kripke structure. A* path *of KS is a sequence of states $\rho = (s_0, s_1, \dots)$ such that for each $i \geq 0, (s_i, s_{i+1}) \in \mathscr{T}$. We denote by:*

- $\rho[i]$ *the i-th state of the path ρ, i.e., $\rho[i] = s_i$;*
- $\rho[..i]$ *the i-th* prefix *of ρ, i.e., $\rho[..i] = (s_0, s_1, \dots, s_i)$;*
- $\rho[i..]$ *the i-th* suffix *of ρ, i.e., $\rho[i..] = (s_i, s_{i+1}, \dots)$.*



(a) The LTS of π (b) The Kripke structure

Figure 6.3: The LTS and the corresponding Kripke structure of π-graph π

6.2.2 Syntax of π-graphs temporal properties

Definition 6.6. *A* temporal property formula φ *is formed as follows:*

$$\varphi ::= \mathsf{true} \mid p \mid \varphi_1 \wedge \varphi_2 \mid \neg\varphi \mid \mathrm{X}\,\varphi \mid \varphi_1 \,\mathrm{U}\, \varphi_2$$

where p is a context property.

For other logical connectives such as disjunction, implication, equivalence, and exclusive are derived from basic operators in Def. 6.6 as follows:

$$\begin{aligned}
\varphi_1 \vee \varphi_2 &\equiv \neg(\neg\varphi_1 \wedge \neg\varphi_2) \\
\varphi_1 \rightarrow \varphi_2 &\equiv \neg\varphi_1 \vee \varphi_2 \\
\varphi_1 \leftrightarrow \varphi_2 &\equiv (\varphi_1 \rightarrow \varphi_2) \wedge (\varphi_2 \rightarrow \varphi_1) \\
\varphi_1 \oplus \varphi_2 &\equiv (\varphi_1 \wedge \neg\varphi_2) \vee (\varphi_2 \wedge \neg\varphi_1)
\end{aligned}$$

Moreover, for a full temporal logic, we add temporal operators always G and eventually F. They are derived from operators X and U as follows:

$$\begin{aligned}
\mathrm{F}\,\varphi &\equiv \mathsf{true}\,\mathrm{U}\,\varphi \\
\mathrm{G}\,\varphi &\equiv \neg\,\mathrm{F}\,\neg\varphi
\end{aligned}$$

Definition 6.7. *Given a temporal property φ and a π-graph π, the derivation of φ on π extends the definition of the derivation of context properties with the following rules:*

$$\begin{aligned}
\mathcal{D}_\pi[\![\mathsf{true}]\!] &\stackrel{\text{def}}{=} \mathsf{true} \\
\mathcal{D}_\pi[\![\varphi_1 \wedge \varphi_2]\!] &\stackrel{\text{def}}{=} \mathcal{D}_\pi[\![\varphi_1]\!] \wedge \mathcal{D}_\pi[\![\varphi_2]\!] \\
\mathcal{D}_\pi[\![\varphi_1 \vee \varphi_2]\!] &\stackrel{\text{def}}{=} \mathcal{D}_\pi[\![\varphi_1]\!] \vee \mathcal{D}_\pi[\![\varphi_2]\!] \\
\mathcal{D}_\pi[\![\neg\varphi]\!] &\stackrel{\text{def}}{=} \neg\mathcal{D}_\pi[\![\varphi]\!] \\
\mathcal{D}_\pi[\![\mathrm{X}\,\varphi]\!] &\stackrel{\text{def}}{=} \mathrm{X}\,\mathcal{D}_\pi[\![\varphi]\!] \\
\mathcal{D}_\pi[\![\varphi_1 \,\mathrm{U}\, \varphi_2]\!] &\stackrel{\text{def}}{=} \mathcal{D}_\pi[\![\varphi_1]\!] \,\mathrm{U}\, \mathcal{D}_\pi[\![\varphi_2]\!]
\end{aligned}$$

6.2.3 Semantics of temporal properties

Some reminders for the intuitive meaning of temporal modalities is depicted in Fig. 6.4. For simplicity, we just illustrate the meaning of the modalities with atomic properties. A property p without any temporal operator is satisfied if it holds in the current state, but p may hold or not in the successor states. Property $\mathrm{X}\,p$ holds if p holds at the state after the current one. Property $p\,\mathrm{U}\,q$ holds if p holds until property q holds. Property $\mathrm{F}\,p$ holds if there is a state in the future such that p holds on it. Finally, property $\mathrm{G}\,p$ holds if p holds in all states.

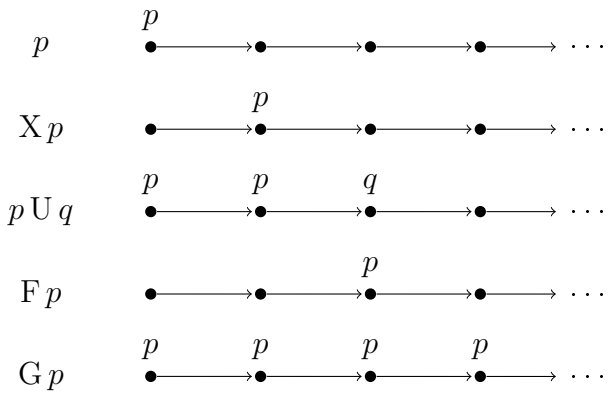


Figure 6.4: Intuitive meaning of temporal modalities

Definition 6.8. *A* word *over a set of local atomic properties LP is a sequence of sets of atomic properties, $w = (w_0, w_1, w_2, \dots)$, i.e., $w_i \in 2^{LP}$ for any $i \geq 0$. We denote $w[i], w[..i]$, and $w[i..]$ the i-th element, i-th prefix and i-suffix of the word w, respectively.*

Definition 6.9. *A word $w = (w_0, w_2, \dots)$ satisfies a temporal property formula φ, denoted by $w \models \varphi$, if and only if $w \models_D \mathcal{D}_\pi[\![\varphi]\!]$, where the satisfaction relation $\models_D$ is defined as follows.*

$$
\begin{array}{lll}
w \models_D \mathsf{true} & & \\
w \models_D p & \textit{iff} & p \in w_0, \textit{ with } p \textit{ is a local atomic property} \\
w \models_D \varphi_1 \wedge \varphi_2 & \textit{iff} & (w \models_D \varphi_1) \wedge (w \models_D \varphi_2) \\
w \models_D \neg\varphi & \textit{iff} & w \not\models_D \varphi \\
w \models_D \mathrm{X}\,\varphi & \textit{iff} & w[1..] \models_D \varphi \\
w \models_D \varphi_1\,\mathrm{U}\,\varphi_2 & \textit{iff} & \exists j \geq 0 \quad w[j..] \models_D \varphi_2, \textit{ and } \forall i, 0 \leq i < j \quad w[i..] \models_D \varphi_1.
\end{array}
$$

Definition 6.10. *Let φ be a temporal property formula over a set of local atomic properties LP. The set of words over LP induced by φ is*

$$\mathsf{words}(\varphi) = \{w \in (2^{LP})^\omega \mid w \models \varphi\},$$

where $(2^{LP})^\omega$ denotes the set of words that arise from the infinite concatenation of words in 2^{LP}.

Definition 6.11. *Let $KS = (\mathscr{S}, s^0, \mathscr{T}, \mathscr{P})$ be a Kripke structure and φ a temporal property formula over a set of local atomic properties LP.*

- *For a path $\rho = (s_0, s_1, \dots)$ of KS, the satisfaction relation is defined by*

$$\rho \models \varphi \quad \textit{iff} \quad (w_0, w_1, \dots) \models \varphi \textit{ with } w_i = \mathscr{P}(s_i) \textit{ for any } i \geq 0;$$

- *For a state s in $\mathscr{S}$, the satisfaction relation is defined by*

$$s \models \varphi \quad \textit{iff} \quad (\forall \rho \in \mathsf{paths}(s),\ \rho \models \varphi),$$

  where $\mathsf{paths}(s)$ denotes all paths that start from s;

- *KS satisfies φ, denoted by $KS \models \varphi$, iff $s^0 \models \varphi$.*

6.2.4 Examples of π-graphs temporal properties

We introduce some examples of properties of π-graphs and their specifications. These properties can be divided into three groups: safety properties, liveness properties and others.

6.2.4.1 Safety properties

Informally, safety properties assert that "Nothing bad will happen". The following are some examples of important safety properties:

1. Deadlock-freedom: In any context, the system can always evolve by performing an atomic action or a synchronization, *i.e.*,

$$\begin{aligned}&\mathrm{G}(\,?_{r:\mathtt{Repl}}\,?_{i:\mathtt{Thrd}}\,?_{v:\mathtt{Vert}}\ \ \mathsf{atomic}(r:\mathtt{Repl}, i:\mathtt{Thrd}, v:\mathtt{Vert}) \quad \vee \\ &\quad ?_{r_1:\mathtt{Repl}}\,?_{i_1:\mathtt{Thrd}}\,?_{v_1:\mathtt{Vert}} \lozenge\, ?_{r_2:\mathtt{Repl}}\,?_{i_2:\mathtt{Thrd}}\,?_{v_2:\mathtt{Vert}} \\ &\qquad \mathsf{sync}(r_1:\mathtt{Repl}, i_1:\mathtt{Thrd}, v_1:\mathtt{Vert} \parallel r_2:\mathtt{Repl}, i_2:\mathtt{Thrd}, v_2:\mathtt{Vert}) \\ &\quad)\end{aligned}$$

2. The property asserts that a given global restricted name x is never sent to the environment (because of some security requirement), meaning that for all the replicators and all the threads in each replicator, the instance of a given name x is always a private name, is defined as follows:

$$\mathrm{G}(*_{r:\mathtt{Repl}}\,*_{i:\mathtt{Thrd}}\ \ \mathsf{name}_{\nu}(r:\mathtt{Repl}, i:\mathtt{Thrd}, \underline{x})).$$

   In the above definition, name x is a constant that is indicated by $\underline{x}$.

3. The number of effective fresh input/output names space is bounded. For example, the number of fresh input names is always less than or equal to 6.

   Since fresh input names are generated by a logical input clock of the form $?k$, where $k \in \mathbb{N}^+$, and a fresh name $?k$ is generated only if all fresh names $?j$, with $j < k$, are being used (*i.e.*, existing in the co-domain of the name environment β) (cf. Section 4.2.2.3 on page 62), K is the bound of effective fresh input names if name $?k$, where $k = K + 1$, is never generated. It means that property

   $$?_{r:\texttt{Repl}}\, ?_{i:\texttt{Thrd}}\, ?_{v:\texttt{Vert}}\ \mathsf{label}_{\alpha}(r : \texttt{Repl}, i : \texttt{Thrd}, v : \texttt{Vert})), \text{ with } \alpha = *(?k)$$

   never holds. This property can be specified using temporal operator G as follows:

   $$\mathrm{G}(\neg\, ?_{r:\texttt{Repl}}\, ?_{i:\texttt{Thrd}}\, ?_{v:\texttt{Vert}}\ \mathsf{label}_{\alpha}(r : \texttt{Repl}, i : \texttt{Thrd}, v : \texttt{Vert})), \text{ with } \alpha = *(?k), k = K + 1.$$

   For fresh output names, the property is similar except that $\alpha = *{<}!k{>}$.

6.2.4.2 Liveness properties

Informally, liveness properties assert that "Something good will eventually happen". The following are examples of some important liveness properties:

1. Eventually, all threads will be released (*i.e.*, the system will go to a state where all local names will be reset). We recall that if a thread i in replicator r is released, then there is no vertex v having control for i, meaning that $\mathsf{active}(\underline{r}, \underline{i}, \underline{v})$ is false. The property is defined as follows:

   $$\mathrm{F}(\neg\, ?_{r:\texttt{Repl}}\, ?_{i:\texttt{Thrd}}\, ?_{v:\texttt{Vert}}\ \mathsf{active}(r : \texttt{Repl}, i : \texttt{Thrd}, v : \texttt{Vert}))$$

2. A local restricted name x in replicator r will eventually be escaped, *i.e.*, its instantiation is not a fresh private name. This property is defined as follows:

   $$\mathrm{F}(\neg\, ?_{i:\texttt{Thrd}}\ \mathsf{name}_{\nu}(r, i : \texttt{Thrd}, x)).$$

   Similarly, if x is a global restricted name, then the property which asserts that x will eventually be escaped is defined as follows:

   $$\mathrm{F}(\neg\, *_{r:\texttt{Repl}}\, ?_{i:\texttt{Thrd}}\ \mathsf{name}_{\nu}(r : \texttt{Repl}, i : \texttt{Thrd}, x)).$$

6.2.4.3 Other properties

We list here the properties which are not in the two previous categories.

- In the definition of replicators, each one has a bounded number of threads. In some cases, this is not the effective bound, *i.e.*, the number of threads that are spawned may be less than the given bound. A number n is not an effective bounded number of a replicator r if in all context, there is no vertex in r that has control for the thread with identifier n.

$$\mathrm{G}(\neg\ ?_{v:\texttt{Vert}}\ \mathsf{active}(r, n, v : \texttt{Vert}))$$

- Checking if there is any output/input on a given channel. This property is specified using property $\texttt{label}_\alpha(r : \texttt{Repl}, i : \texttt{Thrd}, v : \texttt{Vert})$. There is an output on a channel c if and only if there is an action at (r, v) that is activated using thread i with label $\overline{c}\langle * \rangle$. This property is defined as follows:

$$?_{r:\texttt{Repl}}\, ?_{i:\texttt{Thrd}}\, ?_{v:\texttt{Vert}}\ \texttt{label}_\alpha(r : \texttt{Repl}, i : \texttt{Thrd}, v : \texttt{Vert})$$

  where $\alpha = \overline{c}\langle * \rangle$. Similarly for an input action on channel c, the label in this case is $\alpha = c(*)$.

6.3 Model checking with the NECO framework

Now that we have a formalism – the π-graphs – for specifying reconfigurable systems as well as a language to describe properties about the behaviours of these systems, the next step is to be able to model-check these properties in an automated way. It would be difficult and time-consuming to develop a specific model checker [17, 23] for the π-graphs. Thankfully we can exploit the one-to-one correspondence between the π-graphs and Petri nets stated in Section 5.2 on page 95. Since the translated net is a high-level Petri net, two tools appear to be particularly adapted to this case: HELENA [26] and SNAKES+NECO [28][52]. We finally decided to use SNAKES+NECO for the following reasons:

- To conduct the experiment on the semantics of π-graphs, we implemented a π-graphs simulator using programming language Python. Moreover, SNAKES is a Python library that provides all that is necessary to define and execute many sorts of Petri nets including the high-level Petri nets in our translation. Thus, it is reasonable to choose SNAKES for the translation and checking the conformance between the π-graphs and the translated Petri nets.

- NECO is a model checker based on SNAKES. It can verify LTL properties of Petri nets models which are defined in SNAKES in an efficient way relying on the high performance of the SPOT framework [7, 22].

This section presents first how to express LTL properties of Petri nets in NECO. Next, we illustrate how to translate π-graphs properties into the logic of NECO, following our translation scheme. Then, we show how to interpret a counterexample if the property does not hold. Finally, the section ends with a practical experiment of our approach.

Unlike the rest of the thesis, the following presentation will mostly remain informal.

6.3.1 Specifying properties in NECO

A property of a Petri net model in NECO [27] is based on atomic propositions of Petri nets markings. The main atomic propositions are described below:

- Comparisons between two lists of tokens or two integer expressions using operators in $\{=, !=, <=, <, >, >=\}$.

  1. A list of tokens $[\mathtt{tok}_1, \ldots, \mathtt{tok}_n]$ expresses actually a multi-set of values $\{\mathtt{tok}_1, \ldots, \mathtt{tok}_n\}$. In particular, NECO representation of markings uses lists of tokens, *e.g.*, $\mathsf{marking}(p)$ where p is a place name.
  2. An integer expression can be an integer or a sum of integers. In particular, the number of tokens in a list L, which is computed using the function $\mathsf{card}(L)$ is also an integer expression.

- Predefined atomic propositions on transitions to check the firing possibilities of transitions at a given marking.

  – $\mathsf{fireable}(t)$ meaning that transition t can be fired at the current marking.
  – deadlock meaning that there is no transition that can be fired at the current marking.

- Predefined atomic propositions on places to check if a user-defined property f is true on a token tok, denoted by $f(\mathtt{tok})$.

  – $\mathsf{all}(p, f)$ is true for all token tok in the place p, $f(\mathtt{tok})$ is true. If there is no token in p, then $\mathsf{all}(p, f)$ is also true.
  – $\mathsf{any}(p, f)$ is true if there exists for some token tok in place p such that $f(\mathtt{tok})$ is true. If there is no token in p, then $\mathsf{any}(p, f)$ is false.

- Combinations of atomic propositions to form compound properties using:

  – Classical logic connectives: and, or, xor, not, ->, <->.
  – Temporal logic operators: X (next), G (always), F (eventually), and U (until).

Examples of property specifications in NECO

We consider two examples of specifying properties in NECO. Given a Petri net (as in Fig. 6.5) which comprises two places p_1 and p_2, and a transition t. Transition t consumes a token from p_1 and produces the same token to p_2. The guard of t is empty meaning that it is always satisfied. Initially, p_1 contains three tokens $1, 3$ and 5, p_2 contains two tokens 2 and 4.

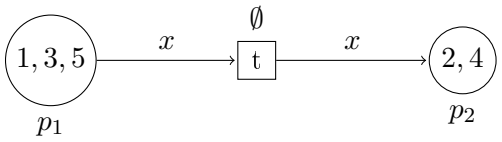


Figure 6.5: An example of Petri net model

First, let us consider the following property P_1 of the Petri net at a given marking,

$$P_1 \stackrel{\text{def}}{=} ([1,3] <= \mathsf{marking}(p_1)) \quad \mathsf{and} \quad (\mathsf{card}(\mathsf{marking}(p_2)) <= 3).$$

It is true if place p_1 contains at least two tokens 1 and 3, and the number of tokens in p_2 is less than or equal to 3.

If we want to check if P_1 holds sometimes in the future, we should add temporal operator F (eventually) in front of P_1. The obtained property P_1' asserts about a path of markings,

$$P_1' \stackrel{\text{def}}{=} \mathrm{F}(([1,3] <= \mathsf{marking}(p_1)) \quad \mathsf{and} \quad (\mathsf{card}(\mathsf{marking}(p_2)) <= 3)).$$

Second, let us consider another property P_2 that holds for a given marking: "All tokens in place p_1 are odd numbers and there is at least one token in place p_2 such that it is an even number". It may be expressed in the following way using user-defined Boolean functions on a given token. These functions must be defined before using predefined atomic propositions $\mathsf{all}(p, f)$ and $\mathsf{any}(p, f)$ of NECO. In general, to specify and check such a property, we have to:

1. Define user-defined functions in a separate Python module. For example, the definitions of two functions odd(tok) and even(tok), which check if the token tok is an odd (resp. even) number should be defined in such a module.

2. Specify the desired properties using these user-defined functions together with constructs "for all tokens" and "there exists a token". In this example, function odd applies to all tokens in place $p1$ and function even applies to some tokens in place $p2$.

The specification of property P_2 is then in NECO:

$$P_2 \stackrel{\text{def}}{=} \mathsf{all}(p_1, \mathsf{odd}) \quad \mathsf{and} \quad \mathsf{any}(p_2, \mathsf{even}).$$

Similarly to the first example, we can add temporal operators to P_2 to obtain new properties that assert a formula about a path of markings.

6.3.2 Translating properties of π-graphs

According to the definition of a temporal property of π-graphs, it is a combination of context properties using classical connectives and temporal logic operators. Moreover, for any context property specification (atomic or compound one), by derivation rules, we can obtain an equivalent formula which is a conjunction or disjunction of corresponding atomic properties. Thus, to translate a temporal property into an equivalent one that can be checked using NECO model checker, it is enough to have a method for translating atomic context properties into the corresponding ones in NECO.

Furthermore, according to the classification of context properties of π-graphs based on the number and the order of parameters, each atomic property may be in one of four groups: RI, RIV, RIN, RIVN. More abstract, these properties can be regrouped into those about threads and those about names. The former relate purely to threads, they can be translated without using user-defined functions. The latter relate to the name context, and the translation needs user-defined functions that apply to the unique name context token in place p_Γ of the translated net.

6.3.2.1 Translation of thread properties

Among atomic context properties of π-graphs specified in Sec. 6.1.2 on page 121, there are two properties that can be translated directly into the equivalent one of the translated net in NECO:

$$\mathtt{active}(r : \mathtt{Repl}, i : \mathtt{Thrd}, v : \mathtt{Vert}), \quad \text{and} \quad \mathtt{avail}(r : \mathtt{Repl}, i : \mathtt{Thrd}, v : \mathtt{Vert}).$$

Recall that in the translated net of a π-graph, each non-terminating vertex v in replicator r has a corresponding transition t_v^r and an output place p_v^r of t_v^r. Thus, if a vertex v in a replicator r has control for a thread i, then it means that the token $r.i$ is present in the place p_v^r. The translation of property $\mathtt{active}(r, i, v)$ is thus defined as follows:

$$\mathtt{active}(r : \mathtt{Repl}, i : \mathtt{Thrd}, v : \mathtt{Vert}) \stackrel{\text{def}}{=} [r.i] <= \mathsf{marking}(p_v^r).$$

The availability of a thread i for performing an action at (r, v) depends on the structural type of the action. If its type is mono or spwn (*i.e.*, $\mathsf{deg}_r^-(v) = 0$), then thread i

is available to the action at (r, v) if it is not present at any vertex in r. Otherwise, *i.e.*, $\mathsf{deg}_r^-(v) = 0$, i is available to the action at (r, v) if it is present at the predecessor of v. The translation of property $\texttt{avail}(r : \texttt{Repl}, i : \texttt{Thrd}, v : \texttt{Vert})$ also based on the marking of places in the translated net as follows:

$$\texttt{avail}(r : \texttt{Repl}, i : \texttt{Thrd}, v : \texttt{Vert})$$
$$\stackrel{\text{def}}{=} \begin{cases} [r.i] <= \mathsf{marking}(p_u^r) & \text{if } \mathsf{deg}_r^-(v) = 1 \text{ where } (u, v) \in E_r \\ \neg(\bigvee_{u \in V_r \setminus \{w \mid \mathsf{deg}_r^+(w)=0\}}([r.i] <= \mathsf{marking}(p_u^r))) & \text{if } \mathsf{deg}_r^-(v) = 0 \end{cases}$$

6.3.2.2 Translation of name context properties

An atomic context property that relates to the name context of π-graphs must be translated indirectly using predefined atomic propositions $\mathsf{all}(p_\Gamma, f)$ and $\mathsf{any}(p_\Gamma, f)$, where f is a user-defined Boolean function that applies to the name context token Γ in place p_Γ.

Since Γ is a singleton in p_Γ (cf. Def. 5.2), the evaluations of $\mathsf{all}(p_\Gamma, f)$ and $\mathsf{any}(p_\Gamma, f)$ are the same at a given marking, we should choose arbitrarily one of them for the translation of properties. Suppose that we choose $\mathsf{all}(p_\Gamma, f)$.

In fact, function f may has other free parameters, let's say $p_1, \ldots, p_n$, besides a bound parameter Γ (which represents the unique name context token in place p_Γ). However, because of the syntax of the two predefined atomic propositions $\mathsf{all}(p_\Gamma, f)$ and $\mathsf{any}(p_\Gamma, f)$, we cannot pass these free parameters. This problem can be solved by defining a new function

$$g : \Gamma \times \mathcal{P}_1 \times \cdots \times \mathcal{P}_n \to \{\texttt{true}, \texttt{false}\}, \tag{6.5}$$

where $\mathcal{P}_i$ is the domain of parameter p_i. Then, for each distinct tuple of effective parameters $(p_1^k, \ldots, p_n^k)$ we define a new function

$$f^k : \Gamma \to \{\texttt{true}, \texttt{false}\} \quad \text{where } f^k(\Gamma^k) \stackrel{\text{def}}{=} g(\Gamma^k, p_1^k, \ldots, p_n^k).$$

In our implementation, a name context token is an instance of class `NameContext`, thus (6.5) may be implemented as a method of `NameContext` and the application

$$g(\Gamma^k, p_1^k, \ldots, p_n^k) \quad \text{becomes a method call} \quad \Gamma^k.g(p_1^k, \ldots, p_n^k).$$

Functions $f^k(\Gamma)$ are user-defined functions as described above.

6.3.3 Processing counterexamples

Given a model $\mathsf{net}(\pi)$ which is the translated net of a π-graph π, and a property φ_π which is the translation of a π-graphs property Φ_π, we check if the model satisfies the property using the model checker NECO. Similarly to other model checkers, if the model

satisfies φ_π then the model checker will give us an answer "OK", otherwise, it gives us a counterexample, which is a trace of markings that does not satisfy the property. The counterexample can be abstracted as follows:

$$\mathsf{M}_0[\varphi_\pi^0] \to \mathsf{M}_2[\varphi_\pi^2] \to \cdots \to \mathsf{M}_n[\varphi_\pi^n], \tag{6.6}$$

where M_i is a marking of the translated net and φ_π^i is the assertion of property φ_π on marking M_i, and there exists a marking $\mathsf{M}_k, 0 \leq k \leq n$, such that the corresponding assertion φ_π^k is false.

Based on the counterexample in (6.6) we can recover which context of the π-graph does not satisfy the property Φ_π. Recall that the translation of π-graphs into Petri nets is isomorphic (cf. Sec. 5.2), thus we can always compute the context from the marking based on Def. 5.10 on page 109. Moreover, the property φ_π^k is a conjunction of atomic properties and the checker also show us which atomic property is not satisfied. Thus, we have more information about the action or names that made that property false.

6.4 Experimental results

This section presents experimental results of using open system logic to specify properties of systems modelled in π-graphs, and using NECO to check these properties. The system we use for this purpose is represented in Figure 6.6, which is presented in Chapter 3. Moreover, we present also the interpretation of a counter-example from NECO in π-graphs, and propose a solution for modifying the model of the system such that it satisfies the property.

Suppose that we want to check if the system satisfies three properties: deadlock-freedom, no internal loop and internal private channel. They are described as follows:

Deadlock-freeness (P1) The system has no deadlock if at any reachable state there exists at least an atomic action or a synchronization that can be activated. It can be defined based on atomic properties `atomic` and `sync` as follows:

$$P1 \stackrel{\text{def}}{=} \mathrm{G}\left(\begin{array}{l} ?_{r:\texttt{Repl}}\, ?_{i:\texttt{Thrd}}\, ?_{v:\texttt{Vert}}\, \texttt{atomic}(r:\texttt{Repl}, i:\texttt{Thrd}, v:\texttt{Vert}) \\ \vee(?_{r_1:\texttt{Repl}}\, ?_{i_1:\texttt{Thrd}}\, ?_{v_1:\texttt{Vert}} \blacklozenge\, ?_{r_2:\texttt{Repl}}\, ?_{i_2:\texttt{Thrd}}\, ?_{v_2:\texttt{Vert}} \\ \quad \texttt{sync}(r_1:\texttt{Repl}, i_1:\texttt{Thrd}, v_1:\texttt{Vert} \parallel r_2:\texttt{Repl}, i_2:\texttt{Thrd}, v_2:\texttt{Vert})) \end{array}\right)$$

NECO provides an atomic proposition `deadlock` asserting that the current state of the system is deadlock, *i.e.*, there is no fireable transition. We can use this proposition to define $P1$.

No internal loop (P2) In each replicator, the information it sends to another replicator (or sent to the environment) should not be received by itself. For example, in

replicator R, the answer y to the requester, which is represented by the action $\overline{k}\langle y\rangle$, should not be received by R itself by an input action, *e.g.*, $k(x)$. It is defined as follows:

$$P2 \stackrel{\text{def}}{=} \mathrm{G}\left(\begin{array}{l}\neg(?_{i_1:\texttt{Thrd}} \blacklozenge ?_{i_2:\texttt{Thrd}} \texttt{sync}(R, i_1 : \texttt{Thrd}, 8 \parallel R, i_2 : \texttt{Thrd}, 3)) \\ \wedge\neg(?_{i_1:\texttt{Thrd}} \blacklozenge ?_{i_2:\texttt{Thrd}} \texttt{sync}(R, i_1 : \texttt{Thrd}, 11 \parallel R, i_2 : \texttt{Thrd}, 3))\end{array}\right)$$

NECO provides an atomic proposition $\mathsf{fireable}(t)$ where t is a transition that asserts that t is fireable in the current state, which may be used to define $P2$. In the translated net, there are two transitions representing the synchronizations between the output at $(R, 1)$, $(R, 8)$ and $(R, 11)$ with the input at $(R, 3)$, named $S1R3R$, $S8R3R$ and $S11R3R$, respectively. Each synchronization can be activated if and only if the corresponding transition can be fired. Thus, we have another definition for P2 as follows:

$$P2 \stackrel{\text{def}}{=} \mathrm{G}(\neg\ \mathsf{fireable}(S8R3R) \wedge \neg\ \mathsf{fireable}(S11R3R) \wedge \neg\ \mathsf{fireable}(S1R3R)).$$

Similarly, a property may concern replicator P where the output $\overline{g}\langle x\rangle$ should not synchronize with the input $d(x)$ or $e(x)$. In this case study, we consider only the property for replicator R.

Internal private channel (P3) All global restricted channels of the system (*i.e.*, $X = \{b, c, d, e, f, g\}$) are always private. It may be defined as follows:

$$P3 \stackrel{\text{def}}{=} \mathrm{G}(\bigwedge_{n\in X} *_{r:\texttt{Repl}} *_{i:\texttt{Thrd}} \texttt{name}_\nu(r : \texttt{Repl}, i : \texttt{Thrd}, n)).$$

6.4.1 Checking results for the original model

We carried numerous experiments on the original model, which is represented in Figure 6.6 on the next page. The results are presented in Table 6.6. Each line corresponds to a case of $(\mathcal{K}_R, \mathcal{K}_D, \mathcal{K}_P)$, in which a satisfied property is marked with symbol ✓, otherwise it is marked with ✗.

In Table 6.6, property P_1 (*i.e.*, deadlock-freedom) does not hold in all cases, other ones hold for all cases. For each case, NECO gives us a counter example for P_1. We consider the one for the first case, *i.e.*, $\mathcal{K}_R = \mathcal{K}_D = \mathcal{K}_P = 1$, which is represented in Listing 6.1. The property that NECO checks is indicated in the first line of the listing, 'G (!DEAD)" (*i.e.*, always not deadlock).

We recall the reader that to check this property, NECO computes the negation of the property, "FDEAD" (eventually deadlock), and checks if there exists a path that satisfies the negation. The prefix of the path consists of a sequence of markings that does not satisfy the negation, *i.e.*, each marking is evaluated to !DEAD, and ends with

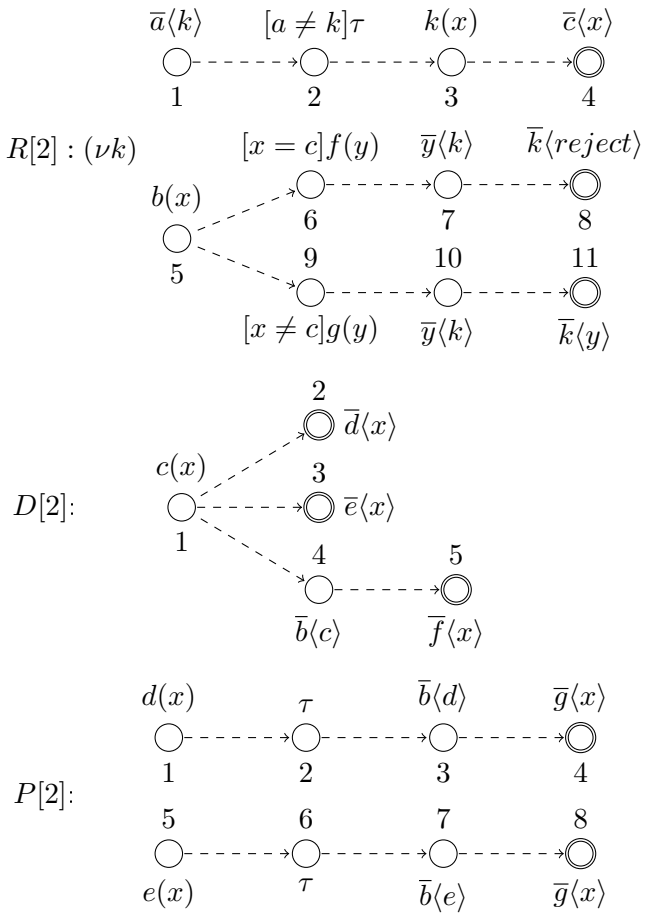


Figure 6.6: The original π-graph model of the RDP system

a marking that satisfies the negation (lines 13 to 15), which is evaluated to DEAD. The cycle part (lines 17 to 20) is consists of one marking, which is exactly the deadlock marking. Moreover, by Def. 5.2, place p3R corresponds to vertex 3 of replicator R, p1D corresponds to vertex 1 of D, p2P corresponds to vertex 1 of P and pG corresponds to the context of the π-graph model. Notice that we only show the places that have token. By Def. 5.3, place p3R has token 1 if the action at vertex 3 of R has control with thread 1. Similarly, that action at vertex 1 of D has control with thread 1 and vertex 1 of P also has control with thread 1. The context token at place pG is represented by three parts which corresponds to the name environment, the dynamic partition and the distinctions (cf. Section 4.2.2.2). Thus, in the corresponding context of the deadlock marking,

- R is waiting for communicating with D,
- D is waiting for communicating with P,
- P is waiting for communicating with R.

No.	Thread bounds			Time (sec)		Num of	Properties		
	$\mathcal{K}_R$	$\mathcal{K}_D$	$\mathcal{K}_P$	compile	explore	states	P_1	P_2	P_3
1	1	1	1	401.36	2.64	162	✗	✓	✓
2	1	1	2	400.97	54.92	2545	✗	✓	✓
3	1	1	3	400.89	1645.40	52080	✗	✓	✓
4	1	2	1	401.31	19.74	1090	✗	✓	✓
5	1	2	2	399.98	602.60	21644	✗	✓	✓
6	2	1	1	402.02	1373.51	50337	✗	✓	✓

Table 6.6: Results of checking on the original model

No.	Thread bounds			Times (sec)		Num of	Properties		
	$\mathcal{K}_R$	$\mathcal{K}_D$	$\mathcal{K}_P$	compile	explore	states	P_1	P_2	P_3
1	1	1	1	401.29	0.71	18	✓	✓	✓
2	1	1	2	393.30	0.72	18	✓	✓	✓
3	1	1	3	395.21	0.70	18	✓	✓	✓
4	1	2	1	400.10	0.75	18	✓	✓	✓
5	1	2	2	399.54	0.71	18	✓	✓	✓
6	2	1	1	399.67	534.41	27211	✓	✓	✓

Table 6.7: Results for the improved model in Figure 6.7

But these three replicators cannot perform the input action at $(R, 5), (D, 1), (P, 1)$ and $(P, 5)$ as they have no available threads. This corresponds to a deadlock in the π-graph.

6.4.2 Checking results for the improved model

From the interpretation of the counter-example above, we propose a solution for modifying the model such that the system is deadlock-free by making the action $b(x)$ at $(R, 5)$ become the successor of the action at $(R, 4)$; the other replicators do not change. This modification ensures that each thread of R has to wait for receiving the answer from D or P after sending a request (by performing action $\overline{c}\langle x\rangle$).

The new model is represented in Figure 6.7 and the checking results for it are given in Table 6.7, meaning that all properties P_1, P_2 and P_3 are satisfied. Moreover, the number of states are the same in all cases in which $\mathcal{K}_R = 1$, meaning that D and P need only one thread for performing actions. This gives rise to a property of the system: if $\mathcal{K}_R = 1$, then the effective thread bounds of both D and P are 1. The property can be specified analogously to that in Section 6.2.4.3 as follows:

$$\mathrm{G}(\neg\, ?_{v:\mathtt{Vert}}\, \mathtt{active}(D, 2, v) \wedge \neg\, ?_{v:\mathtt{Vert}}\, \mathtt{active}(P, 2, v)).$$

Listing 6.1: The original output from NECO of a counterexample of property $P1$, where pG represents the context place p_Γ

```
1  Formula: (G (!DEAD))
2  Prefix:
3    {p0R: [1], p0D: [1], p0P: [1],
4      pG: [{b:$2, c:$1,...}; {$1:1, $2:2,...}; {1-2,...}]
5    }* FDEAD | !DEAD
6
7    {p1R: [1], p0D: [1], p0P: [1],
8      pG: [{k.R.1:!1, c:$1,...}; {!1:1, $1:2,...}; {1-2,...}]
9    }* FDEAD | !DEAD
10   .
11   .
12   .
13   {p3R: [1], p1D: [1], p2P: [1],
14     pG: [{x.D.1:?2, x.P.1:?1, x.R.1:?3,...}; {?1:8, ?2:9, ?3:10,...}; {...}]
15   }* FDEAD | DEAD
16
17 Cycle:
18   {p3R: [1], p1D: [1], p2P: [1],
19     pG: [{x.D.1:?2, x.P.1:?1, x.R.1:?3,...}; {?1:8, ?2:9, ?3:10,...}; {...}]
20   }* 1 | DEAD {Acc[1]}
```

6.5 Synthesis

This chapter deals with specifying properties of systems modelled in π-graphs and checking them using the model checker NECO.

To specify properties, we develop a high-level variant of the linear temporal logic, which allows to specify properties about the dynamic evolution of the systems taking into account interactions within open environments. The logic provides a set of atomic propositions, a set of derivation rules and constraints. Atomic propositions capture precisely the state properties of π-graphs processes. Derivation rules allow to derive all compound properties and temporal properties in this framework into equivalent ones, which are conjunctions and disjunctions of atomic ones. Constraints allow to check the well-formedness of property formulae.

To check these properties, which are specified in this logic, using the model checker NECO, we present informally a method for translating atomic properties into the corresponding ones in NECO. Some properties on π-graphs, such as deadlock-freedom, may have several ways of translating into the equivalent ones on Petri nets. On one hand, these properties may be defined using atomic ones of the logic and then translated into equivalent ones in Petri nets. On the other hand, NECO provides a possibility to translate these properties, such as deadlock or fireable, directly into the ones on Petri nets. In

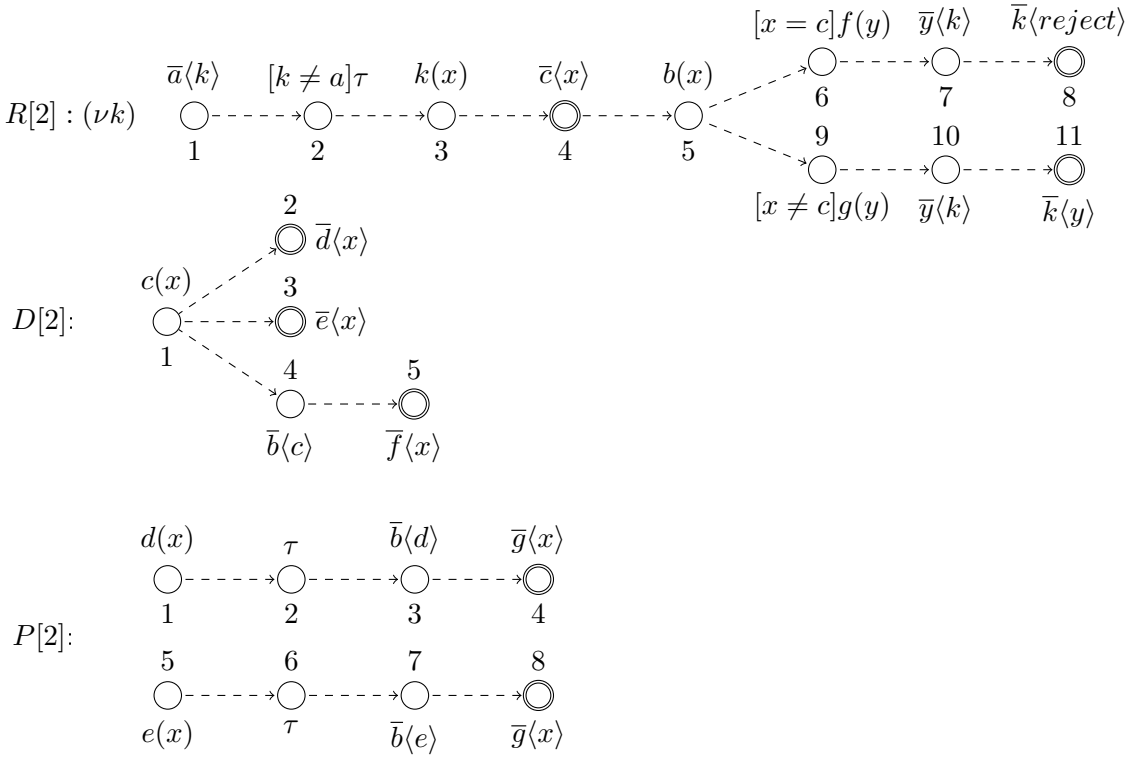


Figure 6.7: The modified π-graph model of the RDP system

general, these two ways of translating of properties give different but hopefully equivalent results (which still should be proved).

The chapter ends with experimental results on checking properties of the system RDP, gives some indications on how to interpret the counter-example and proposes a solution for modifying the model such that the system satisfies the property.

Chapter 7

Conclusion and Perspectives

We summarize the main results of the thesis and then discuss some future works.

7.1 Summary

First, we recall the main objective of this thesis: to develop the theory of π-graphs – our visual variant of the π-calculus – enough so that modelling and verification tools can be developed to support the design and reasoning about open reconfigurable system. Of course, the development of an ad-hoc tool would be an unbearable among of work, especially for the purpose of automated verification. Hence, our approach is to translate the π-graphs models into Petri nets and then reuse existing Petri net tools for the verification part.

The translation consists of first translating the structure of the π-graphs into a corresponding high-level Petri nets structure, and then translating each configuration (state) of π-graphs into a corresponding marking of the translated net. Unlike the previous attempts [21, 48], the Petri nets used for the translation are not of a specific class, and corresponds to a class that is commonly supported by existing Petri nets tools, such as SNAKES+NECO. It is a structural translation thus the size of the translated Petri net is polynomial. Hence, unlike semantic translations, our translation can always be applied without involving any combinatorial explosion (which can of course still occur later when performing model-checking). Moreover, the translation can be performed on-the-fly for any configuration of π-graphs, thus we can check properties while simulating the models. Put in other terms, we can control the verification of properties, hopefully reducing the state-space to explore.

We also proved that the translation is an isomorphism in the sense that for each transition in the π-graph with label μ, there exists an occurrence with the same label in the translated Petri net. And vice-versa, for each occurrence with label μ in the translated

net, there exists a transition with the same label in the π-graph. Moreover, it is easy to recover a configuration of a π-graph from a given reachable marking of the translated Petri net, and vice-versa.

In addition, to specify properties of open systems, we developed the open system logic. The basic part of the logic is a set of atomic properties that capture the state properties of π-graphs processes. All more complex properties in the logic can be derived from the atomic ones. As an extension of our translation to Petri nets, the open system logic properties can be translated to equivalent properties about Petri nets. These translated properties can then be model-checked using existing Petri tools. If a property does not hold for a system, then the model checker will provide a counterexample in terms of Petri nets. Thanks to the isomorphism, we can recover easily the π-graphs configuration that does not satisfy the property as well as the computation path reaching this configuration.

Finally, the whole framework has been implemented during the course of this thesis. Our prototype tool supports the following functionalities:

- encode a π-graphs model in a textual form,
- explore the state-space of a model and simulate transitions,
- export the transitions of a system in the form of a labelled transitions system into a textual representation, which can be used by external tools such as GRAPHVIZ in order to visualize the evolution of the system.
- translate a π-graphs model into Petri nets and manipulate the latter using the SNAKES framework,
- specify properties of π-graphs in the open system logic,
- translate such properties into the logic of NECO model-checker and verify them on the translated nets,
- reinterpret the results in terms of π-graphs.

In conclusion, we reached our primary goals, which demonstrates in our opinion that the *detour* we made within the Petri nets world could be seen as overkill but was ultimately rewarding.

7.2 Future works

In the future, we will continue to develop our approach in the following directions:

First, unlike the translation of systems that we proved formally correct, the translation of the logic properties remains mostly informal. However, the fact that we have

implemented and heavily tested this logical translation makes us quite confident about the feasibility of such a formal proof. Indeed, one major component of the proof would be the correspondence of states and this part of the proof is already available. The second part would be to show that the set of states that satisfy a π-graphs formula is in a one-to-one correspondence with the set of markings of the translated Petri net that satisfy the translated formula.

Our approach does not translate directly an abstract π-calculus but uses an intermediate model of π-graphs. Now that we know what must be put into the intermediate model so that the translation works, it would be interesting to connect our translation to a more abstract calculus. The most interesting aspect would be to study if the logical properties could be abstracted similarly. At the other end of the translation, we intend to characterize a precise Petri nets class together with composition operators that would allow to remove the distinction between the π-graphs on the one side and the Petri nets on the other side. We are confident that this characterization is possible based on the argument that our current translation is actually an isomorphism. This would make our work closer to the Petri box calculus but in the context of open reconfigurable systems.

We also intend to make our prototype tool more user-friendly, especially taking advantages of the visual representation of π-graphs.

Bibliography

[1] Graphviz. `http://www.graphviz.org/`.

[2] LoLA tool. `http://www.informatik.uni-rostock.de/tpp/lola/`.

[3] Mpsat tool. `http://homepages.cs.ncl.ac.uk/victor.khomenko/tools/mpsat/`.

[4] Petri nets world. `http://www.informatik.uni-hamburg.de/TGI/PetriNets/`.

[5] PETRUCHIO tool. `http://petruchio.informatik.uni-oldenburg.de/`.

[6] Luca Aceto, Anna Ingólfsdóttir, Kim G Larsen, Jiři Srba, et al. *Reactive systems: modelling, specification and verification*, volume 8. Cambridge University Press Cambridge, 2007.

[7] Duret-Lutz Alexandre. Spot. `http://spot.lip6.fr`.

[8] Christel Baier, Joost-Pieter Katoen, et al. *Principles of model checking*. MIT press Cambridge, 2008.

[9] Martin Berger. An interview with Robin Milner. `http://www.sussex.ac.uk/Users/mfb21/interviews/milner/`, September 2003.

[10] Eike Best, Raymond Devillers, and Jon G Hall. *The box calculus: a new causal algebra with multi-label communication*. Springer, 1992.

[11] Eike Best, Raymond Devillers, and Maciej Koutny. *Petri net algebra*. Monographs in Theoretical Computer Science. An EATCS Series. Springer, 2001.

[12] Eike Best, Wojciech Frączak, Richard P Hopkins, Hanna Klaudel, and Elisabeth Pelz. M-nets: An algebra of high-level petri nets, with an application to the semantics of concurrent programming languages. *Acta Informatica*, 35(10):813–857, 1998.

[13] Michele Boreale and Rocco De Nicola. Testing equivalence for mobile processes. *Inf. Comput.*, 120(2):279–303, 1995.

[14] Nadia Busi. Analysis issues in Petri nets with inhibitor arcs. *Theoretical Computer Science*, 275(1):127–177, 2002.

[15] Nadia Busi and Roberto Gorrieri. Distributed semantics for the π-calculus based on Petri nets with inhibitor arcs. *The Journal of Logic and Algebraic Programming*, 78(3):138–162, 2009.

[16] Luís Caires. Behavioral and spatial observations in a logic for the pi-calculus. In Igor Walukiewicz, editor, *FoSSaCS*, volume 2987 of *Lecture Notes in Computer Science*, pages 72–89. Springer, 2004.

[17] Edmund M Clarke. The birth of model checking. In *25 Years of Model Checking*, pages 1–26. Springer, 2008.

[18] Mads Dam. Model checking mobile processes. In Eike Best, editor, *CONCUR*, volume 715 of *Lecture Notes in Computer Science*, pages 22–36. Springer, 1993.

[19] Raymond Devillers, Hanna Klaudel, and Maciej Koutny. Context-based process algebras for mobility. In *ACSD*, pages 79–88, 2004.

[20] Raymond Devillers, Hanna Klaudel, and Maciej Koutny. Modelling mobility in high-level petri nets. In Twan Basten, Gabriel Juhás, and Sandeep K. Shukla, editors, *ACSD*, pages 110–119, 2007.

[21] Raymond Devillers, Hanna Klaudel, and Maciej Koutny. A compositional Petri net translation of general π-calculus terms. *Formal Aspects of Computing*, 20(4-5):429–450, 2008.

[22] Alexandre Duret-Lutz and Denis Poitrenaud. Spot: an extensible model checking library using transition-based generalized Büchi automata. In *Modeling, Analysis, and Simulation of Computer and Telecommunications Systems, 2004.(MASCOTS 2004). Proceedings. The IEEE Computer Society's 12th Annual International Symposium on*, pages 76–83. IEEE, 2004.

[23] E Allen Emerson. The beginning of model checking: A personal perspective. In *25 Years of Model Checking*, pages 27–45. Springer, 2008.

[24] Javier Esparza and Mogens Nielsen. *Decidability issues for Petri nets*. BRICS, Computer Science Department, University of Aarhus, 1994.

[25] Sami Evangelista. High level Petri nets analysis with Helena. In Gianfranco Ciardo and Philippe Darondeau, editors, *Applications and Theory of Petri Nets 2005*, volume 3536 of *Lecture Notes in Computer Science*, pages 455–464. Springer Berlin Heidelberg, 2005.

[26] Sami Evangelista and Christophe Pajault. Helena–A high level net analyzer. `http://lipn.univ-paris13.fr/~evangelista/helena/`.

[27] Lucasz Fonc. Neco. `https://code.google.com/p/neco-net-compiler/`.

[28] Łukasz Fronc and Alexandre Duret-Lutz. Ltl model checking with neco. In Dang Van Hung and Mizuhito Ogawa, editors, *Automated Technology for Verification and Analysis*, volume 8172 of *Lecture Notes in Computer Science*, pages 451–454. Springer International Publishing, 2013.

[29] Michel Hack. *Decidability questions for Petri nets.* Garland, 1979.

[30] Ryszard Janicki and P. E. Lauer. *Specification and Analysis of Concurrent Systems: The COSY Approach.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1992.

[31] Kurt Jensen. *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use. Volume 1.* Springer-Verlag, 1997.

[32] Kurt Jensen and Lars M Kristensen. *Coloured Petri nets: modelling and validation of concurrent systems.* Springer, 2009.

[33] Victor Khomenko. A usable reachability analyser. *School of Comp. Sci., Newcastle Univ., Tech. Rep. CS-TR-1140*, 2009.

[34] Hanna Klaudel and Franck Pommereau. M-nets: a survey. *Acta informatica*, 45(7-8):537–564, 2008.

[35] P. E. Lauer. *Path Expression and Petri Nets, or Petri Nets with Fewer Tears.* Newcastle upon Tyne, England: University of Newcastle upon Tyne, Computing Laboratory, MRM 70, Jan, 1974.

[36] Marko Mäkelä. Maria: Modular reachability analyser for algebraic system nets. In *Application and Theory of Petri Nets 2002*, pages 434–444. Springer, 2002.

[37] Roland Meyer. *Structural stationarity in the π-calculus.* PhD thesis, Carl-von-Ossietzky-Univ., Department für Informatik, 2009.

[38] Roland Meyer. A theory of structural stationarity in the π-calculus. *Acta Informatica*, 46(2):87–137, 2009.

[39] Roland Meyer, Victor Khomenko, and Reiner Hüchting. A polynomial translation of π-calculus (FCP) to safe petri nets. *CONCUR 2012–Concurrency Theory*, pages 440–455, 2012.

[40] Robin Milner. *A calculus of communicating systems.* Springer, 1980.

[41] Robin Milner. *Communicating and mobile systems: the pi calculus.* Cambridge university press, 1999.

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

[43] E.-R. Olderog. *Nets, Terms and Formulas: Three Views of Concurrent Processes and Their Relationship*. Cambridge University Press, New York, NY, USA, 1991.

[44] Joachim Parrow. An introduction to the-calculus. *Handbook of process algebra*, pages 479–543, 2001.

[45] F. Peschanski, H. Klaudel, and R. Devillers. Pigraphs with replicators: Finiteness and boundedness results. Technical report, IBISC-Universite d'Evry-Val d'Essonne, 2012.

[46] Frédéric Peschanski and Joël-Alexis Bialkiewicz. Modelling and verifying mobile systems using π-graphs. *SOFSEM 2009: Theory and Practice of Computer Science*, pages 437–448, 2009.

[47] Frédéric Peschanski, Hanna Klaudel, and Raymond Devillers. A decidable characterization of a graphical pi-calculus with iterators. *arXiv preprint arXiv:1011.0220*, 2010.

[48] Frédéric Peschanski, Hanna Klaudel, and Raymond Devillers. A Petri net interpretation of open reconfigurable systems. *Fundamenta Informaticae*, 122(1):85–117, 2013.

[49] James L Peterson. *Petri net theory and the modeling of systems*. Prentice Hall PTR, 1981.

[50] Carl Adam Petri. Communication with automata, new york: Griffiss air force base. Technical report, Tech. Rep. RADC-TR-65-377, 1966.

[51] Franck Pommereau. SNAKES. `https://code.google.com/p/python-snakes/`.

[52] Franck Pommereau. Quickly prototyping Petri nets tools with SNAKES. In *Proceedings of the 1st International Conference on Simulation Tools and Techniques for Communications, Networks and Systems & Workshops*, Simutools '08, pages 17:1–17:10. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2008.

[53] Wolfgang Reisig. *Petri nets: an introduction*. Springer-Verlag New York, Inc., 1985.

[54] Wolfgang Reisig. *Understanding Petri Nets: Modeling Techniques, Analysis Methods, Case Studies*. Springer Science & Business, 2013.

[55] Davide Sangiorgi and David Walker. *The pi-calculus: a Theory of Mobile Processes*. Cambridge university press, 2003.

[56] Karsten Schmidt. LoLA a low level analyser. In *Application and Theory of Petri Nets 2000*, pages 465–474. Springer, 2000.

[57] Dirk Taubner. *Finite representations of CCS and TCSP programs by automata and Petri nets*, volume 369. Springer, 1989.

[58] Björn Victor and Faron Moller. The Mobility Workbench—a tool for the π-calculus. In *Computer Aided Verification*, pages 428–440. Springer Berlin Heidelberg, 1994.