# Declaração

**Nome:** Rúben André Marques da Cruz

**Endereço eletrónico:** pg30869@alunos.uminho.pt

**Telefone:** 934737030

**Nº do Bilhete de Identidade:** 14043408

**Título da Dissertação de Mestrado:**

Web-based Analysis of Reo Connectors

**Designação do Mestrado:**

Mestrado em Matemática e Computação

**Orientadores:**

Doutor José Carlos Soares do Espírito Santo

Doutor José Miguel Paiva Proença

**Ano de Conclusão:** 2018



Universidade do Minho, 31 de Outubro de 2018

**O autor**: Rúben André Marques da Cruz

# Acknowledgements

The work presented in this document marks the end of my academic life. There have been many ups and downs during these long years. Many people have helped me in some way or another, to finish this line. I want to use this little page to thank everyone I think is/was important.

First I want to thank my mother for providing me with the tools and environment to study during these years. I understand the difficulties and sacrifices she had to make for me to be where I am. I also want to thank her for giving me a second chance when I failed the first time, and I hope this work makes her proud.

I also want to thank my little sister, Leila, for being my friend when I needed. I am not an easy person to deal with, and other times she is difficult to deal with as well, but many more times, we support each other as brother and sister, laughing and playing together.

Cristiana has given me so much love, friendship, and support. Her presence in my life is light and warmth, that guides me when I'm cold and everything else is dark. When I doubt myself she gives me the comfort and help to be confident.

To all the friends that I met during all these years, I want to thank for the moments of relaxation.

I also want to thank my supervisors, professor José Carlos Soares Espírito Santo, and professor José Miguel Paiva Proença, for the help and guidance that was given to me throughout this year. Without this guidance I would be lost and could not have done this work. A very special thank you to professor José Miguel Paiva Proença, for giving me this project and making me feel valued for my work.

### Financing

This work is financed by the ERDF – European Regional Development Fund through the Operational Programme for Competitiveness and Internationalisation - COMPETE 2020 Programme and by National Funds through the Portuguese funding agency, FCT - Fundação para a Ciência e a Tecnologia within project POCI-01-0145-FEDER-016692 and POCI-01-0145-FEDER-029946.

# Abstract

**Web-based Analysis of Reo Connectors**

Combining independent software components usually requires additional code which can be difficult to maintain and is prone to errors. Connectors are used to describe how to combine independent components by restricting the possible interactions between their interfaces. The main focus of this work is on the formal analysis of Reo connectors, and a calculus of Reo connectors.

The literature related to the formal analysis of Reo connectors provides different perspectives on it, which sometimes do not fit with each other on every level. The first part of this document presents the Reo connectors, accompanied by background work related to port automata semantics and an mCRL2 encoding of connectors. The remaining documented work focuses on the calculus of Reo connectors for which there are no tools to automatically analyse this calculus, other than a type-checker for an embedded domain specific language. We formalize this calculus in terms of port automata semantics, and provide an encoding, based on this semantics, to mCRL2. We also show that the formalized semantics, and the mCRL2 encoding are correct.

Finally, we present a set of web-based tools for analysing connectors–named ReoLive–requiring only an offline Internet browser with JavaScript support, which also supports a client-server architecture for more complex operations. ReoLive includes tools that generate port automata, mCRL2 processes, and graphical representations of instances of connectors, developed in the Scala language and compiled into JavaScript. The resulting framework is publicly available, and can be easily experimented without any installation or a running server.

# Resumo

**Análise Web de Conetores Reo**

Combinar componentes de *software* independentes requer geralmente código adicional, que pode ser difícil de manter e facilmente conduz a erros. Conetores são usados para descrever a combinação de componentes independentes, através de restrições nas possíveis interações entre as suas interfaces. O principal foco deste trabalho encontra-se na análise formal de conetores Reo, e num cálculo de conetores Reo.

A literatura relacionada com a análise formal de conetores Reo providencia perspetivas diferentes sobre esta, que, por vezes, não se encaixam perfeitamente entre si. A primeira parte deste documento apresenta os conetores Reo, e completa esta apresentação com algum trabalho anterior relacionado com a semântica em autómatos de porta e codificação de conetores em mCRL2. O restante trabalho documentado, centra-se no cálculo de conetores Reo, para o qual não existem atualmente ferramentas para analisar automaticamente este cálculo, para além de um *type-checker* para uma linguagem específica de conetores. Neste documento formalizamos o cálculo em termos de semântica de autómatos de porta, e definimos uma codificação, baseada nesta semântica, em mCRL2. Para além disso, mostramos que a semântica e a codificação estão corretos.

Por último, introduzimos um conjunto de ferramentas baseadas em web para analisar conetores—chamada ReoLive— que requer apenas um explorador de internet offline, que suporte JavaScript, e que também permite uma arquitetura de cliente-servidor para operações mais complexas. O ReoLive incluí ferramentas que geram autómatos de porta, processos mCRL2, e representações gráficas de instâncias de conetores, desenvolvidas na linguagem Scala e que compilam para JavaScript. A *framework* final está disponível publicamente e pode ser experimentada sem qualquer instação ou servidor ativo.

# Contents

# List of Figures

# List of Tables

# List of Acronyms

| | |
|---|---|
| BNF | Backus–Naur form |
| LPS | Linear Process Specification |
| LTS | Labeled Transition System |
| PBES | Parameterized Boolean Equation System |
| PA | Port Automaton |
| IDE | Integrated Development Environment |

# Chapter 1

# Introduction

Service oriented computing aims at providing individual software components, each performing simple tasks, which can be assembled together to generate a complex system. These components are designed to be autonomous, reusable and independent from their application environment, and, as such, coordination between different components is not a trivial task. Usually it is necessary to developed code to generate the coordination. This code can become complex and difficult to maintain.

The Reo coordination language[1] was introduced by Arbab [1] to generate coordination between different components, by composing atomic entities – called channels – into a complex connector. These channels have a predefined behaviour, which defines how the information received is sent to other channels. To verify the behaviour of Reo connectors, Kokash et al. [7] developed an encoding of connectors into models in the mCRL2 specification language,[2] which is used to model and analyse concurrent systems.

Proença and Clarke [8] investigated how one can specify and combine connector families, and how to check if the interfaces of these families match. Their core calculus is a monoidal category, where connectors are morphisms composed sequentially with the morphism composition ';', and in parallel with the tensor operator '$\oplus$'. This calculus was formalised with a tile semantics that describes the behaviour of a connector, and how to combine tiles between two connectors.

[1] http://reo.project.cwi.nl/reo/
[2] https://mcrl2.org

Currently there are no tools to automatically analyse a connector described in this calculus, other than a type checker that verifies if connectors are properly connected. To solve this, we create an encoding of the core calculus into mCRL2, and we try to extend this to the full calculus of connector families proposed in [8]. Our approach follows the same steps as the encoding proposed by Kokash et al. [7]. We could propose an encoding of the core calculus into Reo connectors, as an alternative, effectively reusing the results from Kokash. This approach would create several problems. First, the encoding by Kokash is not currently being maintained. Second, Reo connectors cannot be extended into families, which could place limitations on a possible future encoding of the full calculus. Furthermore, we have a finer control of the semantics and extensions of our calculus using a direct encoding.

We build a web framework –ReoLive– to provide tools to analyse connectors in this calculus. Existing tools to create, edit and analyse Reo connectors are available as a set of plugins in the Eclipse IDE, which are not actively supported. These plugins require the installation of the IDE which is a powerful tool, and therefore, requires a large amount of computational power. We want our framework to be lightweight and independent from resource consuming platforms. Furthermore, we want this framework to be easily extensible, to include new tools developed for the calculus. Currently, our framework should present a type-checking analysis, with behaviour analysis and generate the mCRL2 encoding for the connector instances.

### Organization

In Chapter 2 we present an introductory background on Reo, mCRL2 and the encoding presented in [7]. The following chapter begins by presenting the calculus of connectors presented in [8]. Following this, we describe the semantics of the calculus in terms of port automata semantics, which is used in the following section when defining the encoding of the calculus into mCRL2. For the semantics and the encoding we present the respective proofs of correctness. In Chapter 4 we present the ReoLive framework, which presents a set of tools to analyse instances of the connector calculus. We present the user interface of the framework, its architecture and development details, which gives insight on how

to extend the framework. We conclude this chapter by explaining some problems that arose when generating an encoding for the full calculus of connector families. Finally, Chapter 5 presents some conclusions and future work.

# Chapter 2

# Background

We rely on previous work done in the context of coordination tools, and process algebras. More concretely, the calculus of connectors presented in Chapter 3 uses Reo channels as atomic entities, and we adapt semantic models for Reo connectors to this calculus. In this chapter we begin by presenting the Reo coordination language [1](Section 2.1), followed by some basic mCRL2 syntax and semantics [5](Section 2.2). Finally, we revisit the translation from Reo to mCRL2 by Kokash et al. [7](Section 2.3), providing a more detailed proof of their main result.

## 2.1 The Reo Coordination Language

Reo is a language used to coordinate the communication between independent components. It uses the notion of channels as atomic entities which specify the behaviour of data flowing through the channel. These channels are combined to form complex connectors. In the eclipse IDE, Reo contains a toolset for development and analysis of Reo connectors. We begin this section by explaining the basic Reo syntax and behaviour of channels. Then, we proceed to formalise its semantics in relation to port automata.

### 2.1.1 Basic Concepts

Reo connectors coordinate the flow of information between components. A component is a software implementation running on a machine, which sends to, and receives data from other components. A connector abstracts itself from the behaviour of the components, and only determines how a piece of data flows between the components it connects to.

A connector is a set of channels and a set of nodes, which can be organized in a graph. Channels are the atomic entities of connectors, which have a defined behaviour explaining the flow of information contained in it. Each channel contains channel ends, which can be source ends, or sink ends. Source ends allow information to be received by the channel, while sink ends allow information to flow out of the channel. Channels have exactly two ends, which can be of any type. This means that a channel can have just source ends, sink ends, or both. Channels may have different states, depending on the defined behaviour. We can join channel ends to form nodes. Nodes are groups of channel ends. Each channel end forms, by itself, a node. Furthermore, a channel end belongs to exactly one node. We connect different channels by joining the nodes of the desired channel ends we want to connect. The behaviour of nodes is as follows: a node takes the information available from a sink end belonging to it, and replicates it to all the source ends contained in the node. When a node is connected to a component, it can only contain source ends or sink ends. In this case, it either reads from the component into the source ends, or copies the information from a sink end into the component.

### 2.1.2 Channels

As there are many possible types of channels, with different behaviours, we will only define a small subset of channels which are important to the work of this document. In Fig. 2.1 we present the graphical representation of the set of channels defined.

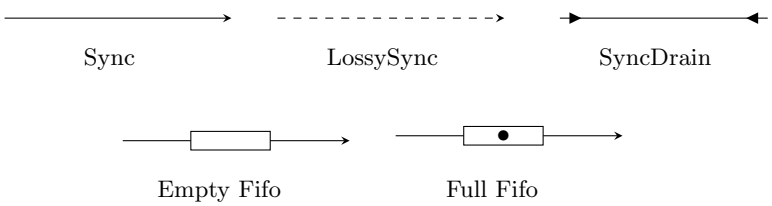



Figure 2.1: Graphical representation of the different channels

### Sync

A sync channel contains two channel ends, one of them is a source end, and the other is a sink end. A sync channel receives data through the source end if and only if the data can immediately be dispersed through the sink end.

### Fifo

A fifo channel contains a source end, and a sink end. It also has two states: empty, and full. When the fifo is empty, data can flow into the channel through the source end, turning it into a full fifo. The full fifo must hold the data until a later moment when it is ready to be dispersed through the sink end. This is not mandatory, which means, if a fifo can never disperse the data through the sink end, this data can remain in the fifo indefinitely.

### Lossy Sync

A lossy sync contains a source end, and a sink end. Data can flow at any moment into the lossy sync. If this data can be immediately dispersed through the sink end, then the data flows out of the lossy; otherwise the data is lost (hence the name).

### Sync Drain

Both channel ends of the sync drain are source ends. This means that data can never flow out of this type of channel. A sync drain can receive data through one of the source ends, if and only if the other source end receives data as well. The main purpose of the channel is to synchronise data flow between different channels. Fig. 2.3 gives an example of the use for this channel.

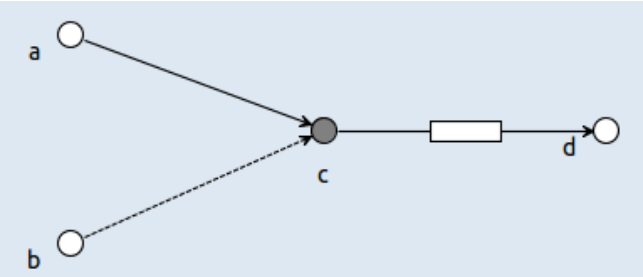



Figure 2.2: Sync and Lossy channels connecting to a fifo channel

### 2.1.3 Composing Channels

We can join the channels presented before to form more complex connectors. We compose the channels by joining the nodes of the channel ends we want to connect. Nodes are specific entities which have a defined behaviour which coordinates the information between the channels. A node behaves by replicating the information of a sink end belonging to the node into all the source ends belonging to it. If more than one sink end wants to send data, the node chooses one of these sink ends non-deterministically and immediately replicates the data received from it, putting the remaining sink ends on hold until a later moment.

Consider the connector in Fig. 2.2. The figure shows 3 channels and 4 nodes forming a connector. Node c contains the sink ends of the sync and lossy sync. Since the fifo is empty, either the lossy sync or the sync channels can send data to the fifo, making it full. When the fifo is full, the lossy can receive data, but since it cannot flow into the source end of the fifo, the data is lost. Since the sync cannot lose data, then it cannot receive data when the fifo is full. The full fifo could disperse data through the sink end, if the source ends contained in the node $d$ (or components connected to it) could receive it.

Consider now Fig. 2.3. In this case, if data can exit through both nodes $f$ and $g$, then the sync channels respectively connected to them can receive data. Since both receive data from the lossy channels connected to $b$, then data reaches the node $d$ through the sync ends connected to $c$ and $e$. Since node $d$ can receive data from only one of them at the same time, then, at each moment, only one of the nodes $c$ or $e$ can receive data. Since there is a sync drain connected to $d$ and $b$, when data reaches $b$ and is replicated to both lossy channels, one of them

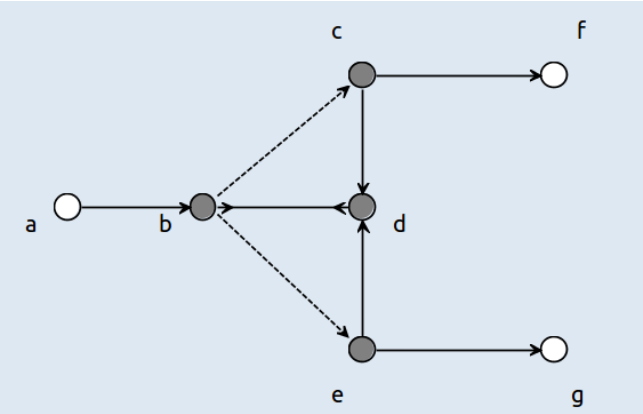



Figure 2.3: The exclusive router connector

cannot lose the data, while the other one must lose it because of the previous condition. So, in this figure, we have a connector which when data reaches the node $a$, it will flow until it reaches either $f$ or $g$, but not both at the same time.

### 2.1.4 Port Automata Semantics

We will now formalise the concepts of channels and channels composition with port automata as a semantic model.

**Definition 2.1.** A port automaton (PA) [6] is a tuple $\mathcal{A} = (Q, \mathcal{N}, \rightarrow, q_0)$ where:

- $Q$ is a set of states;
- $\mathcal{N}$ is a set of port names;
- $\rightarrow \subseteq Q \times 2^{\mathcal{N}} \times Q$ is the transition relation between states;
- $q_0$ is the initial state.

We use the notation $q \xrightarrow{N} r$ to denote $(q, N, r) \in \rightarrow$. Furthermore, port automata have graphical representations, using the states as vertices and the transitions in $\rightarrow$ as edges.

Port automata describe transitions between states through ports. In the case of Reo, the port names are the channel ends, and the transitions describe which channel ends contain information flowing in a moment. In Fig. 2.4 we define the port automata which give semantics to the channels defined in this

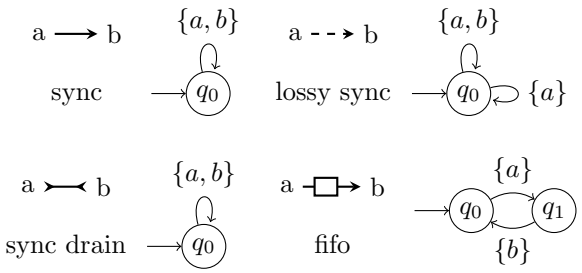



Figure 2.4: Port Automata of Reo channels.

section, based on the constraint automata of Baier et al. [2], but abstracting away the data. Furthermore, port automata do not catch the direction of the flow of information, or classify the channel ends (notice that the port automata for the sync and sync drain channels are equivalent).

**Example 2.1.** Consider a fifo with source end $a$, and sink end $b$. When empty, this fifo allows data to be read through $a$, and when full, data can flow out through $b$. So the automaton of the fifo is $\mathcal{A} = (\{q_0, q_1\}, \{a, b\}, \{q_0 \xrightarrow{\{a\}} q_1, q_1 \xrightarrow{\{b\}} q_0\}, q_0)$, where the state $q_0$ represents the empty fifo and $q_1$ represents the full fifo. This automaton is one of the automata represented in Fig. 2.4.

To compose the port automata of individual channels, we need to define the behaviour of nodes as port automata. Given an arbitrary node, the port automaton associated to this node contains exactly one node, say $q$, and, for each sink end $b$ in the node, a transition $(q, N, q)$, where $N$ contains the sink end $b$ and all the source ends of the node. Consider the following example for a better understanding:

**Example 2.2.** Consider a node $n$ which connects the sink ends $a$ and $b$ with the source ends $c$, $d$, and $e$. The port automata of $n$ is $\mathcal{A} = (\{q_0\}, \{a, b, c, d, e\}, \rightarrow, q_0)$, where the transition function has two transitions:

- $q_0 \xrightarrow{\{a,c,d,e\}} q_0$;
- $q_0 \xrightarrow{\{b,c,d,e\}} q_0$.

To generate the automaton of a Reo connector, we begin by generating the port automata of the individual channels, and nodes, ensuring that port names

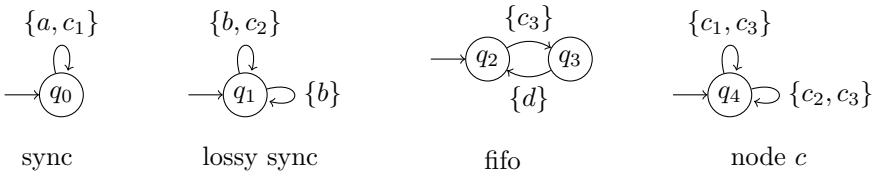



Figure 2.5: Port automata of the channels and node in Fig. 2.2

are correct, i.e., every channel end has a unique port name, and each channel end is used in its channel automaton and in the automaton corresponding to its containing node. Then, we apply the product of port automata as defined in Definition 2.2.

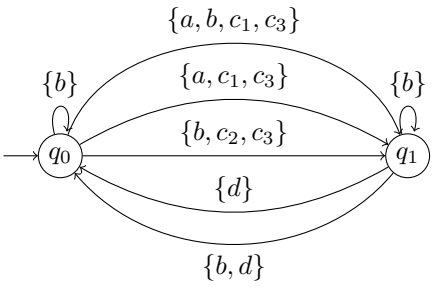
**Definition 2.2** (Product of PA [6])**.** The product of two port automata $\mathcal{A}_i = (Q_i, N_i, \rightarrow_i, q_{0,i})$, for $i \in \{1, 2\}$, is defined as $\mathcal{A}_1 \bowtie \mathcal{A}_2 = (Q_1 \times Q_2, N_1 \cup N_2, \rightarrow, (q_{0,1}, q_{0,2}))$ where $\rightarrow$ is defined by the rules below.

$$\frac{q_1 \xrightarrow{X_1}_1 p_1 \quad q_2 \xrightarrow{X_2}_2 p_2 \quad X_1 \cap N_2 = X_2 \cap N_1}{(q_1, q_2) \xrightarrow{X_1 \cup X_2} (p_1, p_2)} \qquad \frac{q_1 \xrightarrow{X_1}_1 p_1 \quad X_1 \cap N_2 = \emptyset}{(q_1, q_2) \xrightarrow{X_1} (p_1, q_2)} \qquad \frac{q_2 \xrightarrow{X_2}_2 p_2 \quad X_2 \cap N_1 = \emptyset}{(q_1, q_2) \xrightarrow{X_2} (q_1, p_2)}$$

In the following example we explain, with a simple example, the construction of the port automata of a Reo connector.

**Example 2.3.** Consider the Reo connector in Fig. 2.2. Consider that the node $c$ contains the sink ends $c_1$ of the sync channel, $c_2$ of the lossy sync, and the source end $c_3$ of the fifo channel. Then, the port automata of the individual channels are defined as in Fig. 2.5, as well as the automaton of the node $c$.

Using Definition 2.2 we obtain the resulting port automata, by applying the product of the automata of the channels and node:



In the initial state, information can flow through $a$, or $b$. When information flows from $b$ it may be lost, and the state remains the same, or it may flow to sink end $c_2$, changing the state of the fifo. When a transition through $a$ occurs, it must always be accompanied by $c_1$, and $c_3$, changing the state of the fifo. On the state relating to the full fifo, a transition through $b$ may occur, meaning that the lossy loses information, or a transition through $d$ may occur, accompanied or not by $b$. This is equivalent to the behaviour described for the connector when explaining Fig. 2.2.

Another important operation in port automata is hiding ports following [6]:

**Definition 2.3** (Hiding in PA)**.** Let $(A) = (Q, N, \rightarrow, q_0)$ be a port automaton and $X \subseteq N$. Hiding ports $X$ from $A$ yields the port automaton $A \setminus X = (Q, N \setminus X, \dashrightarrow, q_0)$, where $q_i \overset{Y \setminus X}{\dashrightarrow} q_j$ iff $q_i \overset{Y}{\rightarrow} q_j$.

It is necessary throughout this document to define a notion of behavioural equivalence between port automata. In this case, we use the notion of bisimilarity [6, 2] defined below:

**Definition 2.4** (Bisimilarity)**.** Given port automata $\mathcal{A}_1 = (Q_1, \mathcal{N}_1, \rightarrow_1, Q_{0,1})$, and $\mathcal{A}_2 = (Q_2, \mathcal{N}_2, \rightarrow_2, Q_{0,2})$, a bisimulation between $\mathcal{A}_1$, and $\mathcal{A}_2$ is a relation $\sim \subseteq Q_1 \times Q_2$ such that forall $(q_1, q_2) \in \sim$:

1. If $q_1 \xrightarrow{a} p_1$ then there exists $p_2 \in Q_2$ s.t. $q_2 \xrightarrow{a} p_2$, and $(p_1, p_2) \in \sim$.
2. If $q_2 \xrightarrow{a} p_2$ then there exists $p_1 \in Q_1$ s.t. $q_1 \xrightarrow{a} p_1$, and $(p_1, p_2) \in \sim$.

If $(Q_{0,1}, Q_{0,2}) \in \sim$, for some bisimulation $\sim$ between $\mathcal{A}_1$ and $\mathcal{A}_2$, we say that $\mathcal{A}_1$ is bisimular to $\mathcal{A}_2$, denoted $\mathcal{A}_1 \cong \mathcal{A}_2$.

Bisimilarity is a relation that represents the equality of the behaviour of different port automata. This means that different port automata can have similar behaviour. This idea is illustrated in Example 2.4.

**Example 2.4.** Consider the automata in Fig. 2.6. The relation

$$R = \{(q_1, p_1), (q_2, p_2), (q_2, p_4), (q_3, p_3)\}$$

is a bisimulation, and thus $\mathcal{A}_1 \cong \mathcal{A}_2$.

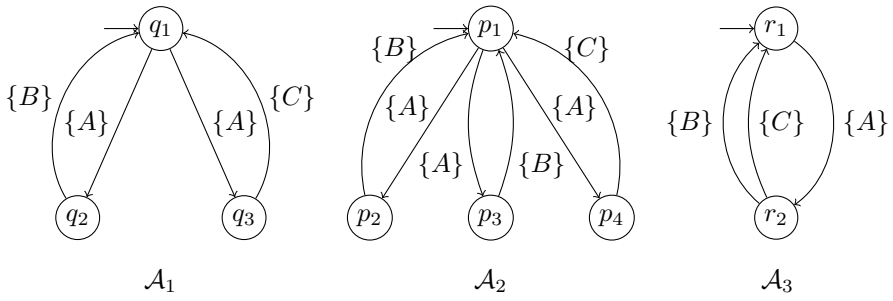



Figure 2.6: Example of bisimilar and non bisimilar PA

We now prove $\mathcal{A}_1$ and $\mathcal{A}_3$ are not bisimilar. The proof is by reduction to absurdity. Consider a bisimulation $R \subseteq Q_1 \times Q_3$, where $Q_1$ and $Q_3$ are the set of states of $\mathcal{A}_1$ and $\mathcal{A}_3$ respectively. Let $(q_1, r_1) \in Q$. Since neither $q_2$ nor $q_3$ can make transitions labelled with $\{B\}$ or $\{C\}$ but $r_2$ can, it follows neither $(q2, r2)$ nor $(q3, r2)$ is in R. On the other hand, since $r_1 \xrightarrow{\{A\}} r_2$, $(q_1, r_1) \in R$, and $R$ is a bisimulation, there is a transition $q_1 \xrightarrow{\{A\}} q$, with $(q, r_2) \in R$. Since neither $(q_2, r_2)$ nor $(q_3, r_2)$ is in $R$, $q$ can only be $q_1$, so $(q_1, r_2) \in R$. But this is absurd, because $r_2$ can do a transition with label $\{B\}$ while $q_1$ cannot. Therefore $\mathcal{A}_1 \ncong \mathcal{A}_3$.

## 2.2 The mCRL2 Toolset

The mCRL2 language is a specification language used to model and verify communicating processes. It has an extensive toolset which can be used to analyse the behaviour and to prove properties over a given model (ex. absence of deadlock). The language used to model the processes is an extension of the process algebra language, with notions of data, and time, functions and predicates.

In this section we present a basic subset of the mCRL2 syntax, which will be used throughout the document. We will briefly explain the operational semantics and bisimulation relation of the process algebra. Finally we give a brief introduction of the mCRL2 toolset. Since the toolset is very extensive, we refer to [5] and to the mCRL2 web site[1], which contain the full documentation of the toolset.

[1] https://mcrl2.org

### 2.2.1 Basic Concepts

The basic elements of any model are actions. Actions represent an atomic event which can be parameterized with data. By combining and synchronising the actions, we get processes. Actions are labelled with an identifier (usually a low case letter).

A multiaction can be constructed according to the following BNF:

$$\alpha ::= \tau \mid a \mid \alpha_1|\alpha_2$$

where $\tau$ is a multiaction containing no actions, and $a$ is an action.

Processes define when multiactions can occur. For instance the process `P = a.(b+c)` denotes a process that specifies that $a$ can occur, followed by either $b$ or $c$. We denote processes with upper case letters. The following list contains some relevant processes and process operations:

- multiaction ($\alpha$): The actions in $\alpha$ must occur simultaneously;
- sequence ($P \ . \ Q$): The first action of process $Q$ can only occur after the last action in $P$ happens;
- parallelism ($a.P \parallel b.Q$): Either $a$, or $b$ can occur, or both can occur simultaneously;
- choice ($P + Q$): Either $P$ or $Q$ can succeed;
- hiding ($\tau_H(P)$): Hides the actions in the set $H$ (or renames to $\tau$), when $P$ executes;
- blocking ($\partial_B(P)$): Using a set of actions $B$, denotes that the actions in $B$ cannot occur in $\partial_B(P)$;
- communication ($\Gamma_C(P)$): Where elements of $C$ have the format $a|b \mapsto c$, when actions $a$ and $b$ occur simultaneously in $P$, they are replaced by $c$;
- renaming ($\rho_R(P)$): Where $R$ is a set of renamings of the form $a \mapsto b$. The occurrences of actions $a$ are replaced in $P$ by $b$;
- deadlock ($\delta$): which denotes the process without any possible actions.
- recursion: a process can refer to itself in its definition.

An mCRL2 model has a defined structure, where we need to define the actions used, the processes, and then specify the initial process of the system. Although this is the structure we use throughout the document, more complex structures, with predicates and functions can be defined in mCRL2. Since we do not need these types of structure, we will simply generate a program with the structure of the following example.

**Example 2.5.** Consider the process of a simple vending machine which, after receiving a coin, provides a beverage. The actions *rcoin* and *sbeverage* represent receiving a coin, and sending a beverage, respectively. The model of this process could be defined as follows:

```
act
  rcoin, sbeverage;
proc
  P = rcoin.sbeverage.P;
init
  P;
```

### 2.2.2 Process Semantics

We denote the set of all processes as $\mathbb{P}$, and the set of all multiactions as $\mathbb{M}$. Given a multiaction $\alpha$, we define $\alpha_{\{\}}$ as the set of actions contained in the multiaction $\alpha$; furthermore we define $\Theta_H(\alpha)$ as the function that, given the set $H$ of actions, hides the respective actions in $\alpha$, and the function $\gamma_C(\alpha)$ as the function that, for each triple $a|b \to c$, replaces in $\alpha$ the actions $a$ and $b$ for the action $c$, if both actions $a$, and $b$ exist in $\alpha$. Finally, the function $R \bullet \alpha$ replaces the actions in the multiaction $\alpha$ using the mapping in $R$. These functions are formally defined in [5].

Using these auxiliary functions we can define the operational semantics of mCRL2 through the transition relation $\to \in \mathbb{P} \times \mathbb{M} \times \mathbb{P}$, and the termination relation $\to_{\checkmark} \in \mathbb{P} \times \mathbb{M}$. We define these relations based on [5] using the deduction rules in tables 2.1 to 2.7, defining a structured operational semantics.

Having defined the operational semantics of process algebras, we can define the concept of bisimilarity of programs. Our definition abstracts the relation $\to \checkmark$, as we can consider the $\checkmark$ operator as the termination process.

$$\frac{}{\alpha \xrightarrow{\alpha} \checkmark}$$

Table 2.1: Deduction rules for multiactions

$$\frac{P \xrightarrow{\alpha} \checkmark}{P \,.\, Q \xrightarrow{\alpha} Q} \qquad \frac{P \xrightarrow{\alpha} P'}{P \,.\, Q \xrightarrow{\alpha} P'.Q}$$

Table 2.2: Deduction rules for sequential composition

**Definition 2.5** (Bisimilarity of Processes [5])**.** Let $P$ and $Q$ be two processes. We say that $P$ and $Q$ are strongly bisimilar, notation $P \cong Q$, if and only if there exists a relation $R \subseteq \mathbb{P} \times \mathbb{P}$ called a strong bisimulation such that for any multiaction $\alpha$:

1. If $P \xrightarrow{\alpha} P'$ then there exists $Q' \in \mathbb{P}$ such that $Q \xrightarrow{\alpha} Q'$, and $P' \cong Q'$;
2. If $Q \xrightarrow{\alpha} Q'$ then there exists $P' \in \mathbb{P}$ such that $P \xrightarrow{\alpha} P'$, and $P' \cong Q'$;

Similarly to the case of the port automata bisimilarity, the relation $R$ denotes an equivalence of process execution, instead of equivalence of process definitions. Groote et al. [5] define theorems that prove the soundness and completeness of the axiomatisation of mCRL2 in terms of strong bisimulation.

### 2.2.3 The Toolset

After having a model created, we can use the mCRL2 toolset to verify it. The first step is to convert the mCRL2 model into a linear process specification (LPS). This conversion removes several components (e.g. parallelism) and simplifies the model. It is from the generated LPS that we can use the remaining toolset.

After having the LPS generated, we can transform it again and turn it into a labelled transition system (LTS). A labelled transition system is a directed multigraph, where the states represent processes, and there is an edge from state $q$ to state $p$, if the process of state $q$ has a transition to the process of

$$\frac{P \xrightarrow{\alpha} \checkmark}{P \mid\mid Q \xrightarrow{\alpha} Q} \qquad \frac{P \xrightarrow{\alpha} \checkmark, Q \xrightarrow{\beta} Q'}{P \mid\mid Q \xrightarrow{\alpha|\beta} Q'} \qquad \frac{Q \xrightarrow{\beta} \checkmark}{P \mid\mid Q \xrightarrow{\beta} P} \qquad \frac{Q \xrightarrow{\beta} \checkmark, P \xrightarrow{\alpha} P'}{P \mid\mid Q \xrightarrow{\alpha|\beta} P'}$$

$$\frac{Q \xrightarrow{\beta} \checkmark, P \xrightarrow{\alpha} \checkmark}{P \mid\mid Q \xrightarrow{\alpha|\beta} \checkmark} \qquad \frac{P \xrightarrow{\alpha} P'}{P \mid\mid Q \xrightarrow{\alpha} P' \mid\mid Q} \qquad \frac{P \xrightarrow{\alpha} P', Q \xrightarrow{\beta} Q'}{P \mid\mid Q \xrightarrow{\alpha|\beta} P' \mid\mid Q'} \qquad \frac{Q \xrightarrow{\beta} Q'}{P \mid\mid Q \xrightarrow{\beta} P \mid\mid Q'}$$

Table 2.3: Deduction rules for parallel composition

$$\frac{P \xrightarrow{\alpha} \checkmark}{\tau_H(P) \xrightarrow{\Theta_H(\alpha)} \checkmark} \qquad \frac{P \xrightarrow{\alpha} P'}{\tau_H(P) \xrightarrow{\Theta_H(\alpha)} P'}$$

Table 2.4: Deduction rules for the hiding operator

$$\frac{P \xrightarrow{\alpha} \checkmark}{\partial_B(P) \xrightarrow{\alpha} \checkmark} \alpha_{\{\}} \cap B = \emptyset \qquad \frac{P \xrightarrow{\alpha} P'}{\partial_B(P) \xrightarrow{\alpha} P'} \alpha_{\{\}} \cap B = \emptyset$$

Table 2.5: Deduction rules for the blocking operator

$$\frac{P \xrightarrow{\alpha} \checkmark}{\rho_R(P) \xrightarrow{R\bullet(\alpha)} \checkmark} \qquad \frac{P \xrightarrow{\alpha} P'}{\rho_R(P) \xrightarrow{R\bullet(\alpha)} P'}$$

Table 2.6: Deduction rules for the renaming operator

$$\frac{P \xrightarrow{\alpha} \checkmark}{\Gamma_C(P) \xrightarrow{\gamma_C(\alpha)} \checkmark} \qquad \frac{P \xrightarrow{\alpha} P'}{\Gamma_C(P) \xrightarrow{\gamma_C(\alpha)} P'}$$

Table 2.7: Deduction rules for the communication operator

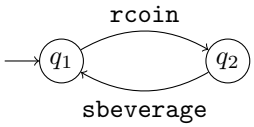



Figure 2.7: Labeled transition system of the process `P`

state $p$, given by the process semantics of Section 2.2.2. This representation simplifies the analysis of the process behaviour. For example, consider Fig. 2.7 which contains the LTS representation of the process `P` defined in Section 2.2.1, where the state $q_1$ represents the process `P`, while state $q_2$ represents the process `sbeverage.P`.

Another important tool that mCRL2 provides is the tool to convert LPS or LTS into parametrized boolean equation systems (PBES). This tool requires a $\mu$-calculus property as input, and when the conversion is done, we can check the validity of the resulting PBES. The $\mu$-calculus property is the property we want to check for the given model (e.g. absence of deadlock).

## 2.3 Modeling Reo Connectors in mCRL2

Just like in any programming language, analysing and verifying the behaviour of a given connector in Reo is important to guarantee the correctness of the communication between components. Manually verifying connectors using a semantic model such as port automata can take a long time and lead to errors. This problem can be solved by an automatic tool such as mCRL2 presented in the previous section. Kokash et al. [7] developed a translation of Reo connectors into the mCRL2 language based on several semantic models, including constraint automata. A section of [7] is dedicated to prove the correctness of the encoding mainly by showing that the algebraic structures of constraint automata and process algebra are preserved by their encoding. The section begins by recalling the encoding, and its correctness. Furthermore, we extend their proof with a soundness result for the bisimilarity, showing that the encoding of bisimilar automata provides bisimilar processes.

| | |
|---|---|
| sync | $Sync = a\|b \ . \ Sync$ |
| fifo | $Fifo = a \ . \ b \ . \ Fifo$ |
| sync drain | $Drain = a\|b \ . \ Drain$ |
| lossy sync | $Lossy = (a + a\|b) \ . \ Lossy$ |
| replicator | $Dupl = a\|b\|c \ . \ Dupl$ |
| merger | $Merger = (a\|c + b\|c) \ . \ Merger$ |

Table 2.8: mCRL2 processes of channels.

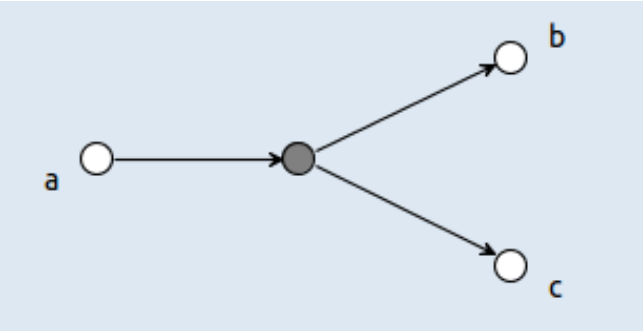



Figure 2.8: Example of a replicator node.

### 2.3.1 Mapping of Reo into mCRL2

We consider the mapping of Reo into mCRL2 based on the port automata semantics specified in Section 2.1.4. Kokash et al.[7] specifies the mapping in terms of the constraint automata semantics, of which we ignore the data constraints for simplicity.

In Table 2.8, we can find the individual processes in mCRL2 of each channel with respect to the PA semantics. We combine the processes of Reo channels, and nodes (which we define later), using parallel composition as well as synchronization of the actions which represent the flow of information in channel ends. This is better explained in the following example.

**Example 2.6.** Consider the replicator node in Fig. 2.8. Using the mapping of the individual channels, we obtain the following processes:

```
Sync1 = a|x1 .  Sync1,      Sync2 = y1|b .  Sync2,
```

`Sync3 = z1|c . Sync3,  Node = x2|y2|z2. Node`

Applying the parallel and communication operators we obtain:

`Replicator =` $\partial_{\{x1,x2,y1,y2,z1,z2\}}$`(`$\Gamma_{\{x1|x2\mapsto x,y1|y2\mapsto y,z1|z2\mapsto z\}}$`(Sync1 || Sync2 || Sync3 || Node))`

The actions $a$, $b$, $c$, $x$, $y$, and $z$ represent the flow of information in the represented channel ends.

Although the previous example provides a correct composition of the channels in mCRL2 semantics, synchronising every action simultaneously becomes inefficient when combining a large number of processes, because it results in a state space explosion. To overcome this problem, we explore the graph structure of Reo, synchronizing the actions as soon as possible.

Consider then a Reo connector $R = (E, V)$, where $E$ is the set of channels, and $V$ is the set of nodes of the connector. Consider the function $Chan(e)$ that receives a channel $e \in E$ and returns its mCRL2 process according to Table 2.8, with actions in the format $X''_{v,e}$ where $v, e$ indicate that the channel end corresponding to the action belongs to channel $e$ and node $v$ for $e \in E$, $v \in V$. For a node $v$ and a channel $e$, we define the predicates $src(v, e)$ to be true if $v$ contains a source end belonging to $e$, and $snk(v, e)$ to be true if $v$ contains a sink end belonging to $e$. The function $Node(v)$ defines the mCRL2 process of a node $v \in V$, and is defined by:

$$Node(v) = \sum_{e:src(v,e)} (X'_{v,e} \mid \otimes_{f:snk(v,f)} X'_{v,f}) \ . \ Node(v),$$

where $\otimes_{i\in I}$ represents the multiaction of the actions $i \in I$. In this encoding, the actions $X'_{v,e}$ and $X''_{v,e}$ are synchronised to form action $X_{v,e}$.

Finally, we define the predicate $lnk(e, v, w)$ to hold if, for the channel $e \in E$, $e$ is connected to the node $v \in V$ through a source or sink end, but not connected to any node in the string of nodes $w \in V^*$.

With these definitions we can define for the Reo connector R=(E,V), the process $P(w)$, which receives a string of nodes $w \in V^*$:

$$P(\epsilon) = \delta$$
$$P(w \ . \ v) = \partial_{H_v}(\Gamma_{C_v}(P(w) \parallel Node(v) \parallel \prod_{e:lnk(e,v,w)} Chan(e)))$$

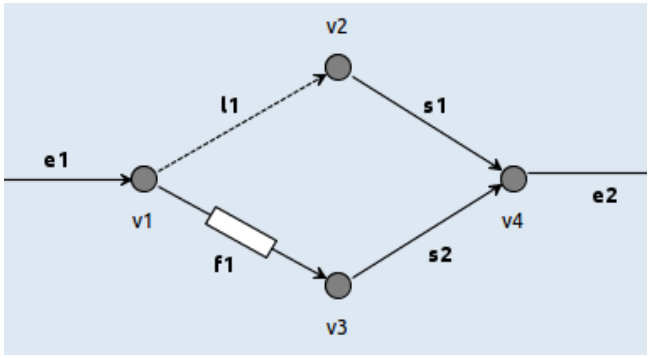



Figure 2.9: Example connector

Where $\prod_{i \in I} P_i$ denotes the parallel composition of the processes $P_i$, and

$$H_v = \left\{ X'_{v,e}, X''_{v,e} \mid e \in E,\ src(v,e) \vee snk(v,e) \right\}, \text{ and}$$
$$C_v = \left\{ X'_{v,e} | X''_{v,e} \mapsto X_{v,e} \mid e \in E,\ src(v,e) \vee snk(v,e) \right\}$$

The order of the nodes in $w$ has been showed by Kokash et al.[7] to be irrelevant, i.e., the process $P(w')$, where $w'$ is the result of applying a permutation to the string $w$, is bisimilar to $P(w)$.

In Example 2.7 we can find an example application of $P$.

**Example 2.7.** Consider the Reo connector in Fig. 2.9, with nodes $v_1, v_2, v_3, v_4$ and channels $f_1, l_1, s_1, s_2, e_1$, and $e_2$. We abstract from the connector prior to $e_1$ and posterior to $e_2$, including these channels. The processes for each channel are:

$$Chan(f_1) = X''_{v_1,f_1} \ . \ X''_{v_3,f_1} \ . \ Chan(f_1)$$
$$Chan(l_1) = X''_{v_1,l_1} \ + \ X''_{v_1,l_1} | X''_{v_2,l_1} \ . \ Chan(l_1)$$
$$Chan(s_1) = X''_{v_2,s_1} | X''_{v_4,s_1} \ . \ Chan(s_1)$$
$$Chan(s_2) = X''_{v_3,s_2} | X''_{v_4,s_2} \ . \ Chan(s_2)$$

And using the $Node$ function, we get the processes for each node:

$$Node(v_1) = X'_{v_1,e_1} | X'_{v_1,l_1} | X'_{v_1,f_1} \ . \ Node(v_1)$$
$$Node(v_2) = X'_{v_2,l_1} | X'_{v_2,s_1} \ . \ Node(v_2)$$
$$Node(v_3) = X'_{v_3,f_1} | X'_{v_3,s_2} \ . \ Node(v_3)$$
$$Node(v_4) = (X'_{v_4,s_1} | X'_{v_4,e_1} \ + \ X'_{v_4,s_2} | X'_{v_4,e_1}) \ . \ Node(v_4)$$

We can now incrementally build the process for our connector, using the string of nodes $v_1 \cdot v_2 \cdot v_3 \cdot v_4$:

$$P(v_1) = \partial_{H_{v_1}}(\Gamma_{C_{v_1}}(Node(v_1) \parallel Chan(e_1) \parallel Chan(f_1) \parallel Chan(l_1)))$$
$$H_{v_1} = \{X'_{v_1,e_1}, X''_{v_1,e_1}, X'_{v_1,f_1}, X''_{v_1,f_1}, X'_{v_1,l_1}, X''_{v_1,l_1}\}$$
$$C_{v_1} = \{X'_{v_1,e_1}|X''_{v_1,e_1} \mapsto X_{v_1,e_1},\ X'_{v_1,f_1}|X''_{v_1,f_1} \mapsto X_{v_1,f_1},\ X'_{v_1,l_1}|X''_{v_1,l_1} \mapsto X_{v_1,l_1}\}$$

$$P(v_1 \cdot v_2) = \partial_{H_{v_2}}(\Gamma_{C_{v_2}}(P(v_1) \parallel Node(v_2) \parallel Chan(s_1)))$$
$$H_{v_2} = \{X'_{v_2,f_1}, X''_{v_2,f_1}, X'_{v_2,s_1}, X''_{v_2,s_1}\}$$
$$C_{v_2} = \{X'_{v_2,f_1}|X''_{v_2,f_1} \mapsto X_{v_2,f_1},\ X'_{v_2,s_1}|X''_{v_2,s_1} \mapsto X_{v_2,s_1}\}$$

$$P(v_1 \cdot v_2 \cdot v_3) = \partial_{H_{v_3}}(\Gamma_{C_{v_3}}(P(v_1 \cdot v_2) \parallel Node(v_3) \parallel Chan(s_2)))$$
$$H_{v_3} = \{X'_{v_3,l_1}, X''_{v_3,l_1}, X'_{v_3,s_2}, X''_{v_3,s_2}\}$$
$$C_{v_3} = \{X'_{v_3,l_1}|X''_{v_3,l_1} \mapsto X_{v_3,l_1},\ X'_{v_3,s_2}|X''_{v_3,s_2} \mapsto X_{v_3,s_2}\}$$

$$P(v_1 \cdot v_2 \cdot v_3 \cdot v_4) = \partial_{H_{v_4}}(\Gamma_{C_{v_4}}(P(v_1 \cdot v_2 \cdot v_3) \parallel Node(v_4) \parallel Chan(e_2)))$$
$$H_{v_4} = \{X'_{v_4,s_1}, X''_{v_4,s_1}, X'_{v_4,s_2}, X''_{v_4,s_2}, X'_{v_4,e_2}, X''_{v_4,e_2}\}$$
$$C_{v_4} = \{X'_{v_4,s_1}|X''_{v_4,s_1} \mapsto X_{v_4,s_1},\ X'_{v_4,s_2}|X''_{v_4,s_2} \mapsto X_{v_4,s_2},\ X'_{v_4,e_2}|X''_{v_4,e_2} \mapsto X_{v_4,e_2}\}$$

### 2.3.2 Correctness of the Translation

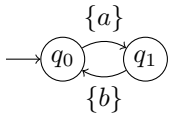
The correctness of the encoding of Reo connectors given by function $P$ is proved by defining an encoding of PA into mCRL2 processes, named *proc*, and showing that the later encoding is sound in terms of the algebraic structure of processes, while the reader is encouraged to verify that the *proc* encoding, followed by the encoding of Reo connectors into PA is equivalent to the $P$ function defined previously. We will present the encoding of PA into mCRL2 and the Theorem 4.1 of [7], with relevance to later work presented in this document. Furthermore we present in Theorem 2.1 that the encoding of PA into mCRL2 is sound in regard to strong bisimulation. In these proofs we consider port automata with disjoint port names, without loss of generality.

Let $\mathcal{A} = (Q, \mathcal{N}, \rightarrow, q_0)$ be a port automata. For any state $q \in Q$,

$$proc(\mathcal{A}, q) = (\sum_{q \xrightarrow{N} r} A \ . \ proc(\mathcal{A}, r)) + \delta,$$

where $A = \otimes_{a \in N} a$.

**Example 2.8.** The following port automata are matched by the respective results of the *proc* function:




$\mathcal{A}_1$

$$proc(\mathcal{A}_1, q_0) = a \ . \ proc(\mathcal{A}_1, q_1)$$
$$proc(\mathcal{A}_1, q_1) = b \ . \ proc(\mathcal{A}_1, q_0)$$


$\{a, b\}$

$\mathcal{A}_2$

$$proc(\mathcal{A}_2, q_0) = (a \ + \ a|b) \ . \ proc(\mathcal{A}_2, q_0)$$

Consider two PA $\mathcal{A}_1 = (Q_1, \mathcal{N}_1, \rightarrow_1, q_0^1)$, and $\mathcal{A}_2 = (Q_2, \mathcal{N}_2, \rightarrow_2, q_0^2)$, and a new set of ports $\mathcal{N}$ such that all three set of ports $\mathcal{N}, \mathcal{N}_1,$ and $\mathcal{N}_2$ are disjoint. A port synchronisation function $\gamma : \mathcal{N} \mapsto \mathcal{N}_1 \times \mathcal{N}_2$, is a pair of injective functions $\gamma_1 : \mathcal{N} \mapsto \mathcal{N}_1$, and $\gamma_2 : \mathcal{N} \mapsto \mathcal{N}_2$, such that $\gamma(n) = (\gamma_1(n), \gamma_2(n))$.[2]

Given $\gamma$, and $N_1 \subseteq \mathcal{N}_1$, $N_2 \subseteq \mathcal{N}_2$, we define the $N_1 \mid_\gamma N_2$ as the union $N_1 \cup N_2$, but replacing $n_1 \in N_1$ by $n \in N$, such that $\gamma_1(n) = n_1$, and $n_2 \in N_2$ by $n \in N$, such that $\gamma_2(n) = n_2$. Notice that since $\gamma_1$, and $\gamma_2$ are not necessarily bijective, not every element of $N_1$ and $N_2$ is necessarily replaced. Then, we define $\mathcal{A}_1 \bowtie_\gamma \mathcal{A}_2 = (Q_1 \times Q_2, \mathcal{N}_1 \mid_\gamma \mathcal{N}_2, \rightarrow, (q_0^1, q_0^2))$, where the transition relation $\rightarrow$ is given by:

$$\frac{q_1 \xrightarrow{N_1}_1 p_1 \quad q_2 \xrightarrow{N_2}_2 p_2 \quad \gamma_1^{-1}[N_1] = \gamma_2^{-1}[N_2]}{(q_1, q_2) \xrightarrow{N_1 \mid_\gamma N_2} (p_1, p_2)} \qquad \frac{q_1 \xrightarrow{N_1}_1 p_1 \quad \gamma_1^{-1}[N_1] = \emptyset}{(q_1, q_2) \xrightarrow{N_1} (p_1, p_2)} \qquad \frac{q_2 \xrightarrow{N_2}_2 p_2 \quad \gamma_2^{-1}[N_2] = \emptyset}{(q_1, q_2) \xrightarrow{N_2} (q_1, p_2)}$$

[2] In the case of the encoding $P$, for a node $v$ and a channel $e$, $\gamma(X_{v,e}) = (X'_{v,e}, X''_{v,e})$.

Furthermore, Kokash et al.[7] also defines the parallel composition with port synchronization based on $\gamma$, $\|_\gamma$. So, for processes $P_1$, $P_2$, we define:

$$P_1 \|_\gamma P_2 = \partial_B(\Gamma_C(P_1 \| P_2))$$

where $B = \gamma_1(\mathcal{N}) \cup \gamma_2(\mathcal{N})$, and $C = \{\gamma_1(n)|\gamma_2(n) \mapsto n \mid n \in \mathcal{N}\}$.

With these functions, Kokash et al.[7] showed in the Theorem 4.1 of [7] that

$$proc(\mathcal{A}_1, q_0^1) \|_\gamma proc(\mathcal{A}_2, q_0^2) \cong proc(\mathcal{A}_1 \bowtie_\gamma \mathcal{A}_2, (q_0^1, q_0^2)).$$

The result proves the soundness of the encoding in terms of parallel composition of processes with action synchronisation.

We want to take the correctness a little further, and show that not only is the *proc* function sound, but it preserves the bisimulation of port automata, in the process algebra.

**Theorem 2.1** (Preservation of Equivalence)**.** Let $\mathcal{A}_1 = (Q_1, \mathcal{N}_1, \rightarrow_1, q_{0,1})$, $\mathcal{A}_2 = (Q_2, \mathcal{N}_2, \rightarrow_2, q_{0,2})$ be two port automata, such that $\mathcal{A}_1 \cong \mathcal{A}_2$. Then $proc(\mathcal{A}_1, q_{0,1}) \cong proc(\mathcal{A}_2, q_{0,2})$.

*Proof.* Let $R \subseteq Q_1 \times Q_2$ be a bisimulation between $\mathcal{A}_1$ and $\mathcal{A}_2$. We define $R' = \{(proc(\mathcal{A}_1, q_{i,1}), proc(\mathcal{A}_2, q_{j,2})) \mid (q_{i,1}, q_{j,2}) \in R\}$. We want to prove that $R'$ is a bisimulation between process algebras, i.e., for all $q_i \in Q_1$ and for all $q_j \in Q_2$:

1. if $(proc(\mathcal{A}_1, q_i), proc(\mathcal{A}_2, q_j)) \in R'$ and $proc(\mathcal{A}_1, q_i) \xrightarrow{a} P$ then there exists a process $Q$ s.t. $(proc(\mathcal{A}_2, q_j) \xrightarrow{a} Q$ and $(P, Q) \in R'$

2. if $(proc(\mathcal{A}_1, q_i), proc(\mathcal{A}_2, q_j)) \in R'$ and $proc(\mathcal{A}_2, q_j) \xrightarrow{a} Q$ then there exists a process $P$ s.t. $(proc(\mathcal{A}_1, q_i) \xrightarrow{a} P$ and $(P, Q) \in R'$

We will focus only on 1, as the proof for 2 would follow the same steps.

Since $(proc(\mathcal{A}_1, q_i), proc(\mathcal{A}_2, q_j)) \in R'$, then $(q_i, q_j) \in R$. Furthemore, because $proc(\mathcal{A}_1, q_i) \xrightarrow{a} P$ and by definition of *proc*, there exists $s_i \in Q_1$ such that $q_i \xrightarrow{N} s_i$, and $a = \otimes_{n \in N}\ n$. Moreover, $P = proc(\mathcal{A}_1, s_i)$. Because $\mathcal{A}_1 \cong \mathcal{A}_2$, there exists $s_j \in Q_2$ such that $q_j \xrightarrow{N} s_j$, and therefore, $proc(\mathcal{A}_2, q_j) \xrightarrow{a} proc(\mathcal{A}_2, s_j)$. Put $Q = proc(\mathcal{A}_2, q_j)$.

By definition of $R'$, we have $(proc(\mathcal{A}_1, s_i),\ proc(\mathcal{A}_2, s_j)) \in R'$, since $(s_i, s_j) \in R$. That is, $(P, Q) \in R'$ as required. This concludes the proof that $R'$ is a bisimulation

Since $R$ is a bisimulation between $\mathcal{A}_1$ and $\mathcal{A}_2$, $(q_{0,1}, q_{0,2}) \in R$. Then, by definition of $R'$, $(proc(\mathcal{A}_1, q_{0,1}), proc(\mathcal{A}_2, q_{0,2})) \in R'$. Since $R'$ is a bisimulation, $(proc(\mathcal{A}_1, q_{0,1}) \cong proc(\mathcal{A}_2, q_{0,2}))$.

□

With this lemma, we guarantee that *proc* is a morphism preserving equality relation of port automata in the corresponding processes.

# Chapter 3

# Connector Calculus for Reo

In Chapter 2 we presented the Reo coordination language, with a port automata semantics, the mCRL2 toolset, and the encoding of Reo connectors in mCRL2, presented in [7]. This chapter presents a calculus of Reo connectors, defined by Proença and Clarke [8]. The core calculus is a monoidal category, whose primitive elements are Reo channels composed sequentially or with a tensor product. The core calculus is extended with parameters to form families of connectors, from which we abstract in the chapter.

The main focus of this chapter is the creation of an encoding of the core calculus into mCRL2, inspired by the encoding presented in Chapter 2. To achieve this, we translate the semantics of the calculus, originally defined by the tile model [4], into an equivalent port automata semantics, which we adapt into an mCRL2 encoding.

This chapter begins by presenting the syntax of the core calculus, and the original semantics using the tile model (Sections 3.1 and 3.2). Following this, we present a translation of the tile semantics into port automata, accompanied by a proof of correctness for the translation (Section 3.3). Finally we present an mCRL2 encoding and respective proof of correctness, supported by the translation presented in Chapter 2 (Section 3.4).

$$
\begin{array}{lll}
c ::= \mathsf{id}_n & \text{identities} & p \in \mathcal{P} ::= \Delta_n \quad \text{duplicator into } n \text{ ports} \\
\mid \gamma_{n,m} & \text{symmetries} & \mid \nabla_n \quad \text{merger of } n \text{ inputs} \\
\mid p \in \mathcal{P} & \text{primitive connectors} & \mid \mathsf{drain} \quad \text{synchronous drain} \\
\mid c_1 \,;\, c_2 & \text{sequential composition} & \mid \mathsf{fifo} \quad \text{buffer} \\
\mid c_1 \oplus c_2 & \text{parallel composition} & \mid \ldots \quad \text{user-defined connectors} \\
\mid \mathsf{Tr}_n(c) & \text{traces (feedback loops)} &
\end{array}
$$

Figure 3.1: Grammar for core connectors, where $n, m \in \mathbb{N}$.

## 3.1 Syntax

We present the basic syntax in Fig. 3.1. Our syntax is a simplification of the original syntax, where we use natural numbers to represent the input ant output ports of the connectors, representing the sum of the tensors used in [8]. The type of a connector represents the input ports and output ports of the connector. Ports are the equivalent to channel ends in Reo, where input ports represent source ends, and output ports represent sink ends. Intuitively, each connector has a sequence of input ports and a sequence of output ports, which we number incrementally from 1.

The primitive elements of our calculus are Reo channels with some modifications. The primitive $\mathsf{id}_1$ is equivalent to the sync channel of Reo, while $\mathsf{id}_n$ represents $n$ $\mathsf{id}_1$ primitives, with no common input/output ports (in this case we say they are in parallel). The primitive $\Delta_n$ presents a replicator with $n$ output ports, which replicates the data received by the input port into all the output ports instantly. The $\nabla_n$ contains $n$ input ports, and, when an input port contains data to be received by the primitive, it copies the respective data to the output port.

We combine primitives using sequential and parallel composition. Composing two connectors sequentially $c_1; c_2$ means connecting the $i$-th sink port $c_1$ to the $i$-th source port of $c_2$, for every sink port of $c_1$ and source port of $c_2$; composing connectors in parallel $c_1 \oplus c_2$ means combining all source and sink ports of both $c_1$ and $c_2$; wrapping a connector $c$ by a trace over $n$ means connecting the last $n$ sink ports of $c$ to its last $n$ source ports.

In Fig. 3.2 the reader can find small examples with the respective graphical

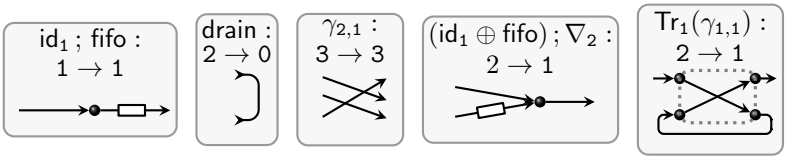
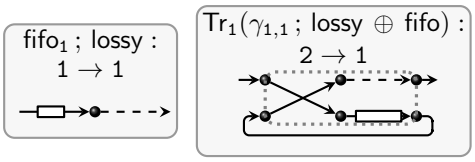



Figure 3.2: Connectors, their interfaces, and their visualization.

representations, where each example contains (1) a connector on top, (2) its interfaces in the middle, and (3) its visual representation below depicting inputs on the left and outputs on the right. In Example 3.1, we present two connectors with similar behaviour, which we demonstrate later in this chapter.

**Example 3.1.** Consider the following examples:



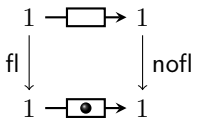

The connector on the left presents a simple fifo connected sequentially to a lossy. The right connector presents a different approach to the connector on the left. Using the $\mathsf{Tr}_1(\_)$ operator, we connect the second input port of $\gamma_{1,1}$, to the output port of the fifo. Since $\gamma_{1,1}$ is composed of $\mathsf{id}_1$ primitives, with abstractable behaviour except for the order of outputs ($\mathsf{id}_1$ is the identity of the sequential composition), this connector behaves as if the fifo is directly connected to the lossy, as we can see in the graphical representation.

## 3.2 Tile Semantics

The semantics of Reo connectors, written using this calculus, uses the Tile Model [4], following the original publication of this calculus [8]. A tile consists of 4 objects, which, in the case of this semantic model, are natural numbers representing the input and output ports of a connector. A tile also consists of horizontal and vertical morphisms which are objects from a category $\mathcal{H}$ and $\mathcal{V}$ respectively. The horizontal morphisms describe the construction of the connector, while the vertical morphisms describe the progress in time of the connector. More concretely, the horizontal morphisms are connectors specified by Fig. 3.1,

and the vertical tiles are either fl, nofl, or a tensor product of these. Visually, we represent each tile as a square, where the objects are the corners, the horizontal morphisms are arrows from left to right between pairs of objects, and the vertical morphisms are arrows from top to bottom between pairs of objects.

**Example 3.2.** The tile




describes a transition of the primitive fifo, such that data flows through the input port, and no data flows through the output port. Furthermore, after the transition, the fifo becomes a fifofull primitive. We can represent the tiles simply as $\mathsf{fifo} \xrightarrow[\mathsf{nofl}]{\mathsf{fl}} \mathsf{fifofull}$. We will use this simple representation as our standard notation throughout this document.

### 3.2.1 Tiles of Primitive Connectors

To fully describe the behaviours of a connector using tiles, we need a set, such that each element describes one of the possible behaviours. In Fig. 3.3 we define the set of tiles for each primitive.

### 3.2.2 Tile Composition

We can compose tiles to specify more complex connectors in three ways: horizontally (;) , vertically ($\circ$), and in parallel ($\oplus$). Tiles can be composed horizontally if the left and right morphisms match respectively; they can be composed vertically if the down and up morphisms match respectively, and they can always be composed in parallel. The formal compositions are defined below:

$$c_1 \xrightarrow[v]{v_1} c_2 \,;\, c_1' \xrightarrow[v_2]{v} c_2' \;=\; (c_1; c_1') \xrightarrow[v_2]{v_1} (c_2; c_2') \qquad \text{(horizontal)}$$

$$c_1 \xrightarrow[v_2]{v_1} c \circ c \xrightarrow[v_2']{v_1'} c_2 \;=\; c_1 \xrightarrow[v_2' \circ v_2]{v_1' \circ v_1} c_2 \qquad \text{(vertical)}$$

$$c_1 \xrightarrow[v_2]{v_1} c_2 \oplus c_1' \xrightarrow[v_2']{v_1'} c_2' \;=\; c_1 \oplus c_2 \xrightarrow[v_2 \oplus v_2']{v_1 \oplus v_1'} c_1' \oplus c_2' \qquad \text{(parallel)}$$

$$\mathsf{id}_1 = \left\{ \mathsf{id}_1 \xrightarrow[\mathsf{fl}]{} \mathsf{id}_1 \,,\; \mathsf{id}_1 \xrightarrow[\mathsf{nofl}]{} \mathsf{id}_1 \right\}$$

$$\gamma_{1,1} = \left\{ \gamma_{1,1} \xrightarrow[\mathsf{nofl}\oplus\mathsf{fl}]{\mathsf{fl}\oplus\mathsf{nofl}} \gamma_{1,1} \,,\; \gamma_{1,1} \xrightarrow[\mathsf{fl}\oplus\mathsf{nofl}]{\mathsf{nofl}\oplus\mathsf{fl}} \gamma_{1,1} \,,\; \gamma_{1,1} \xrightarrow[\mathsf{fl}\oplus\mathsf{fl}]{\mathsf{fl}\oplus\mathsf{fl}} \gamma_{1,1} \,,\; \gamma_{1,1} \xrightarrow[\mathsf{nofl}\oplus\mathsf{nofl}]{\mathsf{nofl}\oplus\mathsf{nofl}} \gamma_{1,1} \right\}$$

$$\Delta_2 = \left\{ \Delta_2 \xrightarrow[\mathsf{fl}\oplus\mathsf{fl}]{\mathsf{fl}} \Delta_2 \,,\; \Delta_2 \xrightarrow[\mathsf{nofl}\oplus\mathsf{nofl}]{\mathsf{nofl}} \Delta_2 \right\}$$

$$\nabla_2 = \left\{ \nabla_2 \xrightarrow[\mathsf{fl}]{\mathsf{fl}\oplus\mathsf{nofl}} \nabla_2 \,,\; \nabla_2 \xrightarrow[\mathsf{fl}]{\mathsf{nofl}\oplus\mathsf{fl}} \nabla_2 \,,\; \nabla_2 \xrightarrow[\mathsf{nofl}]{\mathsf{nofl}\oplus\mathsf{nofl}} \nabla_2 \right\}$$

$$\mathsf{drain} = \left\{ \mathsf{drain} \xrightarrow[\mathsf{fl}]{\mathsf{fl}} \mathsf{drain} \,,\; \mathsf{drain} \xrightarrow[\mathsf{nofl}]{\mathsf{nofl}} \mathsf{drain} \right\}$$

$$\mathsf{lossy} = \left\{ \mathsf{lossy} \xrightarrow[\mathsf{fl}]{\mathsf{fl}} \mathsf{lossy} \,,\; \mathsf{lossy} \xrightarrow[\mathsf{nofl}]{\mathsf{fl}} \mathsf{lossy} \,,\; \mathsf{lossy} \xrightarrow[\mathsf{nofl}]{\mathsf{nofl}} \mathsf{lossy} \right\}$$

$$\mathsf{fifo} = \left\{ \mathsf{fifo} \xrightarrow[\mathsf{nofl}]{\mathsf{fl}} \mathsf{fifofull} \,,\; \mathsf{fifo} \xrightarrow[\mathsf{nofl}]{\mathsf{nofl}} \mathsf{fifo} \right\}$$

$$\mathsf{fifofull} = \left\{ \mathsf{fifofull} \xrightarrow[\mathsf{fl}]{\mathsf{nofl}} \mathsf{fifo} \,,\; \mathsf{fifofull} \xrightarrow[\mathsf{nofl}]{\mathsf{nofl}} \mathsf{fifofull} \right\}$$

Figure 3.3: Behaviour of primitive connectors using tiles.

Furthermore, given a connector $c_1$, and a natural number $n$, the behaviour of the trace connector $\mathsf{Tr}_n(c_1)$ is defined by:

$$\mathsf{Tr}_n(c_1) = \{ \mathsf{Tr}_n(c_1) \xrightarrow[v_2]{v_1} \mathsf{Tr}_n(c_2) \;\mid\; \exists (v : n \to n \in \mathcal{V}) \;.\; c_1 \xrightarrow[v_2 \oplus v]{v_1 \oplus v} c_2 \text{ exists} \} \quad \text{(trace)}$$

### 3.2.3 Bisimulation of Tiles

We define the relation of bisimulation between tiles using the one presented by Gadducci and Montanari [4]:

**Definition 3.1.** Consider the relation $R \subseteq \mathcal{H} \times \mathcal{H}$. We say that $R$ is a bisimulation if $\forall s, t \in \mathcal{H}$:

1. $s \xrightarrow[b]{a} s' \;\wedge\; (s,t) \in R \;\Rightarrow\; \exists_{t' \in \mathcal{H}}\; s.t.\; t \xrightarrow[b]{a} t' \;\wedge\; (s',t') \in R$;
2. $t \xrightarrow[b]{a} t' \;\wedge\; (s,t) \in R \;\Rightarrow\; \exists_{s' \in \mathcal{H}}\; s.t.\; s \xrightarrow[b]{a} s' \;\wedge\; (s',t') \in R$

given $a, b \in \mathcal{V}$.

Given $s, t \in \mathcal{H}$, we say that $s$ is bisimilar to $t$ (notation: $s \cong t$) iff there is a bisimulation $R$ such that $(s,t) \in R$.

We present the tiles for the connectors presented in Example 3.1, as example of bisimilar connectors, in Examples 3.3 and 3.4. Since these connectors contain a fifo primitive and there are two set of tiles to fully describe the behaviour (fifo, fifofull), we present the two sets.

**Example 3.3.** Consider the connector defined by fifo; lossy. The tiles for the individual processes are defined in Fig. 3.3. Using the horizontal composition of tiles, the resulting set of tiles for the connector is:

$$\mathsf{fifo}\ ;\ \mathsf{lossy} = \left\{ \mathsf{fifo};\mathsf{lossy} \xrightarrow[\mathsf{nofl}]{\mathsf{fl}} \mathsf{fifofull};\mathsf{lossy}\ ,\ \mathsf{fifo};\mathsf{lossy} \xrightarrow[\mathsf{nofl}]{\mathsf{nofl}} \mathsf{fifo};\mathsf{lossy} \right\}$$

$$\mathsf{fifofull}\ ;\ \mathsf{lossy} = \left\{ \begin{array}{l} \mathsf{fifofull};\mathsf{lossy} \xrightarrow[\mathsf{fl}]{\mathsf{nofl}} \mathsf{fifo};\mathsf{lossy}\ ,\ \mathsf{fifofull};\mathsf{lossy} \xrightarrow[\mathsf{nofl}]{\mathsf{nofl}} \mathsf{fifo};\mathsf{lossy}\ , \\ \qquad \mathsf{fifofull};\mathsf{lossy} \xrightarrow[\mathsf{nofl}]{\mathsf{nofl}} \mathsf{fifofull};\mathsf{lossy} \end{array} \right\}$$

**Example 3.4.** Consider the connector defined by $c = \mathsf{Tr}_1(\gamma_{1,1}\,;\,\mathsf{lossy} \oplus \mathsf{fifo})$. With the tiles presented in Fig. 3.3 we incrementally build the tiles for $c$. We begin by presenting the tiles for $c_1 = \mathsf{lossy} \oplus \mathsf{fifo}$, and $c_2 = \mathsf{lossy} \oplus \mathsf{fifofull}$, using the parallel composition:

$$c_1 = \left\{ c_1 \xrightarrow[\mathsf{fl}\oplus\mathsf{nofl}]{\mathsf{fl}\oplus\mathsf{fl}} c_2\ ,\ c_1 \xrightarrow[\mathsf{nofl}\oplus\mathsf{nofl}]{\mathsf{fl}\oplus\mathsf{fl}} c_2\ ,\ c_1 \xrightarrow[\mathsf{nofl}\oplus\mathsf{nofl}]{\mathsf{nofl}\oplus\mathsf{fl}} c_2\ ,\ c_1 \xrightarrow[\mathsf{fl}\oplus\mathsf{nofl}]{\mathsf{fl}\oplus\mathsf{nofl}} c_1\ ,\ c_1 \xrightarrow[\mathsf{nofl}\oplus\mathsf{nofl}]{\mathsf{fl}\oplus\mathsf{nofl}} c_1\ ,\ c_1 \xrightarrow[\mathsf{nofl}\oplus\mathsf{nofl}]{\mathsf{nofl}\oplus\mathsf{nofl}} c_1 \right\}$$

$$c_2 = \left\{ c_2 \xrightarrow[\mathsf{fl}\oplus\mathsf{nofl}]{\mathsf{fl}\oplus\mathsf{nofl}} c_2\ ,\ c_2 \xrightarrow[\mathsf{nofl}\oplus\mathsf{nofl}]{\mathsf{fl}\oplus\mathsf{nofl}} c_2\ ,\ c_2 \xrightarrow[\mathsf{nofl}\oplus\mathsf{nofl}]{\mathsf{nofl}\oplus\mathsf{nofl}} c_2\ ,\ c_2 \xrightarrow[\mathsf{fl}\oplus\mathsf{fl}]{\mathsf{fl}\oplus\mathsf{nofl}} c_1\ ,\ c_2 \xrightarrow[\mathsf{nofl}\oplus\mathsf{fl}]{\mathsf{fl}\oplus\mathsf{nofl}} c_1\ ,\ c_2 \xrightarrow[\mathsf{nofl}\oplus\mathsf{fl}]{\mathsf{nofl}\oplus\mathsf{nofl}} c_1 \right\}$$

Using these sets, we can build $c_3 = \gamma_{1,1};\mathsf{lossy} \oplus \mathsf{fifo}$, and $c_4 = \gamma_{1,1};\mathsf{lossy} \oplus \mathsf{fifofull}$, by applying the horizontal composition:

$$c_3 = \left\{ c_3 \xrightarrow[\mathsf{fl}\oplus\mathsf{nofl}]{\mathsf{fl}\oplus\mathsf{fl}} c_4\ ,\ c_3 \xrightarrow[\mathsf{nofl}\oplus\mathsf{nofl}]{\mathsf{fl}\oplus\mathsf{fl}} c_4\ ,\ c_3 \xrightarrow[\mathsf{nofl}\oplus\mathsf{nofl}]{\mathsf{fl}\oplus\mathsf{nofl}} c_4\ ,\ c_3 \xrightarrow[\mathsf{fl}\oplus\mathsf{nofl}]{\mathsf{nofl}\oplus\mathsf{fl}} c_3\ ,\ c_3 \xrightarrow[\mathsf{nofl}\oplus\mathsf{nofl}]{\mathsf{nofl}\oplus\mathsf{fl}} c_3\ ,\ c_3 \xrightarrow[\mathsf{nofl}\oplus\mathsf{nofl}]{\mathsf{nofl}\oplus\mathsf{nofl}} c_3 \right\}$$

$$c_4 = \left\{ c_4 \xrightarrow[\mathsf{fl}\oplus\mathsf{nofl}]{\mathsf{nofl}\oplus\mathsf{fl}} c_4\ ,\ c_4 \xrightarrow[\mathsf{nofl}\oplus\mathsf{nofl}]{\mathsf{nofl}\oplus\mathsf{fl}} c_4\ ,\ c_4 \xrightarrow[\mathsf{nofl}\oplus\mathsf{nofl}]{\mathsf{nofl}\oplus\mathsf{nofl}} c_4\ ,\ c_4 \xrightarrow[\mathsf{fl}\oplus\mathsf{fl}]{\mathsf{nofl}\oplus\mathsf{fl}} c_3\ ,\ c_4 \xrightarrow[\mathsf{nofl}\oplus\mathsf{fl}]{\mathsf{nofl}\oplus\mathsf{fl}} c_3\ ,\ c_4 \xrightarrow[\mathsf{nofl}\oplus\mathsf{fl}]{\mathsf{nofl}\oplus\mathsf{nofl}} c_3 \right\}$$

Finally, we can build $c$, and $c' = \mathsf{Tr}_1(\gamma_{1,1}\,;\,\mathsf{lossy} \oplus \mathsf{fifofull})$, using the trace rule:

$$c = \left\{ c \xrightarrow[\mathsf{nofl}]{\mathsf{fl}} c'\ ,\ c \xrightarrow[\mathsf{nofl}]{\mathsf{nofl}} c \right\}$$

$$c' = \left\{ c' \xrightarrow[\mathsf{nofl}]{\mathsf{nofl}} c'\ ,\ c' \xrightarrow[\mathsf{fl}]{\mathsf{nofl}} c\ ,\ c' \xrightarrow[\mathsf{nofl}]{\mathsf{nofl}} c \right\}$$

To prove the bisimilarity of the previous examples, we consider the sets of tiles. Let $R = \{(\mathsf{fifo};\mathsf{lossy}, c), (\mathsf{fifofull};\mathsf{lossy}, c')\}$. It can easily be seen that $R$ is a bisimulation. Therefore $\mathsf{fifo};\mathsf{lossy} \cong c$.

## 3.3 Connectors as Port Automata

We now define a semantic model for the calculus of connectors using port automata, and address its correctness.

Our port automata follow the definitions presented in Chapter 2. Furthermore, we define *port substitution* of $a$ by $b$ in an automaton $\mathcal{A} = (Q, N, \rightarrow, q_0)$ as the automaton $\mathcal{A}\{a \mapsto b\} = (Q, N\{a \mapsto b\}, \dashrightarrow, q_0)$, where $q_i \overset{X\{a\mapsto b\}}{\dashrightarrow} q_j$ iff $q_i \xrightarrow{X} q_j$, and $X\{a \mapsto b\}$ denotes the set $X$ replacing $a$ by $b$.

### 3.3.1 Encoding Primitives as Port Automata

Recall that the semantics of the Core Calculus is given by the Tiles model, where a tile $c_1 \xrightarrow[snk]{src} c_2$ means that the connector $c_1$ can evolve to a new state given by the connector $c_2$, by firing its source ports based on *src* and its sink ports based on *snk*. Here *src* and *snk* are morphisms built by composing simpler morphisms $\mathsf{fl}$ and $\mathsf{nofl}$, indicating which ports have flow and no flow.

The encoding of a connector $c$ into a PA is written as $\mathcal{PA}(c)$, defined below. Each port is a pair $(n, s)$ where $n \in \mathbb{N}$ is the order number of its source or sink node, and $s \in \{\mathsf{sr}, \mathsf{sk}, \mathsf{mx}\}$ is a constant that marks it as being a source ($\mathsf{sr}$), sink ($\mathsf{sk}$), or mixed ($\mathsf{mx}$) port. The latter is a temporary marking used when composing port automata (defined below).

**Definition 3.2** (Tiles of a connector)**.** Given a core connector $c$, we write $T(c)$ to represent all tiles for $c$ and for the reachable states from $c$. Formally, $T(c)$ is the smallest set such that, for every tile $t = \left(c \xrightarrow[sk]{sr} c'\right)$ we have that $t \in T(c)$ and $T(c') \subseteq T(c)$.

**Definition 3.3** (Reachable connectors)**.** Given a connector $c$, we write $Reach(c)$ to represent all reachable connectors from $c$, i.e., $Reach(c)$ is the smallest set such that $c \in Reach(c)$, and for every tile $c \xrightarrow[sk]{sr} c'$ we have that $Reach(c') \subseteq Reach(c)$.

**Definition 3.4** (Encoding $\mathcal{PA}(c)$)**.** Let $c$ be a connector from $n$ to $m$, and $\ell$ a unique identifier. Its port automaton $\mathcal{PA}(c)$ is $(Q, N, \rightarrow, q_0)$ where

- $Q = Reach(c)$
- $N = \{(i, \mathsf{sr}) \mid i \in \{1 \ldots n\}\} \cup \{(j, \mathsf{sk}) \mid j \in \{1 \ldots m\}\}$

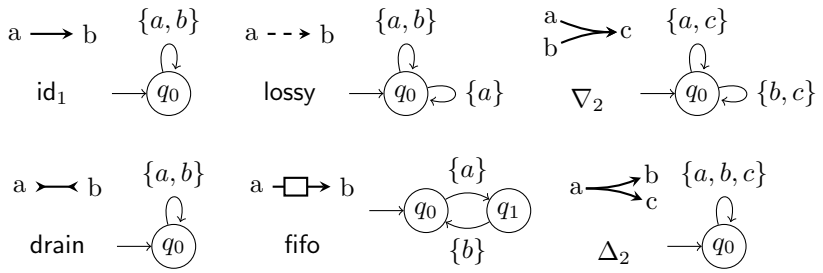



Figure 3.4: Port Automata of primitive connectors.

- $q \xrightarrow{X_{\mathsf{sr}} \cup X_{\mathsf{sk}}} q' \Leftrightarrow \exists t \in T(c) : t = c_1 \xrightarrow[snk]{src} c_2 \ \wedge$
  $X_{\mathsf{sr}} = \{(i, \mathsf{sr}) \mid src = v_1 \oplus \cdots \oplus v_n, i \in \{1 \ldots n\}, v_i = \mathsf{fl}\}$
  $X_{\mathsf{sk}} = \{(i, \mathsf{sk}) \mid snk = v_1 \oplus \cdots \oplus v_m, j \in \{1 \ldots m\}, v_j = \mathsf{fl}\}$

For example, the fifo primitive can be encoded as $\mathcal{PA}(\mathsf{fifo}) = (\{\mathsf{fifo}, \mathsf{fifofull}\}, \{(1, \mathsf{sr}), (1, \mathsf{sk})\}, \rightarrow, \mathsf{fifo})$, where

$$\mathsf{fifo} \xrightarrow{(1,\mathsf{sr})} \mathsf{fifofull} \qquad \mathsf{fifofull} \xrightarrow{(1,\mathsf{sk})} \mathsf{fifo} \qquad \mathsf{fifo} \xrightarrow{\emptyset} \mathsf{fifo} \qquad \mathsf{fifofull} \xrightarrow{\emptyset} \mathsf{fifofull}.$$

In Fig. 3.4 we present example port automata of channels commonly found in the literature, and the respective primitive of our calculus.

### 3.3.2 Composing Port Automata for Connectors

Excluding the case of the trace operation, the port automata of two connectors are composed using the product operation $\bowtie$ with port renaming to fit our definition of $\mathcal{PA}(c)$. More precisely, we define $\bowtie_;$, and $\bowtie_\oplus$ to define the composition through the sequential and parallel operations respectively. The mark $\mathsf{mx}$ marks the ports that are no longer source or sink, and is used for synchronisation purposes.

**Definition 3.5.** Let $c_1, c_2$ be two connectors, such that, there exist $n, m, l$, naturals, such that $c_1 : n \rightarrow l$, and $c_2 : l \rightarrow m$. Let

$$A_1 = \mathcal{PA}(c_1) = (Q_1, \mathcal{N}_1, \rightarrow_1, q_1) \qquad A_2 = \mathcal{PA}(c_2) = (Q_2, \mathcal{N}_2, \rightarrow_2, q_2)$$

be their respective port automata. We define $A_1 \bowtie_; A_2 = (A_1\sigma_1 \bowtie A_2\sigma_2)\backslash X$, where $\sigma_1$, $\sigma_2$ and $X$ define port renamings and hiding of ports that mimic the

connecting of ports from $c_1$ to $c_2$:

$$\sigma_1 = \{(i,\mathsf{sk}) \mapsto (i,\mathsf{mx}) \mid (i,\mathsf{sk}) \in N_1\} \qquad X = \{(i,\mathsf{mx}) \mid (i,\mathsf{sk}) \in N_1\}$$
$$\sigma_2 = \{(i,\mathsf{sr}) \mapsto (i,\mathsf{mx}) \mid (i,\mathsf{sr}) \in N_2\}$$

**Definition 3.6.** Let $c_1, c_2$ be two connectors, such that, for $i \in 1,2$, $c_i : n_i \to m_i$, and $A_i = \mathcal{PA}(c_i) = (Q_i, \mathcal{N}_i, \to_i, q_i)$. We define $A_1 \bowtie_\oplus A_2 = A_1 \bowtie (A_2\sigma)$ where $\sigma$ defines the port renaming:

$$\sigma = \{(i,\mathsf{sr}) \mapsto (i+n_1,\mathsf{sr}) \mid (i,\mathsf{sr}) \in N_2\}$$
$$\cup\ \{(j,\mathsf{sk}) \mapsto (j+m_1,\mathsf{sk}) \mid (j,\mathsf{sk}) \in N_2\}$$

We define the trace operator for port automata as follows:

**Definition 3.7.** Let $c_1 : m_1 \to m_2$ a connector, and $A_1 = \mathcal{PA}(c_1) = (Q_1, \mathcal{N}_1, \to_1, q_1)$ its port automata. Let $n \in \mathbb{N}$, such that $n \leq m_1$, and $n \leq m_2$. We define $Tr_n(A_1) = (Q_1, \mathcal{N}_1 \backslash X, \to_{\mathsf{Tr}}, q_1)$, where

$$X = \{(m_1 - i,\mathsf{sr}) \mid 0 \leq i \leq n-1\} \cup \{(m_2 - i,\mathsf{sk}) \mid 0 \leq i \leq n-1\}$$
$$q \xrightarrow{K \backslash X}_{\mathsf{Tr}} q' \text{ iff } \ q \xrightarrow{K} q' \wedge \forall_{i \in \{0 \ldots n-1\}} (m_1 - i,\mathsf{sr}) \in K \Leftrightarrow (m_2 - i,\mathsf{sk}) \in K$$

The following examples provide an incremental build of the port automata for the connector $\mathsf{Tr}_1(\gamma_{1,1}; \mathsf{fifo} * \mathsf{lossy})$ presented in Example 3.4. Notice that the final PA is bisimilar to $\mathcal{PA}(\mathsf{Tr}_1(\gamma_{1,1}; \mathsf{fifo} * \mathsf{lossy}))$. This property is the main topic of Section 3.3.3.

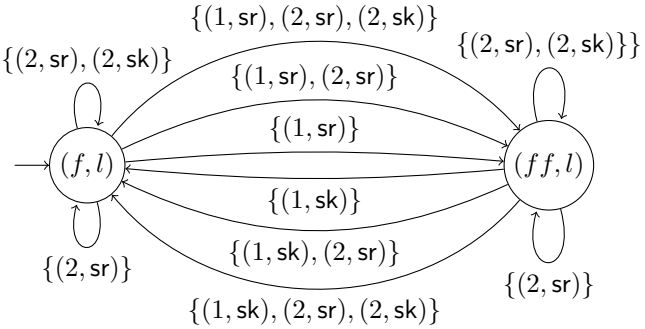
**Example 3.5** ($\mathcal{PA}(\mathsf{fifo}) \bowtie_\oplus \mathcal{PA}(\mathsf{lossy})$)**.** Consider the port automata

- $\mathcal{PA}(\mathsf{fifo}) = (\{\mathsf{fifo}, \mathsf{fifofull}\}, \{(1,\mathsf{sr}), (1,\mathsf{sk})\}, \to_1, \mathsf{fifo})$, and
- $\mathcal{PA}(\mathsf{lossy}) = (\{\mathsf{lossy}\}, \{(1,\mathsf{sr}), (1,\mathsf{sk})\}, \to_2, \mathsf{lossy})$

where

$$\mathsf{fifo} \xrightarrow{(1,\mathsf{sr})}_1 \mathsf{fifofull} \qquad \mathsf{fifofull} \xrightarrow{(1,\mathsf{sk})}_1 \mathsf{fifo} \qquad \mathsf{fifo} \xrightarrow{\emptyset}_1 \mathsf{fifo} \qquad \mathsf{fifofull} \xrightarrow{\emptyset}_1 \mathsf{fifofull}$$
$$\mathsf{lossy} \xrightarrow{(1,\mathsf{sr})}_2 \mathsf{lossy} \quad \mathsf{lossy} \xrightarrow{(1,\mathsf{sr}),(1,\mathsf{sk})}_2 \mathsf{lossy} \quad \mathsf{lossy} \xrightarrow{\emptyset}_2 \mathsf{lossy}$$

The port automata $\mathcal{PA}(\mathsf{fifo}) \bowtie_\oplus \mathcal{PA}(\mathsf{lossy}) = \mathcal{PA}(\mathsf{fifo}) \bowtie (\mathcal{PA}(\mathsf{lossy})\sigma)$, where $\sigma = \{(1,\mathsf{sr}) \to (2,\mathsf{sr}), (1,\mathsf{sk}) \to (2,\mathsf{sk})\}$, and is described graphically (without empty transitions) by:




**Example 3.6** $(\mathcal{PA}(\gamma_{1,1}) \bowtie_{;} (\mathcal{PA}(\mathsf{fifo}) \bowtie_{\oplus} \mathcal{PA}(\mathsf{lossy})))$**.** Consider the automaton created in the previous example, and the automaton

- $\mathcal{PA}(\gamma_{1,1}) = (\{\gamma\}, \{(1,\mathsf{sr}), (1,\mathsf{sk}), (2,\mathsf{sr}), (2,\mathsf{sk})\}, \rightarrow, \gamma)$,

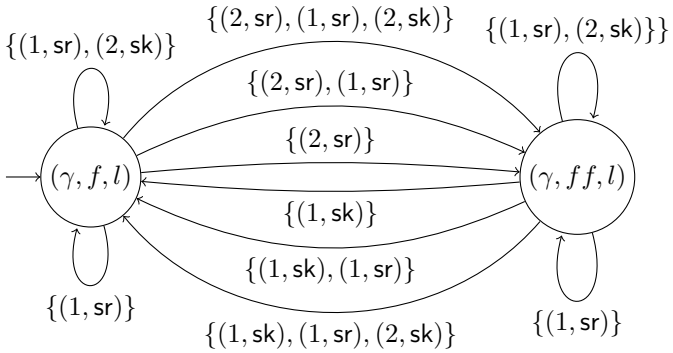
where

$$\gamma \xrightarrow{(1,\mathsf{sr}),(2,\mathsf{sk})} \gamma \qquad \gamma \xrightarrow{(2,\mathsf{sr}),(1,\mathsf{sk})} \gamma \qquad \gamma \xrightarrow{(1,\mathsf{sr}),(2,\mathsf{sk}),(2,\mathsf{sr}),(1,\mathsf{sk})} \gamma \qquad \gamma \xrightarrow{\emptyset} \gamma$$

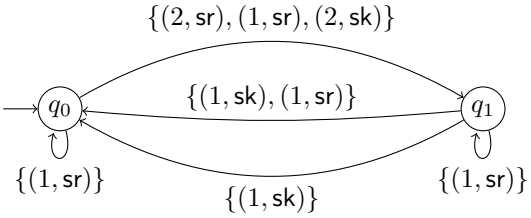
Following the definition of $\bowtie_{;}$, the port automaton $\mathcal{PA}(\gamma_{1,1}) \bowtie_{;} (\mathcal{PA}(\mathsf{fifo}) \bowtie_{\oplus} \mathcal{PA}(\mathsf{lossy})) = ((\mathcal{PA}(\gamma_{1,1})\sigma_1) \bowtie (\mathcal{PA}(\mathsf{fifo}) \bowtie_{\oplus} \mathcal{PA}(\mathsf{lossy})\sigma_2)) \setminus X$, where

$$\sigma_1 = \{(1,\mathsf{sk}) \rightarrow (1,\mathsf{mx}), (2,\mathsf{sk}) \rightarrow (2,\mathsf{mx})\}$$
$$\sigma_2 = \{(1,\mathsf{sr}) \rightarrow (1,\mathsf{mx}), (2,\mathsf{sr}) \rightarrow (2,\mathsf{mx})\} \qquad X = \{(1,\mathsf{mx}), (2,\mathsf{mx})\}$$

and is described graphically (without empty transitions) by:




**Example 3.7** $(Tr_1(\mathcal{PA}(\gamma_{1,1}) \bowtie_{;} \mathcal{PA}(\mathsf{lossy}) \bowtie_{\oplus} \mathcal{PA}(\mathsf{fifo})))$**.** Consider the automaton of the previous example. The port automaton $Tr_1(\mathcal{PA}(\gamma_{1,1}) \bowtie_{;} \mathcal{PA}(\mathsf{lossy}) \bowtie_{\oplus} \mathcal{PA}(\mathsf{fifo}))$ is described graphically (without empty transitions) by:




### 3.3.3 Correctness of $\mathcal{PA}(c)$

We say our encoding is correct with respect to an automaton $A$ if $\mathcal{PA}(c) \cong A$. We begin our proof by showing that $\mathcal{PA}(c)$ preserves the bisimulation equivalence of the tile semantics. We then proceed to use an inductive argument to prove the preservation of the structure of $\mathcal{PA}(c)$ with respect to the operations defined in the previous section. We show that (1) the encodings of primitive channels from Fig. 3.4 are correct with respect to the automata from $\mathcal{PA}(c)$, and (2) the encoding of a connector built with the sequential, parallel, or trace operators is correct with respect to $\bowtie_;$, $\bowtie_\oplus$, and $Tr_n(c)$ respectively.

For simplicity, we ignore all reflexive transitions with empty sets as labels in $\mathcal{PA}(c)$, which must exist for all primitive connectors – because the Port Automata semantics assumes that connectors can decide not to have data flow and remain in the same state.

**Lemma 3.1** (Preservation of Equivalence)**.** Let $c_1, c_2 \in \mathcal{H}$, be two connectors (horizontal morphisms). If $c_1 \cong c_2$, then $\mathcal{PA}(c_1) \cong \mathcal{PA}(c_2)$.

*Proof.* Let $R \subseteq \mathcal{H} \times \mathcal{H}$ be a bisimulation, such that, $(c_1, c_2) \in R$. Let

$$P_1 = \mathcal{PA}(c_1) = (Q_1, \mathcal{N}_1, \rightarrow_1, q_1)$$
$$P_2 = \mathcal{PA}(c_2) = (Q_2, \mathcal{N}_2, \rightarrow_2, q_2)$$

be the respective PA of the connectors $c_1$, and $c_2$. Let $R' \subseteq Q_1 \times Q_2 = \{(c_i, c_j) \mid (c_i, c_j) \in R\}$. We want to prove that $R'$ is a bisimulation, i.e., for all $c \in Q_1$, and $d \in Q_2$:

1. $c \xrightarrow{N} x \wedge (c, d) \in R' \Rightarrow \exists_{y \in Q_2}$ s.t. $d \xrightarrow{N} y \wedge (x, y) \in R'$;
2. $d \xrightarrow{N} y \wedge (c, d) \in R' \Rightarrow \exists_{x \in Q_1}$ s.t. $c \xrightarrow{N} x \wedge (x, y) \in R'$.

We focus our proof in **1**, as **2** is analogous.

Let $c \xrightarrow{N} x$. By definition of $\mathcal{PA}(c_1)$, there exists $c \xrightarrow[b]{a} x \in T(c_1)$, s.t., $N = X_{\mathsf{sr}} \cup X_{\mathsf{sk}}$, where $X_{\mathsf{sr}} = \{(i, \mathsf{sr}) \mid a = v_1 \oplus \cdots \oplus v_n, i \in \{1 \ldots n\}, v_i = \mathsf{fl}\}$, and $X_{\mathsf{sk}} = \{(i, \mathsf{sk}) \mid b = v_1 \oplus \cdots \oplus v_m, j \in \{1 \ldots m\}, v_j = \mathsf{fl}\}$. Then, by definition of tile bisimulation and because $(c, d) \in R'$ implies $(c, d) \in R$, $d \xrightarrow[b]{a} y \in T(c_2)$, and $(x, y) \in R$. Therefore $d \xrightarrow{N} y$, and $(x, y) \in R'$.

Thus, $\mathcal{PA}(c_1) \cong \mathcal{PA}(c_2)$.

Lemma 3.1 shows that bisimilar connectors generate bisimilar port automata, following our encoding. For example, consider the connectors of Examples 3.3 and 3.4. Since we showed that the respective tiles of these connectors are bisimilar, we have no need to find a relation between their encoded port automata, to know that they are bisimilar.

**Lemma 3.2** (Correctness of primitive's encodings)**.** Any primitive from Fig. 3.3 is correct w.r.t. its corresponding automaton from Fig. 3.4, after renaming ports in the latter to follow the same convention as in the encoding (e.g., $(1, \mathsf{sr})$ instead of $a$).

*Proof.* We will only show that this lemma holds for one of the connectors, the fifo, because the other connectors can be shown in a similar way. Recall that after Definition 3.4 we defined $\mathcal{PA}(\mathsf{fifo})$ as an example. The resulting automaton has 4 transitions, and after ignoring the reflexive and empty transitions only two remain. Recall also the port automaton of the fifo in Fig. 3.4. It is enough to observe that $R = \{\langle \mathsf{fifo}, q_0 \rangle, \langle \mathsf{fifofull}, q_1 \rangle\}$ is a strong bisimulation between the two automata, after replacing $a$ by $(1, \mathsf{sr})$ and $b$ by $(1, \mathsf{sk})$. □

**Lemma 3.3** (Correctness of $\mathcal{PA}(c_1; c_2)$)**.** Let $c_1 : n_1 \to n_2$, $c_2 : m_1 \to m_2$ be two connectors. If $\mathcal{PA}(c_1)$ and $\mathcal{PA}(c_2)$ are correct with respect to $A_1$ and $A_2$, respectively, and $c_1; c_2$ is well-typed, i.e., $n_2 = m_1$, then $\mathcal{PA}(c_1; c_2)$ is correct with respect to $A_1 \bowtie_; A_2$,

*Proof.* Let $A_1 = (Q_1, N_1, \to_1, q_1)$, $A_2 = (Q_2, N_2, \to_2, q_2)$, $\mathcal{PA}(c_1) = (Q_3, N_3, \to_3, q_3)$, and $\mathcal{PA}(c_2) = (Q_4, N_4, \to_4, q_4)$. Let $R_1 \subseteq Q_3 \times Q_1$, and $R_2 \subseteq Q_4 \times Q_2$ be bisimulations. We show that $\mathcal{PA}(c_1; c_2)$ is correct with respect to $A_1 \bowtie_; A_2$ by showing that the relation

$$R = \{(x; y, (x', y')) \mid x; y \in \mathit{Reach}(c_1; c_2),\ (x, x') \in R_1,\ (y, y') \in R_2\}$$

is a bisimulation, i.e.:

1. $\forall_{p \in Reach(c_1;c_2)}\ p \xrightarrow{K} p' \wedge (p,q) \in R \Rightarrow \exists_{q' \in Q_1 \times Q_2}\ q \xrightarrow{K} q' \wedge (p',q') \in R$;
2. $\forall_{q \in Q_1 \times Q_2}\ q \xrightarrow{K} q' \wedge (p,q) \in R \Rightarrow \exists_{p' \in Reach(c_1;c_2)}\ p \xrightarrow{K} p' \wedge (p',q') \in R$;

**1.**

Consider a transition $(x;y) \xrightarrow{K} (x';y')$ in $\mathcal{PA}(c_1;c_2)$, with $(x;y,(p,q)) \in R$, where $(p,q) \in Q_1 \times Q_2$.

Then there exists $x;y \xrightarrow[sk]{sr} x';y'$ in $T(c_1;c_2)$, such that $K = \{(i,\mathsf{sr}) \mid sr = v_1 \oplus \cdots \oplus v_{n_1}, i \in \{1 \ldots n_1\}, v_i = \mathsf{fl}\} \cup \{(i,\mathsf{sk}) \mid sk' = v_1 \oplus \cdots \oplus v_{m_2}, i \in \{1 \ldots m_2\}, v_i = \mathsf{fl}\}$.

By definition, there exist $x \xrightarrow[v]{sr} x'$ in $T(c_1)$ and $y \xrightarrow[sk]{v} y'$ in $T(c_2)$, and, therefore, there exist $p \xrightarrow{K_1} p'$ in $A_1$ and $q \xrightarrow{K_2} q'$ in $A_2$ such that, $K_1 = \{(i,\mathsf{sr}) \mid sr = v_1 \oplus \cdots \oplus v_{n_1}, i \in \{1 \ldots n_1\}, v_i = \mathsf{fl}\} \cup \{(i,\mathsf{sk}) \mid v = v_1 \oplus \cdots \oplus v_{n_2}, i \in \{1 \ldots n_2\}, v_i = \mathsf{fl}\}$, and $K_2 = \{(i,\mathsf{sr}) \mid v = v_1 \oplus \cdots \oplus v_{n_2}, i \in \{1 \ldots n_2\}, v_i = \mathsf{fl}\} \cup \{(i,\mathsf{sk}) \mid sk' = v_1 \oplus \cdots \oplus v_{m_2}, i \in \{1 \ldots m_2\}, v_i = \mathsf{fl}\}$. By definition of $R$, $(x,p),(x',p') \in R_1$, and $(y,q),(y',q') \in R_2$, and therefore, $(x';y',(p',q')) \in R$.

Furthermore, there is a transition $(p;q) \xrightarrow{K'} (p';q')$ in $A_1 \bowtie_; A_2$, where $K' = ((K_1\sigma_1 \cup K_2\sigma_2) \setminus X) = ((\{(i,\mathsf{sr}) \mid sr = v_1 \oplus \cdots \oplus v_{n_1}, i \in \{1 \ldots n_1\}, v_i = \mathsf{fl}\} \cup \{(i,\mathsf{sk}) \mid v = v_1 \oplus \cdots \oplus v_{n_2}, i \in \{1 \ldots n_2\}, v_i = \mathsf{fl}\})\sigma_1 \cup (\{(i,\mathsf{sr}) \mid v = v_1 \oplus \cdots \oplus v_{n_2}, i \in \{1 \ldots n_2\}, v_i = \mathsf{fl}\} \cup \{(i,\mathsf{sk}) \mid sk' = v_1 \oplus \cdots \oplus v_{m_2}, i \in \{1 \ldots m_2\}, v_i = \mathsf{fl}\})\sigma_2) \setminus X) = \{(i,\mathsf{sr}) \mid sr = v_1 \oplus \cdots \oplus v_{n_1}, i \in \{1 \ldots n_1\}, v_i = \mathsf{fl}\} \cup \{(i,\mathsf{sk}) \mid sk' = v_1 \oplus \cdots \oplus v_{m_2}, i \in \{1 \ldots m_2\}, v_i = \mathsf{fl}\} = K$

Therefore $R$ is a bisimulation, and **1.** is valid.

**2.**

Consider a transition $(p,q) \xrightarrow{K} (p',q')$ in $A_1 \bowtie_; A_2$, and let $(x;y,(p,q)) \in R$, for $x;y \in Reach(c_1;c_2)$.

Then, one of these cases must be true:

1. $\exists p \xrightarrow{K_1} p'$ in $A_1$, $K_1\sigma_1 \cap N_2\sigma_2 = \emptyset$, and $K = (K_1\sigma_1) \setminus X$
2. $\exists q \xrightarrow{K_2} q'$ in $A_2$, $K_2\sigma_2 \cap N_1\sigma_1 = \emptyset$, and $K = (K_1\sigma_2) \setminus X$
3. $\exists p \xrightarrow{K_1} p'$ in $A_1$ and $\exists q \xrightarrow{K_2} q'$ in $A_2$, such that, $k_1\sigma_1 \cap N_2\sigma_2 = k_2\sigma_2 \cap N_1\sigma_1$, and $K = (((K_1\sigma_1) \cup (K_2\sigma_2)) \setminus X)$

Furthermore, by definition of $R$, $(x,p) \in R_1$, and $(y,q) \in R_2$. Therefore, and by definition of $\mathcal{PA}(c)$:

1. $\exists x \xrightarrow[\mathsf{nofl}\oplus\ldots\oplus\mathsf{nofl}]{sr} x'$ in $T(c_1)$ and $\exists y \xrightarrow[\mathsf{nofl}\oplus\ldots\oplus\mathsf{nofl}]{\mathsf{nofl}\oplus\ldots\oplus\mathsf{nofl}} y'$ in $T(c_2)$, such that $K_1 = \{(i,\mathsf{sr}) \mid sr = v_1 \oplus \cdots \oplus v_{n_1}, i \in \{1 \ldots n_1\}, v_i = \mathsf{fl}\}$. So, $K = K_1$, because $K_1 \cap X = \emptyset$, and $\forall_{i\in\mathbb{N}}(i,\mathsf{sk}) \notin K_1$, meaning $\sigma_1$ has no effect in $K_1$.

2. $\exists y \xrightarrow[sk]{\mathsf{nofl}\oplus\ldots\oplus\mathsf{nofl}} y'$ in $T(c_2)$ and $\exists x \xrightarrow[\mathsf{nofl}\oplus\ldots\oplus\mathsf{nofl}]{\mathsf{nofl}\oplus\ldots\oplus\mathsf{nofl}} x'$ in $T(c_1)$, such that $K_1 = \{(i,\mathsf{sk}) \mid sr = v_1 \oplus \cdots \oplus v_{m_2}, i \in \{1 \ldots m_2\}, v_i = \mathsf{fl}\}$. So, $K = K_2$, because $K_2 \cap X = \emptyset$, and $\forall_{i\in\mathbb{N}}(i,\mathsf{sr}) \notin K_2$.

3. $\exists x \xrightarrow[sk]{sr} x'$ in $T(c_1)$ and $\exists y \xrightarrow[sk']{sr'} y'$ in $T(c_2)$, such that $sk = sr'$, $K_1 = \{(i,\mathsf{sr}) \mid sr = v_1 \oplus \cdots \oplus v_{n_1}, i \in \{1 \ldots n_1\}, v_i = \mathsf{fl}\} \cup \{(i,\mathsf{sk}) \mid sk = v_1 \oplus \cdots \oplus v_{n_2}, i \in \{1 \ldots n_2\}, v_i = \mathsf{fl}\}$, and $K_2 = \{(i,\mathsf{sr}) \mid sr' = v_1 \oplus \cdots \oplus v_{n_2}, i \in \{1 \ldots n_2\}, v_i = \mathsf{fl}\} \cup \{(i,\mathsf{sk}) \mid sk' = v_1 \oplus \cdots \oplus v_{m_2}, i \in \{1 \ldots m_2\}, v_i = \mathsf{fl}\}$. So, $K = ((K_1\sigma_1) \cup (K_2\sigma_2)) \setminus X = (\{(i,\mathsf{sr}) \mid sr = v_1 \oplus \cdots \oplus v_{n_1}, i \in \{1 \ldots n_1\}, v_i = \mathsf{fl}\} \cup \{(i,\mathsf{mx}) \mid sk = v_1 \oplus \cdots \oplus v_{n_2}, i \in \{1 \ldots n_2\}, v_i = \mathsf{fl}\} \cup \{(i,\mathsf{mx}) \mid sr' = v_1 \oplus \cdots \oplus v_{n_2}, i \in \{1 \ldots n_2\}, v_i = \mathsf{fl}\} \cup \{(i,\mathsf{sk}) \mid sk' = v_1 \oplus \cdots \oplus v_{m_2}, i \in \{1 \ldots m_2\}, v_i = \mathsf{fl}\}) \setminus X = \{(i,\mathsf{sr}) \mid sr = v_1 \oplus \cdots \oplus v_{n_1}, i \in \{1 \ldots n_1\}, v_i = \mathsf{fl}\} \cup \{(i,\mathsf{sk}) \mid sk' = v_1 \oplus \cdots \oplus v_{m_2}, i \in \{1 \ldots m_2\}, v_i = \mathsf{fl}\}$

By definition of bisimulation $(x',p') \in R_1$, and $(y',q') \in R_2$, and thus, by definition of $R$, $(x';y',(p',q')) \in R$.

By the composition of tiles we have:

1. $\exists x;y \xrightarrow[\mathsf{nofl}\oplus\ldots\oplus\mathsf{nofl}]{sr} x';y'$ in $T(c_1;c_2)$
2. $\exists x;y \xrightarrow[sk]{\mathsf{nofl}\oplus\ldots\oplus\mathsf{nofl}} x';y'$ in $T(c_1;c_2)$
3. $\exists x;y \xrightarrow[sk']{sr} x';y'$ in $T(c_1;c_2)$

Therefore

1. $\exists (x;y) \xrightarrow{K'} (x';y')$ in $\mathcal{PA}(c_1;c_2)$, where $K' = \{(i,\mathsf{sr}) \mid sr = v_1 \oplus \cdots \oplus v_{n_1}, i \in \{1 \ldots n_1\}, v_i = \mathsf{fl}\} = K$.
2. $\exists (x;y) \xrightarrow{K'} (x';y')$ in $\mathcal{PA}(c_1;c_2)$, where $K' = \{(i,\mathsf{sk}) \mid sr = v_1 \oplus \cdots \oplus v_{m_2}, i \in \{1 \ldots m_2\}, v_i = \mathsf{fl}\} = K$.

3. $\exists(x;y) \xrightarrow{K'} (x';y')$ in $\mathcal{PA}(c_1;c_2)$, where $K' = \{(i,\mathsf{sr}) \mid sr = v_1 \oplus \cdots \oplus v_{n_1}, i \in \{1 \ldots n_1\}, v_i = \mathsf{fl}\} \cup \{(i,\mathsf{sk}) \mid sk' = v_1 \oplus \cdots \oplus v_{m_2}, i \in \{1 \ldots m_2\}, v_i = \mathsf{fl}\} = (\{(i,\mathsf{sr}) \mid sr = v_1 \oplus \cdots \oplus v_{n_1}, i \in \{1 \ldots n_1\}, v_i = \mathsf{fl}\} \cup \{(i,\mathsf{sk}) \mid sk' = v_1 \oplus \cdots \oplus v_{m_2}, i \in \{1 \ldots m_2\}, v_i = \mathsf{fl}\}) = (\{(i,\mathsf{sr}) \mid sr = v_1 \oplus \cdots \oplus v_{n_1}, i \in \{1 \ldots n_1\}, v_i = \mathsf{fl}\} \cup \{(i,\mathsf{sk}) \mid sk = v_1 \oplus \cdots \oplus v_{n_2}, i \in \{1 \ldots n_2\}, v_i = \mathsf{fl}\})\sigma_1 \cup (( \{(i,\mathsf{sr}) \mid sr' = v_1 \oplus \cdots \oplus v_{n_2}, i \in \{1 \ldots n_2\}, v_i = \mathsf{fl}\} \cup \{(i,\mathsf{sk}) \mid sk' = v_1 \oplus \cdots \oplus v_{m_2}, i \in \{1 \ldots m_2\}, v_i = \mathsf{fl}\})\sigma_1) \setminus X = (K_1 \cup (K_2\sigma)) \setminus X)) = K$.

Therefore, $R$ is a bisimulation, and $\mathcal{PA}(c_1;c_2)$ is correct with respect to $A_1 \bowtie_; A_2$. □

**Lemma 3.4** (Correctness of $\mathcal{PA}(c_1 \oplus c_2)$)**.** If, for $i \in \{1,2\}$, $\mathcal{PA}(c_i)$ is correct with respect to $A_i = (Q_i, N_i, \to_i, q_{0,i})$, $c_i : n_i \to m_i$, and $c_1 \oplus c_2$ is well-typed, then $\mathcal{PA}(c_1 \oplus c_2)$ is correct with respect to $A_1 \bowtie_\oplus A_2$.

*Proof.* Let $A_1 = (Q_1, N_1, \to_1, q_1)$, $A_2 = (Q_2, N_2, \to_2, q_2)$, $\mathcal{PA}(c_1) = (Q_3, N_3, \to_3, q_3)$, and $\mathcal{PA}(c_2) = (Q_4, N_4, \to_4, q_4)$. Let $R_1 \subseteq Q_3 \times Q_1$, and $R_2 \subseteq Q_4 \times Q_2$ be bisimulations. We show that $\mathcal{PA}(c_1 \oplus c_2)$ is correct with respect to $A_1 \bowtie_\oplus A_2$ by showing that the relation

$$R = \{(x \oplus y, (x', y')) \mid x \oplus y \in Reach(c_1 \oplus c_2),\ (x,x') \in R_1,\ (y,y') \in R_2\}$$

is a bisimulation, i.e.:

1. $\forall_{p \in Reach(c_1 \oplus c_2)}\ p \xrightarrow{K} p' \wedge (p,q) \in R \Rightarrow \exists_{q' \in Q_1 \times Q_2}\ q \xrightarrow{K} q' \wedge (p',q') \in R$;
2. $\forall_{q \in Q_1 \times Q_2}\ q \xrightarrow{K} q' \wedge (p,q) \in R \Rightarrow \exists_{p' \in Reach(c_1 \oplus c_2)}\ p \xrightarrow{K} p' \wedge (p',q') \in R$;

**1.**

Consider a transition $(x \oplus y) \xrightarrow{K} (x' \oplus y')$ in $\mathcal{PA}(c_1 \oplus c_2)$, and let $(x \oplus y, (p,q)) \in R$, with $(p,q) \in Q_1 \times Q_2$.

Then there exists $x \oplus y \xrightarrow[sk \oplus sk']{sr \oplus sr'} x' \oplus y'$ in $T(c_1 \oplus c_2)$, such that, $(x,p), (x',p') \in R_1$, and $(y,q), (y',q') \in R_2$, and

$$\begin{aligned} K = \{(i,\mathsf{sr}) \mid\ & sr \oplus sr' = v_1 \oplus \cdots \oplus v_{n_1} \oplus v_{n_1+1} \oplus \cdots \oplus v_{n_1+n_2}, \\ & i \in \{1 \ldots n_1 + n_2\}, v_i = \mathsf{fl}\} \cup \\ \{(i,\mathsf{sk}) \mid\ & sk \oplus sk' = v_1 \oplus \cdots \oplus v_{m_1} \oplus v_{m_1+1} \oplus \cdots \oplus v_{m_1+m_2}, \\ & i \in \{1 \ldots m_1 + m_2\}, v_i = \mathsf{fl}\} \end{aligned}$$

By definition of $R$, $(x' \oplus y', (p', q')) \in R$, and by definition of composition of tiles $\exists x \xrightarrow[sk]{sr} x'$ in $T(c_1)$ and $\exists y \xrightarrow[sk]{sr} y'$ in $T(c_2)$. Therefore, by definition of $\mathcal{PA}$, there exist $p \xrightarrow{K_1} p'$ in $A_1$ and $q \xrightarrow{K_2} q'$ in $A_2$, such that, $K_1 = \{(i, \mathsf{sr}) \mid sr = v_1 \oplus \cdots \oplus v_{n_1}, i \in \{1 \ldots n_1\}, v_i = \mathsf{fl}\} \cup \{(i, \mathsf{sk}) \mid sk = v_1 \oplus \cdots \oplus v_{m_1}, i \in \{1 \ldots m_1\}, v_i = \mathsf{fl}\}$, and $K_2 = \{(i, \mathsf{sr}) \mid sr' = v_1 \oplus \cdots \oplus v_{n_2}, i \in \{1 \ldots n_2\}, v_i = \mathsf{fl}\} \cup \{(i, \mathsf{sk}) \mid sk' = v_1 \oplus \cdots \oplus v_{m_2}, i \in \{1 \ldots m_2\}, v_i = \mathsf{fl}\}$.

So, there is a transition $(p \oplus q) \xrightarrow{K'} (p' \oplus q')$ in $(A_1 \bowtie_{\oplus} A_2)$, where $K' = (K_1 \cup K_2\sigma) = \{(i, \mathsf{sr}) \mid sr = v_1 \oplus \cdots \oplus v_{n_1}, i \in \{1 \ldots n_1\}, v_i = \mathsf{fl}\} \cup \{(i, \mathsf{sk}) \mid sk = v_1 \oplus \cdots \oplus v_{m_1}, i \in \{1 \ldots m_1\}, v_i = \mathsf{fl}\} \cup (\{(i, \mathsf{sr}) \mid sr' = v_1 \oplus \cdots \oplus v_{n_2}, i \in \{1 \ldots n_2\}, v_i = \mathsf{fl}\} \cup \{(i, \mathsf{sk}) \mid sk' = v_1 \oplus \cdots \oplus v_{m_2}, i \in \{1 \ldots m_2\}, v_i = \mathsf{fl}\})\sigma = \{(i, \mathsf{sr}) \mid sr = v_1 \oplus \cdots \oplus v_{n_1}, i \in \{1 \ldots n_1\}, v_i = \mathsf{fl}\} \cup \{(i, \mathsf{sk}) \mid sk = v_1 \oplus \cdots \oplus v_{m_1}, i \in \{1 \ldots m_1\}, v_i = \mathsf{fl}\} \cup \{(i, \mathsf{sr}) \mid sr' = v_{n_1+1} \oplus \cdots \oplus v_{n_1+n_2}, i \in \{n_1 + 1 \ldots n_1 + n_2\}, v_i = \mathsf{fl}\} \cup \{(i, \mathsf{sk}) \mid sk' = v_{m_1+1} \oplus \cdots \oplus v_{m_1+m_2}, i \in \{m_1+1 \ldots m_1+m_2\}, v_i = \mathsf{fl}\} = \{(i, \mathsf{sr}) \mid sr \oplus sr' = v_1 \oplus \cdots \oplus v_{n_1} \oplus v_{n_1+1} \oplus \cdots \oplus v_{n_1+n_2}, i \in \{1 \ldots n_1 + n_2\}, v_i = \mathsf{fl}\} \cup \{(i, \mathsf{sk}) \mid sk \oplus sk' = v_1 \oplus \cdots \oplus v_{m_1} \oplus v_{m_1+1} \oplus \cdots \oplus v_{m_1+m_2}, i \in \{1 \ldots m_1 + m_2\}, v_i = \mathsf{fl}\} = K$

Therefore **1.** is valid.

**2.**

Consider a transition $(p, q) \xrightarrow{K} (p', q')$ in $A_1 \bowtie_{\oplus} A_2$, and let $(x \oplus y, (p, q)) \in R$, with $(x \oplus y) \in Reach(c_1 \oplus c_2)$.

Then one of the following conditions is true:

1. $\exists p \xrightarrow{K_1} p'$ in $A_1$, such that $K_1 \cap N_2\sigma = \emptyset$, and $K = K_1$
2. $\exists q \xrightarrow{K_2} q'$ in $A_2$, such that $K_2\sigma \cap N_1 = \emptyset$ and $K = (K_2\sigma)$
3. $\exists p \xrightarrow{K_1} p'$ in $A_1$ and $\exists q \xrightarrow{K_2} q'$ in $A_2$, such that $K_1 \cap N_2\sigma = K_2\sigma N_1$, and $K = K_1 \cup (K_2\sigma)$

By definition of $R$, $(x, p) \in R_1$, and $(y, q) \in R_2$. Therefore, and by definition of $\mathcal{PA}(c)$:

1. $\exists x \xrightarrow[sk]{sr} x'$ in $T(c_1)$ and $\exists y \xrightarrow[\mathsf{nofl}\oplus\ldots\oplus\mathsf{nofl}]{\mathsf{nofl}\oplus\ldots\oplus\mathsf{nofl}} y'$ in $T(c_2)$, such that $K_1 = \{(i, \mathsf{sr}) \mid sr = v_1 \oplus \cdots \oplus v_{n_1}, i \in \{1 \ldots n_1\}, v_i = \mathsf{fl}\} \cup \{(i, \mathsf{sk}) \mid sk = v_1 \oplus \cdots \oplus v_{m_1}, i \in \{1 \ldots m_1\}, v_i = \mathsf{fl}\}$.

2. $\exists y \xrightarrow[sk]{sr} y'$ in $T(c_2)$ and $\exists x \xrightarrow[\mathsf{nofl}\oplus\ldots\oplus\mathsf{nofl}]{\mathsf{nofl}\oplus\ldots\oplus\mathsf{nofl}} x'$ in $T(c_1)$, such that $K_2 = \{(i,\mathsf{sr}) \mid sr = v_1 \oplus \cdots \oplus v_{n_2}, i \in \{1 \ldots n_2\}, v_i = \mathsf{fl}\} \cup \{(i,\mathsf{sk}) \mid sk = v_1 \oplus \cdots \oplus v_{m_2}, i \in \{1 \ldots m_2\}, v_i = \mathsf{fl}\}$.

3. $\exists x \xrightarrow[sk]{sr} x'$ in $T(c_1)$ and $\exists y \xrightarrow[sk']{sr'} y'$ in $T(c_2)$, such that $K_1 = \{(i,\mathsf{sr}) \mid sr = v_1 \oplus \cdots \oplus v_{n_1}, i \in \{1 \ldots n_1\}, v_i = \mathsf{fl}\} \cup \{(i,\mathsf{sk}) \mid sk = v_1 \oplus \cdots \oplus v_{m_1}, i \in \{1 \ldots m_1\}, v_i = \mathsf{fl}\}$, and $K_2 = \{(i,\mathsf{sr}) \mid sr' = v_1 \oplus \cdots \oplus v_{n_2}, i \in \{1 \ldots n_2\}, v_i = \mathsf{fl}\} \cup \{(i,\mathsf{sk}) \mid sk' = v_1 \oplus \cdots \oplus v_{m_2}, i \in \{1 \ldots m_2\}, v_i = \mathsf{fl}\}$.

By definition of bisimulation $(x', p') \in R_1$, and $(y', q') \in R_2$, and therefore, $(x' \oplus y', (p', q')) \in R$.

By the composition of tiles we have:

1. $\exists x \oplus y \xrightarrow[sk\oplus\mathsf{nofl}\oplus\ldots\oplus\mathsf{nofl}]{sr\oplus\mathsf{nofl}\oplus\ldots\oplus\mathsf{nofl}} x' \oplus y'$ in $T(c_1 \oplus c_2)$
2. $\exists x \oplus y \xrightarrow[\mathsf{nofl}\oplus\ldots\oplus\mathsf{nofl}\oplus sk]{\mathsf{nofl}\oplus\ldots\oplus\mathsf{nofl}\oplus sr} x' \oplus y'$ in $T(c_1 \oplus c_2)$
3. $\exists x \oplus y \xrightarrow[sk\oplus sk']{sr\oplus sr'} x' \oplus y'$ in $T(c_1 \oplus c_2)$

Therefore

1. $\exists (x \oplus y) \xrightarrow{K'} (x' \oplus y')$ in $\mathcal{PA}(c_1 \oplus c_2)$, where $K' = \{(i,\mathsf{sr}) \mid sr \oplus \mathsf{nofl} \oplus \ldots \oplus \mathsf{nofl} = v_1 \oplus \cdots \oplus v_{n_1} \oplus v_{n_1+1} \oplus \ldots \oplus v_{n_1+n_2}, i \in \{1 \ldots n_1 + n_2\}, v_i = \mathsf{fl}\} \cup \{(i,\mathsf{sk}) \mid sk \oplus \mathsf{nofl} \oplus \ldots \oplus \mathsf{nofl} = v_1 \oplus \cdots \oplus v_{m_1} \oplus v_{m_1+1} \oplus \ldots \oplus v_{m_1+m_2}, i \in \{1 \ldots m_1 + m_2\}, v_i = \mathsf{fl}\} = \{(i,\mathsf{sr}) \mid sr = v_1 \oplus \cdots \oplus v_{n_1}, i \in \{1 \ldots n_1\}, v_i = \mathsf{fl}\} \cup \{(i,\mathsf{sk}) \mid sk = v_1 \oplus \cdots \oplus v_{m_1}, i \in \{1 \ldots m_1\}, v_i = \mathsf{fl}\} = k_1 = K$.

2. $\exists (x \oplus y) \xrightarrow{K'} (x' \oplus y')$ in $\mathcal{PA}(c_1 \oplus c_2)$, where $K' = \{(i,\mathsf{sr}) \mid \mathsf{nofl} \oplus \ldots \oplus \mathsf{nofl} \oplus sr = v_1 \oplus \cdots \oplus v_{n_1} \oplus v_{n_1+1} \oplus \ldots \oplus v_{n_1+n_2}, i \in \{1 \ldots n_1 + n_2\}, v_i = \mathsf{fl}\} \cup \{(i,\mathsf{sk}) \mid \mathsf{nofl} \oplus \ldots \oplus \mathsf{nofl} \oplus sk = v_1 \oplus \cdots \oplus v_{m_1} \oplus v_{m_1+1} \oplus \ldots \oplus v_{m_1+m_2}, i \in \{1 \ldots m_1 + m_2\}, v_i = \mathsf{fl}\} = \{(i,\mathsf{sr}) \mid sr = v_{n_1+1} \oplus \ldots \oplus v_{n_1+n_2}, i \in \{n_1+1 \ldots n_1+n_2\}, v_i = \mathsf{fl}\} \cup \{(i,\mathsf{sk}) \mid sk = v_{m_1+1} \oplus \ldots \oplus v_{m_1+m_2}, i \in \{m_1+1 \ldots m_1+m_2\}, v_i = \mathsf{fl}\} = (\{(i,\mathsf{sr}) \mid sr = v_1 \oplus \ldots \oplus v_{n_2}, i \in \{1 \ldots n_2\}, v_i = \mathsf{fl}\} \cup \{(i,\mathsf{sk}) \mid sk = v_1 \oplus \ldots \oplus v_{m_2}, i \in \{1 \ldots m_2\}, v_i = \mathsf{fl}\})\sigma = K_2\sigma = K$.

3. $\exists(x \oplus y) \xrightarrow{K'} (x' \oplus y')$ in $\mathcal{PA}(c_1 \oplus c_2)$, where $K' = \{(i, \mathsf{sr}) \mid sr \oplus sr' = v_1 \oplus \cdots \oplus v_{n_1} \oplus v_{n_1+1} \oplus \cdots \oplus v_{n_1+n_2}, i \in \{1 \ldots n_1 + n_2\}, v_i = \mathsf{fl}\} \cup \{(i, \mathsf{sk}) \mid sk \oplus sk' = v_1 \oplus \cdots \oplus v_{m_1} \oplus v_{m_1+1} \oplus \cdots \oplus v_{m_1+m_2}, i \in \{1 \ldots m_1 + m_2\}, v_i = \mathsf{fl}\} = \{(i, \mathsf{sr}) \mid sr = v_1 \oplus \cdots \oplus v_{n_1}, i \in \{1 \ldots n_1\}, v_i = \mathsf{fl}\} \cup \{(i, \mathsf{sk}) \mid sk = v_1 \oplus \cdots \oplus v_{m_1}, i \in \{1 \ldots m_1\}, v_i = \mathsf{fl}\} \cup \{(i, \mathsf{sr}) \mid sr' = v_{n_1+1} \oplus \cdots \oplus v_{n_1+n_2}, i \in \{n_1 + 1 \ldots n_1 + n_2\}, v_i = \mathsf{fl}\} \cup \{(i, \mathsf{sk}) \mid sk' = v_{m_1+1} \oplus \cdots \oplus v_{m_1+m_2}, i \in \{m_1 + 1 \ldots m_1 + m_2\}, v_i = \mathsf{fl}\} = \{(i, \mathsf{sr}) \mid sr = v_1 \oplus \cdots \oplus v_{n_1}, i \in \{1 \ldots n_1\}, v_i = \mathsf{fl}\} \cup \{(i, \mathsf{sk}) \mid sk = v_1 \oplus \cdots \oplus v_{m_1}, i \in \{1 \ldots m_1\}, v_i = \mathsf{fl}\} \cup (\{(i, \mathsf{sr}) \mid sr' = v_1 \oplus \cdots \oplus v_{n_2}, i \in \{1 \ldots n_2\}, v_i = \mathsf{fl}\} \cup \{(i, \mathsf{sk}) \mid sk' = v_1 \oplus \cdots \oplus v_{m_2}, i \in \{1 \ldots m_2\}, v_i = \mathsf{fl}\})\sigma = K_1 \cup K_2\sigma = K$.

Therefore, $R$ is a bisimulation, and $\mathcal{PA}(c_1 \oplus c_2)$ is correct with respect to $A_1 \bowtie_\oplus A_2$.

□

**Lemma 3.5** (Correctness of $\mathcal{PA}(\mathsf{Tr}_n(c))$)**.** If $\mathcal{PA}(c)$ is correct with respect to $A$, where $c : m_1 \to m_2$, and $\mathsf{Tr}_n(c)$ is well-typed, then $\mathcal{PA}(\mathsf{Tr}_n(c))$ is correct with respect to $Tr_n(A)$.

*Proof.* Let $A = (Q_1, N_1, \to_1, q_1)$, and $\mathcal{PA}(c) = (Q_2, N_2, \to_2, q_2)$. Let $R_1 \subseteq Q_2 \times Q_1$ be a bisimulation. We show that $\mathcal{PA}(\mathsf{Tr}_n(c))$ is correct with respect to $Tr_n(A)$ by showing that the relation

$$R = \{(\mathsf{Tr}_n(x), x') \mid x \in Reach(\mathsf{Tr}_n(c)),\ (x, x') \in R_1\}$$

is a bisimulation, i.e.:

1. $\forall_{p \in Reach(\mathsf{Tr}_n(c))}\ p \xrightarrow{K} p' \wedge (p, q) \in R \Rightarrow \exists_{q' \in Q_1}\ q \xrightarrow{K} q' \wedge (p', q') \in R$;
2. $\forall_{q \in Q_1}\ q \xrightarrow{K} q' \wedge (p, q) \in R \Rightarrow \exists_{p' \in Reach(\mathsf{Tr}_n(c))}\ p \xrightarrow{K} p' \wedge (p', q') \in R$;

**1.**

Consider a transition $\mathsf{Tr}_n(x) \xrightarrow{K} \mathsf{Tr}_n(x')$ in $\mathcal{PA}(\mathsf{Tr}_n(c))$, and let $(\mathsf{Tr}_n(x), p) \in R$, with $p \in Q_1$.

Then there exists $\mathsf{Tr}_n(x) \xrightarrow[sk]{sr} \mathsf{Tr}_n(x')$ in $T(\mathsf{Tr}_n(c))$, such that $K = \{(i, \mathsf{sr}) \mid sr = v_1 \oplus \cdots \oplus v_{m_1-n}, i \in \{1 \ldots m_1 - n\}, v_i = \mathsf{fl}\} \cup \{(i, \mathsf{sk}) \mid sk = v_1 \oplus \cdots \oplus v_{m_2-n}, i \in \{1 \ldots m_2 - n\}, v_i = \mathsf{fl}\}$, and $(x, p), (x', p') \in R_1$

By definition of $R$, $(\mathsf{Tr}_n(x'), p') \in R$, and by definition of trace composition there exists $x \xrightarrow[sk \oplus v]{sr \oplus v} x'$ in $T(c)$, and, therefore, $\exists p \xrightarrow{K_1} p'$ in $A$, such that, $K_1 = \{(i, \mathsf{sr}) \mid sr \oplus v = v_1 \oplus \cdots \oplus v_{m_1}, i \in \{1 \ldots m_1\}, v_i = \mathsf{fl}\} \cup \{(i, \mathsf{sk}) \mid sk \oplus v = v_1 \oplus \cdots \oplus v_{m_2}, i \in \{1 \ldots m_2\}, v_i = \mathsf{fl}\}$.

So, there is a transition $p \xrightarrow{K'} p'$ in $Tr_n(A)$, where $K' = K_1 \backslash X = (\{(i, \mathsf{sr}) \mid sr \oplus v = v_1 \oplus \cdots \oplus v_{m_1}, i \in \{1 \ldots m_1\}, v_i = \mathsf{fl}\} \cup \{(i, \mathsf{sk}) \mid sk \oplus v = v_1 \oplus \cdots \oplus v_{m_2}, i \in \{1 \ldots m_2\}, v_i = \mathsf{fl}\}) \backslash X = \{(i, \mathsf{sr}) \mid sr = v_1 \oplus \cdots \oplus v_{m_1}, i \in \{1 \ldots m_1\}, v_i = \mathsf{fl}\} \cup \{(i, \mathsf{sk}) \mid sk = v_1 \oplus \cdots \oplus v_{m_2}, i \in \{1 \ldots m_2\}, v_i = \mathsf{fl}\} = K$

Therefore **1.** is valid.

**2.**

Consider a transition $p \xrightarrow{K} p'$ in $Tr_n(A)$, and let $(x, p) \in R$, with $x \in Reach(\mathsf{Tr}_n(c))$.

Then there exists $p \xrightarrow{K_1} p'$ in $A$, such that $K = K_1 \backslash X$, and for all $i \in \{0 \ldots n-1\}$, $(m_1 - i, \mathsf{sr}) \in K_1 \Leftrightarrow (m_2 - i, \mathsf{sk}) \in K_1$.

By definition of bisimulation, and $\mathcal{PA}(c)$, $\exists y \xrightarrow[sk \oplus v]{sr \oplus v} y' \in T(c)$, such that $K_1 = \{(i, \mathsf{sr}) \mid sr \oplus v = v_1 \oplus \cdots \oplus v_{m_1}, i \in \{1 \ldots m_1\}, v_i = \mathsf{fl}\} \cup \{(i, \mathsf{sk}) \mid sk \oplus v = v_1 \oplus \cdots \oplus v_{m_2}, i \in \{1 \ldots m_2\}, v_i = \mathsf{fl}\}$, with $(y, p), (y', p') \in R_1$. Then, by definition of the trace operator, $\exists x \xrightarrow[sk]{sr} x' \in T(\mathsf{Tr}_n(c))$, such that $x = \mathsf{Tr}_n(y)$, and $x' = \mathsf{Tr}_n(y')$, and, by definition of $R$, $(x', p') \in R$.

By $\mathcal{PA}(\mathsf{Tr}_n(c))$, $\exists x \xrightarrow{K'} x'$ in $\mathcal{PA}(\mathsf{Tr}_n(c)$, where $K' = \{(i, \mathsf{sr}) \mid sr = v_1 \oplus \cdots \oplus v_{m_1 - n}, i \in \{1 \ldots m_1 - n\}, v_i = \mathsf{fl}\} \cup \{(i, \mathsf{sk}) \mid sk = v_1 \oplus \cdots \oplus v_{m_2 - n}, i \in \{1 \ldots m_1 - n\}, v_i = \mathsf{fl}\} = (\{(i, \mathsf{sr}) \mid sr \oplus v = v_1 \oplus \cdots \oplus v_{m_1}, i \in \{1 \ldots m_1\}, v_i = \mathsf{fl}\} \cup \{(i, \mathsf{sk}) \mid sk = v_1 \oplus \cdots \oplus v_{m_2}, i \in \{1 \ldots m_2\}, v_i = \mathsf{fl}\}) \backslash X = K$.

Therefore, $R$ is a bisimulation, and $\mathcal{PA}(\mathsf{Tr}_n(c))$ is correct with respect to $Tr_n(A)$.

□

**Theorem 3.1** (Correctness of $\mathcal{PA}$)**.** Given a well-typed connector $c$, $\mathcal{PA}(c)$ is correct with respect to some port automaton $A$.

*Proof.* This result follows by induction on the structure of connectors, whereas the base case is captured by Lemma 3.2, and the inductive steps are captured by Lemmas 3.3 to 3.5. □

| | | | |
|---|---|---|---|
| $\mathsf{id}_1$ | $Id1 = a\|b \ . \ Id1$ | lossy | $Lossy = (a + a\|b) \ . \ Lossy$ |
| fifo | $Fifo = a \ . \ b \ . \ Fifo$ | $\Delta_2$ | $Dupl = a\|b\|c \ . \ Dupl$ |
| drain | $Drain = a\|b \ . \ Drain$ | $\nabla_2$ | $Merger = (a\|c + b\|c) \ . \ Merger$ |

Table 3.1: mCRL2 processes of primitives, for some actions $a, b, c$.

## 3.4 Modeling Connector Calculus with mCRL2

We now present our encoding of the core connector calculus in mCRL2. Our encoding follows the work done by Kokash et al. [7], presented in Section 2.3. Furthermore, we base our work on the constraint automata semantics, ignoring the data constraints (port automata). Following the encoding definition we present a proof of its correctness.

### 3.4.1 Core Calculus into mCRL2

In Table 3.1 we present the mCRL2 processes of individual channels. These processes are equivalent to the ones presented for the channels of Reo in Table 2.8, with the exception of $\Delta_2$, and $\nabla_2$, which have no correspondent Reo channels. We compose these primitives with the parallel composition and synchronisation of actions, as defined in the following example:

**Example 3.8.** Consider the connector $c = \mathsf{id}_1; \Delta_2; (\mathsf{fifo} \oplus \mathsf{lossy})$. The channels in the connector maps to the following processes:

$$Id_1 = (a|b) \ . \ Id_1 \qquad Fifo = f \ . \ g \ . \ Fifo$$
$$Dupl = c|d|e \ . \ Dupl \qquad Lossy = (h + h|i) \ . \ Lossy$$

Let $\pi_c$ be the set of definitions above. A program for $c$ can be built by placing these definitions in parallel, by imposing communication with $\Gamma$, and by encapsulating internal ports with $\partial$. For example, the program $(P_c, \pi_c)$, with $P_c$ defined below, provides a (naive) encoding of the behaviour of $c$, which only exposes the ports $a$, $g$, and $i$.

$$P_c = \partial_{\{b,c,d,e,f,h\}} (\Gamma_{\{b|c \to bc, d|f \to df, e|h \to eh\}}(Sync \parallel Dupl \parallel Fifo \parallel Lossy))$$

Similar to the case in Example 2.6, applying the block and communication operations as a last step proves to be inefficient, due to state space explosion. Therefore we present an incremental encoding - $\mathcal{MC}$- for a connector. The encoding $\mathcal{MC}$ follows a similar approach to $\mathcal{PA}$, where actions follow the pattern $(n, \mathsf{sr})_\ell$, $(n, \mathsf{sk})_\ell$, or $(n, \mathsf{mx})_\ell$ to indicate the $n$-th source, sink, or mixed port, marked by the identifier unique $\ell$ to distinguish between actions from different processes. The result of this encoding is a pair $(P, \pi)$, where $P$ is the name of a process, the initial process of the encoding, and $\pi$ is a mapping from process names to process definitions, such that the definition of $P$ is defined in $\pi$.

Our encoding abstains from using processes to encode nodes (which are a fundamental aspect of the encoding for Reo in [7]). These are useful in a Reo context, but since our calculus contains only 1 to 1 port connections, nodes can be ignored. A positive side effect is a more efficient encoding.

We start by defining auxiliary functions $\mathsf{Primitive}$ to map primitives into processes, $\mathsf{Hide}$, $\mathsf{Block}$, and $\mathsf{Com}$, used to describe ports that are hidden, are blocked, and communicate, and $\mathsf{Renumber}$ to renumber ports in the tensor composition. The function $\mathsf{Primitive}(p, \ell)$ defines the process of primitive $p$ according to Table 3.1, using the proposed notation for actions marked by $\ell$. We present the case for the $\gamma_{n,m}$ primitive (which is not defined in Table 3.1) with the example $\gamma_{1,1}$, whose process is

$$\mathsf{Primitive}(\gamma_{1,1}, \ell) \;=\; (1, \mathsf{sr})_\ell | (2, \mathsf{sk})_\ell \;||\; (2, \mathsf{sr})_\ell | (1, \mathsf{sk})_\ell,$$

exposing a swap of output ports, according to the behaviour of the primitive.

The auxiliary functions are defined below:

$$\mathsf{Block}(n, \ell_1, \ell_2) = \bigcup_{1 \le i \le n} \{(i, \mathsf{sk})_{\ell_1}, (i, \mathsf{sr})_{\ell_2}\}$$

$$\mathsf{Block}_{\mathsf{Tr}}(n, m, m', \ell_1) = \bigcup_{0 \le i \le n-1} \{(m - i, \mathsf{sr})_{\ell_1}, (m' - i, \mathsf{sk})_{\ell_1}\}$$

$$\mathsf{Com}(n, \ell_1, \ell_2, \ell) = \bigcup_{1 \le i \le n} \{(i, \mathsf{sk})_{\ell_1} | (i, \mathsf{sr})_{\ell_2} \to (i, \mathsf{mx})_\ell\}$$

$$\mathsf{Com}_{\mathsf{Tr}}(n,m,m',\ell_1) = \bigcup_{0\le i\le n-1} \{(m'-i,\mathsf{sk})_{\ell_1} | (m-i,\mathsf{sr})_{\ell_1} \to (i+1,\mathsf{mx})_{\ell_1}\}$$

$$\mathsf{Hide}(n,\ell_1) = \bigcup_{1\le i\le n} \{(i,\mathsf{mx})_{\ell_1}\}$$

$$\mathsf{Rename}(n,m,\ell_1,\ell_2,\ell) = \bigcup_{1\le i\le n} \{(i,\mathsf{sr})_{\ell_1} \to (i,\mathsf{sr})_{\ell}\} \bigcup_{1\le i\le m} \{(i,\mathsf{sk})_{\ell_2} \to (i,\mathsf{sk})_{\ell}\}$$

$$\mathsf{Renumber}(n_1,m_1,\ell,n_2,m_2) = \bigcup_{1\le i\le n_1} \{(i,\mathsf{sr})_{\ell} \to (i+n_2,\mathsf{sr})_{\ell}\} \bigcup_{1\le i\le m_1} \{(i,\mathsf{sk})_{\ell} \to (i+m_2,\mathsf{sk})_{\ell}\}$$

**Definition 3.8.** Encoding $\mathcal{MC}(\cdot)_\ell$

Given a connector $c$ and a unique identifier $\ell$, $\mathcal{MC}(c)_\ell$ is defined below.

$$\mathcal{MC}(p)_\ell = (P_\ell, \{P_\ell = \mathsf{Primitive}(p,\ell)\})$$

where $p$ is a primitive connector

$$\mathcal{MC}(c_1;c_2)_\ell = (P_\ell \ , \ \big\{P_\ell = \rho_{\mathsf{Rename}(n_1,n_2,\ell_1,\ell_2,\ell)}(\tau_{\mathsf{Hide}(n,\ell)}(\partial_{\mathsf{Block}(n,\ell_1,\ell_2)} (\Gamma_{\mathsf{Com}(n,\ell_1,\ell_2,\ell)}(P_1 \parallel P_2))))\big\} \cup \pi_1 \cup \pi_2)$$

where $c_1 : n_1 \to n \quad c_2 : n \to n_2$

$$(P_1,\pi_1) = \mathcal{MC}(c_1)_{\ell_1} \quad (\ell_1 \text{ is fresh})$$

$$(P_2,\pi_2) = \mathcal{MC}(c_2)_{\ell_2} \quad (\ell_2 \text{ is fresh})$$

$$\mathcal{MC}(c_1 \oplus c_2)_\ell = (P_\ell \ , \ \big\{P_\ell = \rho_{\mathsf{Rename}(n_1,m_1,\ell_1,\ell_1,\ell)}(\rho_{\mathsf{Rename}(n_2,m_2,\ell_2,\ell_2,\ell)} (P_1 \parallel \rho_{\mathsf{Renumber}(n_2,m_2,\ell_2,n_1,m_1)}(P_2)))\big\} \cup \pi_1 \cup \pi_2)$$

where $c_1 : n_1 \to m_1 \quad c_2 : n_2 \to m_2$

$$(P_1,\pi_1) = \mathcal{MC}(c_1)_{\ell_1} \quad (\ell_1 \text{ is fresh})$$

$$(P_2,\pi_2) = \mathcal{MC}(c_2)_{\ell_2} \quad (\ell_2 \text{ is fresh})$$

$$\mathcal{MC}(\mathsf{Tr}_n(c))_\ell = (P_\ell \ , \ \big\{P_\ell = \tau_{\mathsf{Hide}(n,\ell)}(\partial_{\mathsf{Block}_{\mathsf{Tr}}(n,m,m',\ell)}(\Gamma_{\mathsf{Com}_{\mathsf{Tr}}(n,m,m',\ell)}(P)))\big\} \cup \pi)$$

where $c : m \to m', n \le m \wedge n \le m'$

$$(P,\pi) = \mathcal{MC}(c)_\ell$$

We illustrate this encoding using the connector of Example 3.8.

**Example 3.9.** Let $x = \mathsf{id}_1; \Delta_2; (\mathsf{fifo} \oplus \mathsf{lossy})$ and $a, b, c, d, e$ be unique identifier; then:

$$
\begin{aligned}
&\mathcal{MC}(\mathsf{fifo} \oplus \mathsf{lossy})_a = (P_a, \pi_a) \\
&\quad \pi_a = \{P_a = \mathit{Fifo}_a \parallel \mathit{Lossy}_a, \\
&\qquad \mathit{Fifo}_a = (1,\mathsf{sr})_a|(1,\mathsf{sk})_a \ . \ \mathit{Fifo}_a, \\
&\qquad \mathit{Lossy}_a = ((2,\mathsf{sr})_a + (2,\mathsf{sr})_a|(2,\mathsf{sk})_a) \ . \ \mathit{Lossy}_a\} \\
&\mathcal{MC}(\Delta_2; (\mathsf{fifo} \oplus \mathsf{lossy}))_b = (P_b, \pi_b) \\
&\quad \pi_b = \{P_b = \rho_{\{(1,\mathsf{sr})_c \to (1,\mathsf{sr})_b, (1,\mathsf{sk})_a \to (1,\mathsf{sk})_b, (2,\mathsf{sk})_a \to (2,\mathsf{sk})_b\}} \\
&\qquad (\tau_{\{(1,\mathsf{mx})_b, (2,\mathsf{mx})_b\}}(\partial_{\{(1,\mathsf{sk})_c, (1,\mathsf{sr})_a, (2,\mathsf{sk})_c, (2,\mathsf{sr})_a\}} \\
&\qquad (\Gamma_{\{(1,\mathsf{sk})_c|(1,\mathsf{sr})_a \to (1,\mathsf{mx})_b, (2,\mathsf{sk})_c|(2,\mathsf{sr})_a \to (2,\mathsf{mx})_b)\}}(\mathit{Dupl}_c \parallel P_a)))), \\
&\quad \mathit{Dupl}_c = (1,\mathsf{sr})_c|(1,\mathsf{sk})_c|(2,\mathsf{sk})_c \ . \ \mathit{Dupl}_c\} \cup \pi_a \\
&\mathcal{MC}(x)_e = (P_e, \pi_e) \\
&\quad \pi_e = \{P_e = \rho_{\{(1,\mathsf{sr})_d \to (1,\mathsf{sr})_e, (1,\mathsf{sk})_b \to (1,\mathsf{sk})_e, (2,\mathsf{sk})_b \to (2,\mathsf{sk})_e\}} \\
&\qquad (\tau_{\{(1,\mathsf{mx})_e\}}(\partial_{\{(1,\mathsf{sk})_d, (1,\mathsf{sr})_b\}}(\Gamma_{\{(1,\mathsf{sk})_d|(1,\mathsf{sr})_b \to (1,\mathsf{mx})_e\}}(\mathit{Id}_d \parallel P_b)))), \\
&\qquad \mathit{Id}_d = (1,\mathsf{sr})_d|(1,\mathsf{sk})_d \ . \ \mathit{Id}_d\} \cup \pi_b
\end{aligned}
$$

### 3.4.2 Correctness of $\mathcal{MC}_\ell(\cdot)$

We now want to prove the correctness of the encoding $\mathcal{MC}$. We begin the proof by showing that the processes of primitives in Table 3.1 are the result of applying *proc* to the PA in Fig. 3.4. Furthermore, we show that *proc* is sound with respect to the operations:

$$
\begin{aligned}
P_1 \|_; P_2 &= \rho_{\mathsf{Rename}(n_1,n_2,\ell_1,\ell_2,\ell)}(\tau_{\mathsf{Hide}(n,\ell)}(\partial_{\mathsf{Block}(n,\ell_1,\ell_2)}(\Gamma_{\mathsf{Com}(n,\ell_1,\ell_2,\ell)}(P_1 \parallel P_2)))) \\
P_1 \|_\oplus P_2 &= \rho_{\mathsf{Rename}(n_1,m_1,\ell_1,\ell_1,\ell)}(\rho_{\mathsf{Rename}(n_2,m_2,\ell_2,\ell_2,\ell)}(P_1 \parallel P_2))) \\
Tr_n(P) &= \tau_{\mathsf{Hide}(n,\ell)}(\partial_{\mathsf{Block}_{\mathsf{Tr}}(n,m,m',\ell)}(\Gamma_{\mathsf{Com}_{\mathsf{Tr}}(n,m,m',\ell)}(P)))
\end{aligned}
$$

where $P_1, P_2, P$ are processes, defined by *proc*. For simplicity, we assume that *proc* is only applied to port automata formed by the $\mathcal{PA}_\ell$ encoding, which is a variation of the $\mathcal{PA}$ encoding, with ports labelled with the unique mark $\ell$, similar to the $\mathcal{MC}$ encoding actions.

**Lemma 3.6** (Correctness of Primitive Encodings)**.** For any primitive $p$ from Fig. 3.4, with the respective automata $A = (Q, \mathcal{N}, \rightarrow, q_0)$, and corresponding process $P$ from Table 3.1, $P \cong proc(A, q_0)$.

*Proof.* We will only show that this lemma holds for one of the connectors, the fifo, because the other connectors can be shown in a similar way. Recall the automaton $A$ for the fifo defined in Section 3.3.1, adapted with the $\ell$ label. The complete specification $proc(A, \mathsf{fifo})$ is defined below:

- $proc(A, \mathsf{fifo}) = (1, \mathsf{sr})_\ell.proc(A, \mathsf{fifofull})$
- $proc(A, \mathsf{fifofull}) = (1, \mathsf{sk})_\ell.proc(A, \mathsf{fifo})$

Recall also the process $Fifo = a \ . \ b \ . \ Fifo$ of the fifo in Table 3.1. It is enough to observe that $R = \{\langle proc(A, \mathsf{fifo}), a \ . \ b \ . \ Fifo\rangle, \langle proc(A, \mathsf{fifofull}), b \ . \ Fifo\rangle\}$ is a strong bisimulation between the two processes, after replacing $a$ by $(1, \mathsf{sr})_\ell$ and $b$ by $(1, \mathsf{sk})_\ell$. □

**Lemma 3.7** (Correctness of $\|_;$)**.** Consider two connectors $c_1 : n_1 \rightarrow n$, $c_2 : n \rightarrow n_2$. Let $\mathcal{A}_1 = \mathcal{PA}_{\ell_1}(c_1)$, and $\mathcal{A}_2 = \mathcal{PA}_{\ell_2}(c_2)$. $proc((\mathcal{A}_1 \bowtie_; \mathcal{A}_2\sigma, (q_1, q_2)) \cong proc(\mathcal{A}_1, q_1) \ \|_; \ proc(\mathcal{A}_2, q_2)$, where $q_1, q_2$ are the initial states of the respective automata, and $\sigma$ is the set of port renamings $\{(i, sr)_{\ell_1} \rightarrow (i, sr)_\ell \mid 1 \leq i \leq n_1\} \cup \{(i, sk)_{\ell_2} \rightarrow (i, sk)_\ell \mid 1 \leq i \leq n_2\}$.

*Proof.* We simply note that for PA $A_1, A_2$, $A_1 \bowtie_; A_2 = (A_1 \bowtie_\gamma A_2) \setminus X$, where $X$ is the hiding set, where $\gamma$ is the renaming function in [7], which we define to be such that $\gamma_1^{-1} = \sigma_1$, and $\gamma_2^{-1} = \sigma_2$, where $\sigma_1$, $\sigma_2$ are defined in the definition of $\bowtie_;$. Also, given two processes $P_1, P_2$, we have that $P_1 \ \|_; \ P_2 = \rho_{\mathsf{Rename}(n_1,n_2,\ell_1,\ell_2,\ell)}(\tau_{\mathsf{Hide}(m,\ell)}(P_1 \ \|_\gamma \ P_2)))$.

In theorem 4.1 of [7], we are presented with the correctness of $\|_\gamma$. Furthermore, the set $X$ used in the $\bowtie_;$ definition hides a set of ports, with an equivalent set of actions hidden by the Hide set. The same idea applies to the renaming sets Rename and $\sigma$. Therefore the lemma is correct. □

**Lemma 3.8** (Correctness of $\|_\oplus$)**.** Consider two connectors $c_1 : n_1 \rightarrow m_1$, $c_2 : n_2 \rightarrow m_2$. Let $\mathcal{A}_1 = \mathcal{PA}_{\ell_1}(c_1)$, and $\mathcal{A}_2 = \mathcal{PA}_{\ell_2}(c_2)$. $proc((\mathcal{A}_1 \bowtie_\oplus$

$\mathcal{A}_2)\sigma, (q_1, q_2)) \cong proc(\mathcal{A}_1, q_1) \parallel_\oplus proc(\mathcal{A}_2, q_2)$, where $q_1, q_2$ are the initial states of the respective automata, and $\sigma$ is the set of port renamings such that

$$\sigma = \{(i, sr)_{\ell_1} \to (i, sr)_\ell \mid 1 \leq i \leq n_1\} \cup \{(i, sk)_{\ell_1} \to (i, sk)_\ell \mid 1 \leq i \leq m_1\} \cup$$
$$\{(i, sr)_{\ell_2} \to (i, sr)_\ell \mid 1 \leq i \leq n_2\} \cup \{(i, sk)_{\ell_2} \to (i, sk)_\ell \mid 1 \leq i \leq m_2\}$$

*Proof.* The proof follows the same steps as the proof for correctness of $\|_;$, but taking an empty function $\gamma$. □

**Lemma 3.9** (Correctness of $Tr_n(P)$)**.** Consider a connector $c : m_1 \to m_2$, with $n \in \mathbb{N}$, such that $n \leq m_1$ and $n \leq m_2$. $proc(Tr_n(\mathcal{PA}_\ell(c)), (q_1)) \cong Tr_n(proc(\mathcal{PA}_\ell(c), q_1))$, where $q_1$ is the initial state $\mathcal{PA}_\ell(c)$.

*Proof.* Let $\mathcal{PA}_\ell(c) = (\mathcal{Q}, N, \to, q_1)$. The proof is done by showing that the relation $R = \{\langle proc(Tr_n(\mathcal{PA}_\ell(c)),\ q_i),\ Tr_n(proc(\mathcal{PA}_\ell(c),\ q_i))\rangle \mid q_i \in \mathcal{Q}\}$ is a bisimulation, i.e.,

1. for all $q_i \in \mathcal{Q}$, if $proc(Tr_n(\mathcal{PA}_\ell(c)),\ q_i) \xrightarrow{\alpha} P$, then $Tr_n(proc(\mathcal{PA}_\ell(c),\ q_i)) \xrightarrow{\alpha} Q$ and $(P, Q) \in R$;
2. for all $q_i \in \mathcal{Q}$, if $Tr_n(proc(\mathcal{PA}_\ell(c),\ q_i)) \xrightarrow{\alpha} Q$, then $proc(Tr_n(\mathcal{PA}_\ell(c)),\ q_i) \xrightarrow{\alpha} P$ and $(P, Q) \in R$.

**1.**

Let $q_i \in \mathcal{Q}$, such that $proc(Tr_n(\mathcal{PA}_\ell(c)),\ q_i) \xrightarrow{\alpha} P$. By definition of $R$, $(proc(Tr_n(\mathcal{PA}_\ell(c)),\ q_i), Tr_n(proc(\mathcal{PA}_\ell(c),\ q_i)) \in R$. By definition of $proc$, there exist $K \subseteq N$ and $q_j \in \mathcal{Q}$, such that $q_i \xrightarrow{K}_{\mathsf{Tr}} q_j$ is a transition in $Tr_n(\mathcal{PA}_\ell(c))$, where $K = \alpha_{\{\}}$. Therefore, $P = proc(Tr_n(\mathcal{PA}_\ell(c)),\ q_j)$.

By definition of $Tr_n(A)$, $q_i \xrightarrow{K'} q_j$ is a transition in $\mathcal{PA}_\ell(c)$, such that $K = K' \backslash X$, and for all $i \in \{0 \ldots n-1\}$, $(m_1 - i, \mathsf{sr}) \in K'$ if and only if $(m_2 - i, \mathsf{sk}) \in K'$, with $X = \{(m_1 - i, \mathsf{sr}) \mid 0 \leq i \leq n-1\} \cup \{(m_2 - i, \mathsf{sk}) \mid 0 \leq i \leq n-1\}$.

By definition of $proc$, $proc(\mathcal{PA}_\ell(c),\ q_i) \xrightarrow{\alpha'} proc(\mathcal{PA}_\ell(c),\ q_j)$, where $\alpha'_{\{\}} = K'$. By definition, $Tr_n(proc(\mathcal{PA}_\ell(c),\ q_i)) = \tau_H(\partial_B(\Gamma_C(proc(\mathcal{PA}_\ell(c),\ q_i))))$,

where:

$$H = \mathsf{Hide}(n, \ell) = \bigcup_{1 \leq i \leq n} \{(i, \mathsf{mx})_\ell\}$$

$$B = \mathsf{Block}_{\mathsf{Tr}}(n, m_1, m_2, \ell) = \bigcup_{0 \leq i \leq n-1} \{(m_1 - i, \mathsf{sr})_\ell, (m_2 - i, \mathsf{sk})_\ell\}$$

$$C = \mathsf{Com}_{\mathsf{Tr}}(n, m_1, m_2, \ell) = \bigcup_{0 \leq i \leq n-1} \{(m_2 - i, \mathsf{sk})_\ell | (m_1 - i, \mathsf{sr})_\ell \rightarrow (i+1, \mathsf{mx})_\ell\}$$

Therefore, $Tr_n(proc(\mathcal{PA}_\ell(c),\ q_i)) \xrightarrow{\alpha''} Tr_n(proc(\mathcal{PA}_\ell(c),\ q_j))$,if $\gamma_C(\alpha')_{\{\}} \cap B = \emptyset$, where $\alpha'' = \theta_H(\gamma_C(\alpha'))$. This is true because for all $i \in \{0 \ldots n-1\}$, $(m_1 - i, \mathsf{sr}) \in K'$ if and only if $(m_2 - i, \mathsf{sk}) \in K'$. Also, because of this property, $C$ can be equivalent to $\{(m_1 - i, \mathsf{sr}) \rightarrow (i+1, \mathsf{mx})_\ell \mid 0 \leq i \leq n-1\} \cup \{(m_2 - i, \mathsf{sk}) \rightarrow (i+1, \mathsf{mx})_\ell \mid 0 \leq i \leq n-1\}$. Thus, $\theta_H(\gamma_C(\alpha'))$ hides the actions equivalent to the ports in $X$, and therefore produces the same outcome as $\theta_X(\alpha')$. So, $\alpha'' = \alpha$. Furthermore, by definition of $R$, $(proc(Tr_n(\mathcal{PA}_\ell(c)), (q_j)), Tr_n(proc(\mathcal{PA}_\ell(c), q_j))) \in R$.

**2.**

Let $q_i \in \mathcal{Q}$, such that $Tr_n(proc(\mathcal{PA}_\ell(c), (q_i))) \xrightarrow{\alpha} Q$. By definition of $R$, $(proc(Tr_n(\mathcal{PA}_\ell(c)), (q_i)), Tr_n(proc(\mathcal{PA}_\ell(c), q_i))) \in R$. By definition of $Tr_n(P)$, and of process algebras, $proc(\mathcal{PA}_\ell(c), (q_i)) \xrightarrow{\alpha'} Q'$, where $Q = Tr_n(Q')$, $\alpha = \theta_H(\gamma_C(\alpha'))$, and $\gamma_C(\alpha') \cap B = \emptyset$, with $H$, $C$, $B$ defined before in this proof.

By definition of $proc$, $q_i \xrightarrow{K} q_j$ is a transition of $\mathcal{PA}_\ell(c)$, where $K = \alpha'_{\{\}}$. Then, $Q' = proc(\mathcal{PA}_\ell(c), (q_j))$. Because $\gamma_C(\alpha')_{\{\}} \cap B = \emptyset$, we know that for all $i \in \{0 \ldots n-1\}$, $(m_1 - i, \mathsf{sr}) \in K$ if and only if $(m_2 - i, \mathsf{sk}) \in K$. Therefore, and by definition of $Tr_n$ for port automata $q_i \xrightarrow{K'} q_j$ is a transition in $Tr_n(\mathcal{PA}_\ell(c))$, where $K' = K \backslash X$,for $X = \{(m_1 - i, \mathsf{sr}) \mid 0 \leq i \leq n-1\} \cup \{(m_2 - i, \mathsf{sk}) \mid 0 \leq i \leq n-1\}$.

By definition of $proc$, $proc(Tr_n(\mathcal{PA}_\ell(c)), q_i) \xrightarrow{\alpha''} proc(Tr_n(\mathcal{PA}_\ell(c)), q_j)$, with $\alpha''_{\{\}} = K' = K \backslash X$. Since for all $i \in \{0 \ldots n-1\}$ $(m_1 - i, \mathsf{sr}) \in K$ if and only if $(m_2 - i, \mathsf{sk}) \in K$, $K \backslash X$ produces the same output as $\theta_H(\gamma_C(\alpha'))_{\{\}}$. Therefore, $\alpha = \alpha''$. Furthemore, $(proc(Tr_n(\mathcal{PA}_\ell(c)), (q_j)), Tr_n(proc(\mathcal{PA}_\ell(c), q_j))) \in R$, by definition of $R$.

So, $R$ is a bisimulation, and $proc(Tr_n(\mathcal{PA}_\ell(c)), (q_1)) \cong Tr_n(proc(\mathcal{PA}_\ell(c), q_1))$, where $q_1$.

□

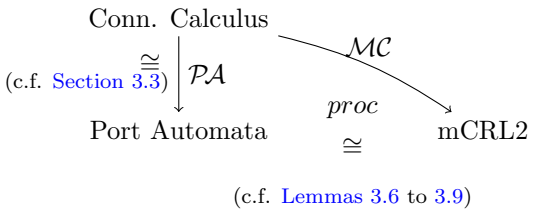



Figure 3.5: Relation between the Connector Calculus, PA, and mCRL2.

In Fig. 3.5 we present a diagram of the encodings presented throughout this document. After proving that the $\mathcal{PA}$ encoding and *proc* function are sound with respect to the algebraic structures, we show that the diagram commutes, i.e., for any connector $c$, $\mathcal{MC}_\ell(c) = proc(\mathcal{PA}_\ell(c))$. Once again we show this with an inductive argument, using the primitives as the base, and the operators as the inductive steps.

**Lemma 3.10** (Commutativity of Primitive Encodings)**.** For any primitive $p$ from Table 3.1, with the respective process $\mathcal{MC}_\ell(p) = (P, \pi)$, and corresponding port automaton $\mathcal{PA}_\ell(p) = (Q, N, \rightarrow, q_0)$, $P \cong proc(\mathcal{PA}_\ell(p), q_0)$.

*Proof.* We will only show that this lemma holds for one of the connectors, the fifo, because the other connectors can be shown in a similar way. Recall the automaton $\mathcal{PA}(\mathsf{fifo})$ defined in Section 3.3.1, adapted with the $\ell$ label. The complete specification $proc(\mathcal{PA}(\mathsf{fifo}), \mathsf{fifo})$ is defined below:

- $proc(A, \mathsf{fifo}) = (1, \mathsf{sr})_\ell.proc(A, \mathsf{fifofull})$
- $proc(A, \mathsf{fifofull}) = (1, \mathsf{sk})_\ell.proc(A, \mathsf{fifo})$

Using the definition of the $\mathcal{MC}_\ell$ encoding we obtain

$$\mathcal{MC}_\ell = (Fifo, \{Fifo = (1, \mathsf{sr})_\ell \ . \ (1, \mathsf{sk})_\ell \ . \ Fifo\})$$

It is enough to observe that

$$R = \{\langle proc(A, \mathsf{fifo}), Fifo\rangle, \langle proc(A, \mathsf{fifofull}), (1, \mathsf{sk})_\ell \ . \ Fifo\rangle\}$$

is a strong bisimulation between the two processes. □

**Lemma 3.11** (Commutativity of Sequential Composition)**.** Let $c_1$, $c_2$ be two connectors, such that $\mathcal{PA}_\ell(c_1) = (Q_1, N_1, \rightarrow_1, q_1)$, $\mathcal{PA}_\ell(c_2) = (Q_2, N_2, \rightarrow_2, q_2)$,

$\mathcal{MC}_\ell(c_1) = (P_1, \pi_1)$, $\mathcal{MC}_\ell(c_2) = (P_2, \pi_2)$. If $proc(\mathcal{PA}_\ell(c_1), q_1) \cong P_1$ and $proc(\mathcal{PA}_\ell(c_2), q_2) \cong P_2$, $c_1; c_2$ is well typed, and $\mathcal{MC}_\ell(c_1; c_2) = (P, \pi)$, then $proc(\mathcal{PA}_\ell(c_1; c_2), (q_1; q_2)) \cong P$.

*Proof.* Using Lemmas 3.3 and 3.7, we have

$$\begin{aligned} proc(\mathcal{PA}_\ell(c_1; c_2), (q_1; q_2)) &\cong proc(\mathcal{PA}_{\ell_1}(c_1), q_1) \| ; proc(\mathcal{PA}_{\ell_2}(c_2), q_2) \\ &= \rho_{\mathsf{Rename}(n_1,n_2,\ell_1,\ell_2,\ell)}(\tau_{\mathsf{Hide}(n,\ell)}(\partial_{\mathsf{Block}(n,\ell_1,\ell_2)}(\Gamma_{\mathsf{Com}(n,\ell_1,\ell_2)} \\ &\qquad (proc(\mathcal{PA}_{\ell_1}(c_1), q_1) \parallel proc(\mathcal{PA}_{\ell_2}(c_2), q_2))))). \end{aligned}$$

Furthermore, using the hypothesis, and the definition of $\mathcal{MC}_\ell$, we have that

$$\begin{aligned} &\rho_{\mathsf{Rename}(n_1,n_2,\ell_1,\ell_2,\ell)}(\tau_{\mathsf{Hide}(n,\ell)}(\partial_{\mathsf{Block}(n,\ell_1,\ell_2)}(\Gamma_{\mathsf{Com}(n,\ell_1,\ell_2)} \\ &\qquad (proc(\mathcal{PA}_{\ell_1}(c_1), q_1) \parallel proc(\mathcal{PA}_{\ell_2}(c_2), q_2))))) \\ &\cong \rho_{\mathsf{Rename}(n_1,n_2,\ell_1,\ell_2,\ell)}(\tau_{\mathsf{Hide}(n,\ell)}(\partial_{\mathsf{Block}(n,\ell_1,\ell_2)}(\Gamma_{\mathsf{Com}(n,\ell_1,\ell_2)}(P_1 \parallel P_2)))) \\ &= P \end{aligned}$$

Therefore this lemma is true. □

**Lemma 3.12** (Commutativity of Parallel Composition)**.** Let $c_1$, $c_2$ be two connectors, such that $\mathcal{PA}_\ell(c_1) = (Q_1, N_1, \rightarrow_1, q_1)$, $\mathcal{PA}_\ell(c_2) = (Q_2, N_2, \rightarrow_2, q_2)$, $\mathcal{MC}_\ell(c_1) = (P_1, \pi_1)$, $\mathcal{MC}_\ell(c_2) = (P_2, \pi_2)$. If $proc(\mathcal{PA}_\ell(c_1), q_1) \cong P_1$ and $proc(\mathcal{PA}_\ell(c_2), q_2) \cong P_2$, $c_1 \oplus c_2$ is well typed, and $\mathcal{MC}_\ell(c_1 \oplus c_2) = (P, \pi)$, then $proc(\mathcal{PA}_\ell(c_1 \oplus c_2), (q_1 \oplus q_2)) \cong P$.

*Proof.* Using Lemmas 3.4 and 3.8, we have

$$\begin{aligned} &proc(\mathcal{PA}_\ell(c_1 \oplus c_2), (q_1 \oplus q_2)) \\ &\cong proc(\mathcal{PA}_{\ell_1}(c_1), q_1) \|_\oplus proc(\mathcal{PA}_{\ell_2}(c_2), q_2) \\ &= \rho_{\mathsf{Rename}(n_2,m_2,\ell_2,\ell_2,\ell)}(proc(\mathcal{PA}_{\ell_1}(c_1), q_1) \parallel proc(\mathcal{PA}_{\ell_2}(c_2), q_2))) \end{aligned}$$

Furthermore, using the hypothesis, and the definition of $\mathcal{MC}_\ell$, we have

$$\begin{aligned} &\rho_{\mathsf{Rename}(n_2,m_2,\ell_2,\ell_2,\ell)}(proc(\mathcal{PA}_{\ell_1}(c_1), q_1) \parallel proc(\mathcal{PA}_{\ell_2}(c_2), q_2))) \\ &\cong \rho_{\mathsf{Rename}(n_2,m_2,\ell_2,\ell_2,\ell)}(P_1 \parallel P_2)) = P \end{aligned}$$

Therefore this lemma is true. □

**Lemma 3.13** (Commutativity of the Trace Operation)**.** Let $c_1$ be a connector, such that $\mathcal{PA}_\ell(c_1) = (Q_1, N_1, \rightarrow_1, q_1)$, $\mathcal{MC}_\ell(c_1) = (P_1, \pi_1)$. If $proc(\mathcal{PA}_\ell(c_1), q_1) \cong P_1$, $\mathsf{Tr}_n(c_1)$ is well typed, and $\mathcal{MC}_\ell(\mathsf{Tr}_n(c_1)) = (P, \pi)$, then $proc(\mathcal{PA}_\ell(\mathsf{Tr}_n(c_1), q_1) \cong P$.

*Proof.* Using Lemmas 3.5 and 3.9, we have

$$
\begin{aligned}
& proc(\mathcal{PA}_\ell(\mathsf{Tr}_n(c_1)), q_1) \\
& \cong Tr_n(proc(\mathcal{PA}_\ell(c_1), q_1)) \\
& = \tau_{\mathsf{Hide}(n,\ell)}(\partial_{\mathsf{Block}_{\mathsf{Tr}}(n,m,m',\ell)}(\Gamma_{\mathsf{Com}_{\mathsf{Tr}}(n,m,m',\ell)}(proc(\mathcal{PA}_\ell(c_1), q_1))))
\end{aligned}
$$

Furthermore, using the hypothesis, and the definition of $\mathcal{MC}_\ell$, we have

$$
\begin{aligned}
& \tau_{\mathsf{Hide}(n,\ell)}(\partial_{\mathsf{Block}_{\mathsf{Tr}}(n,m,m',\ell)}(\Gamma_{\mathsf{Com}_{\mathsf{Tr}}(n,m,m',\ell)}(proc(\mathcal{PA}_\ell(c_1), q_1)))) \\
& \cong \tau_{\mathsf{Hide}(n,\ell)}(\partial_{\mathsf{Block}_{\mathsf{Tr}}(n,m,m',\ell)}(\Gamma_{\mathsf{Com}_{\mathsf{Tr}}(n,m,m',\ell)}(P_1))) \\
& = P
\end{aligned}
$$

Therefore this lemma is true. □

**Theorem 3.2** (Commutativity of $\mathcal{MC}_\ell$ and $proc.\mathcal{PA}_\ell$)**.** Let $c$ be a connector, such that $\mathcal{PA}_\ell(c) = (Q, N, \rightarrow, q)$, $\mathcal{MC}_\ell(c) = (P, \pi)$. Then $P \cong proc(\mathcal{PA}_\ell(c), q)$.

*Proof.* This result follows by induction on the structure of connectors, whereas the base case is captured by Lemma 3.10, and the inductive steps are captured by Lemmas 3.11 to 3.13. □

# Chapter 4

# The ReoLive Web Framework

The ReoLive framework combines tools to analyse connectors and families of connectors, presenting graphics and reports, in a single web-based front-end application. The project is available online to be compiled in `https://github.com/ReoLanguage/ReoLive`.

The user can generate a connector using the Preo language, which is a concrete language to describe connectors, based on the connector calculus presented in [8]. Upon receiving a connector, the framework performs several computations to provide reports and depict graphically the given connector.

The framework offers two modes: a local mode, which can be executed on any machine, and a client-server mode, using an interface which connects the users machine to a server which performs more complex calculations. Each mode has its advantages and disadvantages, which we will present later.

We begin this chapter by presenting the Preo language, and its relation to the connector calculus presented in Chapter 3 (Section 4.1). After this, we introduce the main aspects of the user interface (Section 4.2). Later we report on the architecture of the framework, followed by some implementation details (Sections 4.3 and 4.4 respectively). Section 4.5 presents an example of the results obtained on the exclusive router connector. Finally, in Section 4.6, we explain some of the difficulties which prevented the implementation of a bounded

$$c = p \mid c;c' \mid c*c' \mid c\hat{}n \mid \texttt{Tr}(n)(c) \mid c\{c_1,\ c_2,\ \ldots\} \mid \phi?\ c + c' \mid c|\psi \mid \ \backslash\, x : T\ .\ c$$

$$p = \texttt{id} \mid \texttt{sym}(n, m) \mid \texttt{dupl} \mid \texttt{merger} \mid \texttt{drain} \mid \texttt{fifo} \mid \texttt{lossy}$$

$$T = \texttt{I} \mid \texttt{B}$$

Figure 4.1: Syntax for the Preo language, where $n, m \in \mathbb{N}$.

analysis of connector families, introduced by Proença and Clarke [8].

## 4.1 The Preo Language

The Preo language consists of an ASCII representation of the calculus of families of Reo connectors presented by Proença and Clarke [8], whose core syntax is presented in Chapter 3. The basic syntax of the language is presented in Fig. 4.1, where the markers `I`, and `B` represent the integer and boolean data types respectively, $\phi$, $\psi$ represent a boolean expression, $n$ represents an integer expression, and $x$ is a variable name. The operators presented in a faded colour are used to form parameterized connectors, and can describe connector families, which are not defined in this document.

The operators `;`, `*`, and `Tr` define the sequential, tensor and trace operations respectively, while the `^` operator is a syntactic sugar for an $n$-ary tensor product of the connector $c$ (e.g. `fifo^3` $\equiv$ `fifo*fifo*fifo`). Furthermore, a user can create named connectors using a concept called subconnector, by inserting these named connectors inside the symbols {}. Finally, two keywords: `reader`, and `writer` are defined in the language, which are used to visually simulate components when the connector is graphically represented.

**Example 4.1.** The following instance of the Preo language: `dupl; x; sym(1, 1); x {x = fifo*lossy}` corresponds to the connector

$$\Delta_2\ ;\ (\mathsf{fifo}\ \oplus\ \mathsf{lossy})\ ;\ \gamma_{1,1}\ ;\ (\mathsf{fifo}\ \oplus\ \mathsf{lossy})$$

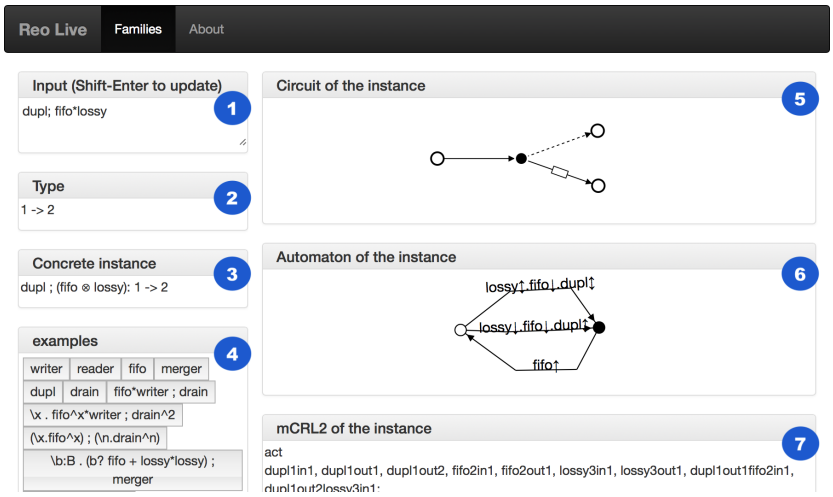



Figure 4.2: Screenshot of the ReoLive Website.

## 4.2 User Interface

We now present the interface for the local version of the framework, disregarding the server interface, since currently both versions have minimal interface differences. As the development of the project progresses, the respective interfaces may diverge from each other. In Fig. 4.2 we can find a screenshot of the current version of the framework. This version can be found pre-compiled online.[1] This web front-end is subdivided into different boxes, each providing a different input or output:

1. Input area;
2. Connector type;
3. A concrete instance and its type;
4. Examples of connectors;
5. Graphical Output of the instance from (3);
6. Graphical Output of the Port Automata of the instance from (3);
7. mCRL2 program, ready to be analysed by mCRL2 tools.

[1] https://reolanguage.github.io/ReoLive/snapshot/

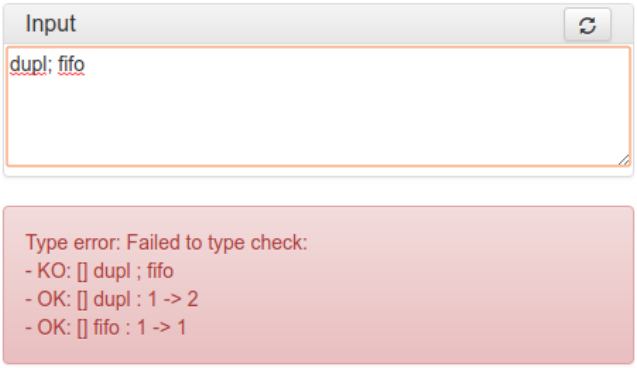



Figure 4.3: Example of an error output

The user interacts directly with the input area, by inserting the intended connector or family of connectors using the preo language. After the user inputs a new connector, a chain of computations is triggered. Upon conclusion, the user is presented with one of two things: an error box, detailing either a parsing error or a runtime error, or informations about the connector, and respective graphical analysis in the visible boxes. The error box is not visible to the user, unless an error occurs, and, when visible, it shows the information about the error. Fig. 4.3 contains the output error of the insertion of `dupl; fifo`, which contains a type error.

When the computations succeed, each box presents some information about the connector. The connector type box presents the input and output ports of each connector. If the connector is parameterized, some restrictions on the parameters may show up in this box. The concrete instance box presents an instance of the language given in the core calculus presented in Fig. 3.1. If the input connector is already a core instance, then this is the instance presented; otherwise an instance is calculated (e.g. `\n.fifo^n` generates the instance `fifo`). The remaining information, presented in the right column, is computed using this core instance of the connector. In (5), we are presented with an interactive graphical representation of the core instance, as a reo connector. (6) presents a simple representation of the port automaton of the instance, based on the encoding of Section 3.3. The port automata does not use unique identifiers on transitions. Instead, transitions contain the name of the primitives where information flows, associated with a downward arrow depicting information flowing

into the primitive, an upward arrow, depicting information leaving the primitive, or both. The mCRL2 box presents the model in mCRL2 of the instance following the encoding of Section 3.4, which the user can copy and process with the mCRL2 toolset. In this model, each action is identified by the name and a unique identifier of the primitive it refers to, as well as information about the type of port. Consider, for example a fifo. If the generated identifier is 0, then the input action of the primitive is identified by `fifo0in1`, referring to the first input (source) port of the fifo with identifier 0. Finally, the examples box provides some useful examples of connectors. When an example is pressed, the connector definition is placed in the input box and the computations are performed.

Any box can be minimized, hiding the output. This is particularly useful in the case of (6), because calculating and displaying the automaton is a computationally expensive operation, and state space explosion of the resulting PA may occur. We partially solved this problem by not performing the computation of (5), (6), and (7), when the respective boxes are minimized, since all may suffer from this problem. The user can minimize the boxes by clicking on the respective title.

## 4.3 Architecture

The ReoLive framework is split into two main projects: the Preo project,[2] and Reolive project.[3] The former provides the back-end components to parse and analyse connectors, while the latter uses the components of the preo project to generate a front-end to display the information. The Preo project was developed using the Scala programming language, and is compiled through the Reolive project either into a standalone JavaScript website, using the Scala.js compiler,[4] or into a client-server pair of programs. In the latter case, there is a client component, compiled into JavaScript which communicates with a server component, developed using the Play framework,[5] and compiled into Java bina-

[2] https://github.com/ReoLanguage/Preo
[3] https://github.com/ReoLanguage/ReoLive
[4] https://www.scala-js.org
[5] https://www.playframework.com

ries. Furthermore, both JavaScript programs use the D3 JavaScript libraries[6] to produce the graph layouts, which manipulate SVG-based diagrams. The overall architecture is summarised in Fig. 4.4.

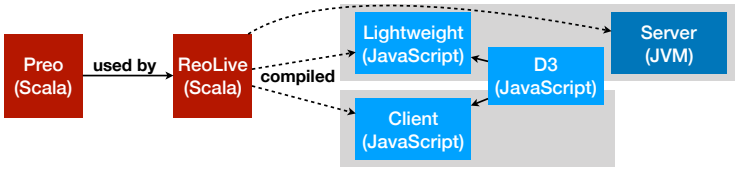



Figure 4.4: Architecture of the ReoLive implementation.

The standalone JavaScript website has the advantage of being an easy to distribute project (with a snapshot provided online), while the client-server mode is more powerful and complete. In this case, the server uses an SMT solver to address complex families of connectors, and it can execute mCRL2 commands to compile the generated model, providing the user with the resulting LTS and LPS (which is not possible to do using the standalone JavaScript). Furthermore, the ability to execute mCRL2 commands opens the possibility to integrate a $\mu$-calculus box on the client site, which can be used to analyse the mCRL2 model on the server side, and provide feedback to the user.

## 4.4 Implementation Details

The framework is composed of individual widgets, coordinated by a central manager. Internally a widget is an object with an associated data type, which stores a state variable with the associated data type which can be queried by other widgets. Each widget provides a visual output of one or more boxes, as highlighted in Fig. 4.2. Each widget provides three functions: `init`, `get`, and `update`. The `init` function loads the visual output in the front-end, and sets the initial state values for its variables. The function `get` queries the current value of the state variable of the widget. `update` updates the state of its variables and the output front-end. Furthermore, widgets may have other widgets as dependencies. When executing the `update` function, the widgets should get the associated state values of the dependencies, calling their `get` function.

[6] `https://d3js.org`

The manager loads the widgets into the front-end, using the `init` function of each widget loaded. Furthermore it generates a function which runs the `update` function of each widget ordering the updates such that the update of the dependencies of each widget is executed before the update of the widget, i.e., if widget $B$ depends on widget $A$, then $B$.`update` must be executed after $A$.`update` is executed. In the client-server component, the access to the server is performed using JavaScript callbacks inside the update method of the widgets.

We present the widgets of the ReoLive framework, with the dependencies, associated data type, and relevant information about them. Each widget is identified to a number corresponding to the respective box displayed in Fig. 4.2.

**Input Widget (1)**: The input widget has no dependencies, and its only function is to store the string containing the input connector, when updating. It receives a function from the central entity, which is executed when the user presses the update button.

**Type Widget (2)**: The dependency of the type widget is the input widget. When updating, it receives the input string, and parses it into a connector, and performs a type check. When success, it displays the resulting type in its visual output.

**Concrete Instance Widget (3)**: The concrete instance depends on the type widget. Its update function gets the connector generated by the type widget and instantiates it, presenting a connector with the core operators presented in Fig. 3.1 – called a core connector – and respective type. This core connector is the data type associated to this widget. This widget and the type widget are only present in the local version, while the type concrete instance replaces them in the server version.

**Type Concrete Instance Widget (2, 3)**: This widget is a mixture between the previous two widgets. It is only loaded in the server version. On load it presents the boxes of the type widget and the concrete instance widget. Instead of parsing the input, it sends the input string to the server, waiting for the type and core connector data from it. This communication turns the server version more powerful than the local version, as the local version has limitations instantiating the connector (e.g. `\n. \ m.(fifo^(n-m))` generates an error) which are not encountered in the server version.

**Examples Widget (4)**: The examples widget has no dependencies, but it must have access to the input widget. When the user selects an example, it copies the example to the input box, and forces an update of the widgets.

**Circuit Widget (5)**: The circuit widget receives the core connector from the concrete instance, and generates a graphical representation of it. The graphical representation is displayed using the d3 library for JavaScript.

**Automaton Widget (6)**: The automaton widget receives the core connector and calculates the respective port automaton. The PA is displayed using the d3 library. The calculations are performed using the encoding presented in Section 3.3.

The resulting automaton tends to grow exponentially, which affects the performance of the site. To reduce the impact of this problem, two measures were taken. The minimization technique to avoid updating has been explained. If the box is not minimized, after a certain number of steps are performed during the computation, a time-out error is reached, the update of the widget is cancelled, and an error message is displayed.

**mCRL2 Widget (7)**: This widget displays the mCRL2 model, using the core connector from the concrete instance widget. This model is calculated following the encoding of Section 3.4, with some changes. The names of the actions are given by the name of the primitive, associated with the process identification number, and an information about its type (source, sink, or mixed). The type can only be mixed if the primitive belongs to a sub-connector. Furthermore, we do not hide the synchronised actions, as opposed to the encoding of the previous chapter. This only happens if the connector is defined in a subconnector instance. On the client-server mode, the widget supports the download of the LTS, and LPS of the model, by executing mCRL2 commands to compile the model, on the server side.

Note that we do not consider the error box displayed as a widget, as it is a stateless component, and its internal structure serves only to display the error messages provided by the other widgets. In Fig. 4.5 we present the widgets mentioned in this section, using the labels from Fig. 4.2, excluding the examples widget. Note that in case of the server version, labels (2), and (3) should be combined.

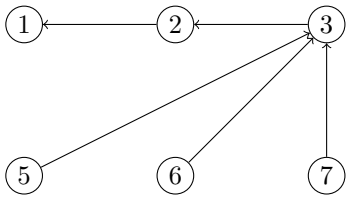



Figure 4.5: Dependency relation of the widgets

This structure provides a modular construction of the framework, making it easier to add new features to the framework. When a new feature is developed in the back-end of the framework, adding it to the framework should only consist of developing a new widget with the respective operations, and dependencies, and instantiating it in the central manager in order for it to be loaded. There is no need for understanding the internal structure of the remaining widgets to add a new one.

## 4.5 Example: analysing an exclusive router

We present an example of an exclusive router, described in the preo language. We explain the computational steps of the framework upon parsing the connector, and provide the output of each widget, with the exception of the mCRL2 model, as they tend to become large. We encourage the reader to test this example in the pre-compiled online version, where the model is presented.

The following code generates an exclusive router connector in the calculus of Reo connectors:

```
dupl ; dupl*id ;
(lossy;dupl)*(lossy;dupl)*id ;
id*merger*id*id ; id*id*sym(1, 1) ; id*drain*id
```

After saving the string value of the input widget, the framework parses the input, generating the respective connector object, and outputting the type of the connector in the type box, whose value is $1 \mapsto 2$. In this case, there are no parameters to instantiate, and the connector only uses core operators of the calculus. Therefore the concrete instance widget displays the connector and the type already mentioned. Then the circuit, automaton, and mCRL2 widgets are

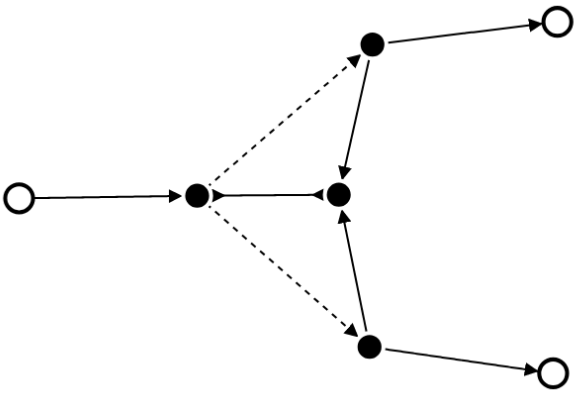


Figure 4.6: Exclusive router graphical output

updated following this order. We can find the output of the circuit and the automaton in Fig. 4.6 and 4.7, respectively. Notice that the port automaton of the connector has two transitions equally labelled. A more precise automaton would differentiate the various primitives. As mentioned before, we do not present the mCRL2 model generated. Instead, we present the generated LTS, presented in Fig. 4.8. For a better presentation we hide the internal ports of the connector in the model, by defining it as a subconnector named `exrouter`. The generated LTS presents a more complete graphical presentation of the connectors behaviour, as opposed to the automaton generated on the framework. Furthermore, consider the $\mu$-calculus formula

```
[true*][exrouter1in1| exrouter17out2 |exrouter16out1] false
```

that specifies that after any transition, information cannot flow out of the connector from both output ports. Executing an analysis using the mCRL2 toolset would show this formula to be true.

## 4.6 Towards Verification of Connector Families

The Preo language, as well as the full version of the connector calculus from Proença and Clarke [8], describe families of connectors. In this document we did not approach the families aspect, although some mentions and examples are presented throughout it, particularly in this chapter, while describing the existing tools to type-check Preo connectors.

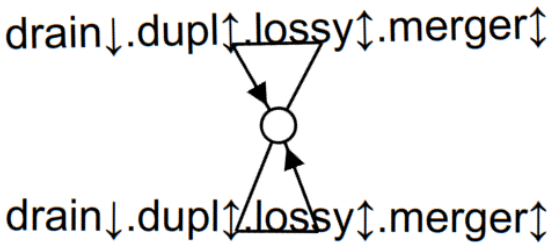



Figure 4.7: Exclusive router PA

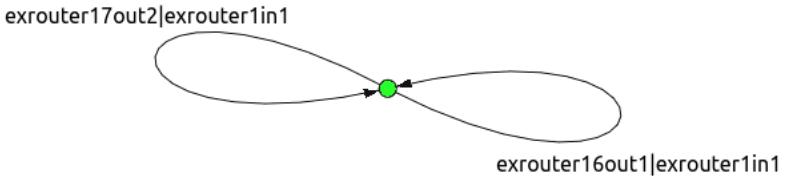



Figure 4.8: LTS of the exrouter

We experimented on how to verify the full calculus of connector families using the mCRL2 toolset, following the ideas from Beek and de Vink [3]. Unfortunately, mCRL2 requires the number of processes running in parallel to be fixed and known upfront, limiting the analysis to only a bounded set of families. The latest experiments consist of generating a small number of instances of a connector and include them in a single mCRL2 model, which can be used for model checking. However, we did not find a satisfactory approach to either select an interesting set of candidates for instances, or to give some control over the instances being selected. Furthermore, modelling families of connectors can easily produce a state explosion that is hard to control. Hence we left these experiments out of the existing framework, although they can be found in experimental branches on our GitHub project.[7] Future work will involve providing some control over the instances that could be of interest when analysing families of connectors, and investigating a suitable (modal) logic to describe properties over families of connectors.

[7] https://github.com/ReoLanguage/Preo/tree/family_modelation_nayve

# Chapter 5

# Conclusion

This thesis formalises the behaviour of a calculus of Reo connectors using port automata. We used this semantic model to generate an encoding of connectors in mCRL2, and this enables the use of the mCRL2 toolset to analyse properties and verify the behaviour of a given connector. For both formalisations we proved the respective correctness, with respect to bisimilarity. We implemented these encodings into a web framework – ReoLive – which, given a connector as input, provides several reports on the properties of the respective connector. This framework supports a family of connectors as input, although most reports presented refer only to one member of the respective family (generated by the framework).

Our work fell short on providing an encoding for families of connectors, as explained in Section 4.6. Our initial ambition was to verify the full spectrum of families of connectors. Unfortunately, this was not possible using the mCRL2 toolset, due to restrictions on the parallel operator, which does not support the specification of an arbitrary number of processes.

We plan to investigate the usage of different tools, such as Alloy,[1] to model and analyse the families. We also plan on extending the portfolio of available modules; for example, add support for a dedicated modal logic to verify connectors, analyse different semantics of Reo connectors other than Port Automata (incorporation of the IFTA tools is planned soon),[2] and add support for the Treo

[1] http://alloytools.org

[2] https://github.com/haslab/ifta

language to specify connectors.[3] Furthermore, we want to take advantage on the client-server version of the ReoLive framework, by introducing an input widget to analyse the connectors on the server, with a $\mu$-calculus property described in this widget.

[3] https://github.com/ReoLanguage/Reo

# Bibliography

[1] Farhad Arbab. Reo: A channel-based coordination model for component composition. *Mathematical Structures in Computer Science*, 14(3):329–366, June 2004.

[2] Christel Baier, Marjan Sirjani, Farhad Arbab, and Jan J. M. M. Rutten. Modeling component connectors in Reo by constraint automata. *Science of Computer Programming*, 61(2):75–113, 2006.

[3] Maurice H. ter Beek and Erik P. de Vink. Using mcrl2 for the analysis of software product lines. In *Proceedings of the 2Nd FME Workshop on Formal Methods in Software Engineering*, FormaliSE 2014, pages 31–37, New York, NY, USA, 2014. ACM.

[4] Fabio Gadducci and Ugo Montanari. The tile model. In *Proof, Language, and Interaction*, pages 133–166. MIT Press, 2000.

[5] Jan Friso Groote, Aad Mathijssen, Michel Reniers, Yaroslav Usenko, and Muck van Weerdenburg. The formal specification language mCRL2. In *Methods for Modelling Software Systems (MMOSS)*, Dagstuhl Seminar Proceedings, 2007.

[6] Christian Koehler and Dave Clarke. Decomposing port automata. In *Proc. SAC '09*, pages 1369–1373, New York, NY, USA, 2009. ACM.

[7] Natallia Kokash, Christian Krause, and Erik P. de Vink. Reo + mCRL2: A framework for model-checking dataflow in service compositions. *FAC*, 24(2):187–216, 2012.

[8] José Proença and Dave Clarke. Typed connector families and their semantics. *Sci. Comput. Program.*, 146:28–49, 2017.