FORMAL MODELS FOR COMPONENT CONNECTORS

David Filipe de Oliveira Costa

FORMAL MODELS FOR COMPONENT CONNECTORS

Printed and bound by GVO printers & designers B.V.
ISBN: 978-90-6464-436-8
IPA Dissertation Series 2010-10

The work reported in this thesis has been carried out at the Centrum Wiskunde & Informatica (CWI), under the auspices of the research school IPA (Institute for Programming research and Algorithmics). The research was funded by an FCT (The Portuguese Foundation for Science and Technology) grant (SFRH/BD/13762) and by the CWI group Foundations of Software Engineering (SEN3).

Dedicated to the loving memory of my grandmothers

Dedicado à memória das minhas duas queridas avós

VRIJE UNIVERSITEIT

FORMAL MODELS FOR COMPONENT CONNECTORS

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad Doctor aan
de Vrije Universiteit Amsterdam,
op gezag van de rector magnificus
prof.dr. L.M. Bouter,
in het openbaar te verdedigen
ten overstaan van de promotiecommissie
van de faculteit der Exacte Wetenschappen
op maandag 13 december 2010 om 11.45 uur
in de aula van de universiteit,
De Boelelaan 1105

door

David Filipe de Oliveira Costa

geboren te Compiègne, Frankrijk

promotoren:	prof.dr. J.J.M.M. Rutten
	prof.dr. F. Arbab
copromotoren:	dr. M. Niqui
	dr. D.G. Clarke

ACKNOWLEDGMENTS

The writing of this thesis represents a lifetime landmark and thus far the most exciting journey of my life. Were it not for the help of many, I would have never been able to survive it, much less, conclude it. Trying to express in words how grateful and in debt I am to some of the people that made it possible is a difficult task, yet one that I am most definitely happy to do.

I am hopelessly in debt to my promotors Jan Rutten and Farhad Arbab. I could not have hoped for better mentors. I only met Jan and Farhad personally when I moved to Amsterdam to start my PhD, and it did not take long for me to realise how fortunate I was to have them both as supervisors. Jan and Farhad have distinct scientific areas of expertise but share common interests. The topic of my thesis is one of their common interests, and the research reported in my thesis stems from their joint work on formal models (semantics) for connectors. I am immensely grateful for all I have learned from them. From Jan, for instance, I learned the importance of universal mathematical concepts and constructions, and how mathematics can be used to capture with elegance and simplicity concrete tangible problems. From Farhad, for example, I learned how important it is to present ones ideas with clarity and coherence. Any signs of simplicity or clarity in my thesis are entirely due to their influence. Jan and Farhad were not just my supervisors, they were also my closest friends, whenever I needed. During the long period I was writing up my thesis, Jan had the patience to meet regularly to discuss my drafts. His support and encouragement were unconditional and without parallel. Farhad had an incredible patiente to listen to my ideas and encouraged me to pursue them. Never complained about how inarticulate I was at times. He believed in me and made sure I knew it. Both taught me a great deal about life in general and most of what I know about research. The advises I received from them will stay with me for the rest of my life.

I want to thank my daily supervisors and co-promotors Dave Clarke and Milad Niqui, who complemented the supervision by Jan and Farhad. Dave joined CWI and started his post-doc shortly after I started my PhD. We worked together on the semantics of $\mathcal{R}$eo. The work we did together was very exciting, and eventually led to the results reported on Chapter 3. Dave has tremendous stamina and is very passionate about what he does. He strives for the best and he is very critical of his work. In our long and frequent discussions, I learned to question my ideas and compare them with the work of others. Shortly after Dave moved to Leuven where he had just been offered a professorship position, Milad joined our group at CWI and took on the role of my daily supervisor. At the time I was struggling to conclude the work on intentional automata and write up Chapters 4 and 5. In our regular meetings, Milad's sharp mind never ceased to amaze me. After a couple of meetings I had overcome the problems I was facing and had found an effective routine that allowed me to produce regular drafts. I just hope we can continue collaborating.

I would like to thank the members of my committee—Christel Baier, Marcello Bonsangue, Marco Aiello, Wan Fokkink, and Erik de Vink—for taking the time to read my manuscript and providing me with comments and remarks that helped me improve the final version of the thesis.

A special thank goes to my former officemates, Freek Burger, Kees Everaars, Juan G. Scholten and José Proença. Freek and Kees took upon themselves to make me feel at home when I first arrived at CWI. On my very first day Kees took me to the bike shop, and that day I left the office riding on the *fietspad* of Amsterdam. Freek convinced me to join the group of ice skaters at CWI, and that is how I learned to *schaatsen*. I have fond memories of the summer Kees lent his house to me and my parents when they first came to visit me. Juan became a close and dear friend, always ready to listen and help. I recall with nostalgia our regular discussions about the most diverse topics while eating a pizza and having a beer at Pescara. José deserves a special thank you. José has been my companion and buddy through most of my PhD. We have been through a lot together. I recall specially the road trip we went on to switzerland to watch Portugal-against-Czech-Republic match, for the European Championship in 2008. The work on the connector animation reported in Chapter 6 has been our project and one I really enjoyed doing. I cannot thank Jose without saying thank you to Alexandra Silva. The three of us met at the University of Minho and have been close friends ever since. I did everything I could to persuade Zé and Xana to do their PhD at CWI and join me. I am so glad they did. I cannot imagine how my life in Amsterdam would have been without them. They make Portugal feel closer and are there whenever I need them.

CWI is an amazing place. It offers to its employees the best conditions. I wish to thank all my friends and colleagues from CWI and especially the Foundations of Software Engineering (SEN3) group for the always pleasant atmosphere, all the great events, and discussions. Susanne van Dam deserves a special thank you, for being a friend always ready to help. She is the pendulum that guarantees the perfect functioning of our group.

I have been lucky to make new friends in Amsterdam since the day I arrived. They are part of what has made and makes my everyday life in the Netherlands great and I thank you all for that—Adam, Ana Sokolova, Arjen, Behnaz, Christian, Clemens, Daniel and Stephanie, Dorina, Elmer and Edith, Elena, Fernando, Florence and Sibren, Frank, Filippo, Georgina and Wolker, Helle and Hartmutz, Ichiro and Kumiko, Immo, Joost and Dina, Katharina, Kees Bloom, Lacramioara, Markéta, Marcello, Marcin, Marjan, Mohammad Dashti, MohammadMahdi Jaghouri, MohammadReza Mousavi, Michiel, Myoung Jin Nam, Natalia, Nikolay, Peter and Wendy, Peter Zoeteweij, Rodrigo, Ronny and Rocio, Stephanie, Stijn Keijzer, Stijn Oonk, Sun, Thijs, Tom, Young Joo.

Living in Amsterdam means that I have to be far away from my family and my friends in Portugal. It hurts to be apart: *As saudades são sempre muitas.* They make me a better person, and I owe everything I have accomplished to them. I thank with all my heart my entire familly, and in particular *a minha mãe* Dina, *o meu pai* Ricardo, *o meu mano caçula* Bruno and his girlfriend Ana for just everything. My family extends to my hometown and forever friends: *o meu outro irmão* João Paulo, *os meus primos* Vitor and Ricardo, *o meu parceiro de carteira* Ricardo, *os meus parceiros de sueca* Rui and Vasco, and *o meu amigo do secundário* Zé. Every time we are together they make me feel like I never left. Thank you!

Many thanks to my friends, including the 5F *family*—Ângelo e Ticha, Miguel e Susana, Jorge, António Pedro, Maria João, Catarina, Dino, Manel, Rui e Joana, Rui e Conceição, Patrícia, Artur, Márcio e Cristina, Ana Luísa, Alexandre, Inês e Gonçalo, Ana Rita, Madaleno, Pi, Ângela, Cândida, Susana Carvalho, Gonçalo Pacheco, Neusa—for being such an amazing and special group of friends. The 5F activities have by now become traditions and are the highlights of most of my trips to Portugal. My dear friends Maria João and Catarina deserve a dedicated thank you. With their visits to Amsterdam and our summer vacation in Italy, they made sure I kept my sanity throughout my PhD.

I wish to thank my professors at the Departamento of Informatica, who have inspired me throughout my studies: José Bernardo Barros, José Nuno Oliveira, Manuel Barbosa, Jorge Sousa Pinto, Alberto Proença, José Valença, Luís Pinto, and last but not least Luís Barbosa who took me under his wing and made it possible for me to apply for a position at CWI and obtain a PhD grant. They are the reason I decided to do a PhD.

I would like to thank my colleagues and friends at Fredhopper—Adrien, Azhar, Barthold, Bram, Christiaan, Erica, Frank, Frederik, George, Martin, Nikolay, Marc, Pavel, Peter, and Ronny for their support and the great working atmosphere. I am specially grateful to Joris, Daniel, and Andreas for their understanding, encouragement and flexibility when I was writing up my thesis.

Finally, my tender loving thanks to Nynke for withstanding the vicissitudes of thesis writing by my side. Her personality and strength have been a constant source of inspiration for me.

Amsterdam, 5 October 2010.
David Costa

CONTENTS

1

INTRODUCTION

The corner stone of all engineering disciplines is the principle of constructing complex systems out of building blocks according to well defined rules. In the paradigm of hardware design, for example, complex systems are obtained by composing interconnected, inherently parallel components, which can represent transistors, logic gates, functional components such as adders, or architectural components such as a processor. In this thesis we study *computer systems* according to the same principle. We study computer systems and their construction according to a component-based paradigm.

1.1 COMPONENT-BASED SYSTEMS

In the component-based paradigm, systems are described as a collection of components that interact with each other to provide new functionalities resulting from the combination of each individual component's functionality. Components are characterised by abstractions that ignore implementation details and describe properties relevant to their composition, e. g. transfer functions, and interfaces. Composition is of paramount importance, not just because it defines the rules according to which components can be put together, but most interestingly because it describes the interaction between the composed components. Composition can be seen as an operation that takes in components, and interaction protocols, and as a result yields a component-based system where components interact according to the interaction protocols.

It is by now generally accepted that paradigms for component-based systems should provide a clean conceptual separation between computation and interaction [47]. This is much more evident as we move towards component-based systems where issues like reusability, maintenance, and heterogeneity are prominent. The notion of *connector* is becoming more and more popular, and is used to refer to the appropriate glue code, necessary for components to connect, interact and interoperate. Strangely, the majority of modelling or programming languages for component-based systems do not offer any means of expressing the connectors explicitly at the implementation level, obliging connectors to be programmed within components or within the application program. Contemporary component-based systems use composition mechanisms based on method invocation, remote procedure calls, or targeted message passing to interact directly with the different components that compose the system. Those mechanisms require a great degree of homogeneity among the components to be used effectively. Most commonly, interactions are defined through include files and import and export statements leaving interaction details entirely hidden inside individual components. These mechanisms evidently limit the reusability of individual components, and the dynamic interchangeability of components. The fact that the interaction details are part of the component limits the use of the component to the specific interaction protocol that it contains.

In this thesis, we treat interaction as a first-class concept. Rather than focusing on the individual components that constitute the system, we focus our attention on the interaction protocols that describe how the individual components interact. Interaction protocols are promoted, and are themselves treated as components, forming a special class of components the, so-called *connectors*.

1.1.1 *What do we mean by connector?*

Connectors can be seen as constituting a distinguished component class. Connectors' behaviour defines the interaction protocols that are used in the design of component-based system. Connectors dictate how the components that constitute a system *connect* and *interact* with each other. Connectors have no relevant role in the computation carried out by the overall system. Computation is the responsibility of the components. Connectors provide systems-independent interaction protocols, whereas components provide systems-specific functionality. Components are kept independent of each other and of their environment; it is the responsibility of the connector to coordinate the activities of individual components to ensure their proper interaction with one another to form a coherent system that behaves according to its requirements.

1.1.2 *Illustrating example*

Consider two individual components, a *bar-code scanner* module and an *LCD*[1] panel. We think about these components as black-boxes and we are only concerned about their interaction. The bar-code scanner interface consists of a single output port through which, just after having read a bar-code, it communicates the product name associated with this code. The LCD panel interface offers a single port to input text to be displayed. The goal is to build a simple component-based system that allows the user to scan a product bar-code and have the name of the product displayed on the LCD. After reading the specification of each component, we realise that there is

[1] liquid crystal display.

a mismatch between the rate at which the text displayed by the LCD can be updated and the rate at which the bar-code reader is able to output the product name. The bar-code scanner is able to scan bar-codes and provide successive product names at a higher rate than the LCD allows the text in its display to be updated. This state of affairs means that we cannot simply *wire-up* the output port of the bar-code scanner with the input port of the LCD panel. What to do when the user starts scanning bar-codes at a pace that exceeds the rate at which the LCD can display the respective product names? We mention a few possible scenarios:

1. Do we force the bar-code scanner to wait for when its output port is not busy to read another bar-code? That is to say, do we synchronise the output port of the bar-code scanner with the input port of the LCD panel?

2. Do we buffer the excess data and display it on a read first, display first order?

3. Do we disregard bar codes that are input while the LCD is busy displaying a previous product name?

4. Do we combine the approaches 2, and 3 and provide a limited buffer where a finite amount of bar codes can be buffered while the LCD is busy displaying a product name? Once the buffer is full, extra bar codes input are disregarded.

As an example, let us consider option 3 and disregard the excess data, that is, we ignore the data provided by the bar-code scanner when the LCD port is not ready to receive data. The result is a system that allows the user to scan a product bar-code and have the name of the product displayed on an LCD panel, with the pecularity that if the user scans bar-codes *too quickly* he will have to read the same bar-code repeatedly until the name of the product is displayed on the LCD panel.

This simple example illustrates a common situation faced by the designers of component based systems: building a desirable system out of independent components does not simply amount to the task of composing them by wiring *properly* their ports together. Special software code, generally called *glue code*, is necessary to *coordinate* their interactions.

Coordination languages are a class of programming languages that offer a solution to the problem of specifying and managing the interactions among computing entities, and specify the glue code. In fact, they generally offer language mechanisms for composing, configuring, and controlling systems made of independent, distributed, active components. However, not all coordination languages ensure that the conceptual separation between computation and coordination is respected at the level of the source code as pointed out by Arbab's classification of coordination models and languages into either *endogenous* or *exogenous* [5]. Endogenous languages provide primitives that must be incorporated within a computation for its coordination. In applications that use such models, primitives that affect the coordination of each module are inside the module itself. Endogenous models lead to intermixing of glue code with computation code. This entangles the semantics of computation with coordination, thus making the coordination part inside the application implicit and sometimes intangible. In contrast, exogenous languages provide primitives that support the coordination of entities from without. In applications that use exogenous models, primitives that affect the coordination of each module are outside the module itself.

In the example above, we designed a connector to deal with the necessary interactions according to the principle of exogenous coordination, which proposes to consider *connectors* as first class citizens, completely separate from components. The connector deals with the necessary interactions between the two components and acts as mediator between the bar-code scanner and the LCD panel, coordinating the two in order to enforce the behaviour of option 3 that we want for the resulting system. This solution keeps both components independent, entirely reusable, and interchangeable. If we were to decide at a later stage to choose a different option, we could still reuse the same two components, and would just have to design a new connector to replace the existing one. Actually, none of the components is aware of the way it is interacting with the others—the input port of the LCD component is dictating the rate at which the resulting system is able to display product names on the LCD panel, and scanned bar codes are disregarded by the connector when the input port of the LCD panel is not ready to receive data.

1.2 RESEARCH CONTEXT

As a first-class concept, connectors are entitled to their own *specification* and *abstractions*. A connector has an *identity* according to the specific interaction protocol it provides. As components, connectors can be *composed* and yield more sophisticated connectors. A natural and promising line of research that has been followed in recent years is that of developing compositional models for component connectors [9, 19, 31, 32, 35, 36]. Typically these models provide an abstract semantic domain to express interaction protocols, provide operations on the domain, and a behavioural equivalence relation of interest that identifies elements of the domain. Special elements of the domain are chosen to specify *basic* interaction protocols and correspond to the behaviour of *so-called primitive connectors*. From these primitive connectors more complex connectors can be constructed, and the interaction protocols of the resulting connectors are obtained by composing the basic interaction protocols of the primitive connectors, using the operations on the semantic domain. Two connectors are said to be equivalent, or behaviourally interchangeable if their interaction protocols are related by the behavioural equivalence defined in the semantic domain.

A compositional model for component connectors means in simple terms that the behaviour (the interaction protocol) of a connector $C_1 \times C_2$ is determined by the combined behaviour of each C_i. More formally, a compositional model is expressed by the compositionality property:

$$C_1 \sim C_1' \text{ and } C_2 \sim C_2' \Longrightarrow C_1 \times C_2 \sim C_1' \times C_2' \qquad (1.1)$$

where the relation $\sim$ denotes the behavioural equivalence of interest, and $\times$ is a composition operator defined using the operations on the semantic domain. Compositionality as presented above provides a means to achieve modularity and to breakdown the complexity of large component connectors.

1.2.1 *$\mathcal{R}$eo*

The main research questions addressed in this thesis, as the title indicates, concern formal models for component connectors. In particular we investigate models for the coordination language $\mathcal{R}$eo. $\mathcal{R}$eo [6, 7] is an exogenous coordination language based on a calculus of channel composition to construct component connectors. $\mathcal{R}$eo constitutes a long term research project of the Foundations of Software Engineering group at CWI. $\mathcal{R}$eo is based on the principles of preceding models like *Ideal Worker Ideal Manager* (IWIM) model [4, 80, 21] and coordination languages like MoCha [12, 88] and MANIFOLD [10, 11]. $\mathcal{R}$eo is specially interesting because of the large range of behaviour it permits to express due to features such as: asynchrony, multi-party synchronisation, mutual exclusion, and context-dependent behaviour. The dataflow behaviour of $\mathcal{R}$eo connectors extends the behaviour that classical models such as Kahn networks, and dataflow languages can express. The semantics of $\mathcal{R}$eo, due to the rich set of features in the language, have however proven to be non-trivial, and several formal models for its semantics have recently been proposed. The original CWI report [6] about $\mathcal{R}$eo presents an informal semantic based on a scheme of *accepts-and-offers* to model the distributed dataflow behaviour of $\mathcal{R}$eo connectors. This semantics turned out to be too difficult to serve as an implementation specification, and consequently too difficult to be implemented [44]. The major difficulty of the proposed scheme is that it describes the semantics of the interaction protocols taking into account details that a formal semantics would (at least initially) abstract away from. For example, the first formal semantics proposed for $\mathcal{R}$eo connectors chose to abstract from some of such details:

1) *coinductive calculus for component connectors* [9] is a simple and transparent relational model, that relies on the coalgebraic proof and definition principles [86]. There the behaviour of connector ports are modelled as *timed-data streams*. An element of the *time*-stream is the time at which the *data value* in the *data*-stream is observed in the corresponding port. The connectors' interaction protocols are modelled as relations on *timed-data streams;* the relations are defined using the coalgebraic definition principle, and the coalgebraic principle of coinduction is used to reason about connector equivalence. Both principles come for

free with the final coalgebra structure that is carried by streams. The model abstracts away from the connector topology, and the direction of the dataflow within the connector. Connectors are reduced to a collection of ports, and the behaviour of the component-based system is expressed as a relation.

2) *constraint automata* [13, 19] are an operational model, based on the theory of automata, in particular I/O automata [72], that builds on the model above, and models the interaction protocols as constraint automata. Constraint automata are acceptors of relations on timed data streams, such an automaton observes the data occurring at the ports and either fires a transition according to the observed data or rejects it if there is no corresponding transition in the automaton. This formalism provides automata operations to model the composition operations of $\mathcal{R}$eo connectors: join and hide; uses the notion of bisimilarity (captured by coinduction on streams) to reason about connector equivalence, and the notion of simulation to reason about behaviour refinement. Constraint automata have been used successfully as the foundation for building tools that automate:

  - the compositional construction of automata models of complex component connectors;
  - the equivalence checking or containment checking of the interaction protocols of two given connectors;
  - the verification of interaction protocols.

Models 1) and 2) live at a level of abstraction where the fundamental concepts and operations in $\mathcal{R}$eo can be formalised. They provide clear insight into the behaviour of connectors, which the informal semantics, and the textual description of $\mathcal{R}$eo lack. Even though important concepts and details are abstracted away in these two models, the expressiveness of the interaction protocols captured by these models remains interesting both theoretically and practically [20]. In particular, both these models are refined enough to differentiate the behaviour of non-deterministic $\mathcal{R}$eo connectors that in other dataflow models, such as Kahn networks, lead to the so-called Brock-Ackerman anomalies [28]. In general, both these models provide sound and systematic methods to reason about connector equivalence, minimality, and expressiveness.

1.3 RESEARCH QUESTIONS

Models 1) and 2) permit to observe each port of the connector and at each step register through which ports data flow. In model 1), each element of the time-stream of the timed-data streams indicates the *time* at which a certain *data value* is observed in the corresponding port. In model 2) each transition of the automata registers the ports that fire and consequently exhibit dataflow. Context-dependent connectors extend the class of dataflow behaviour by including behaviour that depends on the presence or absence of *pending* I/O operations—an I/O operation present on a port. The observation of pending I/O operations permits to express in the model non-monotonic behaviour such as precedence and blocking—referred to as context-dependent behaviour. The models of 1) *coinductive calculus* and 2) *constraint automata* above are unable to capture context-dependent behaviour and consequently are inappropriate to model context-dependent connectors. In order to accurately model this type of behaviour, unlike models 1) and 2), a formalism has to consider the pending I/O operations on a port as observable. That constitutes a critical difference in what is considered observable in the model and raises several research questions.

These are the questions we address:

i *How to introduce the context information reflecting the pending* I/O *operations in the model*?

ii *Can we express context-dependent behaviour and yet keep the model intuitive, transparent and compositional like models 1) and 2)*?

iii *Can we devise context-dependent models that can serve as implementation specifications, unlike the original informal semantics?*

iv *Having an answer for iii, can we refine implementation specifications into fully executable models that permit the automatic synthesis of efficient code in main stream programming languages*?

In this thesis we propose models that address these questions. One important caveat that applies to all the models we propose in this thesis is that the models do not include information about data: data is seen abstractly as mere *signals*. Hence the behaviour is data insensitive. Models 1) and 2) include information about data—data domains—and permit to define data constraints and hence express data sensitive behaviour. The approach suggested by models 1) and 2) provides an elegant solution to express data sensitive behaviour, and can easily be incorporated in our data insensitive models.

Following the models we present in this thesis other models have been proposed that provide answers to some of the research questions stated above. These include Büchi automata [55, 57], Tiles [15], and Guarded Automata [25] models. In the context of service orchestration, a calculus [23] in the Bird and Meertens style is introduced that admits connectors with context-dependent behaviour. Throughout this thesis whenever relevant we will discuss the works above and other related work.

1.4 THESIS OVERVIEW AND CONTRIBUTIONS

We finish this introduction by presenting an outline and a summary of the contributions of each of the chapters in this thesis.

CHAPTER 2. **component connectors** Chapter 2 sets the stage by introducing the concepts that are used throughout the thesis, and fix terminology related with connectors and component-based systems. We introduce $\mathcal{R}$eo and describe the behaviour of the channels used throughout the thesis. We present what by now are considered the classical examples for explaining and evaluating the compositionality of the semantics that address the feature of context-dependency. These examples are insightful to understand the subtleties of reasoning about composition when considering models with context-dependent behaviour; and provide implicitly the compositionality requirements for the semantics of the $\mathcal{R}$eo operations join and hide.

CHAPTER 3. **connector colouring model** In Chapter 3 we devise a simple and intuitive semantic model to define the dataflow behaviour of connectors. This semantics is based on *connector colouring*. Colours are used to denote dataflow and its absence. Colourings with two colours suffice to express the same class of behaviour as models 1) and 2); and capture features like asynchrony, multi-party synchronisation, and mutual exclusion. Each colouring of a connector is a solution to the synchronisation constraints imposed by its channels and nodes.

By refining the set of colours to three colours we are able to capture an additional feature—context-dependency. A connector colouring with three colours represents a solution to the *context-dependent synchronisation* and *context-dependent mutual exclusion* constraints imposed by each sub-constituent of the connector. Colouring a connector in a specific state with given boundary conditions (I/O requests) provides a means to determine the routing alternatives for dataflow.

The colouring metaphor confers a degree of transparency to the model that makes connector colouring semantics easy to understand. The circuit representation of connectors are used to describe the dataflow behaviour—each colouring corresponding to a dataflow behaviour of the connector can be overlaid on top of the circuit representation to provide insight into the dataflow behaviour of the individual primitives of the circuit.

The composition operator for colourings has a number of formal properties, namely, associativity, commutativity, and idempotency. These properties make the colouring scheme with its composition operator suitable for distributed implementations.

In summary our contributions in Chapter 3 include:

- A simple transparent model for context-dependent connectors.
- A solution for synchronous causal loops and ill formed data-dependencies.
- A definition of the hide operation that preserves context-dependent behaviour.
- Algorithms that provide a basis for diverse implementations, including distributed, to compute the dataflow behaviour of $\mathcal{R}$eo connectors.

CHAPTER 4. **intentional automata model** In Chapter 4 we consider a more general definition of connectors and consider as a connector any concurrent reactive system. We present *intentional automata*—an automata model for modelling the behaviour of concurrent reactive systems. What distinguishes this model from others is that it explicitly models the arrival of (I/O) communication requests, and differentiates between communication requests and actual communications. This gives the model an extra degree of expressiveness that will allow to model context-dependent systems. To capture the context information about the presence or absence of I/O operations the transitions in the automata register the I/O operation requests and the I/O operation firings.

We define operations to compose intentional automata and encapsulate information on intentional automata. We propose equivalence relations based on the notions of bisimulation and weak-bisimulation. We show that the weak-bisimulation relation constitutes a congruence for the defined operations.

In summary our contributions in Chapter 4 are as follows.

- An automata model for context-dependent connectors with:
  - composition and information hiding operations;
  - a congruence equivalence relation.

CHAPTER 5. **reo automata** In this chapter, using two properties that result from axiomatizing the dataflow behaviour of a $\mathcal{R}$eo connector port we are able to characterise a *well-defined* class of intentional automata which we call *$\mathcal{R}$eo automata*.

The $\mathcal{R}$eo automata class admits a novel representation which we call *configuration tables*. Configuration tables resemble colouring tables that are introduced in Chapter 3. Configuration tables are a very succinct representation and constitute a definition principle for $\mathcal{R}$eo automata.

We define operations on configuration tables that are faithful with respect to the operations on intentional automata, and show that the $\mathcal{R}$eo automata class is closed under these operations. With this argument we claim that the $\mathcal{R}$eo automata class of intentional automata captures the context-dependent behaviour of $\mathcal{R}$eo connectors. The operations on the configuration tables have a considerably lower computational cost when compared to the cost of applying the intentional automata operations on the $\mathcal{R}$eo automata denoted by the respective configuration tables.

We conclude this chapter by comparing the models we have presented in this thesis with other existing models.

In summary our contributions in Chapter 5 are as follows.

- Axiomatisation of the dataflow behaviour of $\mathcal{R}$eo connector ports.
- Identification of the intentional automata class—$\mathcal{R}$eo automata—resulting from the axiomatisation of a $\mathcal{R}$eo connector port.
- Configuration table models for $\mathcal{R}$eo connectors and the definitions of join and hide operations on configuration tables.

CHAPTER 6. **connector simulation and animation** Constructing connectors in $\mathcal{R}$eo is conceptually analogous to the design of asynchronous electronic circuits. The same way one uses basic logic gates to build more complex digital circuits, $\mathcal{R}$eo uses an extensible set of channels as primitive connectors from which designers build complex connectors. Among other things, this analogy emphasises the importance of visual environments for design, analysis, verification and optimisation of $\mathcal{R}$eo connectors, as counterparts of tools and facilities available in modern electronic Computer-Aided Design (CAD) systems. In Chapter 5 we investigate dynamic, graphical representations that enable the understanding of the behaviour of $\mathcal{R}$eo connectors without requiring expertise in the formal semantics of $\mathcal{R}$eo. Our interest is on methods to automatically generate these graphical representations and integrate them in the $\mathcal{R}$eo development tools.

We present a framework, called Connector Animation, to simulate the dataflow behaviour of $\mathcal{R}$eo connectors by means of visual animations. The visual animations are based on the connector colouring semantics.

We define an animation specification language based on four simple dataflow actions. The dataflow actions can be combined and permit to define animation specifications. An animation specification consists of a collection of basic dataflow actions that respect the data dependencies dictated by the colouring.

We characterise what constitutes a formal refinement map from the dataflow behaviour prescribed by a colouring into an animation specification.

Our main result is that the composition of colourings carries over to refinement maps, meaning that whenever two colourings compose, we can compose the animation specifications that refine each colourings and obtain an animation specification that refines the composed colouring.

We describe how the animation specifications are compiled into generic executable animation descriptions. An animation description consists of an abstract animation code tailored for $\mathcal{R}$eo connectors and suitable for being mapped into standard animation languages, such as Flash©.

We finish the chapter describing the two existing implementations of Connector Animation, and summarising how the synthesised animations are used has a complementary tool for $\mathcal{R}$eo connector developers.

In summary our contributions in Chapter 6 are as follows.

- Definition of formal refinement of constructive 3-colouring models into animation specifications.
- Compositionality result for refinement maps.
- Visual elements to capture and express dataflow actions on top of the connector colouring visual elements.
- Automatic synthesis from animation specifications of rich multimedia animations suitable for debugging and pedagogical purposes.

2

COMPONENT CONNECTORS

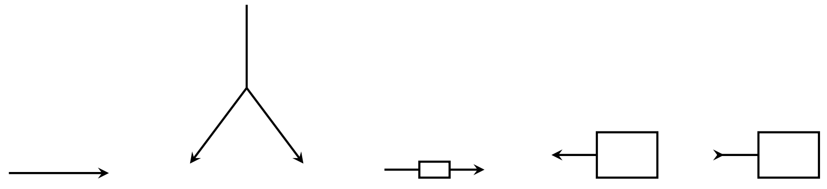
We fix our terminology and present the basic concepts concerning component connectors that we use throughout the thesis. We introduce the language for component connectors: $\mathcal{R}$eo. We fix a set of $\mathcal{R}$eo connector primitives to serve as the building blocks of a connector language called Conlang. The language Conlang uses the two $\mathcal{R}$eo operation join and hide to build connectors. The language Conlang will constitute our reference language to build component connectors and to assess the abstract semantic domains we propose to specify the dataflow behaviour of component connectors. Through examples we illustrate the compositionality of the dataflow behaviour of $\mathcal{R}$eo connectors.

2.1 BASIC CONCEPTS

An abstract yet useful way of thinking about a connector is to view it as a black box that has a well-defined interface, and a name identifying it. Internal details are abstracted away from consideration. The interface describes the collection of communication *ports* through which the connector interacts with its environment. The number of ports of a connector determines the connector's *degree*. The connectors depicted in Figure 1 are examples of abstract connectors with different degree. Depending on the actual level of abstraction, a port can have an additional polarity attribute indicating whether it is an input port or an output port. In that case the output ports have an arrow pointing away from the connector, such as the connectors 1a, and 1d; and the input ports have either no arrow (1a) or an explicit arrow pointing inwards, such as connector 1e.

2.1.1 *Configuration*

The *configuration* of a connector corresponds to the (abstract) structure that describes the global state of the connector, namely, the memory of the connector and the environment in which the connector is currently being evaluated. Hence a connector configuration is partitioned into two parts: the *internal configuration* and the *external*



(a) degree 2 (b) degree 3 (c) degree 2 (d) degree 1 (e) degree 1

Figure 1: Example of connectors with different degree.

configuration. The internal configuration describes the internal memory of the connector; whereas the external configuration describes the status of the ports of the connector.

2.1.2 *Ports and I/O operations*

Connector ports are the only medium for interacting with a connector, and I/O operations correspond to the well-defined operations that can be performed on a port. A connector can have at most one component connected at each of its ports performing I/O operation requests. A connector, upon the arrival of an I/O operation *request* at one of its ports, has to decide whether the I/O operation can be *fired* or has to be *delayed* because the interaction constraints that the connector imposes are not satisfiable in the present *configuration*. An I/O operation that is being delayed is referred to as a *pending operation*, and a port with a pending operation is referred to as a *pending port*.

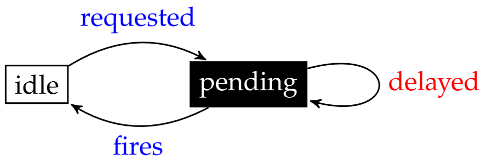


Figure 2: Connector port life-cycle

In figure 2 we have a labelled transition diagram that illustrates the life-cycle of a connector port. The two states denote the two possible configurations of a port: either the port is *idle* or the port is *pending*. The edges indicate the way a port can change its *configuration*. The labels associated with the edges are the (observable) actions that lead to the change of the port configuration. An I/O operation request on an idle port causes the port to change its status to pending. The port is now pending and the connector is responsible to decide whether the I/O operation request is delayed or whether it fires. The blue labels indicate actions that yield *dataflow* on the port and the red labels indicate actions that enforce *no-dataflow* on the port.

2.1.3 *Memory*

Connectors can have *memory*. A connector with memory has the capability of storing data in its buffer cells. One buffer cell has the capability of storing one datum element. A buffer cell has two configurations: *full*, when it contains one datum element, and *empty*, when the buffer cell contains no data. A connector without memory has no capability of storing data. Data input in a memoryless connector cannot be stored in the connector. The data either flows through the connector to another port where it is output or it is lost by the connector.

2.1.4 *Dataflow Behaviour*

The behaviour of a connector is described in terms of the data that flows in and out of its ports, called in this thesis the *dataflow behaviour* of a connector.

We are interested in descriptions of the dataflow behaviour that are discrete in time. The assumption is that the connector configuration can be observed and we can take snapshots of the connector at a pace fast enough to obtain (at least) a snapshot as often as the configuration of the connector changes.

At each time unit the connector performs an *evaluation step*: it evaluates its configuration and according to its interaction constraints changes to another (possibly different) configuration.

A connector can fire multiple ports in the same evaluation step. If multiple ports fire in the same evaluation step we say that the ports fire *synchronously*, regardless of the precise order of their firing and no matter how long the evaluation step actually takes. Hence synchronous means solely that a set of ports fire atomically—in a single indivisible step. We refer to this type of behaviour as *synchronous dataflow behaviour*.

If the set of ports of a connector can be partitioned into two set of ports that never fire together in the same evaluation step we say that any two ports each from one of those sets are *mutually exclusive*. Mutual exclusion therefore means that ports from different sets can never fire together. We refer to this type of behaviour as *asynchronous dataflow behaviour*.

More sophisticated dataflow behaviour can be obtained by allowing a connector to observe and propagate information about pending I/O operations on its ports. In this case we can have dataflow behaviour that depends on the presence or absence of pending operations on the ports of a connector. We refer to this type of behaviour as *context-dependent behaviour*.

2.2 Reo

Reo has been proposed by F. Arbab [6, 7] as an exogenous coordination language based on a calculus of channel composition to construct component connectors. Each connector in Reo represents an interaction protocol that constrains the components that connect to its ports. Reo supports synchronous, asynchronous, and context-dependent dataflow behaviour.

2.2.1 *Channels*

The simplest connectors of degree 2 in Reo are *channels*. Reo does not define any specific channels. Users can define the channel types and their dataflow behaviour. A *channel* in Reo is a medium of communication with exactly two *ends*, and a constraint that defines its interaction protocol through these ends. Reo recognises two types of channel ends: *source* ends, through which data enter channels, and *sink* ends through which data come out of channels. That is all Reo defines about channels. Users define the different channel types and their dataflow behaviour in terms of specific constraints that relate their data exchanges through their respective ends. These constraints define, for example, whether a channel is synchronous or asynchronous, whether or not it has a buffer, whether or not its buffer is bounded, whether or not it retains the order of the data items it receives, whether it loses some of its data, or generates fresh data items, etc. Reo does not even require a channel to have a source and a sink. It is perfectly content with a channel that has two sources or two sinks, with whatever behaviour a user may define for it. Reo supports two I/O operations to perform requests—*write* and *take*—one requests an input from a sink end, and the other requests an output from a source end, respectively[1].

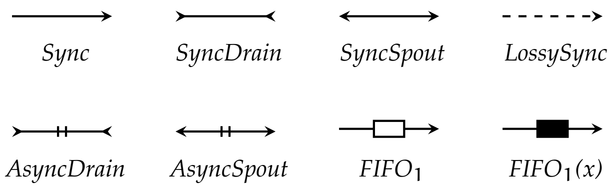


Table 1: Reo channel types.

Table 1 contains the Reo channel types we use throughout this thesis. We provide their formal semantics in the following chapters. At this stage, we give an informal

[1] The terms *source* and *sink* designate the senses of the ends of a channel from the point of view of the channel itself. Obviously, the sense of a channel end must be reversed from the point of a user of a channel, i. e., a component that performs an I/O operation on a channel end. Thus, a component writes to the source end of a channel and takes from the sink end of a channel.

description of their dataflow behaviour. *Sync* denotes a synchronous channel. Data flows through this channel if and only if it is possible to synchronously accept data on one end and pass it out through the other end. *SyncDrain* denotes a synchronous drain. Data flows into both ends of this channel only if it is possible to synchronously accept the data on both ends. *SyncSpout* denotes a synchronous spout. Unspecified data flows out of both ends of this channel only if it is possible to synchronously take the data from both ends. *LossySync* denotes a lossy synchronous channel. If a *take* is pending on the sink end of this channel and a *write* is requested on its source end, then the channel behaves as a synchronous channel. However, if no *take* is pending, the *write* fires, and the data is lost. Observe that this channel has *context-dependent behaviour*, as it behaves differently depending upon whether there is a *take* pending on its sink end—if it were context independent, the data could be lost regardless of whether its sink end has or does not have a *take* pending. *AsyncDrain* denotes an asynchronous drain. Data can flow into only one end of this channel at the exclusion of data flow at the other end[2]. *AsyncSpout* denotes an asynchronous spout. Unspecified data can flow out of only one end of this channel at the exclusion of data flow at the other end[3]. $FIFO_1$ denotes an empty FIFO with one buffer cell. Data can flow into the source end of this buffer, but no flow is possible at its sink end (since its buffer is empty). After data flow into the buffer cell, it becomes a full FIFO. $FIFO_1(x)$ denotes a full FIFO with one full buffer cell. Data can flow out of the sink end of this buffer, but no flow is possible at the source end (since its buffer is full). After data flow out of the buffer, it becomes an empty FIFO.

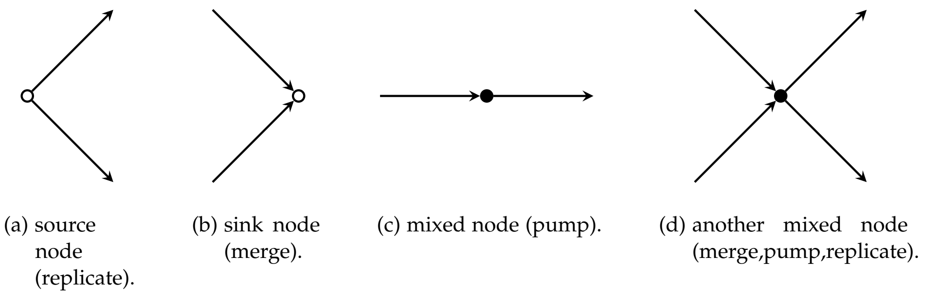
2.2.2 *Nodes*

More complex connectors can be constructed out of simpler ones through connector composition. Initially, every single channel end is a singleton *node*. Channels are composed by conjoining their ends to form *nodes* with multiple channel ends. A node may contain any number of channel ends. Nodes are classified into three different types depending on the types of their coincident ends: a *source node* contains only source channel ends; a *sink node* contains only sink channel ends; and a *mixed node* contains both kinds of channel end. Figure 3 depicts examples of ℛeo nodes. We also refer to mixed nodes as *internal nodes* and to sink and source nodes as *boundary nodes*. Internal nodes are represented by black filled circles • and boundary nodes are represented by white filled circles ∘. Components can only perform write operations on source nodes, and take operations on sink nodes, but no I/O operation is allowed on a mixed node.

[2] The term *asynchronous drain* is perhaps a bit of a misnomer for the behaviour of this channel. *ExclusiveDrain* is a more appropriate name, but since all ℛeo literature already uses this name, we also refer to this channel as *AsyncDrain*.

[3] As the counterpart of *AsyncDrain*, *ExclusiveSpout* would be a better name for this channel, see footnote 2.

Note: Connector ports in ℛeo correspond to nodes. In the context of ℛeo connectors we might interchangebly use both terms.



(a) source node (replicate).

(b) sink node (merge).

(c) mixed node (pump).

(d) another mixed node (merge,pump,replicate).

Figure 3: ℛeo nodes.

A *write* operation to a source node fires only if all source channel ends coincident on the node accept the data item, in which case the data item is written to every source end coincident on the node. A source node thus acts as a *replicator* (Figure 3a). A *take* operation on a sink node fires only if at least one of the sink channel ends coincident on the node offers a suitable data item; if more than one coincident channel end offers suitable data, one is selected non deterministically, at the *exclusion* of all others. A sink node, thus, acts as a merger (Figure 3b). A mixed node behaves, like a self-contained "pumping-station" that combines the behaviour of a sink (merger) and a source node (replicator). A mixed node selects a value through one of its sink ends and replicates it to all of its source ends (Figure 3c and 3d). The subtlety is that nodes have no buffer to hold any data. Therefore, before a mixed node selects a value out of one of its coincident sink ends, it must ensure that this value can be replicated into all of its coincident source ends.

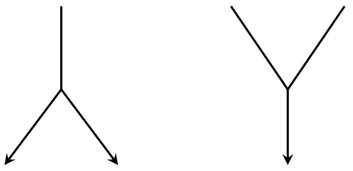
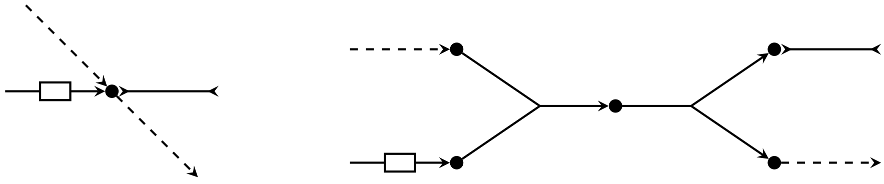


Figure 4: *Replicator* and *Merger* primitive connectors.

To model the behaviour of ℛeo nodes accurately, we make the merge and replicate behaviour that is inherent in ℛeo nodes explicit and, without loss of generality, model them using two additional primitive connectors: a *Replicator* and a *Merger* (Figure 4). The *Replicator* primitive captures the replicator behaviour of a source node with two source ends. Whereas the *Merger* primitive models the behaviour of a sink node with two sink ends. Informally, data will flow through a *Replicator* if it can synchronously accept data on its source end and pass it through both of its sink ends. A merger permits the synchronous flow of data from exactly one of its source ends through its sink end, at the exclusion of flow on its other source end, making a non-deterministic choice between its two source ends if required. Figure 5 illustrates how a mixed node with two source ends and two sink ends (5a) is expressed in terms of the *Merger* and the *Replicator* primitives (5b). We will use the *Merger* and *Replicator* primitives to represent ℜeo nodes when formalising the semantics of ℜeo. In that case internal nodes always connect two channel ends; and boundary nodes always consist of one channel end.



(a) mixed node. (b) mixed node replaced by *Merger* and *Replicator* primitives.

Figure 5: A mixed ℜeo node and its representation with *Merger* and *Replicator*.

Channels as the ones in Table 1 constitute examples of connectors with degree 2. The *Merger* and *Replicator* connectors listed in Figure 4 are examples of connectors of degree 3. In the context of this thesis, connectors of degrees greater than 3 are composite connectors; these are compositionally built out of connectors of degrees 2 and 3. For example the *Barrier Synchroniser* depicted in Figure 6 is a connector of degree 4 compositionally built out of connectors of degrees 2 and 3. The two internal nodes • are a compact representation of two replicator connectors.

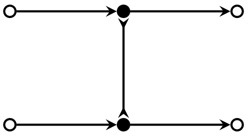


Figure 6: The *Barrier Synchroniser* connector.

2.2.3 *Abstract Components*

From the perspective of a ℜeo connector a component connected to a ℜeo boundary node is an instance of either a *Writer* or a *Taker* component. A *Writer* is an abstract component that has the capability to perform *write* operations. A *Taker* is an abstract component that has the capability to perform *take* operations. From that perspective without loss of generality *Writer* and *Taker* can also be seen as connectors, namely connectors of degree 1, where one has a single source node and the other has a single sink node.

(a) *Writer*. (b) *Taker*.

Figure 7: Abstract components.

2.2.4 *Primitive Connectors*

The connectors of degree 1 *Writer* and *Taker*, the connectors of degree 2 given by the channels of Table 1, and finally the connectors of degree 3 *Replicator* and *Merger*, constitute our set of primitive connectors that we denote by Primitives.

2.2.5 *$\mathcal{R}$eo Operations*

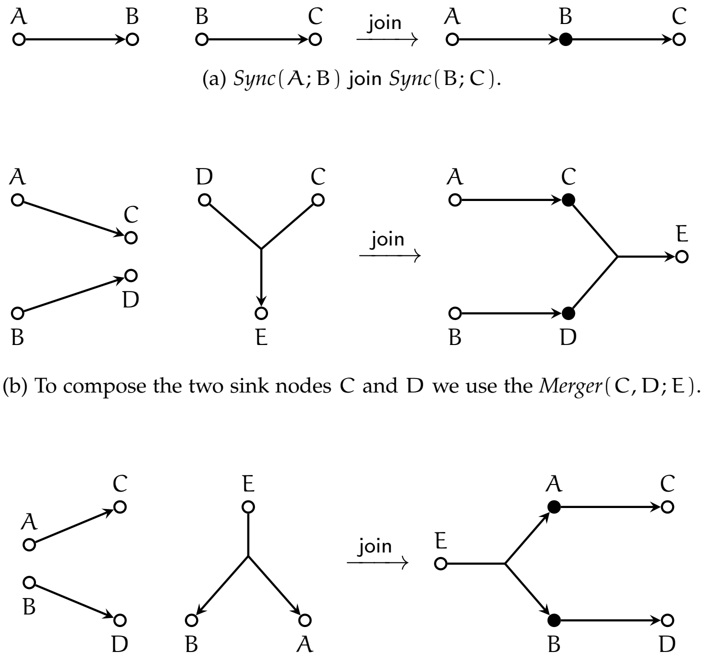
The calculus of connector composition in $\mathcal{R}$eo has two fundamental operations. The operation join that permits to compose connectors, and the operation hide that permits to perform information hiding on connectors.

join

In this thesis _ join _ is a binary operation that takes two $\mathcal{R}$eo connectors and returns another $\mathcal{R}$eo connector that results from composing the boundary nodes that share the same name in both connectors as follows: to compose one sink node with a source node we just conjoin the nodes, as depicted in Figure 8a; to compose two sink (source) nodes we use the *Merger* (*Replicator*) primitive, as depicted in Figure 8b (8c). It is important to note that the result of joining two boundary nodes is an internal node. Hence the resulting node is black.

hide

In this thesis, hide _ is a unary operation that takes one $\mathcal{R}$eo connector and returns another $\mathcal{R}$eo connector that results from removing all the information about the internal nodes of the given connector. In Figure 9a it is depicted how the result of the hide operation is represented visually.

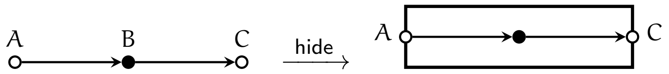


(a) *Sync*(A; B) join *Sync*(B; C).

(b) To compose the two sink nodes C and D we use the *Merger*(C, D; E).

(c) To compose two source nodes A and B we use the *Replicator*(E; B, A).

Figure 8: Composite connectors constructed with the _ join _ operation.



(a) hide (*Sync*(A; B) join *Sync*(B; C)).

Figure 9: Composite connector constructed with the hide _ operation.

2.2.6 *Connector Language:* CONLANG

$\mathcal{R}$eo does not have a fixed set of channels. This means for example that depending on the type of channels that are considered one obtains a different language to build connectors. In this thesis we fix a concrete $\mathcal{R}$eo *setting* as our language to build connectors and we call it CONLANG. CONLANG has a fixed number of primitive connectors denoted by Primitives; two rules to build connectors: $\mathcal{R}$eo operations join and hide; and considers the interaction protocol depicted in Figure 2 as the interaction protocol enforced by each connector port. Following the principle of component-based systems, CONLANG permits us to build arbitrarily complex connectors—the primitives are the building blocks; the $\mathcal{R}$eo operations define the construction rules; and the interaction protocol in a port defines the observations that can be made on a port. CONLANG will be the concrete language of connectors used to illustrate the use of the abstract semantic models introduced throughout this thesis.

Note: The set of channels taken as primitive connectors in CONLANG is in fact one that is commonly used throughout the literature. There is a plethora of $\mathcal{R}$eo connectors built using these channels indicating that a large class of dataflow behaviour can be expressed through composition using these primitives.

2.3 COMPOSITIONALITY OF DATAFLOW BEHAVIOUR

A formal model (semantics) for component connectors in this thesis consists of three parts: an abstract domain S where the elements $s \in S$ are denotations of dataflow behaviours; operations on S that are used to define the semantics of the construction rules to build connectors; and a semantical map

$$[\![\_]\!] : \text{CONLANG} \longrightarrow S$$

that assigns a denotation $s \in S$ to each connector in CONLANG. The map $[\![\_]\!]$ has the following *compositional* property:

$$[\![\mathsf{hide}\ (\mathsf{join}\ c_1\ c_2)]\!] = \exists_S\ ([\![c_1]\!] \times_S [\![c_2]\!])$$

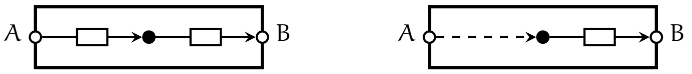
where $\exists_S : S \to S$, and $\times_S : S \times S \to S$ are operations on S.

Once we have the semantical map defined for the primitive connectors and the operations join and hide, we have the semantics of arbitrarily complex composite connectors in the language. Two examples of such composite connectors are depicted in figure 10.

Special care must be taken when defining the semantics of the operations on S, to ensure that the compositional semantics capture the intended behaviour of the composite connectors. For example, according to the informal semantics of $\mathcal{R}$eo [7] the intended dataflow behaviour of the composite connectors in Figure 10 are as follows:

- the composite connector 10a has the behaviour of a FIFO buffer with two memory cells. When the two buffer cells are empty, data can flow two times consecutively through port A. After that, the buffer cells are full. Before data can flow on A again, data must flow through C. Data can flow on C if at least one buffer cell is full, in which case one buffer cell is emptied. The order in which the data flow through C is the same as the order in which data flow through A.

- the composite connector 10b, has dataflow behaviour similar to the *FIFO*$_1$ except that when the buffer cell is full, data can still flow through A, in which case, and only then, the data is lost.



(a) A composite connector built using two *FIFO*$_1$.

(b) A composite connector built using one *LossySync* and one *FIFO*$_1$.

Figure 10: Example of composite connectors.

The compositional semantics provided by the coinductive calculus [9] and the constraint automata [13] capture the intended behaviour of composite connector 10a, but cannot capture the intended behaviour of the composite connector 10b. The reason for the latter is that the intended behaviour of the composite connector 10b depends on the context-dependent behaviour of the *LossySync*, which cannot be captured by these two semantics. In the coinductive calculus and the constraint automata the semantics of the *LossySync* indicate that data written to the source end is non-deterministically lost irrespective of whether or not it can flow through the sink end of the channel.

The composite connector 10b is, by now, the classical example used to illustrate the subtleties that context-dependent behaviour entails. Such subtleties prove that it is non-trivial to formalise a compositional semantics that captures the intended (context-dependent) dataflow behaviour. In Chapters 3 and 5 we introduce semantics models that are compositional and capture the intended (context-dependent) dataflow behaviour.

As a final example, in this case of a larger connector, we introduce the *exclusive router*, depicted in Figure 11. An *exclusive router* is built by composing five *Sync*s, two *LossySync*s and one *SyncDrain*. The intended dataflow behaviour of this connector is that data obtained through its input node a is delivered to exactly one of its output

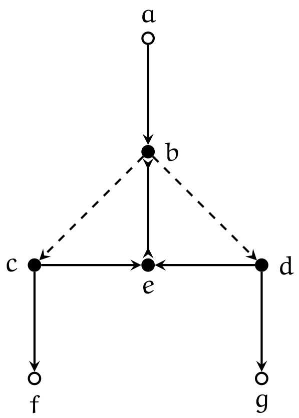


Figure 11: Exclusive router connector.

nodes f or g. If both f and g are willing to accept data, then the merger node e non-deterministically selects which side of the connector will succeed in passing data. The *SyncDrain* and the two *Sync*s in the node e conspire to ensure that data flows at precisely one of c and d, and hence f and g, whenever data flows at b. In this connector, the context-dependent behaviour of the *LossySync* is not essential to ensure that data is routed through either f or g. If the *LossySync* semantics is that it can lose its data non-deterministically, the *SyncDrain* ensures that if either *LossySync* decides to lose the data remains in a.

Note: The research concerning $\mathcal{R}$eo coordination language goes far beyond the study of compositional semantics for the dataflow behaviour of connectors. For instance it includes the study of timed behaviour [14, 62], probabilistic models [17], stochastic models [18], model checking [56, 63], and verification [20], quality of service [16, 61], dynamic reconfiguration [66, 67, 68], and modelling of biological systems [34]. These topics are out of the scope of this thesis. The interested reader can find detailed treatment of each of these topics in the cited references.

3

CONNECTOR COLOURING MODEL

In this chapter we present our first model for the dataflow semantics of $\mathcal{R}$eo connectors. This semantics is based on *connector colouring* for determining the dataflow behaviour of a $\mathcal{R}$eo connector by resolving its *context-dependent synchronisation* and *mutual exclusion* constraints. Colouring a $\mathcal{R}$eo connector in a specific state with given boundary conditions (I/O requests) provides a means to determine the routing alternatives for dataflow.

3.1 OVERVIEW

This chapter is based on joint work [35, 36] with Dave Clarke and Farhad Arbab. These papers introduce connector colouring and cover two fundamental features: synchronisation and context-dependency. We start this chapter presenting the same two features but in a different setting, namely we consider the temporal behaviour of connectors instead of just the one step behaviour. In a setting where the temporal behaviour is taken into account the model indicates at each step which colouring table holds the colouring to apply. The temporal behaviour of $\mathcal{R}$eo connectors was left implicit in the original presentation. Next, we introduce two other important features that were left as future work in those papers:

- a solution to detect and invalidate *causality loops* emerging from ill-formed connectors;
- the definition of the $\exists[\_]\,\_$ operation on connector colouring models to provide semantics to the $\mathcal{R}$eo hide operation.

The connector colouring model with 3 colours differentiates the possible behaviour of a connector at a finer level of granularity than the previous $\mathcal{R}$eo models [8, 13] do. By considering the context (the presence or absence) of pending I/O operations at the boundary nodes of a connector to determine the set of its actual behaviour alternatives, connector colouring permits to capture context-dependent behaviour and therefore provide semantics to context-dependent connectors.

The composition operation in connector colouring $\_ \times \_$ has a number of formal properties, namely, associativity, commutativity, and idempotency. These properties

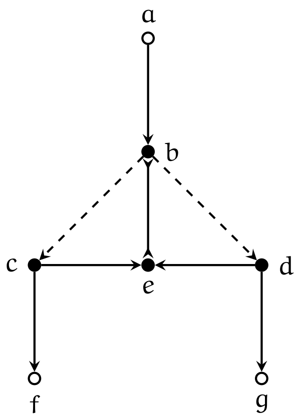


Figure 12: Exclusive router connector.

make it quite suitable for a distributed implementation. Compared to less formal implementation schemes [44, 7] that require history computations and backtracking to resolve various cycles in synchronous segments of a $\mathcal{R}$eo connector, our model requires less mutual exclusion in a distributed implementation, does not require backtracking, and allows concurrent parties to combine each other's partially computed results. The connector colouring model with 3 colours serves as the basis for an actual distributed implementation of $\mathcal{R}$eo.

3.1.1 *Chapter structure*

We introduce connector colouring in Section 3.2 and start by modelling the *synchronous propagation of constraints* using 2 colours. In Section 3.4 we change the set of colours and use 3 colours to accurately define the semantics of context-dependent connectors. We proceed by refining connector colouring with 3 colours to its constructive subset in Section 3.5 to tackle causality loops. In Section 3.6 we define the hide operation. An outline of an existing implementation and an approach for making it distributed are given in Section 3.7. In Section 3.8 we discuss related work.

3.2 CONNECTOR COLOURING

We illustrate through an example how we can visually describe the dataflow behaviour of a $\mathcal{R}$eo connector, and therefore motivate the upcoming notion of connector colouring. We consider the *exclusive router* that we introduced in Section 2.3. The connector is depicted in Figure 12 for convenience. Remember that the intended dataflow behaviour of this connector is that data obtained through its input node

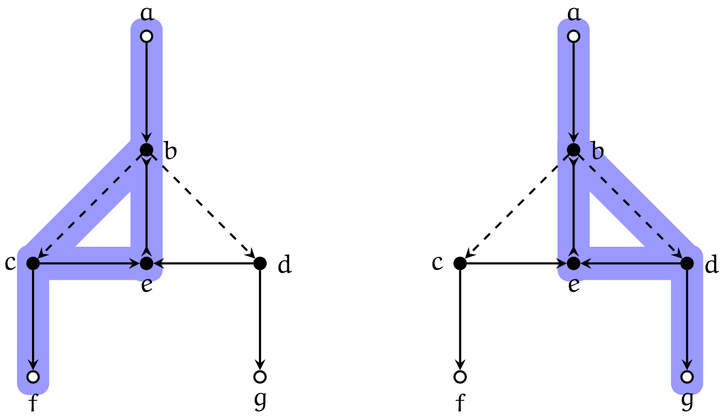


Figure 13: Possible dataflow behaviour of the exclusive router. The thick highlighted lines mark the part of the connector where data flow synchronously. In the other parts no data flows.

a is delivered to exactly one of its output nodes f or g. If both f and g are willing to accept data, then the merger node e non-deterministically selects which side of the connector will succeed in passing data. The *SyncDrain* and the two *Sync*s at the node e conspire to ensure that data flows at precisely one of c and d, and hence f and g, whenever data flows at b. An informal, graphical way of depicting the possible dataflow through the exclusive router is by colouring where data flows, as illustrated in Figure 13, where the highlighted lines mark the parts of the connector where data flow and the other parts correspond to the parts where no data flow. Note that the colour abstracts away from the direction of dataflow, as (the ends of) channels themselves determine this. This idea of colouring underlies our model and provides a visual representation that parallels $\mathcal{R}$eo's visual syntax for connectors.

3.2.1 *Colours*

Our model is based on the idea of marking places where data flows and places where data does not flow by colours. Let $\mathfrak{Colour}$ denote the set of colours. A reasonable minimal set of colours is $\mathfrak{Colour} = \{$'—— ', '- - - - '$\}$, where the colour '—— ' marks places in the connector where data flows, and the colour '- - - - ' marks the absence of data flow.

$\mathcal{R}$eo semantics dictates that data is never stored or lost at nodes [7]. Connector colouring models consider nodes with at most two ends attached, and use the *Replicator* and the *Merger* to model the behaviour of $\mathcal{R}$eo nodes with multiple ends attached (see Section 2.2.2). Thus, the state of dataflow at one end attached to a node must be the same as at the other end attached to the node. Either data will flow out of one end, through the node, and into the other end, or there will be no dataflow at all. Hence, the two ends plugged together into a node will be given the same colour, and thus we just colour the node. Colouring nodes determines the colouring of their attached ends, which in turn determines the colouring of the connector, and thus the data flow through the entire connector. Colouring all the nodes of a connector, in a manner consistent with the colourings of its constituents, produces a valid description of dataflow through the connector. Channels and other primitive connectors then determine the actual dataflow based on the colouring of their ends. With two colours, each colouring of a connector is a solution to the synchronisation constraints imposed by its channels and nodes.

The following definition formalises the notion of a colouring of the underlying graph of a connector. We consider a denumerable set of node names denoted $\mathcal{N}ode$. We use small letters $a, b, c, d, \ldots$ and $n_1, \ldots, n_k$ with $k \in \mathbb{N}$ to denote elements of $\mathcal{N}ode$, and capital letters $X, Y, Z, \ldots$ as variables that range over $\mathcal{N}ode$.

Definition 3.2.1 (Colouring)**.** A *colouring* $c : N \to \mathcal{C}\mathsf{olour}$ for $N \subseteq \mathcal{N}ode$ is a function that assigns a colour to every node in N.

Notation 3.2.2. We use the notation $c : \{n_1 \mapsto c_1, ..., n_k \mapsto c_k\}$, to comprehensively enumerate a colouring $c : N \to \mathcal{C}\mathsf{olour}$, with $N = \{n_1, ..., n_k\}$ and $c(n_i) = c_i$ for $1 \leqslant i \leqslant k$. We refer as 2-colouring to a colouring defined into the set $\mathcal{C}\mathsf{olour} = \{$'—— ', '- - - - '$\}$.

Consider a connector with a single channel, a *FIFO*$_1$, connecting an input node n_1 and an output node n_2. One of the possible colourings of this connector is $c_1 : \{n_1 \mapsto$ ——, $n_2 \mapsto$ - - - -$\}$, which describes the situation where data flows through the input node n_1 and no data flows through the output node n_2.

3.2.2 *Colouring table*

The primitive elements that constitute connectors, channels, mergers, replicators and I/O operations, typically have multiple possible colourings to model the alternative ways that they can behave in the different contexts in which they can be used. Furthermore, a connector can have states and behave differently in each state. The collection of possible colourings for each state of a connector is represented by a *colouring table*. A single colouring table captures the full behaviour of a stateless connector, whereas for stateful connectors we have a colouring table for each state.

Definition 3.2.3 (Colouring table)**.** A *colouring table*, T, over nodes $N \subseteq \mathcal{N}ode$ is a set of colourings with domain N.

Colouring a connector involves composing the colourings of its constituents so that they agree on the colour of their common nodes, as depicted in Figure 14. To capture this notion, we define the binary operation, $\_ \cdot \_$ to compose colouring tables.

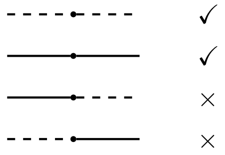


Figure 14: The valid compositions out of the possible combinations of 2-colourings.

Definition 3.2.4 (Colouring table composition)**.** The composition of two tables T_1 and T_2, denoted $T_1 \cdot T_2$, is defined as:

$$T_1 \cdot T_2 \doteq \{c_1 \cup c_2 \mid c_1 \in T_1, c_2 \in T_2, \\ n \in \mathrm{dom}(c_1) \cap \mathrm{dom}(c_2) \Rightarrow c_1(n) = c_2(n)\}.$$

Here $c_1 \cup c_2$ is the set-theoretic union on the graphs of the functions c_1 and c_2:

$$(c_1 \cup c_2)(n) = \begin{cases} c_1(n) & \text{if} \quad n \in \mathrm{dom}(c_1) \text{ and } n \notin \mathrm{dom}(c_1) \\ c_2(n) & \text{if} \quad n \in \mathrm{dom}(c_2) \text{ and } n \notin \mathrm{dom}(c_2) \end{cases}$$

The result $c_1 \cup c_2$ is a function, because for nodes in the intersection of the domains both colourings $\mathrm{dom}(c_1)$ and $\mathrm{dom}(c_2)$, due to the side condition, take the same value. Important to notice additionally is that the resulting colouring $c = c_1 \cup c_2$, admits a unique decomposition in c_1 and c_2, by restricting c to $\mathrm{dom}(c_1)$ we obtain c_1, and similarly for c_2.

The colouring table composition operation $\_ \cdot \_$ satisfies the following useful properties, where $1 = \{\emptyset\}$ is the colouring table with an empty colouring and $0 = \emptyset$ is the empty colouring table.

Proposition 3.2.5. *Given colouring tables* T, T_1, T_2, T_3 *then:*

1. $T_1 \cdot (T_2 \cdot T_3) = (T_1 \cdot T_2) \cdot T_3$ *(associativity)*
2. $T_1 \cdot T_2 = T_2 \cdot T_1$ *(commutativity)*
3. $T_1 \cdot T_1 = T_1$ *(idempotency)*
4. $T \cdot 1 = 1 \cdot T = T$ *(unit)*
5. $T \cdot 0 = 0 \cdot T = 0$ *(zero)*

A consequence of these properties is that the composition operation for colouring table can form the basis of a distributed algorithm: associativity and commutativity allow colouring tables to be computed in any order, and idempotency enables the smooth handling of redundantly computed information, such as when two different concurrent computations of a colouring reach the same part of a connector.

3.2.3 *next* function

A colouring table for a $\mathcal{R}$eo connector describes a behaviour alternative in a particular *configuration* (or snapshot) of the connector, which depends on its state and the presence or absence of I/O requests at its boundary. A colouring prescribes a *possible step* of the connector based on its *present configuration*. As a consequence of performing a step according to a prescribed colouring, the connector may or not change state and respectively change the colouring table applicable in the next step of the connector. For each colouring the *next* function determines the colouring table to apply in the next step.

Definition 3.2.6 (*next* function)**.** Given a finite set I, and an I-indexed set of colouring tables over nodes N, $\{T_1, \ldots, T_i \mid i \in I\}$, denoted $S_I(N)$. A *next* function over $S_I(N)$ is a function of type $\bigcup S_I(N) \rightarrow S_I(N)$, where $\bigcup S_I(N)$ denotes the set $\{(i, c) \mid c \in T_i, T_i \in S_I(N)\}$.

Note that different colouring tables may contain the same colouring, in which case the index in I is necessarily different.

Notation 3.2.7. Consider a *next* function η over $S_I(N)$, and $i, j \in I$. Whenever the context is clear about the colouring and the colouring table in question, we write S instead of $S_I(N)$, and $\eta(c) = T$ instead of $\eta(i, c) = T_j$.

For a stateless connector the set S contains a single colouring table T, and the *next* function is the constant function $\underline{T}$, $\underline{T}(c) = T$, for all $c \in T$. After performing a colouring in T the connector does not change state and therefore the colouring table T is valid in the next step.

A stateful connector requires a non-singleton set S. The size of S is given by the number of states of the connector, thereby S contains one colouring table for each state of the connector. In a given state modelled by a table $T_i \in S$ the connector performs one of the colourings $c \in T_i$. The *next* function determines what is the colouring table applicable in the next state: $\eta(\langle i, c \rangle)$.

For deriving the *next* function of a composite connector we compose the *next* functions from the constituent primitives.

Definition 3.2.8 (*next* function composition)**.** Given the indexed sets of colouring tables $K_L(N_1)$, and $R_J(N_2)$, consider the *next* functions $\eta_1 : \bigcup K_L(N_1) \rightarrow K_L(N_1)$ and $\eta_2 : \bigcup R_J(N_2) \rightarrow R_J(N_2)$. The composition of η_1 and η_2 denoted $\eta_1 \otimes \eta_2$, is a *next* function $\eta : \bigcup S_I(N) \rightarrow S_I(N)$ where:

1. $I = L \times J$, is given by the cartesian product of indexing sets L and J;
2. $N = N_1 \cup N_2$;
3. $S_I(N) = \{T_{\langle l,j \rangle} \mid T_{\langle l,j \rangle} = T_l \cdot T_j,\ T_l \in K_L(N_1),\ T_j \in R_J(N_2)\}$;

4. $\eta(\langle l, j\rangle, c_1 \cup c_2) = \eta_1(\langle l, c_1\rangle) \cdot \eta_2(\langle j, c_2\rangle)$, for all $(\langle l, j\rangle, c_1 \cup c_2) \in \bigcup S_I(N)$.

By Definition 3.2.4 (Colouring table composition), each element in the I-indexed set $S_I(N)$ is indeed a colouring table over N. By Definition 3.2.6 (*next* function) $\eta_1(\langle l, c_1\rangle) \in K_L(N_1)$ and $\eta_2(\langle j, c_2\rangle) \in R_J(N_2)$ therefore by point 3. above, $\eta_1(\langle l, c_1\rangle) \cdot \eta_2(\langle j, c_2\rangle)$ is indeed in $S_I(N)$. Hence η is indeed by definition a *next* function over $S_I(N)$.

3.3 2-COLOURING: SYNCHRONISATION

The set of all the *next* function over indexed sets of colouring tables defines the colouring semantics domain. The colouring semantics domain is parameterised on the set $\mathfrak{C}\mathsf{olour}$. We call *2-colouring domain* the colouring domain that considers $\mathfrak{C}\mathsf{olour} = \{$'——— ','- - - - '$\}$.

3.3.1 *Primitives*

We now provide a formal definition of a *primitive* and define the 2-colouring semantics for the primitives in CONLANG.

Definition 3.3.1 (Primitive)**.** A *primitive* is a labelled tuple

$$(n_1^{j_1}, \ldots, n_k^{j_k})_p$$

where for $0 < l \leqslant k$, $n_l \in \mathcal{N}ode$, $j_l \in \{i, o\}$, $k \geqslant 1$ is the arity of the primitive, and p is the name of the primitive, such that for any n_q and n_r in $(n_1^{j_1}, \ldots, n_k^{j_k})_p$, $q \neq r \Rightarrow n_q \neq n_r$.

The semantics of a primitive P is denoted by a triple $\langle P, S_I(N), \eta\rangle$ where $S_I(N)$ is an I-indexed set of colouring tables over a set of nodes N, and η is a *next* function defined over $S_I(N)$.

The superscript labels i or o of a node n indicate the direction of the end which is connected to n. For example, $(a^i, b^o)_{Sync}$ denotes a *Sync* whose first end is an input end connected to node a, and whose second end is an output end connected to node b. A colouring table for this primitive has colourings with domain $\{a, b\}$. Labels i and o help ensure that connectors are well-formed (Definition 3.3.2). We often omit such labels, tacitly assuming the well-formdness of connectors.

I/O operations

At a boundary node an I/O operation can either be present or absent. When present we say that the boundary node has an I/O request. The colouring of a boundary node with an I/O request in a given step denotes whether the I/O request will be

	T_1	
■	c_1 : ■——	c_2 : ■- - -

(a) 2-colouring table of an I/O operation present.

	T_0
□	c_1 : □- - -

(b) 2-colouring table of an I/O operation absent.

Table 2: 2-colouring tables for I/O operations.

fired or delayed in that step. If the I/O operation is absent we say the boundary node has no I/O request, and in this case the colouring denotes that no data flows.

We model an I/O operation as a primitive $((n^j)_{I/O}, \{T_1, T_0\}, \eta)$, where $j \in \{i, o\}$. It has two 2-colouring tables: $T_1 = \{c_1 : \{n \mapsto \text{——}\}, c_2 : \{n \mapsto \text{- - - -}\}\}$ for when an I/O request is present; and $T_0 = \{c_1 : \{n \mapsto \text{- - - -}\}\}$ for when an I/O request is absent. Both colouring tables are depicted graphically in Table 2. Colouring c_1 in table T_1 is the only colouring that triggers a state change, therefore, we define the *next* function η as follows: $\eta(\langle 1, c_1 \rangle) = T_0$, $\eta(\langle 1, c_2 \rangle) = T_1$, and $\eta(\langle 0, c_1 \rangle) = T_0$.

The colouring tables of the I/O operation model the *Writer* and *Taker* generic components. The colouring table T_1 captures the possible behaviours when an I/O request is made by a component: either the I/O operation succeeds and data will flow or the connector will not allow it to succeed and data does not flow. The colouring table T_0 captures the possible behaviour when no I/O operation is requested on a node by a component: no data flows through that node.

Replicator and Merger

The behaviour of *Replicator* and *Merger* is dictated by the semantics of $\mathcal{R}$eo nodes. $(a^i, b^o, c^o)_{Replicator}$ and $(a^i, b^i, c^o)_{Merger}$ are two stateless primitives. Their single 2-colouring tables are given in Tables 3 and 4. The *next* functions are given by the constant functions $\underline{T_{Replicator}}$ and $\underline{T_{Merger}}$ respectively.

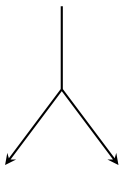
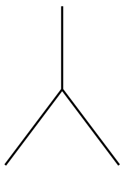
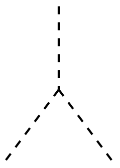
	$T_{Replicator}$			
	c_1 :		c_2 :	

Table 3: 2-colouring table for *Replicator*.

A *Replicator* only allows data to flow synchronously through all of its ends or none at all. When data flows, the data is replicated from a to b and c.

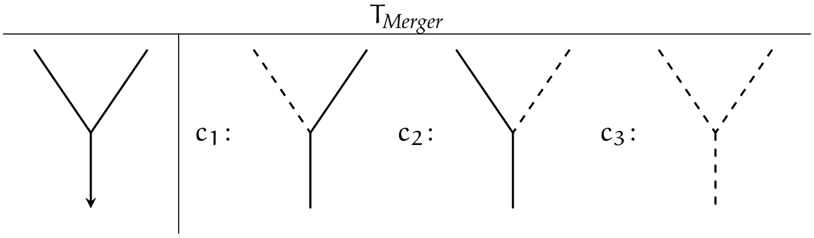


Table 4: 2-colouring table for *Merger*.

A *Merger* allows data to flow synchronously from either a to c or from b to c with the exclusion of data flow at the other end. If both alternatives are possible, one is selected non-deterministically.

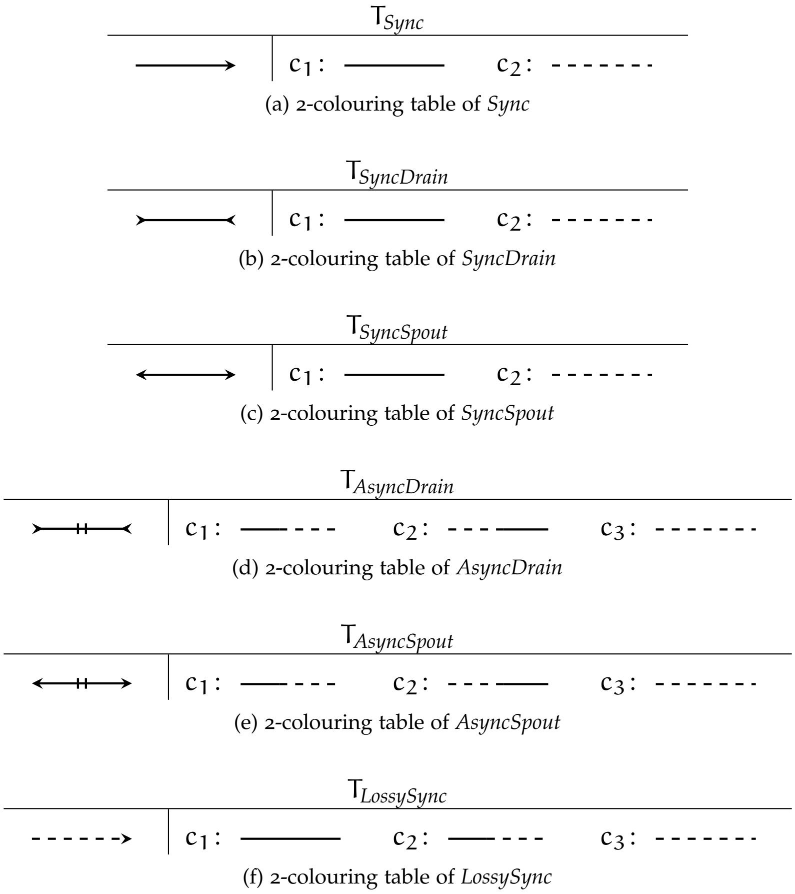
Channels

Table 5 gathers the 2-colouring tables for the selection of stateless channels in this thesis. Their *next* functions are given by the constant function that returns their respective (single) colouring tables. As each channel connects two nodes, there are two colours (which may be identical as in colouring c_1 of T_{Sync}) for each channel colouring.

Channels that are completely synchronous, such as $(a^i, b^o)_{Sync}$, $(a^i, b^i)_{SyncDrain}$, and $(a^o, b^o)_{SyncSpout}$, have the property that either data flows synchronously at both of their ends or no data flows—abstracting away from the direction of data flow. Data flows from one end to the other through the *Sync*, flows into both ends of a *SyncDrain*, and flows out of both ends of a *SyncSpout*, as indicated by the arrows in the diagrams. The $(a^i, b^o)_{LossySync}$ permits data to flow either all the way through the channel, or just at its input end (in which case, the data is lost), or no data flow. (This is not the whole story. We revisit this channel in Section 3.4.) The asynchronous channels, $(a^i, b^i)_{AsyncDrain}$ and $(a^o, b^o)_{AsyncSpout}$, permit data flow at one end at a time only or no data flow at all. The data flow direction is analogous to their synchronous counterparts.

$(a^i, b^o)_{FIFO_1}$ is a stateful primitive. It has two states and therefore two colouring tables: T_0 when its buffer is empty, and T_1 for when its buffer is full. These are depicted in Table 6. The colourings c_1 in table T_0 and c_1 in table T_1 are responsible for state changes. The *next* function for this channel, η, is defined as follows: $\eta(\langle 0, c_1 \rangle) = T_1$, $\eta(\langle 0, c_2 \rangle) = T_0$, $\eta(\langle 1, c_1 \rangle) = T_0$, and $\eta(\langle 1, c_4 \rangle) = T_1$.

An empty *FIFO*$_1$ can accept data on its input end. A full *FIFO*$_1$ can deliver data out of its output end. The other ends of these channels permit no data flow. The *FIFO*$_1$ represents the simplest example of how the buffer content affects the dataflow behaviour.



(a) 2-colouring table of *Sync*

(b) 2-colouring table of *SyncDrain*

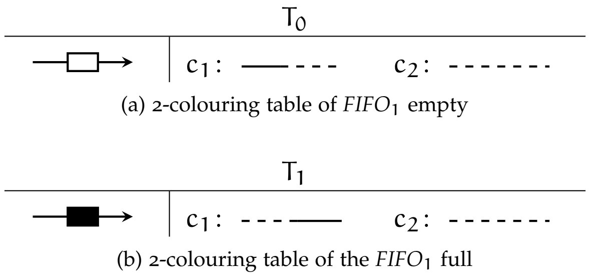
(c) 2-colouring table of *SyncSpout*

(d) 2-colouring table of *AsyncDrain*

(e) 2-colouring table of *AsyncSpout*

(f) 2-colouring table of *LossySync*

Table 5: 2-colouring tables for the stateless channels.



(a) 2-colouring table of $FIFO_1$ empty

(b) 2-colouring table of the $FIFO_1$ full

Table 6: 2-colouring tables for $FIFO_1$.

3.3.2 *Component connectors*

A connector is a collection of primitives composed together, complying with structural conditions. A connector has by definition a non-empty set of boundary nodes. These are the nodes and the only nodes that allow components or other connectors to exchange data with the connector.

Definition 3.3.2 (Connector denotation)**.** A component connector C is denoted by a tuple $\langle N, B, E, S_I(N), \eta \rangle$, where:

- N is a set of nodes, where all the nodes in N appear in E;
- $\emptyset \neq B \subseteq N$ is the set of boundary nodes;
- E is a set of primitives;
- $S_I(N)$ is an I-indexed set of colouring tables over N, where I is the set of colouring table names;
- η is a *next* function over $S_I(N)$.

A primitive can straightforwardly be considered as a connector. A primitive corresponds to a connector where all its nodes are boundary nodes.

Proposition 3.3.3 (Primitive connector)**.** *Consider a primitive* $(n_1^{j_1}, \dots, n_k^{j_k})_p$ *denoted by*

$$\langle (n_1^{j_1}, \dots, n_k^{j_k})_p, S_I(N), \eta \rangle.$$

Primitive P can be naturally denoted as a connector $P = \langle N, B, E, S_I(N), \eta \rangle$*, where* $N = B = \{n_1, \dots, n_k\}$*,* $E = \{(n_1^{j_1}, \dots, n_k^{j_k})_p\}$*, and vice-versa.*

For example the primitive channel *Sync* denoted by $\langle (a^i, b^o)_{Sync}, \{T_{Sync}\}, \underline{T_{Sync}} \rangle$ is modelled with the connector model

$$Sync = \langle \{a, b\}, \{a, b\}, \left\{ (a^i, b^o)_{Sync} \right\}, \{T_{Sync}\}, \underline{T_{Sync}} \rangle.$$

More complex connectors are obtained by composing existing connectors. The colouring tables (*next* function) of a connector is computed from the colouring tables (*next* functions) of its primitive constituents.

Definition 3.3.4 (join operation)**.** Consider connectors $C_1 = \langle N_1, B_1, E_1, K_L(N_1), \eta_1 \rangle$ and $C_2 = \langle N_2, B_2, E_2, R_J(N_2), \eta_2 \rangle$ such that $(N_1 \setminus B_1) \cap (N_2 \setminus B_2) = \emptyset$, and for each $n \in B_1 \cap B_2$, n^i appears in E_1 and n^o appears in E_2, or vice versa. join (C_1, C_2) is denoted by $C_1 \times C_2$, where:

$$C_1 \times C_2 = \langle N_1 \cup N_2, (B_1 \cup B_2) \setminus (B_1 \cap B_2), E_1 \cup E_2, S_I(N), \eta_1 \otimes \eta_2 \rangle$$

with

$$S_I(N) = \{T_{\langle l,j \rangle} \mid T_{\langle l,j \rangle} = T_l \cdot T_j,\ T_l \in K_L,\ T_j \in R_J\}.$$

Internal nodes of the two connectors, denoted by the sets $(N_1 \setminus B_1)$ and $(N_2 \setminus B_2)$, must be disjoint $(N_1 \setminus B_1) \cap (N_2 \setminus B_2) = \emptyset$, and a common node n between the two is a connectors boundary nodes $n \in B_1 \cap B_2$. The colouring tables and *next* functions are composed using the respective composition operation, '$\cdot$', and '$\otimes$' respectively.

Operation $\times$ is partial and, when defined, is associative and commutative with the empty connector $1 = \langle \emptyset, \emptyset, \emptyset, \emptyset, \underline{\emptyset} \rangle$ as identity.

Proposition 3.3.5. $C_1 \times C_2$ *if defined it defines a connector (Definition 3.3.2).*

3.3.3 *Example: composite connector*

We illustrate how the connector model of a composite connector is calculated compositionally from its primitives by means of an example. Let us consider the connector C depicted in Figure 15.

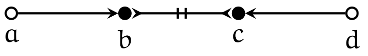


Figure 15: Connector C.

We consider the channels that constitute the connector as connectors and denote them as C_1, C_2, and C_3.

$$\begin{aligned} C_1 = \langle \{a,b\}, \{a,b\}, \left\{(a^i, b^o)_{Sync}\right\}, \{T_1 = \{&c_1 : \{a \mapsto \text{——}, b \mapsto \text{——}\},\\ &c_2 : \{a \mapsto \text{- - - -}, b \mapsto \text{- - - -}\}\},\\ \eta_1 = \underline{T_1}\rangle \end{aligned}$$

$$\begin{aligned} C_2 = \langle \{b,c\}, \{b,c\}, \left\{(b^i, c^i)_{AsyncDrain}\right\}, \{T_2 = \{&c_3 : \{b \mapsto \text{——}, c \mapsto \text{- - - -}\},\\ &c_4 : \{b \mapsto \text{- - - -}, c \mapsto \text{——}\},\\ &c_5 : \{b \mapsto \text{- - - -}, c \mapsto \text{- - - -}\}\},\\ \eta_2 = \underline{T_2}\rangle \end{aligned}$$

$$\begin{aligned} C_3 = \langle \{c,d\}, \{c,d\}, \left\{(c^o, d^i)_{Sync}\right\}, \{T_3 = \{&c_6 : \{c \mapsto \text{——}, d \mapsto \text{——}\},\\ &c_7 : \{c \mapsto \text{- - - -}, d \mapsto \text{- - - -}\}\},\\ \eta_3 = \underline{T_3}\rangle \end{aligned}$$

From the definitions of C_1, C_2 and C_3 we compute C.

$$\begin{aligned} C &= C_1 \times C_2 \times C_3\\ &= \langle \{a,b,c,d\},\ \{a,d\},\ \{(a,b)_{Sync},\ (b,c)_{AsyncDrain},\ (c,d)_{Sync}\},\\ &\quad \{T_1 \cdot T_2 \cdot T_3\},\ \eta_1 \otimes \eta_2 \otimes \eta_3\rangle \end{aligned}$$

The crux of computing $C_1 \times C_2 \times C_3$ is in computing the table $T_1 \cdot T_2 \cdot T_3$. The composed *next* function $\eta_1 \otimes \eta_2 \otimes \eta_3$ equals $\underline{T_1 \cdot T_2 \cdot T_3}$.

First compute $T_1 \cdot T_2$.

$$
\begin{aligned}
T_1 \cdot T_2 &= \{c_1 \cup c_3, c_2 \cup c_4, c_2 \cup c_5\} \\
&= \{\{a \mapsto \text{——}, b \mapsto \text{——}, c \mapsto \text{- - - -}\}, \\
&\quad \{a \mapsto \text{- - - -}, b \mapsto \text{- - - -}, c \mapsto \text{——}\}, \\
&\quad \{a \mapsto \text{- - - -}, b \mapsto \text{- - - -}, c \mapsto \text{- - - -}\}\}.
\end{aligned}
$$

The remaining combinations, namely, c_1 with c_4 or c_5, and c_2 with c_3, are incompatible at node b and thus do not appear in the table.

Finally, compute $(T_1 \cdot T_2) \cdot T_3$.

$$
\begin{aligned}
(T_1 \cdot T_2) \cdot T_3 = \{&\{a \mapsto \text{——}, b \mapsto \text{——}, c \mapsto \text{- - - -}, d \mapsto \text{- - - -}\}, \\
&\{a \mapsto \text{- - - -}, b \mapsto \text{- - - -}, c \mapsto \text{——}, d \mapsto \text{——}\}, \\
&\{a \mapsto \text{- - - -}, b \mapsto \text{- - - -}, c \mapsto \text{- - - -}, d \mapsto \text{- - - -}\}\}.
\end{aligned}
$$

In Table 7 we graphically depict the colouring table of connector C as we just computed. Indeed, this is the preferred way of presenting colouring tables, because they give a pictorial representation of the data flow in a manner which resembles the shape of the connector.

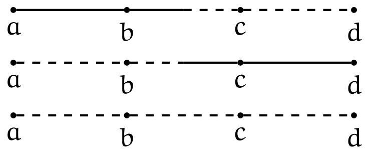


Table 7: 2-colouring table of connector C.

3.3.4 *Dataflow through a connector*

By adding an I/O operation to the boundary nodes of the connector just computed, we have enough context information to determine how the connector can route data. Figure 16 contains the two possible 2-colourings when there is an I/O request (write) on node a and no I/O operation on node d.

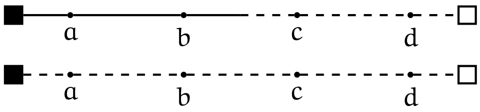


Figure 16: 2-colourings possibilities when there is an I/O request on node a and no I/O operation on node d.

The first entry in the colouring table indicates that the *write* operation fires and data flows at node a and node b. The second entry indicates that the *write* operation is delayed and there is total absence of data flow throughout the connector.

In this example, only the first colouring should be possible in this configuration. The second possibility goes against the intended behaviour, because in case it is picked there will be no data flow, although there is no concrete reason to prevent data to flow. This happens because connector colouring with 2 colours is not sensitive to the context in which a connector appears. We shall see in the next section that this problem has other manifestations, as it arises not only at the boundary of a connector, but also when channels exhibiting context-dependent behaviour appear in the middle of connectors—how can they determine the context in order to behave correctly? Then we extend our colouring scheme to accurately describe context-dependent behaviour and to propagate it through connectors.

3.4 3-COLOURING: SYNCHRONISATION AND CONTEXT DEPENDENCY

In this section we address the issue of context-dependent behaviour. We demonstrate that the 2-colouring scheme applied to a connector involving a *LossySync* fails to give the expected dataflow behaviour. We observed a similar situation in the example at the end of Section 3.3.3. We argue that this occurs because context information is not captured by the semantics and therefore cannot be propagated to enable channels to choose their most appropriated context-dependent behaviour. Previous semantic models of $\mathcal{R}$eo connectors [13, 9] as argued, remain at a coarser level of abstraction and fail to address this issue.

3.4.1 *LossySync channel's inadequacy due to context-dependency*

A *LossySync* has the following context-dependent behaviour (as described in Section 2.2.1). If both a write I/O operation is pending on its input end and a take I/O operation is pending on its output end, then it behaves as a *Sync*—the write and take simultaneously fire, and the data flows through the channel. If, on the other hand, no pending take is present, then the write fires but the data is lost. Limitations of the 2-colouring semantics reveal themselves if we consider the prime example: the connector *LossyFIFO* with an I/O request present on the input end of the *LossySync* while the *FIFO*$_1$ is empty. The connector is depicted in Figure 17. This connector admits two alternative colourings gathered in Table 8.

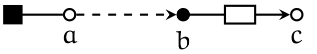


Figure 17: *LossyFIFO* connector with an empty *FIFO*$_1$ and an I/O request at node a.

The first colouring indicates that the I/O operation fires, the data flows through a and that the *LossySync* acts as a *Sync* sending the data through b into the *FIFO*$_1$. This is the expected behaviour in this configuration.

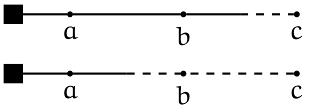


Table 8: 2-colouring table of the *LossyFIFO* in Figure 17.

The second colouring indicates that data flows through node a, but not at node b, indicating that it is lost in the *LossySync*. An empty *FIFO*$_1$ is, however always able to accept data. Another way of seeing this is that an empty *FIFO*$_1$ always fires a *take* to the part of the connector that is connected to its input node. Indeed, the only reason that it should not succeed in receiving data is if the connector connected to its input node gives it a reason not to—such as by not sending it any data. One can therefore interpret the situation as a violation of the intended semantics of the *LossySync* channel, because the information that the data can be accepted on its output end is not appropriately propagated to it. The *LossySync* cannot detect the presence of the pending *take* issued by the input-enabled,empty *FIFO*$_1$ buffer. Similar situations arise when dealing with a *LossySync* in isolation or in the context of other connectors.

The behaviour of a context-dependent connector depends on the presence or absence of I/O requests on its boundary nodes. For mixed nodes, however, no I/O request information is present, so it is not obvious what the context is. The key to resolving this is to determine what context information can be consistently propagated while addressing synchronisation constraints. Rather than propagating only the presence of an I/O request, our approach focuses on propagating their absence too, or more generally, on any reason to delay data flow (until the next state), such as unsatisfiable synchronisation constraints or exclusion constraints issued, for example, by choices made by mergers.

In the next section, we present a 3-colouring semantics which uses colours to propagate *reasons to delay*. Using this scheme the undesirable colouring for the *LossyFIFO* connector, described above can be ruled out due to the mismatch of colours at node b. This means that the possibility of losing data in the *LossySync* is no longer a behaviour of the *LossyFIFO* in the context depicted in Figure 17.

3.4.2 *3-colouring*

To address the problem just described, we modify the set $\mathcal{C}\mathsf{olour}$. As we wish to trace the reason to delay, we replace the colour that denotes no-dataflow by two colours both of which use a dashed line marked with an arrow. The arrow reflects the direction that a reason to delay comes from, that is, it points away from the reason, in the direction that the reason propagates. Thus we now work with colours, $\mathcal{C}\mathsf{olour}$ = {——, - ▷ -, - ◁ -}, we refer to colourings defined into this set as

3-colourings, and to colourings with the no-dataflow colours as no-dataflow colourings. In fact, the colours depend on how the arrow lies in relation to the node being coloured. A no-dataflow colouring with the arrow pointing toward a node, - -▷- -•, means *gives a reason to the node to delay*, and a colouring with the arrow pointing the opposite way, - -◁- -•, means *requires a reason from the node to delay*.[1]

We can compose two no-dataflow colourings at a given node if *at least one* of the colours involved gives a reason to justify no dataflow. Of the four possible combinations of the two no-dataflow colours, three are considered valid combinations, as given in Figure 18.

- -◁ - •- -◁ -	✓	*(one colour giving a reason)*
- -▷ - •- -▷ -	✓	*(one colour giving a reason)*
- -▷ - •- -◁ -	✓	*(two colours giving a reason)*
- -◁ - •- -▷ -	×	*(no colours giving a reason)*

Figure 18: The valid compositions of no-dataflow 3-colourings out of all combinations.

The last entry in Figure 18 is not permitted as it determines no-dataflow at the node without providing a colouring with a reason to delay, actually none of them give a reason.

A disadvantage of the 3-colouring semantics is that its colouring tables often contain superfluous entries. Namely, colourings that provide reason to delay at boundary nodes entail colourings in which a reason to delay is also accepted at these same nodes. The point is that there must be a reason. The following *flip rule* (Definition 3.4.1), reduces tables to their essential colourings. It can be used as a guide for constructing tables for primitives, and it is also used in the implementation to reduce table sizes.

Definition 3.4.1 (The Flip Rule)**.** If a colouring table T has an entry c that maps a boundary node n to the colour - -▷- -• n, that is, gives a reason to exclude, then the colouring that is the same as c except that n is mapped to - -◁- -• n, that is, requires a reason to exclude from n, is redundant and can be removed from the table.

The rationale behind the flip rule is as follows. If a node can provide a reason to delay the request of an I/O operation, then it is perfectly reasonable for it not to bother if such a request is made or not. Indeed we do not want to distinguish these two situations and treat them as semantically independent.

Consider two colourings in a table that differ only in the colour of the boundary node n; that is, one colouring has - -▷- -• n, the other has - -◁- -• n. The idea is

[1] To be precise, the colours also depend on the direction of data flow. Giving a reason to an input node is the same colour as requiring a reason from an output node, and giving a reason to an output node is the same colour as requiring a reason from an input node.

that these two colourings should both compose with a colouring containing colour n •- ◁- - . The resulting colourings should have no dataflow at node n, each marked with a different no-dataflow colour, and be elsewhere identical. Removing the colouring containing - ◁- -• n from the table does not reduce the table's composability with other tables. On the other hand, removing - ▷- -• n does reduce composability. Thus, the colouring containing - ◁- -• n is compositionality-wise *superfluous*, whereas the one containing - ▷- -• n is not.

The treatment of superfluous colourings as semantically equivalent must be reflected in the scheme of composition we defined so far for colouring tables (Definition 3.2.4) and connectors (Definition 3.3.4).

The third entry of Figure 18 is not compatible with the composition operation we defined for colouring tables (Definition 3.2.4). Observe that the flip rule applied to any colouring table T induces a lattice ordered by redundancy. Denote the largest element, the one with the most redundancy, as T^+. In addition, the flip rule induces an equivalence class on tables (take the symmetric, transitive closure of the lattice), which we denote $\equiv$.

We can define a second composition operation which excludes the need to have the third entry in Figure 18 and applies to the maximal tables (T^+). This notion is compatible with previous definitions. Furthermore, the two notions are equivalent in the following sense. We define the new composition operation $\cdot_1$ in terms of the previously defined operation $\cdot$ (Definition 3.2.4).

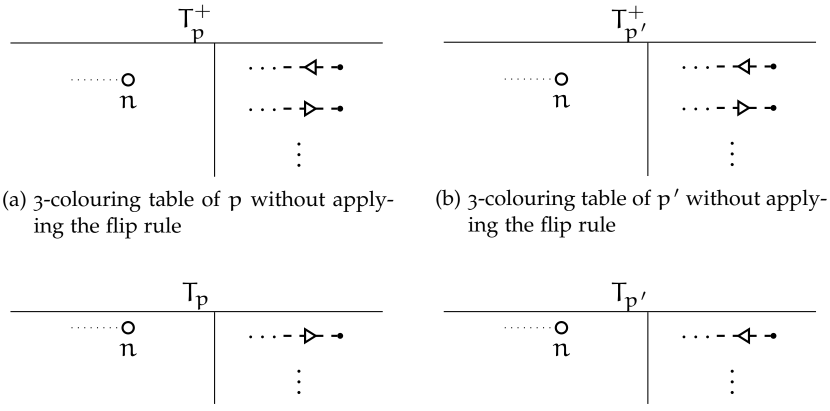
Definition 3.4.2. Given colouring tables T_1 and T_2 reduced with the flip rule, then: $T_1 \cdot T_2 \equiv (T_1^+) \cdot_1 (T_2^+)$. □

This means that we can use whichever notion of composition is most convenient and that we can think in terms of reduced tables using the flip rule.

Example 3.4.3 (Use of The Flip Rule). Consider two stateless primitives $(\dots, n^o)_p$ and $(n^i, \dots)_{p'}$ connected at node n. Primitives p and p' have the colouring tables T_p and $T_{p'}$ partially defined as (a) and (b) in Table 9.

A connector $C = \langle \{n, \dots\}, \{\dots\}, \{p, p'\}, \{T_p \cdot_1 T_{p'}\}, \underline{T_p} \times \underline{T_{p'}} \rangle$ composed out of these two primitives connected at node n has the colouring $c_1 : \cdots$ - ▷- $\overset{n}{\text{-•-}}$ ▷- - $\cdots$ in its colouring table $T_p \cdot_1 T_{p'}$. By expanding the tables using the opposite of the flip rule we obtain T_p^+ and $T_{p'}^+$ depicted in (a) and (b) in Table 9. The flip rule allows to recover the second colouring in table T_p^+ from the first colouring in table T_p. Composing with operation $\cdot$ the second colouring of T_p^+ with the first colouring of table $T_{p'}^+$ we obtain colouring c_1.

Note, that after a composition has been performed, the directions of the arrows on mixed nodes no longer matters: the colouring simply represents no data flow. This fact is used to reduce table sizes.



(a) 3-colouring table of p without applying the flip rule

(b) 3-colouring table of p' without applying the flip rule

(c) 3-colouring table of p applying the flip rule

(d) 3-colouring table of p' applying the flip rule

Table 9: 3-colouring tables of primitives p and p' with and without applying the flip rule.

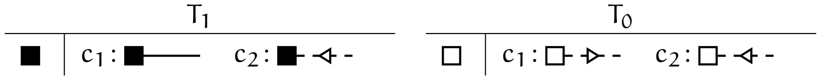
We are now set to define new colouring tables for all the primitives using the colouring domain that considers

$$\mathcal{C}\mathsf{olour} = \{`\text{——}', `\text{-} \rhd \text{-}', `\text{-} \lhd \text{-}'\}$$

called the *3-colouring domain*. The definitions of *next* function and connector model do not need to be changed.

I/O operations

The dataflow behaviour of an I/O operation primitive, $(n^j)_{I/O}$, is denoted by the 3-colouring tables depicted in Table 10, and the *next* function $\eta = \{\langle 1, c_1 \rangle \mapsto T_0, \langle 1, c_2 \rangle \mapsto T_1, \langle 0, c_1 \rangle \mapsto T_0, \langle 0, c_2 \rangle \mapsto T_0\}$.

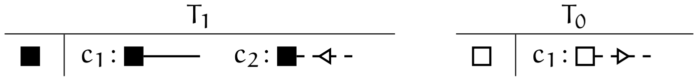


(a) 3-colouring table for an I/O operation present.

(b) 3-colouring table for an I/O operation absent.

Table 10: 3-colouring tables for I/O operations.

Colouring c_1 in table T_1 indicates that the I/O operation request fires and data flows. This is the only colouring that triggers a change of colouring table according to the *next* function η. After firing there is no I/O operation request therefore the



(a) 3-colouring table for an I/O operation present.

(b) 3-colouring table for an I/O operation absent.

Table 11: 3-colouring tables for the I/O operations after applying the flip rule.

colouring table to apply is T_0. Colouring c_2 in table T_1 indicates that the I/O operation request is delayed because the connector gives a reason to delay the request. Both entries in the T_0 table indicate that no data flow is possible. Furthermore, colouring c_1 in table T_0 states that the absence of an I/O request can be used to justify a delay. Colouring c_2 of the same table represents the case where the reason to delay is already present and propagated from the connector to the boundary node.

The one possible colouring missing from the T_1 table, namely similar to c_2 with the arrow going the other way, does not make sense. It would read: there is an I/O request which is a cause of delay. This case is therefore not admissible.

Using the flip rule, the colouring tables for the I/O primitive can be replaced by the colouring tables in Table 11.

The flip rule can be used to recover the colouring missing from table T_0, yielding the colouring c_2 from c_1. From now on, we always use tables that are reduced by the application of the flip rule and we omit from defining the *next* function for colourings that are recovered by application of the flip rule. For such a colouring the *next* function takes tacitly the same value as for the colouring that yields this colouring by application of the flip rule.

Replicators and Mergers

We update the colouring tables for mergers and replicators. The flip rule accounts for all other sensible possibilities.

The new colouring table for a replicator is given in Table 12. The last three entries indicate situations where no data can flow. In each case, a reason to delay coming from one end is sufficient to cause delay in the entire replicator. The reason for delay is propagated to the other ends.

The new colouring table for a merger is given in Table 13. The first two entries in the table deal with choices made by the merger. Data flowing down one input branch is sufficient reason to delay data flow in the other input branch. The third entry corresponds to no take being present at the output end: no data flow is possible in the merger, and the reason to delay is propagated to the input ends. The final entry corresponds to no data flow due to no data availability at either of the two input ends.Again the reason to delay is propagated.

Note that neither colouring table includes an entry with all arrows pointing outward. This would indicate that the reason came from nowhere.

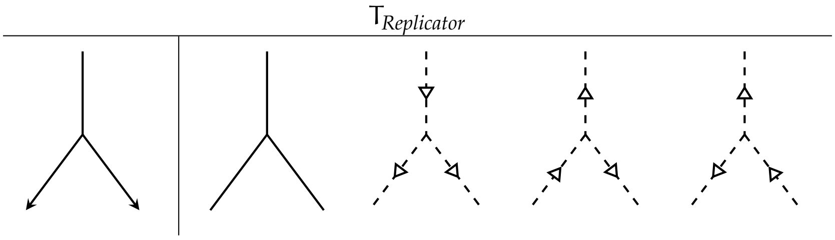


Table 12: 3-colouring table for the *Replicator*.

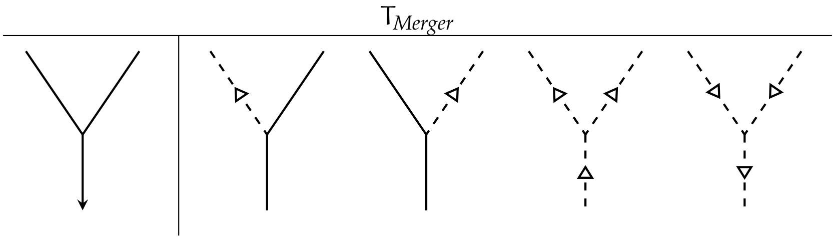


Table 13: 3-colouring table for the *Merger*.

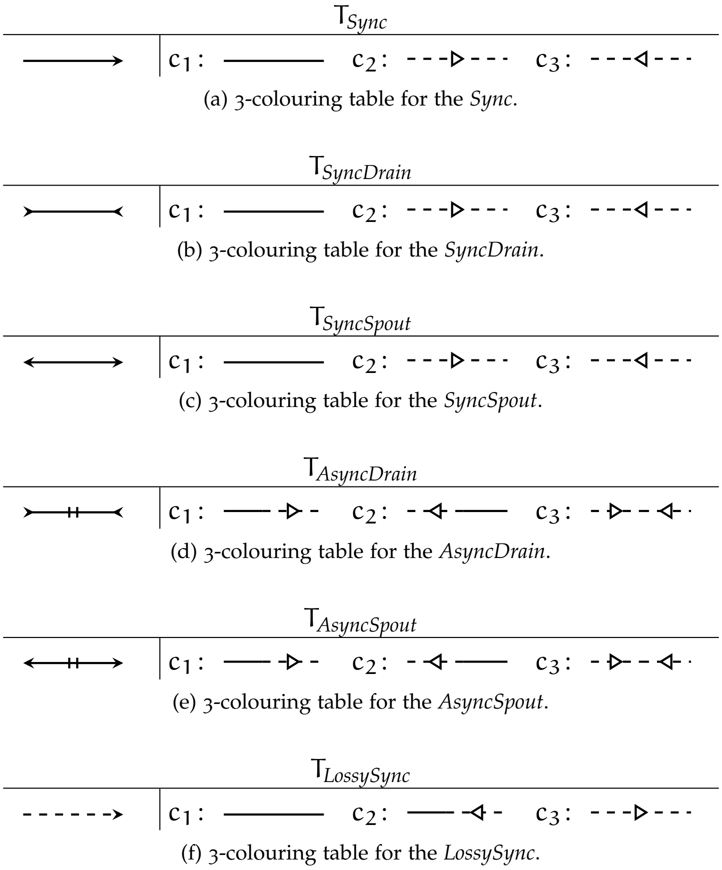
Channels

The 3-colouring tables for stateless channels are gathered in Table 14. The colouring, - - - ▷- - - is shorthand for the colouring - ▷- -- ▷- - , which means that the reason for delay is propagated from one end of the channel to the other. We highlight a few points of interest in this table, focusing only on reasons to delay, leaving the reader to ponder over the rest.

Failure at one node of a *Sync*, *SyncDrain* or *SyncSpout*, is enough to prevent data flow due to the synchrony constraint imposed by the channel between the two nodes. The reason is propagated to the other node. The second entry of the table for a *LossySync* states that it will lose the data only when a reason to delay is propagated into its output node, which amounts to saying that the channel is unable to transfer the data. For the two asynchronous channels, *AsyncDrain* and *AsyncSpout*, accepting data on one node is sufficient reason for delaying the other node. No data flows if both ends have a reason to delay. Note that a non-deterministic choice may be required to decide between the first two possibilities.

The 3-colouring table for the *FIFO*$_1$ channel is presented in Table 15. An empty *FIFO*$_1$ buffer does not enable dataflow on its output node, giving a reason for delay.

Dually, a full *FIFO*$_1$ buffer has a reason to delay its input node. The *next* function is defined as $\eta(\langle 0, c_1 \rangle) = \eta(\langle 1, c_2) = T_1$ and $\eta(\langle 0, c_2 \rangle) = \eta(\langle 1, c_1 \rangle) = T_0$.



(a) 3-colouring table for the *Sync*.

(b) 3-colouring table for the *SyncDrain*.

(c) 3-colouring table for the *SyncSpout*.

(d) 3-colouring table for the *AsyncDrain*.

(e) 3-colouring table for the *AsyncSpout*.

(f) 3-colouring table for the *LossySync*.

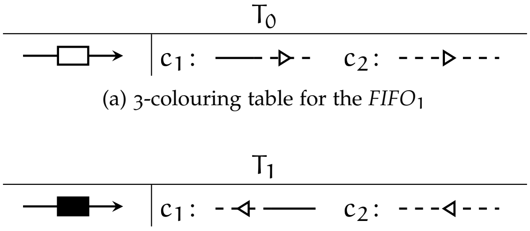
Table 14: 3-colouring tables for the stateless channels.

We now recall the example in Section 3.3.3 and the *LossyFIFO* connector presented in the beginning of this section. In both cases, Figures 19 and 20, we can see that the incorrect 3-colourings alternatives are now respectively ruled out. In the first example (from Section 3.3.3), the colours do not match on node b.

Similarly, in the *LossyFIFO* example, the colours also do not match on node b.

These examples illustrate how the propagation of I/O context in the 3-colouring setting ensures the intended behaviour of context-dependent connectors and channels like the *LossySync*.

Next, we present a further example where we exploit the 3-colouring model to define a connector that provides an elegant solution to express priority behaviour.


T_0
c_1 :
c_2 :
T_1
c_1 :
c_2 :

(a) 3-colouring table for the *FIFO*$_1$

(b) 3-colouring table for the *FIFO*$_1$*(x)*

Table 15: 3-colouring tables for the *FIFO*$_1$.

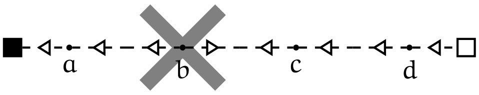


Figure 19: Invalid colouring of the connector C.

3.4.3 *Example: Priority merger*

We introduce a new primitive, a *priority merger*, *PMerger* for short, and use it to model a *priority router*. A $(a^i, !b^i, c^o)_{PMerger}$ behaves similarly to a $(a^i, b^i, c^o)_{Merger}$, allowing flow of data from at most one of its input ends to its output end. The difference is that whenever data is available on both of its input ends, such as when there is a *write* pending on both input nodes, then the merger gives priority to a specific node (marked with an exclamation mark '!'). The graphic representation for the priority merger and its colouring table are presented in Table 16. Compare the first two entries in the table. The first entry means that allowing flow in the right input can give a reason to delay the left input. On the other hand, the second entry means that data can flow from the left input only if a reason to delay is given on the right input. This reason states that no data flow is possible on that input. It is a stateless connector and its *next* function is given by the constant function $\underline{T_{PMerger}}$.

Let us now construct a *priority router*. It behaves like an exclusive router, except that rather than making a non-deterministic choice when each of its output nodes has a pending *take* operation, it makes a choice dictated by the priority merger primitive. The priority router is given in Figure 21, along with the only 3-colouring possible in the configuration where I/O requests are present on all of its boundary

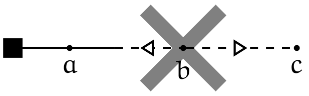


Figure 20: Invalid colouring of the connector *LossyFIFO*.

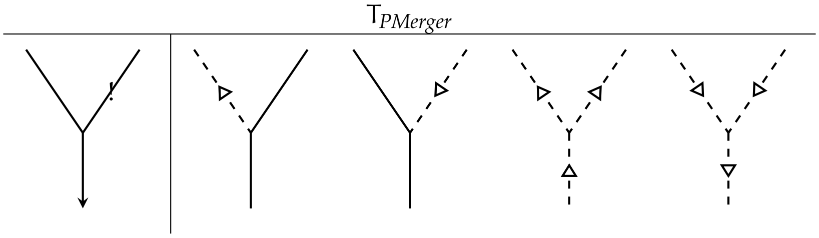


Table 16: 3-colouring table of *PMerger*.

nodes. This means that in the presence of two competing *takes* on the output nodes of the connector, the left hand one (which has priority) will always succeed. This is because the colouring table of the priority merger (appropriately rotated to give priority to the left branch), in the presence of the I/O requests, cannot give a reason for the left hand branch to delay, though it can give a reason for the right hand branch to delay.

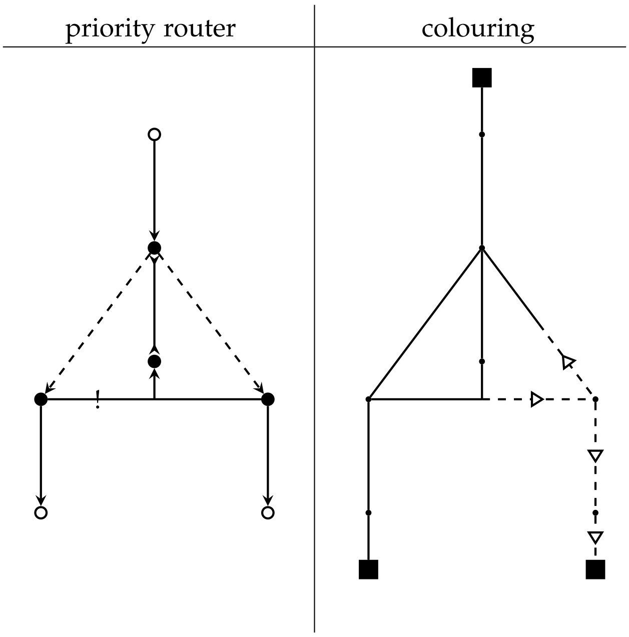


Figure 21: A priority router and one of its colourings exploiting context-dependency.

Note, however, that priority is not globally decided. It may be the case that a decision made by a different part of the connector makes priority irrelevant or even inverts the decision—it all depends upon the connector.

3.4.4 *Causality Loops*

The model presented thus far still produces incorrect colourings for some connectors containing *loops*. Colourings for synchronous loops in a connector tend to result in so-called *causality loops*. These occur whenever a chain of cause-effect events is circular, giving, for example, a colouring that prescribes data to flow, but for which there is no source of data. These anomalous behaviours are a standard problem in synchronous languages [24]. In our setting, the problem is somewhat more complicated, because, not only do we need to consider causality loops that concern data flow, but also causality loops concerning reasons for delay. Both kinds of loops need a source of either data flow or reason to delay to be valid, depending on the kind of the loop. Figure 22 shows examples of the two kinds of loops. (I) a loop for which the colouring (I-c) states that data can flow, even though there is no source providing data: where does the data flowing at c come from? And (II) a loop for which the colouring (II-c) states that there is a reason for delaying, even though there is no source providing a reason: where does the reason to delay observed at a come from?

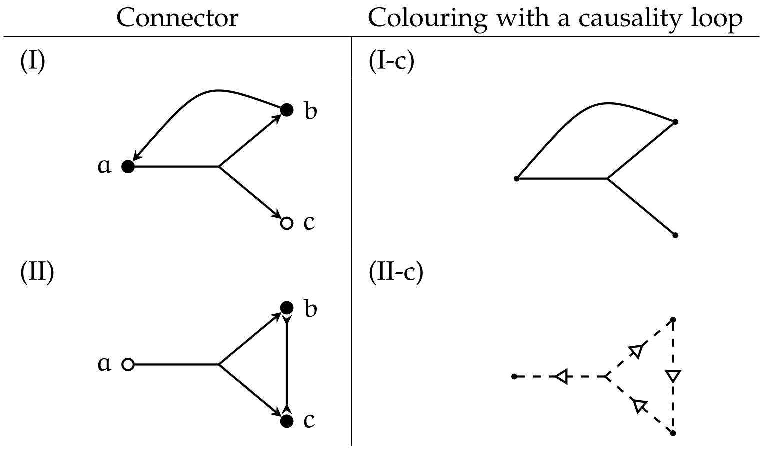


Figure 22: Reo connectors (I) and (II) with causality loops. In colouring (I-c) the loop follows the data flow of the channels. In colouring (II-c) the loop follows the reason to delay arrows.

The basic approach to finding causality loops is to trace all paths *backwards* in the causality graph to see whether there is, in our case, either an actual source of data or delay. Various solutions have been proposed to treat causality loops [70, 39].

These solutions can be adapted to compute colourings in a *compositional manner* such that every path in a colouring has a proper source, where (a) in a solid colouring, the path is given by the direction of the data flow and the source is a source of data, and (b) in a no-dataflow colouring, the path is given by the direction of the colouring arrows and a source is a reason to delay.

3.5 constructive 3-colouring

We wish to restrict our colourings to the subset without causality loops. We call this subset *constructive 3-colouring*. Removing the colourings that contain causality loops from colouring tables results in a more sensible semantics for $\mathcal{R}$eo connectors, as colourings that correspond to anomalous situations are removed. We start by observing that causality loops are reasoned about at the level of individual colourings rather then at the level of colouring tables; other valid colourings for the connectors may exist in the colouring table.

3.5.1 *Causal relation*

To detect colourings with causality loops we propose to associate each colouring with a *causal relation*. A causal relation is any binary relation on the domain of the colouring. We consider a 3-colouring *constructive* if the transitive closure of its causal relation is a strict partial order, or equivalently, irreflexive. This means that, for all nodes in the domain of the colouring, there does not exist a dataflow (no-dataflow) path (with the same arrow direction) that connects a node with itself. In the sequel we let R be a causal relation, and R^+ its transitive closure. Informally, the causal relation of a colouring corresponds to all the pairs of adjacent nodes of the graph of the colouring that fulfil either one of these conditions:

a) both nodes are coloured with the dataflow colour and the direction of the data flow in the underlying primitives is the same;

b) both nodes are coloured with the no-dataflow colour and the arrows point in the same direction.

Note that even though we have two types of causality loops, a single causal relation on the domain of the colouring is sufficient to detect either of the two types of causality loops. A node is either coloured with a dataflow colour or with a no-dataflow colour, partitioning the domain of the colouring. Conditions a) and b) only relate nodes of the same partition, therefore, the causal relation is also partitioned and each type of causality loop has its exclusive partition.

Instead of providing a formal definition for the causal relation of a colouring and use it to determine the causal relation for each colouring, we update the Definition 3.2.3 of a colouring table and Definition 3.2.4 for the composition of colouring tables. The idea is to construct the causal relation for each colouring in a colouring table compositionally out of the causal relations of each colouring of the primitive connectors, and filter out from the colouring table the resulting colourings that have a causal relation that is not a strict partial order.

Definition 3.5.1 (Constructive colouring table)**.** A *colouring table*, T, over nodes $N \subseteq \mathcal{N}ode$ is a set of constructive colourings with domain N. We denote a constructive colouring by a pair (c, R) of a colouring c, and its causal relation R that defines a strict partial order.

The definition of the composition operation for composing colouring tables filters out non-constructive colourings from the composite table.

Definition 3.5.2 (Constructive composition)**.** The composition of two constructive tables T_1 and T_2, denoted $T_1 \cdot T_2$, is defined as:

$$\begin{aligned} T_1 \cdot T_2 = \{(c_1 \cup c_2, R \cup S) \mid \; & (c_1, R) \in T_1,\ (c_2, S) \in T_2, \\ & n \in (\mathrm{dom}(c_1) \cap \mathrm{dom}(c_2)) \Rightarrow c_1(n) = c_2(n), \\ & (R \cup S)^+ \text{ is a strict partial order}\}. \end{aligned}$$

The notation $(\_)^+$ denotes the standard transitive closure of a binary relation.

The other definitions are valid in the constructive setting. The colouring tables and the composition operation for connectors $\_ \times \_$ just need to be considered according to the updated definitions.

A connector that only has colourings with causality loops, has the empty colouring table as semantics, making the connector then behave as the *zero* of the composition operation.

We examine the colourings with causality loops from Figure 22 and illustrate the use of the causal relation to identify them. The connectors (I) and (II) of Figure 22 are composed out of one *Replicator* and one *Sync*; and one *Replicator* and one *SyncDrain* respectively. In Table 17 each of the colourings for these primitives is associated with its respective causal relation. We observe that each colouring is a constructive colouring, since for each R, R^+ is a strict partial order. In fact we have $R = R^+$.

Colouring (I-c) in Figure 22 has a causal relation given by:

$$R = \{(a, b), (a, c), (b, a)\}.$$

Let us calculate the transitive closure of R. We have that:

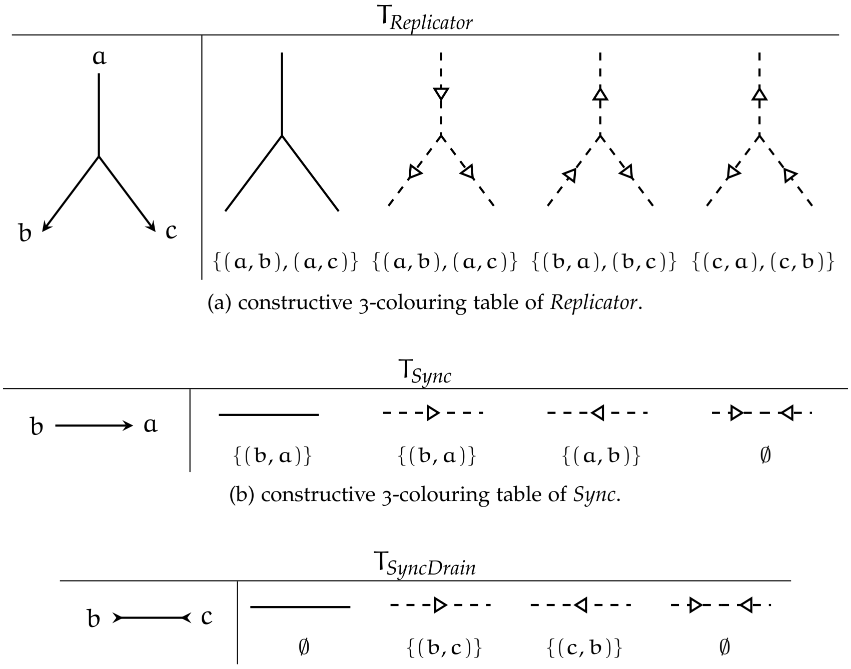
$$R \cup R^2 = \{(a, b), (a, a), (a, c), (b, a), (b, b), (b, c)\}$$

and $R^3 = R$ therefore $R^+ = R \cup R^2$. Since aR^+a and bR^+b, R^+ is not a strict partial order. Colouring (II-c) has a causal relation given by

$$S = \{(c, a), (c, b), (b, c)\}.$$

We have that:

$$S \cup S^2 = \{(c, a), (c, b), (c, c), (b, c), (b, a), (b, b)\}$$



(a) constructive 3-colouring table of *Replicator*.

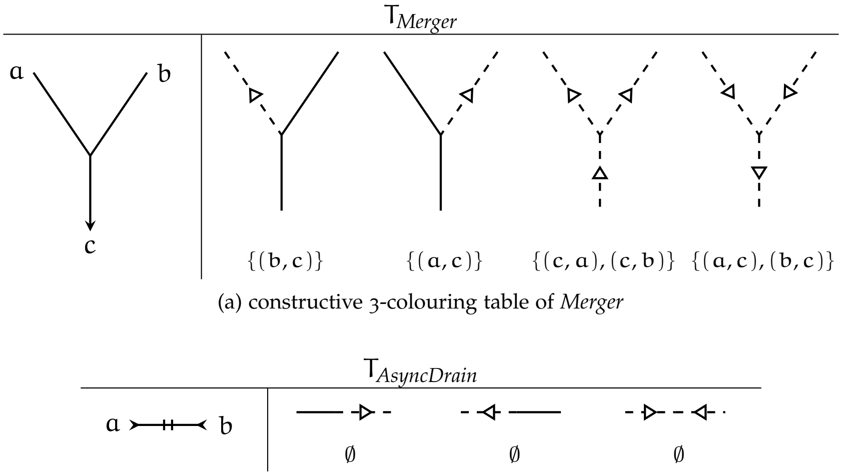
(b) constructive 3-colouring table of *Sync*.

(c) constructive 3-colouring table of *SyncDrain*.

Table 17: Constructive 3-colouring tables.

and $S^3 = S^2$ therefore $S^+ = S \cup S^2$. Since cS^+c and bS^+b, S^+ is not a strict partial order. Even though they are composed out of primitives with only constructive colourings, both connectors have non-constructive colourings. The *constructive* semantics detects in a compositional manner causality loops and disposes of them.

In Table 18 we have the causal relations for the colourings of two more primitives: *Merger* and *AsyncDrain*. Not surprisingly all the colourings of the *AsyncDrain* have the empty causal relation, reflecting the asynchronous relationship between the two ends of the channel. The causal relations for the colourings of the remaining primitives are similar to the ones presented in Tables 17 and 18.



(a) constructive 3-colouring table of *Merger*

(b) constructive 3-colouring table of *AsyncDrain*

Table 18: Other constructive 3-colourings.

3.6 OPERATION HIDE: ABSTRACTING FROM INTERNALS

The hide operation hides internal nodes of a connector. Hiding an internal node makes any dataflow on that node no longer *observable*. In connector colouring terms, it translates in two points:

a) the information about hidden nodes is removed from all colourings—the domain of colourings are restricted to nodes that are not hidden;

b) the *next* function abstracts away from colourings that assign dataflow only to hidden nodes—*hidden colourings*.

The hide operation is instrumental for defining context-dependent connectors with memory compositionally. Connector colouring permits to define component connectors with finite memory compositionally out of *FIFO*$_1$ primitive channels. We will illustrate how to construct a composite connector with two *FIFO*$_1$ which captures the intended semantics of a FIFO buffer with two memory cells. Before we can actually do this exercise however, we have to introduce what we call *hidden colourings*, and define the hide operation.

In the sequel we consider a connector denoted by $C = (N, B, E, S_I(N), \eta)$, and we denote by $i(c)$ a colouring c in the colouring table $T_i \in S_I(N)$. A hidden colouring corresponds to colourings that assign the data flow colour only to hidden nodes; these colourings denote dataflow behaviour that occur only on a connector's hidden nodes.

Definition 3.6.1 (Hidden colouring)**.** Consider connector C, a set of hidden nodes $H \subseteq (N \setminus B)$, and the set containing the nodes where colouring $i(c)$ assigns the dataflow colour:

$$Z = \{n \mid n \in N, i(c)(n) = \text{——}\}.$$

Colouring $i(c)$ is a H-*hidden colouring* if and only if, $Z \neq \emptyset$ and $Z \subseteq H$.

If data can flow through several *contiguous* hidden nodes according to several hidden colourings the hide operation abstracts away from the dataflow through the intermediate hidden nodes and is solely concerned with the reachability of this dataflow. Similarly to the transitive closure of a directed acyclic graph (DAG) which determines the reachability relation of the DAG, we can determine the reachability of the dataflow carried out by hidden colourings by calculating the transitive closure of the hidden-colourings.

Notation 3.6.2. We denote by $\xrightarrow{H}_S$ the relation on $S_I(N) \times S_I(N)$:

$$\left\{ (T_i, T_j) \mid \eta(i, c) = T_j, c \text{ is a } H\text{-\textit{hidden colouring}} \right\}.$$

The transitive closure of $\xrightarrow{H}_S$ is denoted by $(\xrightarrow{H}_S)^+$.

Intuitively a pair of colouring tables $(T, T') \in \xrightarrow{H}_S$ if the connector C is capable of changing from a configuration, in which the colouring table T applies, to another configuration, where colouring table T' applies, by performing only H-*hidden colourings*.

Once we have determined the reachability of the dataflow carried out by hidden colourings we can remove the information about hidden-nodes from colourings. For that purpose we use the *domain restriction* of functions. Given a colouring c with domain N, and node $h \in N$, we denote by $c \rhd (N \setminus \{h\})$ the restriction of colouring c to the domain subset $N \setminus \{h\}$. For a colouring table T over nodes N, and a node $h \in N$, we use the notation $T \rhd (N \setminus \{h\})$ to denote that for each colouring $c \in T$, we have $c \rhd (N \setminus \{h\})$.

We now have the necessary setup to define the semantics of the hide operation.

Definition 3.6.3 (hide operation)**.** Given a connector C and a node $h \in (N \setminus B)$. $\mathsf{hide}(C, h)$ is denoted by $\exists[h]C$ where:

$$\exists[h]C = (N \setminus \{h\}, B, E, S', \eta')$$

- $S' = \{T \rhd (N \setminus \{h\}) \mid T \in S\}$;
- $\eta' : \bigcup S_I(N \setminus \{h\}) \to S_I(N \setminus \{h\})$

$$\eta'(\langle i, c \rhd (N \setminus \{h\})\rangle) = \begin{cases} T_j \rhd (N \setminus \{h\}) & \text{if } (\eta(\langle i, c\rangle), T_j) \in (\xrightarrow{\{h\}}_S)^+ \\ \eta(\langle i, c\rangle) \rhd (N \setminus \{h\}) & \text{otherwise.} \end{cases}$$

Proposition 3.6.4. $\exists[h]C$ *is a well-defined connector (Definition 3.3.2).*

If all the internal nodes of a connector are hidden we are left with only boundary nodes. Hence the connector C defines a new primitive connector (Proposition 3.3.3).

Let us illustrate the hide operation by constructing the *FIFO*$_2$ depicted in Figure 23 using two *FIFO*$_1$ and the operations join and hide. We consider a generic *FIFO*$_1$ (X, Y) connector, where X and Y are variables ranging over $\mathcal{N}ode$, denoted by:

$$\mathit{FIFO}_1(X, Y) = \langle\ \{X, Y\}, \{X, Y\}, (X^i, Y^o)_{\mathit{FIFO}_1}, \{T_0, T_1\}, \{\ \eta(\langle 0, c_1\rangle) = \eta(\langle 1, c_2) = T_1 \\ \eta(\langle 0, c_2\rangle) = \eta(\langle 1, c_1\rangle) = T_0\}\ \rangle$$

Tables T_0 and T_1 are depicted in Figure 15.

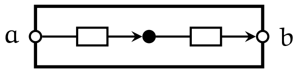


Figure 23: *FIFO*$_2$ = $\mathsf{hide}(\mathit{FIFO}_1(a, c)\ \mathsf{join}\ \mathit{FIFO}_1(c, b))$

Example 3.6.5 (Construction of a *FIFO*$_2$ out of two *FIFO*$_1$ using the join and hide operations)**.** We first calculate *FIFO*$_1$ (a, c) join *FIFO*$_1$ (c, b), denoted by

$$\mathit{FIFO}_1(a, c)\ \times\ \mathit{FIFO}_1(c, b).$$

Which yields the connector

$$\left\langle \{a, b, c\}, \{a, b\}, \left\{(a^i, c^o)_{\mathit{FIFO}_1}, (c^i, b^o)_{\mathit{FIFO}_1}\right\}, S_I(N), \eta' \right\rangle$$

where:

- $N = \{a, b, c\}$

- $I = \{0,1\} \times \{0,1\}$
- $S_I(N) = \{T_{\langle 0,0\rangle} = T_0 \cdot T_0, T_{\langle 1,0\rangle} = T_1 \cdot T_0, T_{\langle 0,1\rangle} = T_0 \cdot T_1, T_{\langle 1,1\rangle} = T_1 \cdot T_1\}$. Each of the colouring tables in $S_I(N)$ is depicted in Table 19.
- $\eta' : \bigcup S_I(N) \to S_I(N)$

  $$\begin{aligned}
  \eta'(\langle 0,0\rangle, c_1) &= \eta(\langle 0, c_1\rangle) \cdot \eta(\langle 0, c_2\rangle) = T_1 \cdot T_0 = T_{\langle 1,0\rangle}\\
  \eta'(\langle 0,0\rangle, c_2) &= \eta(\langle 0, c_2\rangle) \cdot \eta(\langle 0, c_2\rangle) = T_0 \cdot T_0 = T_{\langle 0,0\rangle}\\
  \eta'(\langle 1,0\rangle, c_1) &= \eta(\langle 1, c_3\rangle) \cdot \eta(\langle 0, c_1\rangle) = T_0 \cdot T_1 = T_{\langle 0,1\rangle}\\
  \eta'(\langle 0,1\rangle, c_1) &= \eta(\langle 0, c_1\rangle) \cdot \eta(\langle 1, c_3\rangle) = T_1 \cdot T_0 = T_{\langle 1,0\rangle}\\
  \eta'(\langle 0,1\rangle, c_2) &= \eta(\langle 0, c_1\rangle) \cdot \eta(\langle 1, c_4\rangle) = T_1 \cdot T_1 = T_{\langle 1,1\rangle}\\
  \eta'(\langle 0,1\rangle, c_3) &= \eta(\langle 0, c_2\rangle) \cdot \eta(\langle 1, c_3\rangle) = T_0 \cdot T_0 = T_{\langle 0,0\rangle}\\
  \eta'(\langle 0,1\rangle, c_4) &= \eta(\langle 0, c_2\rangle) \cdot \eta(\langle 1, c_4\rangle) = T_0 \cdot T_1 = T_{\langle 0,1\rangle}\\
  \eta'(\langle 1,1\rangle, c_1) &= \eta(\langle 1, c_4\rangle) \cdot \eta(\langle 1, c_3\rangle) = T_1 \cdot T_0 = T_{\langle 1,0\rangle}\\
  \eta'(\langle 1,1\rangle, c_2) &= \eta(\langle 1, c_4\rangle) \cdot \eta(\langle 1, c_4\rangle) = T_1 \cdot T_1 = T_{\langle 1,1\rangle}
  \end{aligned}$$

Next we calculate the model for $\mathsf{hide}(\mathit{FIFO}_1(a,c)\,\mathsf{join}\,\mathit{FIFO}_1(c,b))$ denoted by

$$\exists[c]\left\langle \{a,b,c\}, \{a,b\}, \left\{(a^i, c^o)_{\mathit{FIFO}_1}, (c^i, b^o)_{\mathit{FIFO}_1}\right\}, S_I(N), \eta' \right\rangle$$

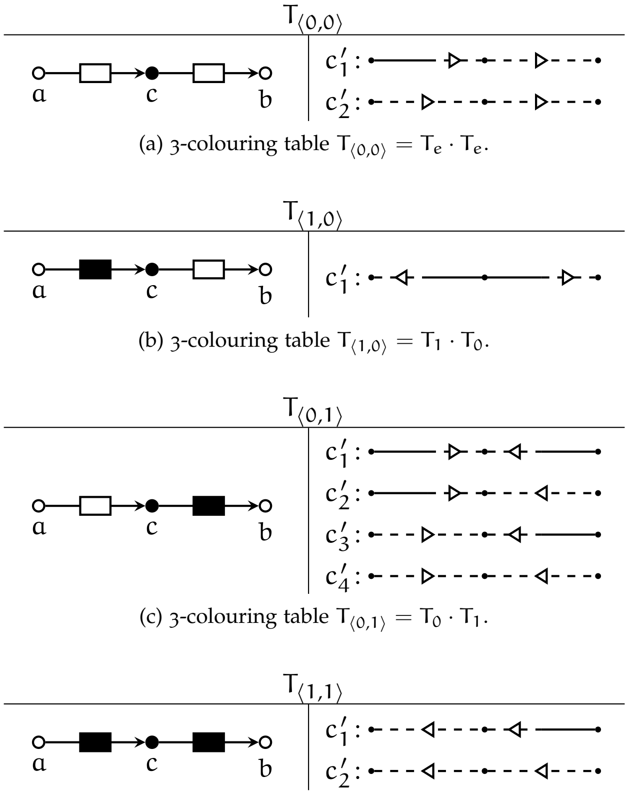
which yields a new connector

$$\left\langle N', \{a,b\}, E', S'_I(N'), \eta'' \right\rangle$$

where:

- $N' = \{a,b,c\} \setminus \{c\}$
- $E' = \left\{(a^i, c^o)_{\mathit{FIFO}_1}, (c^i, b^o)_{\mathit{FIFO}_1}\right\}$
- $S' = \left\{T'_{\langle 0,0\rangle}, T'_{\langle 1,0\rangle}, T'_{\langle 0,1\rangle}, T'_{\langle 1,1\rangle}\right\}$. Each colouring table is depicted in Table 20.
- To define η'' we are required to calculate $(\xrightarrow{\{c\}}_{S'})^+$. The only $\{c\}$-*hidden colouring* is $c'_1 \in T'_{\langle 1,0\rangle}$ and $\eta'(\langle 1,0\rangle, c'_1) = T'_{\langle 0,1\rangle}$, we have that $\xrightarrow{\{c\}}_{S'} = \{(T'_{\langle 1,0\rangle}, T'_{\langle 0,1\rangle})\}$. Hence, $(\xrightarrow{\{c\}}_{S'})^+ = \{(T'_{\langle 1,0\rangle}, T'_{\langle 0,1\rangle})\}$.

  $\eta'' : \bigcup S'_I(N') \to S'_I(N')$


$T_{\langle 0,0 \rangle}$
a
c
b
c'_1 :
c'_2 :
$T_{\langle 1,0 \rangle}$
a
c
b
c'_1 :
$T_{\langle 0,1 \rangle}$
a
c
b
c'_1 :
c'_2 :
c'_3 :
c'_4 :
$T_{\langle 1,1 \rangle}$
a
c
b
c'_1 :
c'_2 :

(a) 3-colouring table $T_{\langle 0,0 \rangle} = T_e \cdot T_e$.

(b) 3-colouring table $T_{\langle 1,0 \rangle} = T_1 \cdot T_0$.

(c) 3-colouring table $T_{\langle 0,1 \rangle} = T_0 \cdot T_1$.

(d) 3-colouring table $T_{\langle 1,1 \rangle} = T_1 \cdot T_1$.

Table 19: 3-colouring tables of $\mathit{FIFO}_1(a, c) \times \mathit{FIFO}_1(c, b)$.

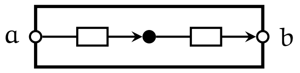
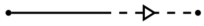
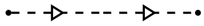
$$
\begin{aligned}
\eta''(\langle 0,0\rangle, c_1') &= T'_{\langle 0,1\rangle}, && \text{because } \eta'(\langle 0,0\rangle, c_1') = T'_{\langle 1,0\rangle}, \text{ and}\\
& && (T'_{\langle 1,0\rangle}, T'_{\langle 0,1\rangle}) \in (\xrightarrow{\{c\}}_{S'})^{+};\\
\eta''(\langle 0,0\rangle, c_2') &= T'_{\langle 0,0\rangle};\\
\eta''(\langle 1,0\rangle, c_1') &= T'_{\langle 0,1\rangle};\\
\eta''(\langle 0,1\rangle, c_1') &= T'_{\langle 0,1\rangle}, && \text{because } \eta'(\langle 0,1\rangle, c_1') = T'_{\langle 1,0\rangle}, \text{ and}\\
& && (T'_{\langle 1,0\rangle}, T'_{\langle 0,1\rangle}) \in (\xrightarrow{\{c\}}_{S})^{+};\\
\eta''(\langle 0,1\rangle, c_2') &= T'_{\langle 1,1\rangle};\\
\eta''(\langle 0,1\rangle, c_3') &= T'_{\langle 0,0\rangle};\\
\eta''(\langle 0,1\rangle, c_4') &= T'_{\langle 0,1\rangle};\\
\eta''(\langle 1,1\rangle, c_1') &= T'_{\langle 0,1\rangle}, && \text{because } \eta'(\langle 0,1\rangle, c_1') = T_{\langle 1,0\rangle}, \text{ and}\\
& && (T_{\langle 1,0\rangle}, T'_{\langle 0,1\rangle}) \in (\xrightarrow{\{c\}}_{S})^{+};\\
\eta''(\langle 1,1\rangle, c_2') &= T'_{\langle 1,1\rangle}.
\end{aligned}
$$

Observe that after hiding node c the colourings do not include the node c in the domain of the colourings. Colouring table $T_{\langle 1,0\rangle}$ is not contained in the codomain of η'' and is therefore not applicable. As a consequence the dataflow behaviour denoted by $c_1' \in T'_{\langle 1,0\rangle}$ is, as we intended, never observed, since $c_1' \in T'_{\langle 1,0\rangle}$ is identified as a $\{c\}$-*hidden colouring*.

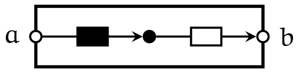
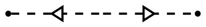
We argue that the connector *FIFO*$_2$ denoted by $\exists[c](\mathit{FIFO}_1(a,c) \times \mathit{FIFO}_1(c,b))$ captures the intended behaviour of a *FIFO* buffer with two cells. The colouring table $T_{\langle 0,0\rangle}$ captures the dataflow behaviour of a *FIFO*$_2$ that has both buffer cells empty; $T_{\langle 0,1\rangle}$ captures the dataflow behaviour of a *FIFO*$_2$ that has one buffer cell full; $T_{\langle 1,1\rangle}$ captures the dataflow behaviour of a *FIFO*$_2$ that has both buffer cell full. We explain the colouring tables $T_{\langle 0,0\rangle}$ and $T_{\langle 0,1\rangle}$, $T_{\langle 1,1\rangle}$ follows similarly.

When the two buffer cells are empty, colouring table $T_{\langle 0,0\rangle}$ applies. If a *write* is present, the colouring $c_1' \in T_{\langle 0,0\rangle}$ applies, the *write* fires, data flows through a and is stored in a buffer cell. As a result the connector has now one buffer cell full and the other empty. Intuitively we expect the colouring table $T_{\langle 0,1\rangle}$ to apply and indeed this is the case because $\eta''(\langle 0,0\rangle, c_1') = T_{\langle 0,1\rangle}$. When $T_{\langle 0,1\rangle}$ applies three interesting dataflow behaviour can occur, denoted by colourings $c_1', c_2', c_3' \in T_{\langle 0,1\rangle}$:

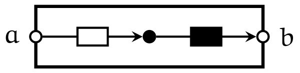
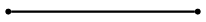
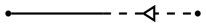
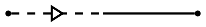
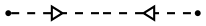
- Colouring c_1' captures the dataflow behaviour in which a *write* and a *take* are present in each of the boundary nodes, a and b respectively. Both operation fire and the data stored in one buffer cell flows through b and synchronously the data input by the *write* flows through a and is stored in one buffer cell. After c_1' executes, the colouring table $T'_{\langle 0,1\rangle}$ applies again. Notice that this is expected since again the connector has one buffer cell full and one empty.

$T'_{\langle 0,0\rangle}$	
	$c'_1 \rhd \{a,b\}$:  $c'_2 \rhd \{a,b\}$: 

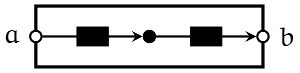
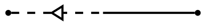
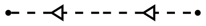
(a) 3-colouring table $T'_{\langle 0,0\rangle}$ calculated from $T_{\langle 0,0\rangle}$ by performing the restriction $\rhd\{a,b\}$ in the domain of each of the colourings.

$T'_{\langle 1,0\rangle}$	
	$c'_1 \rhd \{a,b\}$: 

(b) 3-colouring table $T'_{\langle 1,0\rangle}$ calculated from $T_{\langle 1,0\rangle}$ by performing the restriction $\rhd\{a,b\}$ in the domain of each of the colourings.

$T'_{\langle 0,1\rangle}$	
	$c'_1 \rhd \{a,b\}$:  $c'_2 \rhd \{a,b\}$:  $c'_3 \rhd \{a,b\}$:  $c'_4 \rhd \{a,b\}$: 

(c) 3-colouring table $T'_{\langle 0,1\rangle}$ calculated from $T_{\langle 0,1\rangle}$ by performing the restriction $\rhd\{a,b\}$ in the domain of each of the colourings.

$T'_{\langle 1,1\rangle}$	
	$c'_1 \rhd \{a,b\}$:  $c'_2 \rhd \{a,b\}$: 

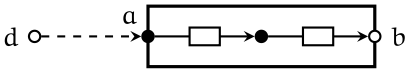
(d) 3-colouring table $T'_{\langle 1,1\rangle}$ calculated from $T_{\langle 1,1\rangle}$ by performing the restriction $\rhd\{a,b\}$ in the domain of each of the colourings.

Table 20: 3-colouring tables of $\exists[c](\mathit{FIFO}_1(a,c) \times \mathit{FIFO}_1(c,b))$.

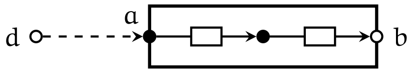
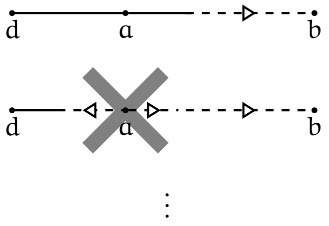
- Colouring c'_2 captures the dataflow behaviour in which a *write* is present in a and no operation is present in b. The *write* fires and the data is stored in the empty buffer cell. After c'_2 executes, both buffer cells of the connector are full and the colouring table $T_{\langle 1,1\rangle}$ given by $\eta''(\langle 0,1\rangle, c'_2)$ applies as expected.

- Colouring c'_3 captures the dataflow behaviour in which a *take* is present in b and no operation is present in a. The *take* fires and the data stored flows through b. After c'_2 executes, both buffer cells of the connector are empty and the colouring table $T_{\langle 0,0\rangle}$ given by $\eta''(\langle 0,1\rangle, c'_3)$ applies as expected.

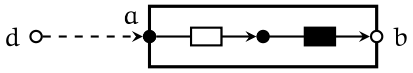
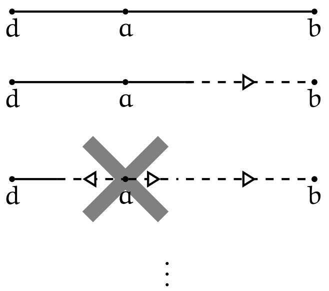
Another important property of the *FIFO*$_2$ semantics is that the composite connector:



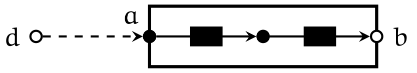
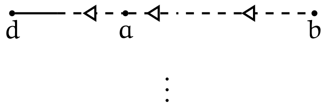
yields the intended context dependent behaviour. We do not present all the details but we discuss the colourings which illustrate that the context-dependent behaviour of the *LossySync* channel is properly captured:

$T_{LossySync} \cdot T'_{\langle 0,0\rangle}$	
	

When the *FIFO*$_2$ has its two buffer cells empty the *LossySync* cannot lose data that flows though its input end, as denoted by the colouring in the first first entry. The colouring that would denote the *LossySync* losing data is not possible.

$T_{LossySync} \cdot T'_{\langle 0,1\rangle}$	
	

When the *FIFO*2 has one buffer cell full the *LossySync* continues without being able to lose data that flows though its input end, as denoted by the colouring in the first first entry. The colouring that would denote the *LossySync* losing data continues to be not possible.

$T_{LossySync} \cdot T'_{\langle 1,1 \rangle}$	
	

When the *FIFO*2 has both buffer cell full then the *LossySync* loses the data whenever data flows through its input end.

Notice that if colouring table $T'_{\langle 1,0 \rangle}$ was applicable the *LossySync* would always lose data in $T_{LossySync} \cdot T'_{\langle 1,0 \rangle}$, which would violate the intended behaviour.

3.7 IMPLEMENTING CONNECTOR COLOURING

In this section we discuss how connector colouring forms the basis of a non-distributed implementation of ℛeo connectors, and how it can be extended to a distributed implementation based on MoCha mobile channel middleware [12] which was developed at CWI. Before presenting the details, we outline some requirements that a distributed implementation ought to satisfy. We then present the general scheme which we expect the algorithms implementing connector colouring to follow. Next, we describe a non-distributed implementation, called *Reolite* , which implements most of the original ℛeo proposal. Finally we present a distributed algorithm for connector colouring based on *spanning trees* [92].

3.7.1 *Requirements for a Distributed Implementation of ℛeo*

A distributed implementation of ℛeo in our perspective must fulfil specific requirements. These requirements are presented and discussed in detail in a Technical Report [44] that preceded the work presented in this chapter. Here we just briefly recall those requirements.

NO GLOBAL VIEW In a geographically distributed environment, different parts of a ℛeo connector may reside on remote hosts. A global view of a connector's state can result in single point-of-failure vulnerability, and the delays necessary for *maintaining* a consistent global view may inhibit the desirable parallelism inherent in physically distributed systems. Without a global view, the constituents of a connector have only a limited knowledge about the connector, and must delegate requests to other parts of the connector in order to obtain the information required to transport data.

COMMUNICATION INFRASTRUCTURE AND TOPOLOGY In $\mathcal{R}$eo, channels encapsulate all communication-related activities. Since channels provide the only infrastructure for communication, then only the paths defined by the interconnection of channels, the connector topology, can be used to send the control information required to determine the data flow of a $\mathcal{R}$eo connector.

PROPAGATION OF SYNCHRONISATION CONSTRAINTS $\mathcal{R}$eo channels and nodes impose synchronisation and exclusion constraints on data flow across the entire connector. Data flows through each "synchronous" part of a connector atomically. The state of the entire connector and its boundary may be required to determine how data can flow.

One approach to determine the flow of data is to optimistically send data along channels and rollback any changes when synchronisation constraints cannot be met. Aside from requiring a rollback capability on every channel, which may not be feasible in practice, this approach may, in general, result in too much wasted resources or network flooding when trying to find a suitable data flow.

The alternative we present pre-computes the routes of possible dataflow, and then, non-deterministically chooses one to take, whenever required.

CONCURRENCY In a distributed environment, multiple parties may interact with a connector *at the same time*. This means that more than one computation to determine a connector's data flow can be active, leading to a situation where different computations are competing for parts of the connector. Without proper handling of these situations, these concurrent data flow computations can face race conditions, livelocks, deadlocks, or simply waste resources.

3.7.2 *Algorithm Scheme for Connector Colouring*

Any algorithm using connector colouring as the basis for deciding how to route data through a $\mathcal{R}$eo connector will need to perform the following steps, though not necessarily strictly in the order presented. We assume that the configuration of a connector including its pending I/O operations is locked when the colouring table is being computed, although at any other time parties may delay, timeout, retry, and new parties may join—changing the configuration of the connector and its environment.

COMPUTE COLOURING TABLE FOR COMPLETE CONNECTOR Collect all the colouring tables from all the channels and nodes of the connector. Compute the composite colouring table.

SELECT ROUTE TO EMPLOY The computed colouring table may contain 0, 1 or many colourings. If the table has no elements, no communication occurs. If the table has only one element, then that is selected. Otherwise, select a colouring non-deterministically.

DISTRIBUTE THE CHOSEN COLOURING TO ALL PARTIES The chosen colouring is distributed to all parties so that they have a consistent view, according to the synchronisation constraints, of what the data flow will be.

SEND DATA Each data source (e. g. *write/FIFO*$_1$*(x)* buffer) which has been selected to have dataflow can send its data as soon as it gets the final colouring table. All choices that a primitive needs to make are determined by the chosen colouring.

A number of variations are possible. Rather than globally computing the table, it could be computed using a parallel algorithm such as *all-reduce* [89]. If the local tables are $T_1, \dots, T_n$, *reduce* computes $T_1 \cdot T_2 \cdots T_n$; the *all* part corresponds to sending this information to all parties. In practice one does both steps together, relying on the properties of the operation '$\cdot$'. To deal with the case that multiple entries are possible, simply order the entries in the table and choose the first. Entries should be placed in the table non-deterministically. Alternatively, some form of negotiation might be required to choose a colouring. This falls into the class of problems known as *reaching consensus* in a distributed network [71].

We now will describe how connector colouring is implemented in *Reolite* .

3.7.3 Reolite *: A Non-Distributed $\mathcal{R}$eo Implementation*

Reolite [33] is a concurrent, non-distributed, Java$^{©}$ [75] implementation of $\mathcal{R}$eo, based on connector colouring.[2] The implementation is rudimentary and serves as a proof-of-concept which demonstrates the feasibility of connector colouring compared to an earlier approach based on *accepts* and *offers* [7, 44]. The previous approach to implementing $\mathcal{R}$eo was so complex that it cost approximately a man year of effort to implement a system which neither worked particularly well nor was easy to reason about. Based on connector colouring, *Reolite* was up and running within a fortnight.

Reolite permits a number of components, running their own threads, to interact with a connector, which itself is managed globally by a *single* thread. Interaction occurs only between a component's thread and the connector whenever a component attempts to *write* to or *take* from a channel-end. The connector is protected by a global lock, which means that whenever the connector thread is calculating the

[2] Lacking are (1) operations for connecting components to and disconnecting components from connectors, and for moving nodes, as these have no effect on the connector behaviour, (2) the hide operation, and (3) channels that are data sensitive, such as a filter.

colouring table or performing dataflow, the connector cannot be changed—even registering a new pending *write* or *take* is impossible. The interaction between a component and the connector is best described by detailing the two kinds of thread. Note that this locking scheme is too coarse grained to be scalable, but it works well for the proof-of-concept.

COMPONENT THREAD Whenever a component performs a write or take on a channel-end, it begins interacting with the connector as follows:

```
1. start timer for timeout, if specified
2. obtain connector lock
3. if writing then register that data is being written to the channel-end
   if taking then register that data is requested from the channel-end
4. release connector lock
5. notify connector and block
6. when awakened
   if awakened by connector (assumption: this end was chosen in a colouring)
      kill timer
      return, with data if operation was a take
   if awakened by timeout
      obtain connector lock
      if write/take has since succeeded
         release connector lock
         return, with data if operation was a take
      else
         deregister write/take
         release connector lock
         throw timeout exception
```

Note that a timeout will never occur if the connector is busy, which avoids inconsistency across the connector. Thus, it may be the case that a timeout expires (under the hood), but that data is, nevertheless, transported.

CONNECTOR THREAD The connector thread is reactive on the activity performed by the components threads.

```
1. obtain connector lock
```

```
2. collect colourings from all channels and input and output ends (both
   those that have pending operations and those that do not).
3. compute colouring table
4. select a colouring
5. loop until data has flowed at all coloured ends
       select a source of data that is coloured with the flow colour
       pass its data into that respective channel, which may create new
       sources of data
       reset input and output ends that have had their requests satisfied
       remove such ends from colouring
   end loop
6. release lock.
```

Channels have data pushed into their input end(s). This data will appear at their output end(s) based on each channel's implementation in accordance with the colouring selected for the channel (otherwise, the channel is not implemented correctly).

In addition to this algorithm, the connector periodically gains control and performs any actions that it can—this is necessary to, for example, push data through chains of $FIFO_1$ buffers.

The implementation of *Reolite* also enables dynamic reconfiguration of connectors (using join and split [7]), and permits channel-ends to be passed through connectors, enabling complex dynamic coordination patterns. New components can be added to an existing system, *assuming* that the connector *first* knows the name of one of the channel-ends. This deficiency has been removed in MoCha [12], which permits the advertisement and discovery of channel-end names.

3.7.4 *Distributed Algorithm*

We now present an informal description of a somewhat idealised version of a possible distributed algorithm. The algorithm which follows may be initiated at any boundary node by an I/O request, or by a buffer trying to forward its data. In addition, the algorithm may be concurrently initiated at different nodes by different parties. We first describe the algorithm from the perspective of one such party, assuming that no interference with other parties occurs. Then, after, we describe how to deal with multiple parties computing concurrently.

Single initiation thread

The algorithm follows the topology of the connector using remote procedure calls rather than message passing. Calls traverse the graph of a connector by passing from a node, to the channel-ends forming the node, then to a channel, which may propagate the call to its other end, and then to some node again. Information describing the state of the algorithm may be stored in channel-ends. The initial state is SLEEPING.

The first phase of the algorithm is called *collect*. It proceeds as follows. Starting with the initiating node as root, a spanning tree of the full connector graph is computed. This is achieved simply by traversing the graph of the connector, marking each channel-end as it is visited (state COLLECT), ceasing further progress whenever an already visited end is met. The forward leg of this phase, thus, traverses a spanning tree of the $\mathcal{R}$eo connector. The return leg collects the colouring tables of every channel and node. The complete colouring table is then computed at the root of the tree, and an entry of the table is chosen.

The algorithm then enters the second phase, called *propagate*, in which the chosen colouring and the data to be sent are propagated through the connector. The interesting behaviour occurs at channel-ends, and there are essentially three cases to deal with:

1. For calls following the direction of the dataflow where the channel-end is coloured with the dataflow colour, the colouring is passed onwards (through the channel or through the node) along with the data. On the return leg, the state is reset to SLEEPING.

2. For calls against the direction of dataflow where the channel-end is coloured with the dataflow colour, the colouring is passed onwards, and on the return leg, the data is returned and the state is reset to SLEEPING.

3. For calls where there is no dataflow, the colouring is propagated, potentially to places where dataflow is possible, and the state is immediately reset to SLEEPING.

Nodes get data from some output-end coloured with the dataflow colour and propagate it to every input-end coloured with the dataflow colour. Channels accept data from their input-ends coloured with dataflow, process it according to their specification, so long as this process is consistent with the selected colouring, and forward data on their output-ends coloured with dataflow.

At the end of this phase, data will be passed through the connector as prescribed by the selected colouring, and the transient state stored by the algorithm in the channel-ends will be reset to its initial state.

Multi initiation threads

In a setting in which multiple parts of the connector can initiate the algorithm above, extra considerations need to be taken into account. The first point to note is that the compositional nature of the colourings enable one to consider different computations to be cooperatively computing the same global colouring scheme, which is in contrast to Internet routing algorithms where packets compete for passage through the network [97]. This means that partial colouring table computations initialised concurrently can be combined, and that way reduce de absolute time required to compute the colouring table of the connector. The second point to note is that in this setting a scheme to guarantee a global ordering on the channel-end names is required. Although this is a theoretically difficult issue, a number of schemes exist for doing this in practice, such as composing channel end names out of network card MAC addresses of machines, their IP numbers, and/or their process identifiers.

As in the setting above, every channel-end stores state information used by different threads to determine what the other threads are performing. Initially, all threads are in the SLEEPING state. State COLLECT denotes that the channel-end has already been visited in the *collect* phase—this state also stores the data indicating the identifier of the initiating thread that has visited the end. Finally, the state PROPAGATE indicates that the algorithm is propagating the computed colouring information and potentially data—this state stores the colouring table, the direction of propagation (against or along the flow of data) and the data value (if available).

When a channel-end is passed by the *collect* phase, the algorithm marks the channel-end into the state COLLECT and with the identifier of the initiating channel-end. If the collect phase passes a channel-end that is in the COLLECT state and has the same ID as its own, it knows that it has hit a loop and it starts returning the collected results. If it passes a channel-end in state COLLECT that is marked with a different thread identifier, then the thread with the highest identifier continues, and the thread with the lowest identifier backs off. Otherwise, it continues constructing the spanning tree—stopping whenever it detects that it has completed the colouring. If a collecting thread passes a channel-end which is currently propagating data (state PROPAGATE), then this collecting thread also backs off. Whenever a computation backs off, it waits until the initiating channel-end is reset to the SLEEPING state. The computation will then reinitiate in case the pending I/O request associated with the initiating channel-end was not coloured with the flow colour by the colouring that just executed.

When a channel-end is passed by the *propagate* phase and the channel-end is in the COLLECT state, the propagate algorithm continues as normal, setting the state to PROPAGATE (with the direction, colouring table, and any data stored as well). If the end is in the SLEEPING state, it means that another thread in the propagate phase has passed this way, so this thread returns—the data that it needs to propagate will be stored by another thread in one of the ends previously visited by this thread. If the propagate phase passes a channel-end in the PROPAGATE state, it does one of two things. If this thread has data, it writes the data into the state of the channel-end and returns. If this thread is waiting for data, it suspends until the channel-end receives data from some other thread. In all cases, at the end of the propagate phase, the states of the channel ends are set back to the original SLEEPING state.

3.7.5 *Discussion*

The complexity of sending data though a $\mathcal{R}$eo connector depends on two factors: the size of the *synchronous slices*[3], and the size of the *synchronous slice*'s colouring table, which domain size is determined by the size of the synchronous slices. Assume that there are n channel-ends in a synchronous slice. The algorithm then, in the worst case, passes sequentially through these n channel-ends. Assume that the size of the domain of the colouring table is denoted by $T(n)$. It is clear, then, that the complexity of the algorithm is $O(n \times T(n))$ for each data item that needs to be sent. In the worst case the table is exponential in the size of the synchronous slices, so the algorithm has exponential complexity. Often, table sizes tend to be linear in n, and thus the overall complexity is $O(n^2)$.

The fact that the size of colouring tables can easily become large can be managed by a series of optimisations:

1. store the colouring tables as sets, rather than lists, to avoid duplication;
2. all colourings that differ only by the arrow direction of the internal nodes' colouring can be grouped into a single colouring where the arrows of the no-dataflow colours are removed, since for effects of composition the arrow direction of the no-dataflow colour only matters at boundary nodes;
3. the tables should be kept always reduced according to the *flip-rule* (Definition 3.4.1);
4. the colouring tables should be kept constructive—by removing colourings that are not constructive due by causality loops in the connector (Section 3.5).

Combining these optimisations reduces drastically the size of, otherwise large colouring tables. For example to compute the colouring table of the exclusive router depicted in Figure 12, the optimisations above reduce the size of the colouring table from over 1000 entries to just the expected 4.

The complexity of the algorithm is due to the requirements that we set out for the distributed implementation of $\mathcal{R}$eo in Section 3.7.1, rather than the algorithm itself. Nevertheless there are a number of different ways of managing this complexity. First, avoiding the deployment of synchronous slices across different machines helps localise the cost. Making connectors less synchronous is another way, but this means using a different connector. Ultimately, the trade-off between the degree

[3] *synchronous slice*–for the CONLANG channels a synchronous slice corresponds to a contiguous part of the connector not including $FIFO_1$ channels.

of synchronisation and acceptable performance can only be determined through benchmarking various candidate connectors.

The algorithm works in the setting where each party knows nothing of its neighbours, except how to find them (via the topology of the connector). Different parties can compute concurrently, though the topology of the connector may limit how much concurrency can be exploited in computing a colouring table.The colouring table is computed as a solution to the synchronisation constraints *before* any data flows, rather than optimistically sending data that may need to be retracted or ignored. Although the colouring table is completely obtained in one node (of a synchronous slice), we argue that the algorithm still satisfies the *no global view* constraint, as the algorithm need not maintain this view. It simply uses it to perform a step. We argue, thus, that the algorithm satisfies the 4 criteria presented in Section 3.7.1.

There are two other issues to consider: dynamic changes in a $\mathcal{R}$eo connector and partial failure of the network. Dynamic changes to a $\mathcal{R}$eo connector, such as reconfiguration, can be catered for, as these must observe the locking scheme and can only occur when the node being modified is in the Sleeping state. The algorithms for reconfiguration and mobility implemented in the MoCha middleware [12] can be adapted to our setting. The main point where the algorithm suffers is dealing properly with partial failure. Put simply, the algorithm does not deal with partial failure. But it is possible to deploy a connector so that it does not suffer from problems of partial failure, by ensuring that no synchronous slice of a connector spans multiple machines in a network.

3.7.6 *Summary of Implementation Status*

In the context of his PhD, José Proença, devised a distributed runtime framework called Dreams [83]. Dreams stands for distributed runtime evaluation of atomic multiple steps. The framework is used currently to develop a distributed implementation of $\mathcal{R}$eo. The implementation consists of a (distributed) *coordination engine* where each part of the engine runs independently, and interacts with the other parts of the engine exchanging descriptions of the *coordination behaviour* and data values. In the case of $\mathcal{R}$eo, a connector is deployed and its coordination behaviour is given by the 3-colouring semantics, implemented using propositional logic [37]. At each execution step of the connector the overall dataflow behaviour is calculated according to the 3-colouring semantics, by composing the colouring tables from each part of the distributed connector. Dreams architecture is based on the Actor model [2], hence each part of the coordination engine is modelled as an *actor*. Communication between actors is asynchronous, and the atomicity inherent to $\mathcal{R}$eo is achieved via a distributed consensus algorithm.

The Dreams framework addresses three main concerns disregarded by previous (centralised) implementations of $\mathcal{R}$eo, such as *Reolite* :

(1) decoupling of the execution;

(2) scalability;

(3) reconfigurability.

Decoupling is achieved by identifying independent communication events which can be executed safely in parallel. *Scalability* is a direct consequence of the decoupling. Complex connectors can be deployed and executed under the assumption that only a smaller part of the connector are required to synchronise each time data flows through the connector. *Reconfigurability* is facilitated because the engine symbolically encodes the possible dataflow behaviour of a connector and permits to manipulate it whenever instructions for reconfiguration are issued by the runtime. This approach contrasts with previous approaches where the possible dataflow behaviour is compiled in some abstract state machine according to some automata model. Once compiled the behaviour can no longer be changed.

3.8 RELATED WORK

$\mathcal{R}$eo is capable of defining connectors with sophisticated behaviour using very few primitive channels [7, 8, 9]. Predecessors to $\mathcal{R}$eo, namely MoCha [12] and Manifold [26], did not impose synchronisation constraints to the degree that $\mathcal{R}$eo does, and hence were simpler to implement but less expressive. $\mathcal{R}$eo enables synchronisation and exclusion constraints to propagate across a connector, whereas these models could not. Older coordination languages and models such as Linda [29] and Gamma [22] cannot directly accommodate the degree of synchronisation that is expressible in $\mathcal{R}$eo. This fact remains true for all of the coordination models covered in a recent survey [81]: in this sense $\mathcal{R}$eo's approach to coordination is unique, due to the propagation of to synchronisation and exclusion constraints, across the entire connector.

The notion of connector is not unique to $\mathcal{R}$eo, as it appears in the study of software architecture [74], and also in the guise of a coordination model for active objects [38]. The main distinguishing feature of $\mathcal{R}$eo is that it enables the simple compositional expression of synchronisation and exclusion constraints, whereas the other work on connectors focuses more on connecting behavioural interfaces of components.

A number of informal and formal models exist for $\mathcal{R}$eo. The first operational description of $\mathcal{R}$eo [7] describes connector behaviour in the presence and absence of requests at channel-ends in a context-dependent manner. This operational model based on what values connectors offered and accepted proved, however, to be too difficult to reason about and to implement [44]. Semantic models based on a coinductive calculus [9] and on constraint automata [13] paved the way to reasoning about connectors and their expressiveness, and for the mechanical verification of their properties. These two models were proven equivalent [13, Section 4.7] under mild assumptions.

We now compare connector colouring with constraint automata. One aspect of the constraint automata model is that the transitions in automata are labelled with the collection of nodes that synchronously succeed in their data exchange in a given step, at the exclusion of all other nodes present in the connector being modelled. Calculating this set based on the configuration of a connector (which is equivalent to the state of the constraint automata) is precisely what connector colouring achieves. That is, the 2-colouring model of a connector produces a set of colourings that can be equated with the transitions in the corresponding constraint automata. In Chapter 5 we compare both models in more detail, together with other models. The 3-colouring semantics has the novelty of capturing the context-dependent behaviour of connectors. The model also has the advantage of being visually appealing, as the colouring can overlay the connector. In fact in Chapter 6 we exploit the visual representation of colourings to simulate and animate $\mathcal{R}$eo connectors.

Bruni *et al* [32] propose a semantic model for CommUnity connectors, the core of which is a denotation for each primitive connector based on ticks and unticks corresponding to the presence and absence of data flow. This clearly is similar to the 2-colouring semantics, although the connectors we consider can have both loops and a larger set of primitives. As far as we are aware, these languages and formalisms do not have quite the range of expressiveness covered by the channels present in $\mathcal{R}$eo, such as *LossySync* with its subtle behaviour, nor do they require or express context-dependent behaviour, as we address in this thesis.

The *synchronised hyperedge replacement* approach [69] of modelling distributed systems using graph transformations has some similarities with $\mathcal{R}$eo, in that the synchronisation is transitive across large chunks of the graph or connector. Transformations in a graph change the structure of the graph, whereas in $\mathcal{R}$eo the structure of the connector is more static, though the internal states of buffers may change. Computation in a $\mathcal{R}$eo connector is realised when data travels though a connector and its state changes, whereas the transformation of a graph *is* computation in the graph model. Our model also resembles the Tile model [46]. Indeed, the Tile model is a general framework for the compositional description of transition systems. Farhad et al. propose an encoding of the 2-colouring and 3-colouring semantics in the Tile model [15]. The main motivation is that in the Tile Model the dynamic reconfiguration of connectors can be uniformly captured. The proposed encoding however, does not include $\mathcal{R}$eo's hide operation and does not deal with loops in connectors, yielding ill-defined semantics for those connectors.

Milner's classic SCCS [76] also appears to be an appropriate model for *implementing* the 2- and 3-colouring semantics, by mapping colours to SCCS actions, after polarising the ends joined at a node. For example, we can model the 2-colouring behaviour of a *LossySync* with ends connected to nodes named $\mathfrak{a}$ and $\mathfrak{b}$ as:

$$\mathit{LossySync}(\mathfrak{a},\mathfrak{b}) \doteq \text{‹}(\mathit{Flow}(\mathfrak{a}) \times \overline{\mathit{Flow}(\mathfrak{b})} + \mathit{Flow}(\mathfrak{a})) : \mathit{LossySync}(\mathfrak{a},\mathfrak{b})$$

Modelling the 3-colouring scheme of the same *LossySync* requires more than a simple use of the *delay* operator (δ). Actions need to be expanded to also include no-dataflow colours, in order to properly propagate the constraints they encode. One possible encoding of the *LossySync* is the following, which uses $\overline{\mathit{NoFlow}(\mathfrak{b})}$ and $\mathit{NoFlow}(\mathfrak{b})$ to denote the giving and the requiring of a reason to delay, respectively:

$$\begin{aligned}\mathit{LossySync}(\mathfrak{a},\mathfrak{b}) \doteq (\ & \mathit{Flow}(\mathfrak{a}) \times \overline{\mathit{Flow}(\mathfrak{b})} + \\ & \mathit{Flow}(\mathfrak{a}) \times \mathit{NoFlow(b)} + \\ & \mathit{NoFlow}(\mathfrak{a}) \times (\overline{\mathit{NoFlow}(\mathfrak{b})} + \mathit{NoFlow}(\mathfrak{b}))\ ) : \mathit{LossySync}(\mathfrak{a},\mathfrak{b})\end{aligned}$$

This approach to encoding $\mathcal{R}$eo in SCCS is worth further investigation.

Colouring is a natural concept and appears in various contexts throughout the literature. For example, a variant of Petri nets called Coloured Petri Nets [58, 59] exists. There different colours correspond to abstractions of various data values; thus colours are types or sorts. Colouring also appears in graph algorithms: these algorithms aim, in general, to colour different connected parts of the graph differently [3]. So, for example, a 3-colourable graph is one where each vertex can be assigned one of three colours so that no edge joins two equi-coloured vertices. Both of these uses of colouring are distinct from ours.

We believe the 3-colouring semantics is new, and has successfully been used as the basis for coordination models that enforce synchronisation and exclusion constraints in a manner that depends upon the way in which components interact with the coordination layer, especially in a distributed environment.

4

INTENTIONAL AUTOMATA MODEL

In this chapter we broaden our scope and zoom out from $\mathcal{R}$eo. We present *intentional automata*—an automata model for modelling the behaviour of concurrent systems. What characterises this operational model as opposed to others is that it explicitly models the arrival of communication (or I/O) requests, and distinguishes between communication requests and the actual communications. This gives the model an extra degree of expressiveness that becomes useful in modelling systems that need to behave differently depending on the presence or absence of pending requests in their context/environment. Connectors are a prime example of such systems, in particular context-dependent connectors. Models for quality of services (QoS) also require the distinction between the arrival of communication requests and when they fire. Typically in these models the time between the arrival and the firing of a request is precisely the delay that the model is intended to capture, and hence this information cannot be abstracted away from the model [16]. Although intentional automata have been used as the basis for models of QoS, this topic is out of the scope of this chapter. We devote our attention here solely to context-dependent connectors.

CHAPTER OVERVIEW

This chapter is organised as follows: Section 4.1 sets our basic terminology and definitions before introducing intentional automata in its two variants: deterministic and non-deterministic. Section 4.2 proposes semantic equivalences for connectors modelled with intentional automata. In Section 4.3 operations are defined to compose and perform information hiding on intentional automata. The intentional automata observational equivalence is shown to be a congruence with respect to these operations. Finally Section 4.5 concludes the chapter and discusses future work.

4.1 INTENTIONAL AUTOMATA MODELS FOR CONNECTORS

In this section we present intentional automata based models for component connectors. We associate a name to each port. Each connector defines a finite set of ports $\Sigma \subseteq \mathcal{N}ames$.

Definition 4.1.1 (request-set)**.** Consider a connector C with a set of ports Σ. A *request-set* is a subset $R \subseteq \Sigma$ of the set of ports. We denote the set of all request-sets by $\mathcal{R} = \mathcal{P}(\Sigma)$.

Given a connector C with a set of ports Σ, the different ways that the environment can interact with C are given by the set of requests $\mathcal{R} = \mathcal{P}(\Sigma)$. For a request-set $R \in \mathcal{R}$, every port in R has a request, whereas there are no requests on ports in $\Sigma \backslash R$. For instance, consider the connector of degree 2, *Sync*, with a set of ports $\Sigma = \{A, B\}$. The set $\mathcal{P}(\{A, B\}) = \{\emptyset, \{A\}, \{B\}, \{A, B\}\}$ contains all the request-sets the environment can perform on *Sync*. The empty request-set $\emptyset$ denotes that none of the ports of the connector receives a request from its environment. The request-set $\{A\}$ denotes that a request is present on port A, and no request is present on port B. Similarly, the request-set $\{B\}$ denotes that request is present on port B, but no request is present on port A. Finally the request-set $\{A, B\}$ denotes that a request is present on port A and another one is present on port B simultaneously. Presence of a request on a port is considered to be an observable property on the port.

Definition 4.1.2 (firing-set and firings)**.** Consider a connector C with a set of ports Σ. A *firing-set* is a subset $F \subseteq \Sigma$ of the set of ports. We denote the set of all firing-sets by $\mathcal{F} = \mathcal{P}(\Sigma)$.

A connector C processes one *request-set* at a time, and in response produces a (possibly empty) *firing-set* $F \subseteq \Sigma$. For example, the empty firing-set $\emptyset$ denotes *quiescence*—no firing at any of the ports. The firing-set $\{A, B\}$ denotes the simultaneous firing of ports A and B. The firing of a port is an observable event on the port.

A (possibly empty) request-set is related to a (possibly empty) firing-set. They comprise an interaction pair that constitutes an *experiment*. Experiments constitute the mechanism used to describe the behaviour of a connector.

Definition 4.1.3 (Experiment)**.** Consider a connector C with a set of ports Σ. An *experiment* $(R, F) \in \mathcal{R} \times \mathcal{F}$ is an ordered pair of a request-set R and a firing-set F.

Notation 4.1.4. We use R and R_i, with $i \in \mathbb{N}$, to range over the elements of $\mathcal{R}$. Similarly, we use F and F_i, with $i \in \mathbb{N}$, to range over the elements of $\mathcal{F}$.

4.1.1 *Deterministic Intentional Automata*

Finite automata are a natural means to describe the dynamic behaviour of systems, in particular reactive systems [1]. Finite automata are formal and rigorous and can be supported by tools for design, analysis, simulation and verification of their behaviour. We propose a particular variant of finite automata, called *intentional automata*, to model connectors.

To model a connector with an intentional automaton one can think of the automaton as being an abstract state machine, where each *state* corresponds to a possible *configuration* of the connector and the *transitions* indicate the *experiments* that take the connector from one configuration to another, not necessarily different, configuration.

Definition 4.1.5. A *deterministic intentional automaton* over the set of ports Σ is a system $\mathcal{A} = (Q, \Sigma, \delta, q_0)$, with a finite *set of states* Q; a *transition function* $\delta : Q \longrightarrow (1 + (\mathcal{F} \times Q))^{\mathcal{R}}$ that associates for every state $q \in Q$, a function $\delta(q) \in (1 + (\mathcal{F} \times Q))^{\mathcal{R}}$, where $1 = \{\Uparrow\}$, $\mathcal{R}$ are the *requests* of $\mathcal{A}$ and $\mathcal{F}$ are the *firings* of $\mathcal{A}$, and an initial state $q_0 \in Q$.

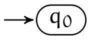
An intentional automaton $\mathcal{A} = (Q, \Sigma, \delta, q_0)$ defined over a finite non-empty set of *port names* Σ consists of a set Q of *abstract states* and a *transition function* $\delta : Q \longrightarrow (1 + (\mathcal{F} \times Q))^{\mathcal{R}}$, where $1 = \{\Uparrow\}$, $\mathcal{R} = \mathcal{P}(\Sigma)$ is the requests of $\mathcal{A}$ and $\mathcal{F} \subseteq \mathcal{P}(\Sigma)$ is the firings of $\mathcal{A}$. The transition function δ encodes the (semantics) dynamic behaviour of $\mathcal{A}$ assigning to each state $q \in Q$ a function $\delta(q) \in (1 + (\mathcal{F} \times Q))^{\mathcal{R}}$, that is, $\delta(q) : \mathcal{R} \longrightarrow 1 + (\mathcal{F} \times Q)$. For each request-set $R \in \mathcal{R}$, $\delta(q)$ either maps R to a firing set F and a new state q' denoted with the pair $(F, q') \in \mathcal{F} \times Q$; or $\delta(q)$ is *undefined* for the request set R and we have $\delta(q)(R) = \Uparrow$.

Notation 4.1.6. We write $\delta_q(R)$ instead of $\delta(q)(R)$; $\delta_q \Uparrow R$ to denote $\delta(q)(R) = \Uparrow$; and $\delta_q \Downarrow R$ when $\delta_q(R)$ is defined and in $\mathcal{F} \times Q$.

LABELLED TRANSITION DIAGRAM A labelled transition diagram for an intentional automaton $\mathcal{A} = (Q, \Sigma, \delta, q_0)$ has its vertices labelled by the states $q \in Q$; there is a directed edge represented with an arrow labelled $R \mid F$ from the vertex labelled q to the vertex labelled q' precisely when $\delta_q(R) = (F, q')$:

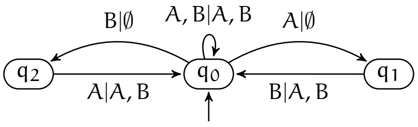
$$q \xrightarrow{R|F} q' \equiv \delta_q(R) = (F, q').$$

The initial state is distinguished by an inward-pointing arrow:



To avoid having too many arrows cluttering up a diagram, if $R_1|F_1, \ldots, R_n|F_n$ label n edges from the state q to the state q' then we simply draw one arrow from q to q' labelled $R_1|F_1 + \ldots + R_n|F_n$ instead of n arrows labelled $R_1|F_1$ to $R_n|F_n$. For all states $q \in Q$, we omit from the diagram the transitions for all request-sets R for which $\delta_q \Uparrow R$, and for the empty request-set if $\delta_q(\emptyset) = (\emptyset, q)$. We often omit the delimiting set of brackets of the request-set R and firing-set F when labelling an edge: for instance, we write $q \xrightarrow{A,B|A} q'$, instead of $q \xrightarrow{\{A,B\}|\{A\}} q'$. Irrespective of the direction of arrows, the label $R|F$ reads always from left to right: the two edges $q \xrightarrow{A,B|A} q'$ and $q' \xleftarrow{A,B|A} q$ both depict the transition $\delta_q(\{A, B\}) = (\{A\}, q')$.

Example 4.1.7. As an example of a labelled transition diagram consider the deterministic intentional automaton $\mathcal{A}_1 = (Q, \Sigma, \delta, q_0)$ depicted below:



where:

- $Q = \{q_0, q_1, q_2\}$,
- $\Sigma = \{A, B\}$, $\mathcal{R} = \mathcal{P}(\Sigma)$, $\mathcal{F} = \{A, B\}$,
- $\delta : Q \longrightarrow (1 + (\mathcal{F} \times Q))^{\mathcal{R}}$ is given by:
  $\delta_{q_0}(\emptyset) = (\emptyset, q_0)$, $\delta_{q_0}(\{A\}) = (\emptyset, q_1)$, $\delta_{q_0}(\{B\}) = (\emptyset, q_2)$, and
  $\delta_{q_0}(\{A, B\}) = (\{A, B\}, q_0)$;
  $\delta_{q_1}(\emptyset) = (\emptyset, q_1)$, $\delta_{q_1} \Uparrow \{A\}$, $\delta_{q_1}(\{B\}) = (\{A, B\}, q_0)$, $\delta_{q_1} \Uparrow \{A, B\}$;
  $\delta_{q_2}(\emptyset) = (\emptyset, q_2)$, $\delta_{q_2}(\{A\}) = (\{A, B\}, q_0)$, $\delta_{q_2} \Uparrow \{B\}$, $\delta_{q_2} \Uparrow \{A, B\}$.

Taking the automaton $\mathcal{A}_1$ as a model for the *Sync* connector we can read the automaton as follows: in the initial state q_0 the connector responds to request-set $\{A, B\}$ by firing both ports A and B without changing state; if only port A receives a request the connector changes to state q_1 without firing any port; if only port B receives a request the connector changes to state q_2 without firing any port. States q_1 and q_2 distinguish the situation in which the requests on ports A and B arrive at different times, respectively, on port A and on port B, first. Once in one of these states the connector responds only on requests performed on the other port and consequently fires both ports and returns to the initial state q_0. State q_1 represents the situation in which a request is present on port A but no request on port B has yet arrived. In state q_1, the connector responds only to requests on port B, by firing both ports A and B, and returning to state q_0. Note that $\delta_{q_1} \Uparrow \{A\}$ and $\delta_{q_1} \Uparrow \{A, B\}$. (Similarly for state q_2.)

4.1.2 *Non-deterministic Intentional Automata*

The automata given by Definition 4.1.5 are deterministic because whenever the environment externally picks a request-set in $\mathcal{R}$, this uniquely determines the firing-set in $\mathcal{F}$ and the state in Q that the automaton evolves to.

Deterministic automata therefore allow us to give semantics to connectors with deterministic behaviour only. However, often a connector has a range of possible outcomes in response to the same request-set. In these cases, the environment fails to fully control the internal evolution of the connector. The environment is still allowed to select the request-set from $\mathcal{R}$ in the current state of the connector; however, the environment does not have the ability to influence the selection between the alternative ways in which the connector can respond to this request-set. That is to say, the environment cannot determine the firing-set in $\mathcal{F}$ as the response and the state in Q that the automaton evolves into.

For the deterministic case we have that the response to a request-set in $\mathcal{R}$ by the automaton is either undefined, $\Uparrow$, or determines a single element in $\mathcal{F} \times Q$. In contrast, in the non-deterministic case, the automaton offers for each request-set in $\mathcal{R}$ a set of responses in $\mathcal{P}(\mathcal{F} \times Q)$. Thus, for non-deterministic intentional automata the transition function is of type $Q \longrightarrow \mathcal{P}(\mathcal{F} \times Q)^{\mathcal{R}}$.

Definition 4.1.8. A *non-deterministic intentional automaton* over the set of ports Σ is a system $\mathcal{A} = (Q, \Sigma, \hat{\delta}, I)$, with a *set of states* Q; a *transition function* $\hat{\delta} : Q \to \mathcal{P}(\mathcal{F} \times Q)^{\mathcal{R}}$ that associates for every state $q \in Q$, a function $\hat{\delta}_q \in \mathcal{R} \longrightarrow \mathcal{P}(\mathcal{F} \times Q)$, where $\mathcal{R}$ are the *requests* of $\mathcal{A}$ and $\mathcal{F}$ are the *firings* of $\mathcal{A}$, and a non-empty set of initial states $I \subseteq Q$.

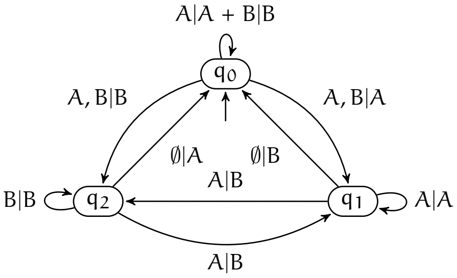
Notation 4.1.9. Whenever $I = Q$ we write simply $\mathcal{A} = (Q, \Sigma, \hat{\delta})$ to denote the intentional automaton $\mathcal{A} = (Q, \Sigma, \hat{\delta}, I)$. We use $\hat{\delta}_q \Uparrow R$ to denote $\hat{\delta}_q(R) = \emptyset$ and $\hat{\delta}_q \Downarrow R$ when $\hat{\delta}_q(R) \neq \emptyset$.

$\longrightarrow$ TRANSITION RELATION The transition function of the non-deterministic intentional automata $\hat{\delta} : Q \longrightarrow \mathcal{P}(\mathcal{F} \times Q)^{\mathcal{R}}$ can be equivalently represented by a relation $\longrightarrow_{\hat{\delta}} \subseteq Q \times \mathcal{R} \times \mathcal{F} \times Q$ defined by:

$$q \xrightarrow{R|F}_{\hat{\delta}} q' \equiv (F, q') \in \hat{\delta}_q(R).$$

Additionally, we write $q \not\rightarrow^{R}_{\hat{\delta}}$ whenever $\hat{\delta}(q, R) = \emptyset$. Often we write just $\longrightarrow$ (without subscript) whenever the corresponding transition function is clear from the context.

Example 4.1.10. Below is depicted the labeled transition diagram of a non-deterministic intentional automaton $\mathcal{A}_2 = (Q, \Sigma, \hat{\delta}, I)$:



where:

- $Q = \{q_0, q_1, q_2\}$,
- $\Sigma = \{A, B\}$, $\mathcal{R} = \mathcal{P}(\Sigma)$, $\mathcal{F} = \{\{A\}, \{B\}\}$
- $I = \{q_0\}$,
- $\hat{\delta} : Q \longrightarrow \mathcal{P}(\mathcal{F} \times Q)^{\mathcal{R}}$ is defined by:

  $\hat{\delta}_{q_0}(\emptyset) = \{(\emptyset, q_0)\}$, $\hat{\delta}_{q_0}(\{A\}) = \{(\{A\}, q_0)\}$, $\hat{\delta}_{q_0}(\{B\}) = \{(\{B\}, q_0)\}$, and $\hat{\delta}_{q_0}(\{A, B\}) = \{(\{A\}, q_1), (\{B\}, q_2)\}$;

  $\hat{\delta}_{q_1}(\emptyset) = \{(\{B\}, q_0)\}$, $\hat{\delta}_{q_1}(\{A\}) = \{(\{A\}, q_1), (\{B\}, q_2)\}$, $\hat{\delta}_{q_1} \Uparrow \{B\}$, and $\hat{\delta}_{q_1} \Uparrow \{A, B\}$;

  $\hat{\delta}_{q_2}(\emptyset) = \{(\{A\}, q_0)\}$, $\hat{\delta}_{q_2} \Uparrow \{A\}$, $\hat{\delta}_{q_2}(\{B\}) = \{(\{B\}, q_2), (\{A\}, q_1)\}$, and $\hat{\delta}_{q_2} \Uparrow \{A, B\}$.

Automaton $\mathcal{A}_2$ can be taken as a model for the *AsyncDrain* connector. In its initial state, the *AsyncDrain* accepts independent requests at ports A and B, responding by firing the port that received the request, without changing state. In case requests on both ports are present at the same time, the connector has two possible alternatives: it either moves to state q_1 choosing port A to fire; or it moves to state q_2 firing port B. State q_1 represents the situation that a request on port B is pending. In state q_1 the *AsyncDrain* either fires port B and goes back to state q_0; or receives a request on port A that it fires promptly and stays in state q_1. (Similarly for state q_2.)

INTERNAL TRANSITIONS Internal steps (or internal activity) of a connector account for the non-observable activity of a connector and are modelled in the automata by *internal transitions*. Internal activity in the connector takes place without involving the ports of the connector. Thus, internal transitions of an intentional automaton defined over the set of ports Σ are those that take place without involving ports in Σ. A transition with an empty request-set denotes a transition in the automaton that takes place when there is no requests in any of the connector ports. A transition with an empty firing-set denotes a quiescent transition in the automaton (no firing at any of the connector ports). A transition $q \xrightarrow{\emptyset|\emptyset} q'$, with $q \neq q'$, allows the automaton to change from a state q to state q' when no requests are present in the ports of the connector and as a result no ports of the connector are fired. It involves none of the ports of the automaton and is thus an internal transition.

Definition 4.1.11 (internal transition)**.** We call a transition of type $q \xrightarrow{\emptyset|\emptyset} q'$, with $q \neq q'$, we call an *internal transition*. Given an intentional automaton $\mathcal{A}$, we denote the set of all internal transitions of $\mathcal{A}$ by $\xrightarrow{\emptyset|\emptyset}_{\mathcal{A}}$.

The class of non-deterministic intentional automata subsumes the class of deterministic intentional automata: a deterministic intentional automaton is just a special case of a non-deterministic intentional automaton. In what follows, we use non-deterministic intentional automata.

4.2 CONNECTOR EQUIVALENCE

In this section, we introduce the important notions of behavioural equivalence and observational equivalence on intentional automata.

We base our notion of behavioural equivalence for intentional automata on what we perceive as observable behaviour of connectors. Observable behaviour is any manifestation of behaviour through the connector interface, that is, its visible ports. The observable behaviour of a connector consists of the interactions between the environment and the connector through its ports. The concrete structure and the implementation of the connector are completely disregarded. From the automata perspective this means that we identify automata according to their dynamics rather than their structure, the actual names of their states, or the number of their transitions. The first requirement for our behavioural equivalence is given by Requirement 4.2.1.

Requirement 4.2.1. We identify two automata as behaviourally equivalent unless there is some sequence of *requests* that the environment can perform on them that leads to different *experiments*.

Standard notions of equivalence carry over from classical theory of automata to intentional automata and thus constitute candidates for our notion of equivalence. Naturally, we start by inspecting the linear behaviour of intentional automata. The linear behaviour of automata is commonly characterised by traces [40] or more classically as the (accepted) language of the automata [79, 85].

However we seek a notion of equivalence that covers both deterministic and non-deterministic intentional automata. For non-deterministic automata, a linear equivalence identifies non-deterministic intentional automata with different branching behaviour, i. e. different experiment capabilities, as equivalent. Hence the notion of linear equivalence identify still too many automata that we want to consider as behaviourally distinct.

4.2.1 *Bisimilarity*

The classic notion of bisimulation equivalence, introduced by David Park in [82], formalises the Requirement 4.2.1 elegantly. We adapt Park's notion to our setting:

Definition 4.2.2 (bisimulation and bisimilarity)**.** Consider two intentional automata $\mathcal{A}_1 = (Q_1, \Sigma_1, \delta_1, I_1)$ and $\mathcal{A}_2 = (Q_2, \Sigma_2, \delta_2, I_2)$. A relation $\mathcal{Z} \subseteq Q_1 \times Q_2$ is called a bisimulation if for $q_1 \in Q_1$ and $q_2 \in Q_2$, if $(q_1, q_2) \in \mathcal{Z}$, then:

- $q_1 \xrightarrow{R|F}_{\delta_1} q_1'$ implies there is a $q_2' \in Q_2$ s.t. $q_2 \xrightarrow{R|F}_{\delta_2} q_2'$ with $(q_1', q_2') \in \mathcal{Z}$;
- $q_2 \xrightarrow{R|F}_{\delta_2} q_2'$ implies there is a $q_1' \in Q_1$ s.t. $q_1 \xrightarrow{R|F}_{\delta_1} q_1'$ with $(q_1', q_2') \in \mathcal{Z}$.

Two states $q_1 \in Q_1$ and $q_2 \in Q_2$ are *bisimilar*, written $q_1 \sim q_2$, if there is a bisimulation that relates them. We call $\sim$ the (strong) *bisimilarity relation*. If $\pi_1\mathcal{Z} = Q_1$ and $\pi_2\mathcal{Z} = Q_2$ we call $\mathcal{Z}$ a *full bisimulation*[1].

Proposition 4.2.3. *In Definition 4.2.2, if* $\mathcal{A}_1$ and $\mathcal{A}_2$ *are the same automaton* $\mathcal{A}_1 = \mathcal{A}_2$ *then* $\sim$ *is an equivalence relation. The union of all bisimulations is the largest bisimulation and corresponds to the bisimilarity relation* $\sim$*.*

Definition 4.2.4 (Behavioural equivalence of intentional automata)**.** Two intentional automata $\mathcal{A}_1 = (Q_1, \Sigma_1, \delta_1, I_1)$ and $\mathcal{A}_2 = (Q_2, \Sigma_2, \delta_2, I_2)$ are behaviourally *equivalent*, denoted as $\mathcal{A}_1 \sim \mathcal{A}_2$, *iff* there is a *bisimulation* $\mathcal{Z}$ that contains $I_1 \times I_2$).

The final condition states that each initial state in $\mathcal{A}_1$ is bisimilar to an initial state in $\mathcal{A}_2$, and vice versa. In case the two automata do not have a distinguished set of initial states we have by default that $I_1 = Q_1$ and $I_2 = Q_2$ which implies that $\mathcal{Z}$ must be a full bisimulation for the two automata to be equivalent.

Notation 4.2.5. If two states q_1 and q_2 are not bisimilar, we write $q_1 \nsim q_2$. Analogously we write $\mathcal{A}_1 \nsim \mathcal{A}_2$ to denote that the automata $\mathcal{A}_1$ and $\mathcal{A}_2$ are not behaviourally equivalent.

4.2.2 *Weak-bisimilarity*

The bisimilarity relation keeps track of the internal transitions, identifying states with bisimilar internal behaviour as equivalent. In the flavour of Milner's notion of observational equivalence [77] which identifies processes up-to internal actions, we use the derived transition relation $\Longrightarrow$ and the notion of *weak-bisimilarity* relation to identify intentional automata up-to internal transitions.

[1] π_1 and π_2 are the first and second projection maps of the cartesian product.

Definition 4.2.6 ($\Longrightarrow$ transition relation)**.** Let $\mathcal{A}$ be a non-deterministic intentional automaton with transition relation $\longrightarrow_\delta \subseteq Q \times \mathcal{R} \times \mathcal{F} \times Q$. We define the derived transition relation $\Longrightarrow_\delta \subseteq Q \times Q$ as the reflexive, transitive closure of $\xrightarrow{\emptyset|\emptyset}_\delta$:

$$q \Longrightarrow_\delta q' = q(\xrightarrow{\emptyset|\emptyset}_\delta)^* q'. \tag{4.1}$$

Definition 4.2.7 (weak-bisimulation and weak-bisimilarity)**.** Consider two intentional automata $\mathcal{A}_1 = (Q_1, \Sigma_1, \delta_1)$ and $\mathcal{A}_2 = (Q_2, \Sigma_2, \delta_2)$. A relation $\mathcal{W} \subseteq Q_1 \times Q_2$ on the two sets of states is called a weak-bisimulation if for $q_1 \in Q_1$ and $q_2 \in Q_2$, if $(q_1, q_2) = \mathcal{W}$, then:

- $q_1 \xrightarrow{\emptyset|\emptyset}_{\delta_1} q_1'$ implies there is a $q_2' \in Q_2$ s.t. $q_2 \Longrightarrow q_2'$ and $(q_1', q_2') \in \mathcal{W}$;
- $q_2 \xrightarrow{\emptyset|\emptyset}_{\delta_2} q_2'$ implies there is a $q_1' \in Q_1$ s.t. $q_1 \Longrightarrow q_1'$ and $(q_1', q_2') \in \mathcal{W}$;
- $q_1 \xrightarrow{R|F}_{\delta_1} q_1'$ and $R \cup F \neq \emptyset$ implies there is a $q_2' \in Q_2$ s.t. $q_2 \Longrightarrow\xrightarrow{R|F}_{\delta_2}\Longrightarrow q_2'$ with $(q_1', q_2') \in \mathcal{W}$;
- $q_2 \xrightarrow{R|F}_{\delta_2} q_2'$ and $R \cup F \neq \emptyset$ implies there is a $q_1' \in Q_1$ s.t. $q_1 \Longrightarrow\xrightarrow{R|F}_{\delta_1}\Longrightarrow q_1'$ with $(q_1', q_2') \in \mathcal{W}$.

Two states q_1 and q_2 are *weakly bisimilar*, written $q_1 \approx q_2$, if there is a weak-bisimulation that relates them. We call $\approx$ the *weak-bisimilarity relation*. If $\mathcal{W} = Q_1 \times Q_2$, we call $\mathcal{W}$ a *full weak-bisimulation*.

Proposition 4.2.8. *In Definition 4.2.7, if $\mathcal{A}_1$ and $\mathcal{A}_2$ are the same automata $\mathcal{A}_1 = \mathcal{A}_2$ then $\approx$ is an equivalence relation. The union of two weak-bisimulations is a weak-bisimulation. The union of all weak-bisimulations is the largest bisimulation and corresponds to the weak-bisimilarity relation $\approx$.*

Definition 4.2.9 (Observational equivalence of intentional automata)**.** Two intentional automata $\mathcal{A}_1 = (Q_1, \Sigma_1, \delta_1, I_1)$ and $\mathcal{A}_2 = (Q_2, \Sigma_2, \delta_2, I_2)$ are *observationally equivalent*, denoted $\mathcal{A}_1 \approx \mathcal{A}_2$, if and only if there is a weak-bisimulation $\mathcal{W} \subseteq Q_1 \times Q_2$ that contains $I_1 \times I_2$.

Notation 4.2.10. If two states q_1 and q_2 are not weakly-bisimilar, we write $q_1 \not\approx q_2$. Likewise we write $\mathcal{A}_1 \not\approx \mathcal{A}_2$ to denote that automata $\mathcal{A}_1$ and $\mathcal{A}_2$ are not observationally equivalent.

It follows immediately from the definitions that $\sim$ is included in $\approx$.

4.3 OPERATIONS ON INTENTIONAL AUTOMATA

In this section, we discuss operations on intentional automata to construct and reason about models for composite connectors. The behaviour of composite connectors can be expressed in terms of the behaviour of their component connectors. The automata operations are then used to calculate the automata semantics for a composite connector in terms of the automata semantics of its constituent connectors. These operations can be split into two categories: *composition* and *information hiding* operations. We start by discussing and defining one composition operation to compose models to define composite connectors; and we discuss and define two *information hiding* operation for abstracting away the internal ports and the internal behaviour of composite connectors.

4.3.1 *Product*

For intentional automata we define a product operation that follows the definition of the product operation $\bowtie$ in constraint automata [19]. When modelling connectors in constraint automata the information about the requests is abstracted away. The information that is registered contains solely the firings of the connector. In intentional automata both types of information are present: the requests and the firings. The product operation necessarily needs to be adapted to properly handle the information concerning the requests. We do not force the two automata to be defined over the same set of ports and because of that the product has the particularity that it allows behaviour of non-shared ports in the resulting automata.

PRODUCT OPERATION ON INTENTIONAL AUTOMATA $\_ \times \_$ is a binary infix operation on two intentional automata as its operands. Given two intentional automata $\mathcal{A}_1$ and $\mathcal{A}_2$, we write $\mathcal{A}_1 \times \mathcal{A}_2$, to denote the product automaton $\mathcal{A}$ which synchronises the behaviour of $\mathcal{A}_1$ and $\mathcal{A}_2$ on their shared ports and allows simultaneous and independent behaviour on their non-shared ports.

Definition 4.3.1 (product of intentional automata)**.** The product of two intentional automata $\mathcal{A}_1 = (Q_1, \Sigma_1, \delta_1, I_1)$ and $\mathcal{A}_2 = (Q_2, \Sigma_2, \delta_2, I_2)$ is:

$$\mathcal{A}_1 \times \mathcal{A}_2 = (Q_1 \times Q_2, \Sigma_1 \cup \Sigma_2, \delta, I_1 \times I_2)$$

where, δ is given by its transition relation $\longrightarrow_\delta$, defined by the following deduction rules:

$$\frac{q_1 \xrightarrow{R_1|F_1}_{\delta_1} q_1' \qquad q_2 \xrightarrow{R_2|F_2}_{\delta_2} q_2' \qquad F_1 \cap \Sigma_2 = F_2 \cap \Sigma_1}{(q_1, q_2) \xrightarrow{R_1 \cup R_2 | F_1 \cup F_2}_{\delta} (q_1', q_2')} \tag{4.2}$$

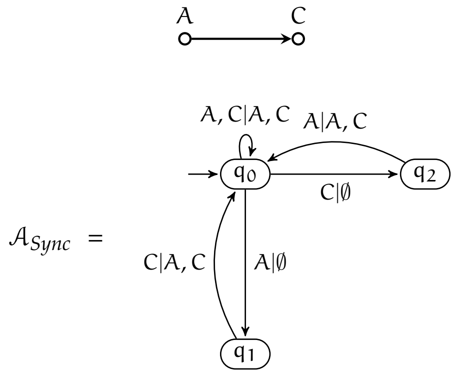


Figure 24: An intentional automaton model of the synchronous channel.

$$\frac{q_1 \xrightarrow{R_1|F_1}_{\delta_1} q_1' \qquad F_1 \cap \Sigma_2 = \emptyset}{(q_1, q_2) \xrightarrow{R_1|F_1}_{\delta} (q_1', q_2)} \tag{4.3}$$

and dually,

$$\frac{q_2 \xrightarrow{R_2|F_2}_{\delta_2} q_2' \qquad F_2 \cap \Sigma_1 = \emptyset}{(q_1, q_2) \xrightarrow{R_1|F_1}_{\delta} (q_1, q_2')} \tag{4.4}$$

Rule 4.2 applies when one transition from each automaton fires ports common to both automata. This rule models the synchronisation of the two connectors. Rules 4.3 and 4.4 apply in case a transition fires ports that belong to only one of the automata, which therefore are not forced to synchronise and can fire independently.

The product operation defined above is commutative and associative.

Example 4.3.2. Consider two automata modelling respectively a *Sync* channel with ports A and C depicted in Figure 24, and a *LossySync* channel with ports C and B depicted in Figure 25. We calculate the product automaton of the two automata depicted in Figure 26. Note that we abuse the notation and write for example q_0q_4 instead of (q_0, q_4).

There are transitions for which none of the rules to obtain the transition relation of the product automaton applies. Consider two automata $\mathcal{A}_1 = (Q_1, \Sigma_1, \delta_1)$ and $\mathcal{A}_2 = (Q_2, \Sigma_2, \delta_2)$ such that: on the one hand, $\mathcal{A}_1$ has a transition t of the form $q_1 \xrightarrow{R|F_1}_{\delta_1} q_2$ where $F_1 \cap \Sigma_2 = Y$; and on the other hand, $\mathcal{A}_2$ has no transition with a firing-set F_2 such that $F_2 \cap \Sigma_1 = Y$. In this case, obviously, for transition t,

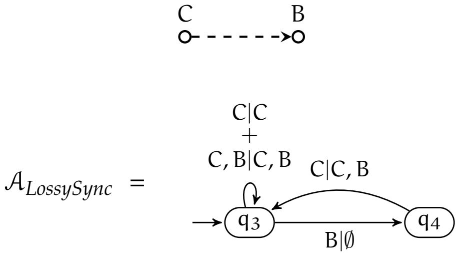


Figure 25: An intentional automaton model of the lossy synchronous channel.

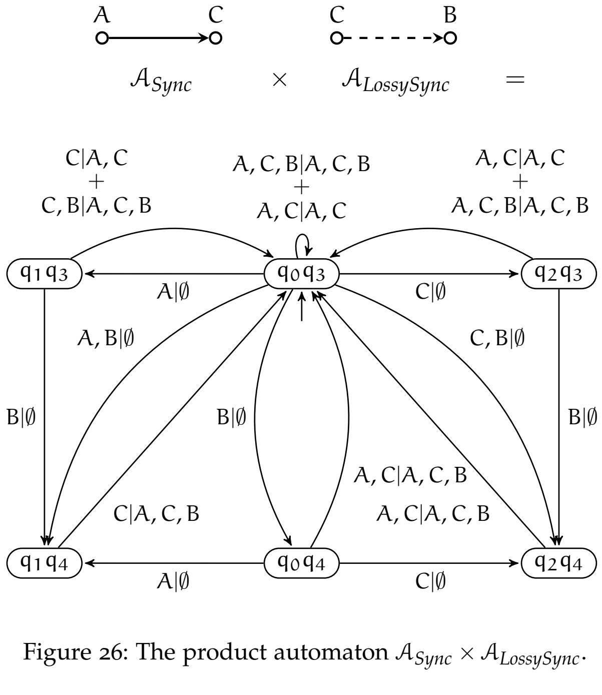


Figure 26: The product automaton $\mathcal{A}_{Sync} \times \mathcal{A}_{LossySync}$.

Rules 4.2, 4.3, and 4.4 cannot be applied. Indeed, the product operation does not include in the product automaton transitions that involve ports common to both automata, but do not agree on the firing of these common ports ($F_1 \cap \Sigma_2 = F_2 \cap \Sigma_1$).

4.3.2 *Hiding*

A model for a connector represents ports that are present in the interface of the connector. In a scenario where we consider constructing composite connectors out of basic or simpler connectors, we end up having models that include ports that do not belong to the interface of the composite connector. These are the ports through which the connector components that constitute the composite connector interact and communicate with each other. After the construction of the composite model, we may wish to *hide* a port to *internalise* the behaviour of an internal port. To serve this purpose we introduce the hiding operation on intentional automata.

The hiding operation internalizes a port h and abstracts away from the behaviour resulting from the possible different arrival times of requests on the internalised port h. The connector has, thereafter, independent control of the requests on these ports. Hidden ports are no longer available for the environment to perform experiments. Thus, the request-sets do not contain hidden ports. Additionally, hidden ports do not produce an observation when they fire, meaning that the firing-sets do not contain hidden ports either. A hidden port prioritises its possible behaviour as follows: if a request on a non-hidden port is performed by the environment, and the synchronisation with a hidden port h is a possible behaviour in the original automaton, then the post-hiding automaton considers the behaviour as if a request on h is present, and synchronisation occurs. On the other hand, if a request on h is performed, but there is no transition that can fire h then the post-hiding automaton behaves as if h had not received a request. States in the automaton that are reachable by the arrival of requests on hidden ports may become unreachable from the set of initial states in the post-hiding automaton.

HIDING OPERATION ON INTENTIONAL AUTOMATA $\exists[\_]\_$ is a binary prefix operation with a port and an intentional automaton involving that port as its operands, where the port belongs. For an intentional automaton $\mathcal{A}$ defined over ports Σ, and a port $h \in \Sigma$, we write $\exists[h]\mathcal{A}$ to denote the automaton that results from hiding the port h in automaton $\mathcal{A}$.

Definition 4.3.3 (hiding on intentional automata)**.** Consider the intentional automaton $\mathcal{A} = (Q, \Sigma, \delta, I)$, and $h \in \Sigma$. The hiding of port h of the intentional automaton $\mathcal{A}$ is defined as:

$$\exists[h]\mathcal{A} = (Q, \Sigma \setminus \{h\}, \delta', I)$$

where δ' is defined as follows:

$$\delta'_q(R) = \begin{cases} \{(F \setminus \{h\}, q') \mid (F, q') \in \delta_q(R \cup \{h\}), h \in F\} & \text{if this set is non-empty} \\ \{(F \setminus \{h\}, q') \mid (F, q') \in \delta_q(R)\} & \text{otherwise} \end{cases} \tag{4.5}$$

Example 4.3.4. Consider the automaton $\mathcal{A}_{Sync} \times \mathcal{A}_{LossySync}$ in Figure 26 from Example 4.3.2. The automaton resulting from hiding port C is given by $\exists[C](\mathcal{A}_{Sync} \times \mathcal{A}_{LossySync})$. The resulting labelled transition diagram is depicted in Figure 28. To help tracking the effects of the hiding operation, in Figure 27 we dim in gray the transitions that are abstracted away from $\mathcal{A}_{Sync} \times \mathcal{A}_{LossySync}$ in the post-hiding automaton $\exists[C](\mathcal{A}_{Sync} \times \mathcal{A}_{LossySync})$, and the occurrences of port C in request and firing sets, since this information is similarly abstracted away.

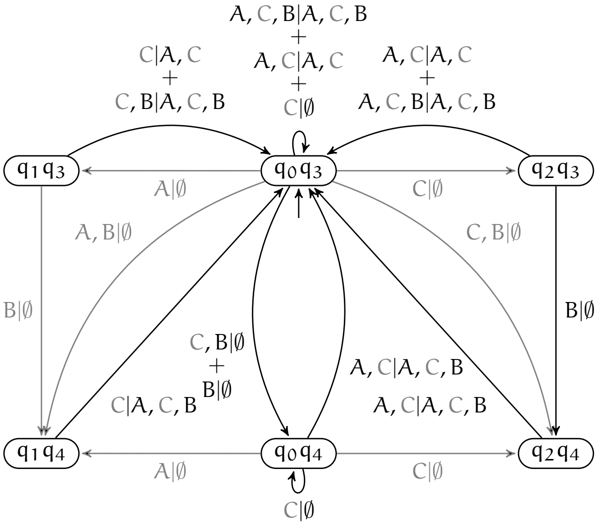


Figure 27: The labelled transition diagram obtained by dimming in gray the occurrences of port C in request and firing sets, and the transitions that are abstracted away from $\mathcal{A}_{Sync} \times \mathcal{A}_{LossySync}$ in the post-hiding automaton $\exists[C](\mathcal{A}_{Sync} \times \mathcal{A}_{LossySync})$.

Consider the state (q_0, q_3) in Figure 27 and its outgoing transitions. In Figure 29 we depict only the corresponding part of the diagram, and we mark the transitions with symbols †, §, ‡, $*$ to highlight the relation between the transitions that $\exists[C](\mathcal{A}_{Sync} \times \mathcal{A}_{LossySync})$ abstracts away from $\mathcal{A}_{Sync} \times \mathcal{A}_{LossySync}$. The same symbol relates two transitions whose request-sets differ only by the port C. For example, $(q_0, q_3) \xrightarrow{(*)\ C|\emptyset} (q_2, q_3)$ relates with $(q_0, q_3) \xrightarrow{\emptyset|\emptyset} (q_0, q_3)$. Remember, that the latter transition, for convenience, is not depicted.

On the one hand, from the pairs of transitions † and § the transitions that are abstracted away are the ones that do not have C in their request-sets. For these

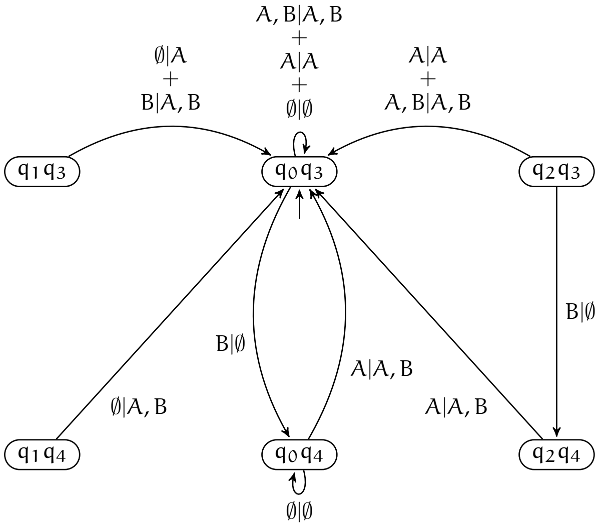


Figure 28: The labelled transition diagram of the post-hiding automaton $\exists[C](\mathcal{A}_{Sync} \times \mathcal{A}_{LossySync})$.

pairs, the transitions with C in their request-sets have firing-sets that include C. Therefore, the *if* condition of equation 4.5 applies. On the other hand, from the pairs of transitions ‡ and ∗ the transitions that are abstracted away are the ones that have C in their request-sets. For these pairs, the transitions with C in their request-sets have firing-sets that do not include C. Therefore, the *otherwise* condition applies, and the new transitions (1) and (2) are constructed.

Equation 4.5 ensures that in the post-hiding automaton, if a hidden port can fire in the context of any request-set *then* it will indeed fire.

The hiding operation is inspired by the $\mathcal{R}$eo mixed node's *pumping-station* behaviour [7]. This behaviour consists of the capability of a mixed node in a $\mathcal{R}$eo connector to push data from one of its source ends into all of its sink ends. Mixed nodes (hidden ports in intentional automata terms) cannot store data, therefore, data must flow through them. In intentional automata terms this means that there are two possible behaviours:

1) a hidden port fires together with the other ports of the connector, which models the pumping of data through the connector, as in Example 4.3.4;

2) a hidden port fires alone, which models the pumping of data through the mixed node only, for example, when pushing data from one buffer cell to another buffer cell. Example 4.3.5 which follows next, illustrates exactly this behaviour.

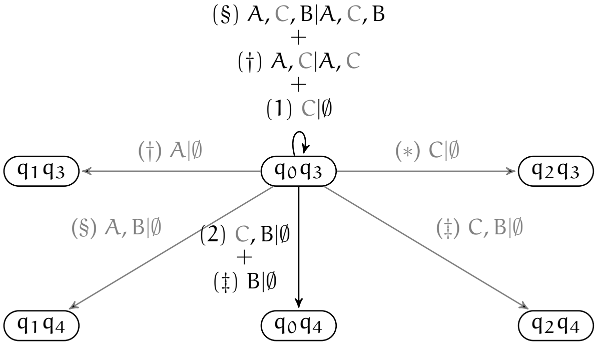
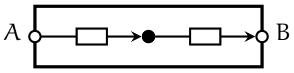


Figure 29: The symbols †, §, ‡, ∗ highlight the relation between the transitions outgoing from state q_0q_3 that $\exists[C](\mathcal{A}_{Sync} \times \mathcal{A}_{LossySync})$ abstracts away from $\mathcal{A}_{Sync} \times \mathcal{A}_{LossySync}$.

The hiding operation can be interpreted as a formalisation of the pumping station behaviour of mixed/internal $\mathcal{R}$eo nodes.

We now present an example in which we illustrate the behaviour of internal ports described in alternative 2).

Example 4.3.5. Consider the automaton model for a *FIFO*1 channel. The parameterised model is depicted in Figure 30. In this example we calculate an automaton model for the composite connector:



We compose the automaton models of two *FIFO*1 channels using the product operation, and subsequently internalise their common port C with the hiding operation. The resulting model is given by the automaton $\exists[C](\mathcal{A}_{FIFO_1}(A, C) \times \mathcal{A}_{FIFO_1}(C, B))$, depicted in Figure 32. The product automaton $\mathcal{A}_{FIFO_1}(A, C) \times \mathcal{A}_{FIFO_1}(C, B)$, is depicted in Figure 31. Consider the transition

$$\langle q_f, q_e \rangle \xrightarrow{\emptyset|\emptyset} \langle q_e, q_f \rangle \text{ in } \exists[C](\mathcal{A}_{FIFO_1}(A, C) \times \mathcal{A}_{FIFO_1}(C, B))$$

and the transition

$$\langle q_f, q_e \rangle \xrightarrow{C|C} \langle q_e, q_f \rangle \text{ in } \mathcal{A}_{FIFO_1}(A, C) \times \mathcal{A}_{FIFO_1}(C, B).$$

The former transition models the flow of data stored in the buffer cell of *FIFO*1(A, C), through the node C into the buffer cell of *FIFO*1(C, B), and this behaviour is observable. On the other hand in the latter transition this behaviour is modelled by an internal transition resulting from the hiding of the port C. In fact, note that $\langle q_e, q_f \rangle$

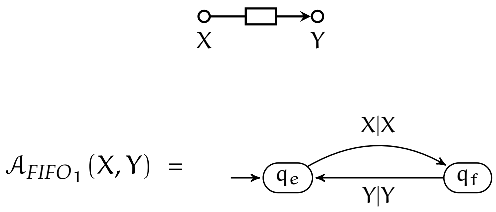


Figure 30: An intentional automaton model of the $FIFO_1$ parameterised on ports X and Y. The states q_e and q_f model the buffer cell configurations, q_e denotes the empty buffer cell, and q_f denotes the full buffer cell.

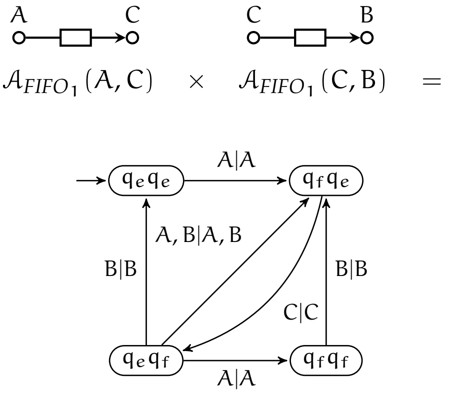


Figure 31: The product automaton $\mathcal{A}_{FIFO_1(A,C)} \times \mathcal{A}_{FIFO_1(C,B)}$.

is identified with $\langle q_f, q_e\rangle$ i. e.$\langle q_f, q_e\rangle \approx \langle q_e, q_f\rangle$, meaning that it is not possible to observationally distinguish whether the automaton is in state $\langle q_f, q_e\rangle$ or $\langle q_e, q_f\rangle$.

We proceed to show how the product and hiding operations on intentional automata behave with respect to $\approx$.

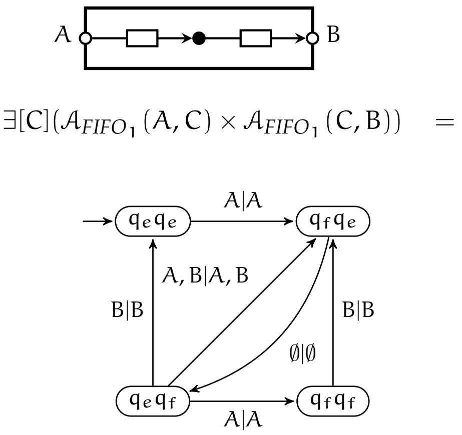
Lemma 4.3.6. *Weak-bisimilarity is preserved by the product and hiding operations on intentional automata; that is, if $\mathcal{A}_1 \approx \mathcal{A}_2$ then $\mathcal{A}_1 \times \mathcal{A} \approx \mathcal{A}_2 \times \mathcal{A}$, and $\exists[h]A_1 \approx \exists[h]A_2$, where $\mathcal{A}_1, \mathcal{A}_2, \mathcal{A}$ are intentional automata and h is a port in $\mathcal{A}_1$ and $\mathcal{A}_2$.*

Proof. We prove the result only for the product operation; the proof for the hiding operation follows similarly. Consider the intentional automata $\mathcal{A}_1 = (Q_1, \Sigma_1, \delta_1)$, $\mathcal{A}_2 = (Q_2, \Sigma_2, \delta_2)$, and $\mathcal{A} = (Q, \Sigma, \delta)$. We proceed to prove that:

$$\text{if } \mathcal{A}_1 \approx \mathcal{A}_2 \text{ then } \mathcal{A}_1 \times \mathcal{A} \approx \mathcal{A}_2 \times \mathcal{A}$$

Assuming $\mathcal{A}_1 \approx \mathcal{A}_2$, it suffices to show that a relation

$$\mathcal{W} = \{(\langle q_1, q\rangle, \langle q_2, q\rangle) \mid q_1 \in Q_1,\ q_2 \in Q_2,\ q \in Q, \text{ and } q_1 \approx q_2\}$$



$$\exists[C](\mathcal{A}_{FIFO_1}(A,C) \times \mathcal{A}_{FIFO_1}(C,B)) \quad =$$

Figure 32: Model of the composite connector $\exists[C](\mathcal{A}_{FIFO_1(A,C)} \times \mathcal{A}_{FIFO_1(C,B)})$.

is a weak-bisimulation. We have two possible types of transitions:

CASE 1 $\langle q_1, q\rangle \xrightarrow{\emptyset|\emptyset} \langle q_1', q'\rangle$. From the definition of $\_\times\_$ we have three sub-cases to check:

1.1 $q_1 \xrightarrow{\emptyset|\emptyset} q_1'$, and $q = q'$. Then $q_2 \Longrightarrow q_2'$ for some $q_2' \in Q_2$ with $q_1' \approx q_2'$, and hence $\langle q_2, q\rangle \Longrightarrow \langle q_2', q\rangle$, and $(\langle q_1', q\rangle, \langle q_2', q\rangle) \in \mathcal{W}$ as required.

1.2 $q \xrightarrow{\emptyset|\emptyset} q'$, and $q_1 = q_1'$. Then also $\langle q_2, q\rangle \Longrightarrow \langle q_2, q'\rangle$, and $(\langle q_1, q'\rangle, \langle q_2, q'\rangle) \in \mathcal{W}$ as required.

1.3 $q_1 \xrightarrow{\emptyset|\emptyset} q_1'$ and $q \xrightarrow{\emptyset|\emptyset} q'$. Then $q_2 \Longrightarrow q_2'$ for some $q_2' \in Q_2$ with $q_1' \approx q_2'$, so $\langle q_2, q\rangle \Longrightarrow \langle q_2', q'\rangle$, and $(\langle q_1', q'\rangle, \langle q_2', q'\rangle) \in \mathcal{W}$ as required.

CASE 2 $\langle q_1, q\rangle \xrightarrow{R|F} \langle q_1', q'\rangle$, $R \cup F \neq \emptyset$. From the definition of $\_\times\_$ we have three sub-cases to check:

2.1 $q_1 \xrightarrow{R|F} q_1'$, and $q = q'$ because $F \cap \Sigma = \emptyset$. Then $q_2 \Longrightarrow\xrightarrow{R|F}\Longrightarrow q_2'$ for some $q_2' \in Q_2$ with $q_1' \approx q_2'$, and hence $\langle q_2, q\rangle \Longrightarrow\xrightarrow{R|F}\Longrightarrow \langle q_2', q\rangle$, and $(\langle q_1', q\rangle, \langle q_2', q\rangle) \in \mathcal{W}$ as required.

2.2 $q \xrightarrow{R|F} q'$, and $q_1 = q_1'$ because $F \cap \Sigma_1 = \emptyset$. Then also $\langle q_2, q\rangle \Longrightarrow\xrightarrow{R|F}\Longrightarrow \langle q_2, q'\rangle$, and $(\langle q_1, q'\rangle, \langle q_2, q'\rangle) \in \mathcal{W}$ as required.

2.3 $q_1 \xrightarrow{R_1|F_1} q_1'$, and $q \xrightarrow{R'|F'} q'$ because $R = R_1 \cup R'$, and $F = F_1 \cap \Sigma = F' \cap \Sigma_1$. Then $q_2 \Longrightarrow\xrightarrow{R_1|F_1}\Longrightarrow q_2'$ for some $q_2' \in Q_2$ with $q_1' \approx q_2'$, so $\langle q_2, q\rangle \Longrightarrow\xrightarrow{R|F}\Longrightarrow \langle q_2', q'\rangle$, and $(\langle q_1', q'\rangle, \langle q_2', q'\rangle) \in \mathcal{W}$ as required.

Hence, by a symmetric argument, $\mathcal{W}$ is a weak-bisimulation. □

4.3.3 *Elimination of internal transitions*

In order to obtain the intended semantics of composite connectors, in particular context-dependent connectors, it is necessary to *eliminate* the internal transitions $\xrightarrow{\emptyset|\emptyset}$ from the models. For instance, recall Example 4.3.5 where we calculated the model for the composite connector $\exists[C](\mathcal{A}_{FIFO_1(A,C)} \times \mathcal{A}_{FIFO_1(C,B)})$[2]. The operation $\exists[\_]\_$ introduced the internal transition:

$$\langle q_f, q_e\rangle \xrightarrow{\emptyset|\emptyset} \langle q_e, q_f\rangle$$

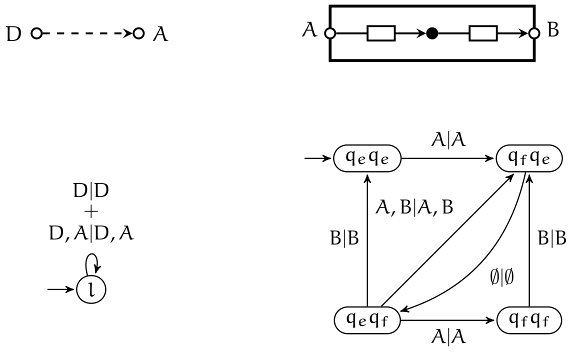
which permits the connector to *silently* change from state $\langle q_f, q_e\rangle$ to state $\langle q_e, q_f\rangle$. This transition can be interpreted as modelling the flow of the data between the FIFO buffers—from the first buffer cell into the second buffer cell. Notice that, in the state $\langle q_f, q_e\rangle$ the connector must take the internal transition and evolve to state $\langle q_e, q_f\rangle$ before it can input more data ($\xrightarrow{A|A}$), or output the data it currently stores ($\xrightarrow{B|B}$). There are other models as well wherein, just like in Example 4.3.5, it is necessary to abstract away the information entailed by internal transitions.

Example 4.3.7. Consider the automaton model with internal transitions in Figure 33a and the automaton model in Figure 33b. The latter is the automaton model derived in Example 4.3.5. The former corresponds to an underspecified version of the *LossySync* model of Figure 25, which, nevertheless, is detailed enough for this example. If we compose these two models as depicted in Figure 34 we witness the inadequacy of the obtained composite model 34b to capture the intended semantics. Consider the state $\langle l, q_f q_e\rangle$ which models the internal configuration of the connector wherein the first cell of its FIFO buffer is full and the second is empty. In this state we have two transitions that are annotated with $\star$ and $\dagger$:

$$\langle l, q_f q_e\rangle \xrightarrow{D|D^\star} \langle l, q_f q_e\rangle \text{ and } \langle l, q_f q_e\rangle \xrightarrow{D|D^\dagger} \langle l, q_e q_f\rangle \text{ respectively.}$$

Both transitions indicate that in state $\langle l, q_f q_e\rangle$ when the connector receives a request on D the port fires. The difference between these two transitions is the state into which the connector evolves. In case that the former transition is non-deterministically

[2] We recall the automaton model $\exists[C](\mathcal{A}_{FIFO_1(A,C)} \times \mathcal{A}_{FIFO_1(C,B)})$ in Figure 33 at the right.



(a) $\mathcal{A}_{LossySync}(D, A)$.

(b) $\mathcal{A}_{FIFO_2}(A, B)$.

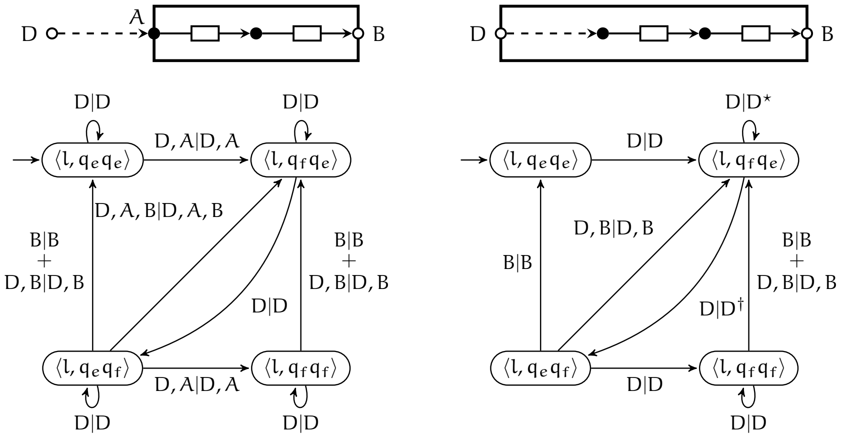
Figure 33: $\mathcal{A}_{LossySync}(D, A)$ is an automaton model of the *LossySync*. It is an *underspecified* version of the automaton model in Figure 25. $\mathcal{A}_{FIFO_2}(A, B)$ is a copy of the automaton model in Figure 4.3.5.

chosen the connector remains in the same state; whereas if the latter is chosen the connector changes to state $\langle l, q_e q_f \rangle$. In either case the data input in D is lost because the automaton remains with only one full buffer cell. According to the intended semantics, both transitions are inadequate, the expected behaviour is context-dependent and states that unless the FIFO buffer is completely full when the request-set $\{D\}$ is issued, D must fire and the data input must flow through the *LossySync* into the FIFO buffer. Despite the fact that the automaton model $\mathcal{A}_{FIFO_2}(A, B)$ captures the observational behaviour intended for a FIFO buffer with 2 cells; the model does not behave as expected when composed with context-dependent connectors like the *LossySync*. We argue that in order to reason compositionally with context-dependent connectors the internal transitions must be eliminated from the models.

For a semantics that benefits from the elimination of the internal transitions it is necessary to consider an extra operation on intentional automata to eliminate internal transitions. Based on the transition relation $\Longrightarrow$ we define the *⇒-closure* of a state. Given a state q, we denote by $[q]_\delta$ the set containing q and all the states q' related to q by the transition relation $\Longrightarrow_\delta$:

$$[q]_\delta = \{q\} \cup \left\{ q' \mid q \Longrightarrow_\delta q' \right\}.$$

States contained in $[q]_\delta$ are states that are reachable from state q by performing only internal activity. Intuitively, one can think of the set $[q]$ as defining a superstate where the automaton can transit across the states contained in it using only internal activity.



(a) $\mathcal{A}_{LossySync}(D, A) \times \mathcal{A}_{FIFO_2}(A, B)$. (b) $\exists[A](\mathcal{A}_{LossySync}(D, A) \times \mathcal{A}_{FIFO_2}(A, B))$.

Figure 34: Composite connector models of Example 4.3.7.

internal transitions elimination on intentional automata We define a unary operation $-/_{\Rightarrow}$ on intentional automata that eliminates internal transitions by applying the *$\Rightarrow$-closure* on the states of the automaton.

Definition 4.3.8 (Elimination of internal transitions on intentional automata)**.** Consider the intentional automaton $\mathcal{A} = (Q, \Sigma, \delta, I)$.

$$\mathcal{A}/_{\Rightarrow} = (Q, \Sigma, \delta_{/_{\Rightarrow}}, I)$$

where $\delta/_{\Rightarrow}$ is defined as follows:

$$\frac{q \in [q_1]_\delta \qquad q \xrightarrow{R|F}_\delta q' \qquad q_2 \in [q']_\delta \qquad R \cup F \neq \emptyset}{q_1 \xrightarrow{R|F}_{\delta_{/_{\Rightarrow}}} q_2} \tag{4.6}$$

The *$\Rightarrow$-closure* of a state is inspired by the ϵ-closure of a state in the theory of Finite Automata with Epsilon Transitions [54, Section 2.5.3]. We simply take the internal transition on intentional automata $\xrightarrow{\emptyset|\emptyset}$ as corresponding to the epsilon transition $\xrightarrow{\epsilon}$ in finite automata with epsilon transitions (ϵ-FA). The operation $-/_{\Rightarrow}$ is similar to the construction to compute the extended transition function on ϵ-FA [54, Section 2.5.4].

Remark 4.3.1. Internal transitions can arise in an intentional automata model as a result of hiding a port. With that in mind, it is sensible to consider the construction to eliminate internal transitions, as defined by the operation $-/_{\Rightarrow}$, to be also part of the hiding operation. In this case instead of having a separated operation to perform the elimination of internal transitions, we would have a hiding operation similar to the operation $\exists[\_]$ that, additionally, performs the elimination of the internal transitions. The hiding operation on constraint automata, for example, includes the elimination of the internal transitions. We opted for having a dedicated operation to eliminate the internal transitions and separate it from the hiding operation mainly because it makes the definition of the hiding operation easier to write, and makes more clear the parallel between the operation to eliminate internal transitions on intentional automata, with the operation to eliminate epsilon transitions from the classical theory of Finite Automata.

Example 4.3.9. We illustrate the operation to eliminate internal transitions by calculating the model $\mathcal{A}_{FIFO_2}(A, B)/_{\Rightarrow}$ from Figure 35. We explain some of the transitions in $\mathcal{A}_{FIFO_2}(A, B)/_{\Rightarrow}$. We start by calculating the *⇒-closure* of each state: $[q_e q_e] = \{q_e q_e\}, [q_f q_e] = \{q_f q_e, q_e q_f\}, [q_e q_f] = \{q_e q_f\}, [qfqf] = \{q_f q_f\}$. Next, we explain some transitions. For example, $q_f q_e \xrightarrow{A|A}_{/\Rightarrow} q_f q_f$ follows because $q_e q_f \in [q_f q_e]$ and $q_e q_f \xrightarrow{A|A} q_f q_f$; and similarly $q_f q_e \xrightarrow{B|B}_{/\Rightarrow} q_e q_e$ follows from $q_e q_f \in [q_f q_e]$ and $q_e q_f \xrightarrow{B|B} q_e q_e$. We have $q_e q_e \xrightarrow{A|A}_{/\Rightarrow} q_e q_f$ because $q_e q_e \xrightarrow{A|A} q_e q_f$ and $q_e q_f \in [q_f q_e]$.

We revisit now Example 4.3.7 using the automaton model without internal transitions $\mathcal{A}_{FIFO_2}(A, B)/_{\Rightarrow}$. The resulting composite connector is in Figure 36. Note that contrary to the obtained composite connector when using $\mathcal{A}_{FIFO_2}(A, B)$ the model $\exists[A](\mathcal{A}_{LossySync}(D, A) \times \mathcal{A}_{FIFO_2}(A, B)/_{\Rightarrow})$ captures the intended semantics. When the FIFO buffer of the connector is not completely full and the request-set $\{D\}$ is issued, D fires and the data flow through the *LossySync* channel into the FIFO buffer as intended. When the connector has its FIFO buffer completely full (state $q_f q_f$) if the request-set $\{D\}$ is received the port fires and the data is lost ($q_f q_f \xrightarrow{D|D} q_f q_f$).

From the definitions of observational equivalence $\approx$ and the operation $-/_{\Rightarrow}$ the following lemma follows directly.

Lemma 4.3.10. *Consider intentional automata $\mathcal{A}_1$ and $\mathcal{A}_2$ such that $\mathcal{A}_1 \approx \mathcal{A}_2$, then $\mathcal{A}_1/_{\Rightarrow} \approx \mathcal{A}_2/_{\Rightarrow}$ and $\mathcal{A}_1 \approx \mathcal{A}_1/_{\Rightarrow}$.*

4.3.4 *Observational equivalence is a congruence*

The operations defined on intentional automata in the previous section preserve the equivalence relation $\approx$, and Lemmas 4.3.6, 4.3.10 guarantee that the equivalence relation $\approx$ is fully substitutive. $\approx$ is respected by all the possible contexts. This makes the relation $\approx$ a congruence.

Theorem 4.3.11. *The relation $\approx$ is a congruence with respect to the operations on intentional automata: $\_ \times \_$, $\exists[\_]\,\_$ and $\_/_{\Rightarrow}$.*

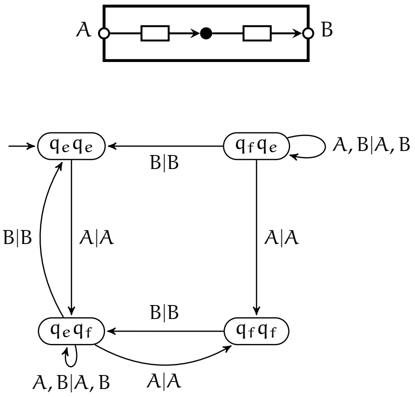


Figure 35: $\mathcal{A}_{FIFO_2}(A, B)/_{\Rightarrow}$.

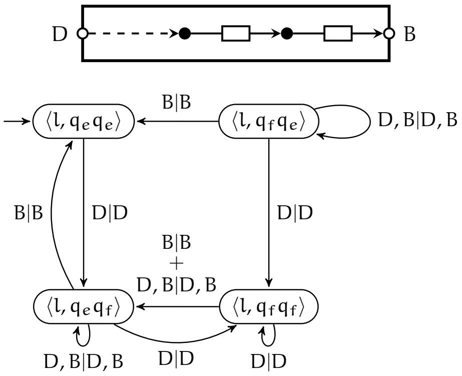

D
B
B|B
$\langle l, q_eq_e\rangle$
$\langle l, q_fq_e\rangle$
D, B|D, B
B|B
D|D
D|D
B|B
+
D, B|D, B
$\langle l, q_eq_f\rangle$
$\langle l, q_fq_f\rangle$
D, B|D, B
D|D
D|D

Figure 36: $\exists[A](\mathcal{A}_{LossySync}(D, A) \times \mathcal{A}_{FIFO_2}(A, B)/_{\Rightarrow})$.

4.4 MINIMAL MODELS

Intentional automata models are not necessarily *minimal*. Structurally, there can be states that are *unreachable* from the initial state(s). Semantically there can be states that are behaviourally equivalent modulo $\approx$, or just modulo $\sim$.

4.4.1 *Reachability*

As we mentioned in the previous section, an intentional automaton $\mathcal{A}$ with states that are reachable by the arrival of a request on a port h may become unreachable from the set of initial states in the hiding automaton $\exists[h]\mathcal{A}$. For instance the states (q_1, q_3), (q_1, q_4), (q_2, q_3), and (q_2, q_4) in the automaton model $\exists[C](\mathcal{A}_{Sync} \times \mathcal{A}_{LossySync})$ depicted in Figure 28 are not reachable from the initial state (q_0, q_3). If we consider an operation to remove the states unreachable from the initial state (q_0, q_3), the result is a new automaton whose labelled transition diagram is depicted in Figure 37.

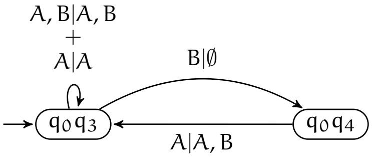


Figure 37: Labelled transition diagram obtained from $\exists[C](\mathcal{A}_{Sync} \times \mathcal{A}_{LossySync})$ depicted in Figure 28 by removing the states unreachable from the initial state (q_0, q_3).

Although they may clutter the model, unreachable states in an automaton model of a component connector do not have any impact on its semantics. When depicting the labelled transition diagrams we usually depict only the reachable states of an automaton.

4.4.2 *Collapsing observationally equivalent states*

The quotient construction induced by a congruence relation, such as observational equivalence, is an important general construction that is considered when the intention is to obtain minimal models. States that are indistinguishable are collapsed into one to yield a minimal model.

Semantically, the quotient construction is not as important as the construction that eliminates internal transitions. A model without internal transitions and its minimal counterpart obtained by applying the quotient construction have the same semantics—the two models are observationally equivalent, although the former potentially has a smaller state space.

4.5 DISCUSSION

Intentional automata are very much in the spirit of constraint automata [19]. Intentional automata transitions extend constraint automata transitions to accommodate information about the I/O operation requests, complementing the information about the I/O operation firings. This extra information, as we illustrate through this chapter, allows us to capture context-dependent behaviour. This simple extension permits us to use intentional automata as an operational model for context-dependent connectors, in the same way as constraint automata is used as an operational model for *context-independent* connectors.

We have presented notions of equivalence for intentional automata. Like in constraint automata we use bisimulation as equivalence relation. Unlike constraint automata we use weak-bisimulation as observational equivalence and show that it is a congruence with respect to the operations. The main reason to consider observational equivalence is because we consider internal transitions in the model. In constraint automata, all the transitions are observable and the hiding operation that removes hidden ports also eliminates what could be interpreted as internal transitions (transitions labeled with an empty set of ports).

The product operation on intentional automata, like in constraint automata, follows the standard construction for building finite automata for intersection. It also has similarities with composition operators of process algebra, namely, the parallel composition of labelled transition systems with synchronisation over common actions and interleaving over other actions, as in TCSP [30]. The hiding operation on constraint automata performs two separate constructions: (a) it removes all information about the hidden port; (b) it performs the elimination of the transitions labelled only with the hidden port. The hiding operation on intentional automata performs also two separate constructions: (i) it removes all information about the hidden port; (ii) it prioritises the transitions in which the hidden port has a request and fires, over the transitions in which the hidden port has a request but does not fire. Construction (i) is similar to (a), whereas construction (ii) has no counterpart in constraint automata. Construction (b) is not performed by the hiding operation on intentional automata. In intentional automata, the transitions labelled only by hidden ports become internal transitions. The internal transition in intentional automata are eliminated by the operation $-/_{\Rightarrow}$. This operation performs a construction similar to construction (b) on constraint automata; and both are similar to the elimination of ϵ-transitions in ordinary non-deterministic finite automata. Construction (ii) represents the main difference between the operations on the two automata models. In fact, construction (ii) uses information present only in intentional automata transitions regarding requested I/O operations, and based on this information, it prioritises the transitions.

Important questions about intentional automata are left for future work, namely the formalisation of the notion of refinement/underspecification. Such a notion permits to answer questions such as: Given two models of the same connector, does one model refine the other? Or vice/versa: Is one model an underspecification of the other? For example, a notion of underspecification should permits us to say that the intentional automata model we defined for *LossySync* in Figure 33a is a partial model of the model presented in Figure 25.

The study of properties of the operations on intentional automata such as commutativity, associativity, and distributivity laws are also out of the scope of this chapter and left for future work.

5

REO AUTOMATA MODEL

In this chapter we identify a specific class of intentional automata models for context-dependent $\mathcal{R}$eo connectors: *$\mathcal{R}$eo automata*. In this particular class the automata models by definition have a number of properties that are motivated by the axiomatisation of the behaviour life-cycle of a $\mathcal{R}$eo connector port.

Interestingly, it turns out that the $\mathcal{R}$eo automata admit a concise representation as *abstract configuration tables*. We use abstract configuration tables as a definition principle for $\mathcal{R}$eo automata. We define operations on configuration tables, and show that they are behaviourally faithful to the operations on intentional automata, and that the $\mathcal{R}$eo automata are closed under these operations. With this argument we claim that the $\mathcal{R}$eo automata class of intentional automata captures the context-dependent behaviour of $\mathcal{R}$eo connectors.

Due to the fact that abstract configuration tables are succinct, the operations on the configuration tables have a considerably lower computational cost, when compared to the cost that the intentional automata operations have on the $\mathcal{R}$eo automata denoted by the configuration tables.

CHAPTER OVERVIEW

In Section 5.1 we introduce the notion of configuration on intentional automata. In Section 5.2 we axiomatise the behaviour of a $\mathcal{R}$eo connector port, and motivated by the obtained axioms, we introduce the $\mathcal{R}$eo automata class of models. We present the $\mathcal{R}$eo automata models for the CONLANG primitives in Section 5.3. Next, in Section 5.5.1, we define the operations on configuration tables, and study the closure properties of the $\mathcal{R}$eo automata class with respect to these operations. We briefly discuss other classes of intentional automata models in Section 5.6. Finally, in Section 5.7, we conclude this chapter with a brief discussion to compare the models introduced in this thesis with other existing models and present the most relevant questions left to address in future work.

5.1 $\mathcal{R}$EO CONNECTORS CONFIGURATION

Traditionally, semanticists consider the state of a system to have some minimal structure conveying information about the configuration of the system, namely, about the internal memory of the system and the environment in which the system is currently being evaluated. The intentional automata models of connectors we considered in the previous chapter, have abstract states. When modelling $\mathcal{R}$eo connectors, states correspond to configurations of the connector and therefore we will consider states that have structure according to the configuration of a $\mathcal{R}$eo connector.

A connector *configuration* is partitioned into two parts: the *internal configuration* and the *external configuration*. An *internal configuration* is an abstract representation of the internal memory of the connector. An internal configuration is denoted by an element $s \in S$, where S is the set of all internal configurations of the connector. The *external configuration* of a connector describes the status of the connector's interface and is denoted by a set $P \subseteq \Sigma$, where Σ is the set of ports of the connector. The intuition is that in a given configuration (s, P), the set P indicates the ports of the connector that have a *pending request*. We shall refer to these as *pending ports*. These are ports that have received a request in previous evaluation steps and for which the request has not been handled until now. Obviously, if a port is not in P, then this port has no pending requests.

Definition 5.1.1 (Configuration)**.** Consider a connector with a set of internal configurations S and a set of ports Σ. A *configuration* is a pair (s, P) consisting of an internal configuration $s \in S$ and an external configuration $P \subseteq \Sigma$. The set of all *configurations* of the connector is given by $S \times \mathcal{P}(\Sigma)$. Given a configuration (s, P), we say p is *a pending port* or *port* p *has a pending request* if $p \in P$. All the configurations of a connector are initial, unless a subset of configurations $I \subseteq S \times \mathcal{P}(\Sigma)$ is defined as initial.

We now turn our attention from the configurations to the evaluation steps of a connector. Recall the dynamics of intentional automata, i. e., the transition relation $\longrightarrow_\delta \subseteq Q \times \mathcal{R} \times \mathcal{F} \times Q$. We define the set of states Q as the set of configurations $S \times \mathcal{P}(\Sigma)$. A transition:

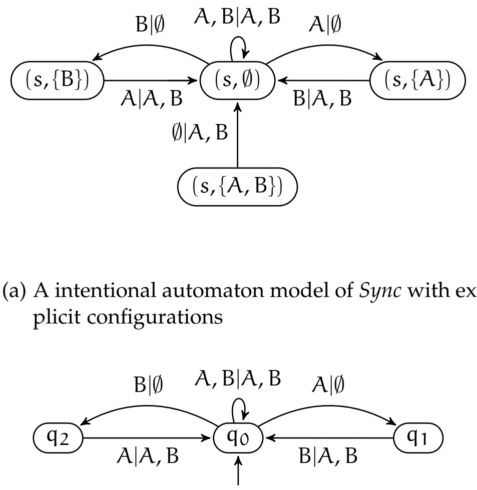
$$(s, P) \xrightarrow{R|F} (s', P')$$

models an *evaluation step* of a connector. The connector changes from configuration (s, P) to configuration (s', P') by evaluating the request-set R and producing the firing-set F.

Definition 5.1.2 (Automaton model of a connector)**.** Consider a connector C with a set of ports Σ, and a set of internal configurations S. An *automaton model* of C is a non-deterministic intentional automaton $\mathcal{A} = (Q, \Sigma, \delta)$ where $Q = S \times \mathcal{P}(\Sigma)$. We call an *evaluation step* of C a transition $(s, P) \xrightarrow{R|F} (s, P') \in\longrightarrow_\delta$. We write $\mathcal{A}_C$ to denote the automaton model of C.

In Figure 38 (a), we have the transition diagram of $\mathcal{A}_{Sync}$, an automaton model of the $\mathcal{R}$eo synchronous channel. *Sync* is a memoryless connector and therefore has a singleton set $S = \{s\}$ as its set of internal configurations. It has two ports, $\Sigma = \{A, B\}$. Thus its $Q = S \times \mathcal{P}(\Sigma) = \{(s, \emptyset), (s, \{A\}), (s, \{B\}), (s, \{A, B\})\}$ is its set of configurations of $\mathcal{A}_{Sync}$. The initial configuration is given by the singleton set $I = \{(s, \emptyset)\}$. In its initial configuration *Sync* has the two ports A and B available. If a *write* I/O operation is performed on port A and simultaneously a *take* I/O operation is performed on port B, then the two operations succeed simultaneously and both ports again become available for other operations. This is modelled in the automaton by the evaluation step $(s, \emptyset) \xrightarrow{A,B|A,B} (s, \emptyset)$. Alternatively, when *Sync* is in the initial configuration, if a *write* is performed on port A, and no *take* is performed on port B, then the *write* does not succeed. Instead it remains pending on port A. In the automaton model, this alternative is modelled by the evaluation step $(s, \emptyset) \xrightarrow{A|\emptyset} (s, \{A\})$. A last alternative, when *Sync* is in the initial configuration, is if a *take* operation is performed on port B and no *write* operation is performed on A. In this case the *take* operation does not succeed and remains pending on port B. This behaviour is modelled in the automaton model by the evaluation step $(s, \emptyset) \xrightarrow{B|\emptyset} (s, \{B\})$. In the configuration $(s, \{A\})$, *Sync* has only the port B available. When a *take* operation is performed on port B, then the pending *write* operation on port A and the *take* operation in port B succeed simultaneously, and the connector returns to its initial configuration with both ports available. This behaviour corresponds to the evaluation step $(s, \{A\}) \xrightarrow{B|A,B} (s, \emptyset)$. Conversely, in the configuration $(s, \{B\})$, *Sync* has only the port A available. If a *write* operation is performed on port A then the pending *take* operation on port B and the *write* operation on port A succeed simultaneously and the connector returns to its initial configuration with both ports available. This is modelled by the evaluation step $(s, \{B\}) \xrightarrow{A|A,B} (s, \emptyset)$. The configuration $(s, \{A, B\})$ is an unreachable configuration.

In Figure 38 (b), we recall the intentional automaton $\mathcal{A}$ over the set of ports $\Sigma = \{A, B\}$ defined in Example 4.1.7. Remember that we interpreted $\mathcal{A}$ as a model for the $\mathcal{R}$eo synchronous channel. If we compare this transition diagram with the transition diagram of $\mathcal{A}_{Sync}$, we notice that the only difference is that the states in $\mathcal{A}$ abstract from the configuration structure present in the states of $\mathcal{A}_{Sync}$. Equating $q_0 = (s, \emptyset)$, $q_1 = (s, \{A\})$, and $q_2 = (s, \{B\})$ we indeed obtain the same model. The information in the states is only suggestive like comments. The information captured in each state is redundant and can be derived from its incoming and outgoing transitions.



(a) A intentional automaton model of *Sync* with explicit configurations

(b) The intentional automaton model of *Sync* from Example 4.1.7

Figure 38: The intentional automata model of *Sync* with and without configurations.

We choose to have the information explicitly to improve the understandability of the models and to simplify the formulation of the definitions that follow.

Moreover we cannot identify classes of automata models for connectors according to properties of the evaluation steps. The evaluation steps, modelled by the dynamics of the automata, have different properties depending on the type of connector model. In the next section we discuss the properties of $\mathcal{R}$eo connectors. We define a class of automata for $\mathcal{R}$eo connectors and we treat this class in detail. We briefly discuss other classes in Section 5.6.

5.2 $\mathcal{R}$eo automata

Ports in $\mathcal{R}$eo connectors interact in a particular manner with the environment[1], which allows us to infer important invariants for the evaluation steps of automata models for $\mathcal{R}$eo connectors.

In an *evaluation step* of a $\mathcal{R}$eo connector:

① a port can fire only if it either has already a pending request, or receives a request in this step;

② when it receives a request, a port either fires in this step or becomes pending;

③ a port with a pending request, either fires in this step or it remains pending;

[1] Recall the connector port life cycle in Figure 2.

④ a port is pending after the evaluation step only if the port receives a request, or it was already pending;

⑤ a port with a pending request is unavailable to receive requests;

⑥ a port that fires cannot become/remain pending.

Consider an evaluation step $(s, P) \xrightarrow{R|F} (s', P')$ of an automaton model for a $\mathcal{R}$eo connector. We examine how the properties of $\mathcal{R}$eo ports ①, ②, ③, ④, ⑤ and ⑥ can be formalised. According to ①, a port $p \in F$ (p fires) only if $p \in P$ (p is pending) or $p \in R$ (p receives a request). We have that $F \subseteq R \cup P$. According to ②, a port $p \in R$ (p receives a request) implies that $p \in F$ (p fires) or $p \in P'$ (p becomes pending). We have that $R \subseteq F \cup P'$. According to ③, a pending port $p \in P$ implies $p \in F$ (p fires) or $p \in P'$ (p remains pending). We have that $P \subseteq F \cup P'$. According to ④, a port $p \in P'$ (p is pending after the evaluation step) only if $p \in R$ (p receives a request) or $p \in P$ (p was already pending). Hence, $P' \subseteq R \cup P$. According to ⑤, a port $p \in P$ (p is pending) implies $p \notin R$ (p cannot receive a request). That is $P \cap R = \emptyset$. Finally, according to ⑥, a port $p \in F$ (p fires) implies $p \notin P'$ (p cannot become/remain pending). Thus, $F \cap P' = \emptyset$.

The evaluation steps of an automata model, $(s, P) \xrightarrow{R|F} (s', P')$, for a $\mathcal{R}$eo connector are such that:

① $F \subseteq R \cup P$ ② $R \subseteq F \cup P'$ ③ $P \subseteq F \cup P'$ ④ $P' \subseteq R \cup P$

⑤ $P \cap R = \emptyset$ ⑥ $F \cap P' = \emptyset$.

Properties ① to ④ can be equivalently and more concisely expressed by:

❶ $P \cup R = F \cup P'$

Lemma 5.2.1. *① to ④ ⇔ ❶.*

Proof. ⇒) Assuming ①, ②, ③, and ④ we want to prove ❶:

$P \cup R = F \cup P'$.

⇒) $P \cup R \subseteq F \cup P'$

$$\begin{aligned} p \in P \cup R &\Leftrightarrow p \in R \vee p \in P && \text{(def. } \cup\text{)} \\ &\Rightarrow p \in F \cup P' \vee p \in P && \text{②} \\ &\Rightarrow p \in F \cup P' \vee p \in F \cup P' && \text{③} \\ &\Leftrightarrow p \in F \cup P' && \text{(def. } \vee\text{)} \end{aligned}$$

⇐) $F \cup P' \subseteq P \cup R$

$$\begin{aligned} p \in F \cup P' &\Leftrightarrow p \in F \vee p \in P' && \text{(def. } \cup\text{)} \\ &\Rightarrow p \in R \cup P \vee p \in P' && \text{①} \\ &\Rightarrow p \in R \cup P \vee p \in R \cup P && \text{④} \\ &\Leftrightarrow p \in R \cup P && \text{(def. } \vee\text{)} \end{aligned}$$

⇒) ❶ ⇒ ① *to* ④ is trivial. □

Property ❶ can be interpreted in various ways. For one, if we interpret it elementwise we can read it as saying that a $\mathcal{R}$eo connector, in a given configuration, takes each pending port ($p \in P$) and requested port ($p' \in R$), and either fires it ($p \in F$ and/or $p' \in F$) or keeps it pending ($p \in P'$ and/or $p' \in P$, accordingly).

Assuming ❶ we rule out the possibility that a pending request can *time-out*. In the context of this thesis we say that a pending request *times-out* if an evaluation step relates one state where we have a pending request p to another state where p is not pending, without firing p.

Definition 5.2.2 (A request time-out)**.** A pending request on a port p *times-out*, if we have an evaluation step $(s, P) \xrightarrow{R|F} (s', P')$ where $p \in P \cup R$, $p \notin F \cup P'$.

We discuss models that consider *time-out* in Section 5.6.

So far, we have identified three properties that we expect from our automata models of $\mathcal{R}$eo connectors: ❶, ⑤, and ⑥. These are properties that follow from the original description of $\mathcal{R}$eo [7]. For the additional property we are about to propose this is not necessarily the case.

Browsing through the $\mathcal{R}$eo literature one encounters a plethora of connector specifications, ranging from channels to composite connectors. We observe that all of them, as expected, abide by ❶, ⑤, and ⑥. Additionally, and remarkably, none of the $\mathcal{R}$eo connector specifications distinguishes between the set of pending ports and the set of requested ports when deciding which ports to fire. Considering an evaluation step of an automaton model for a $\mathcal{R}$eo connector, this translates to the following equation:

$$\delta_{(s,P)}(R) = \delta_{(s,\emptyset)}(R \cup P) \tag{5.1}$$

Equivalently, given two evaluation steps $(s, P) \xrightarrow{R|F} q$ and $(s, \emptyset) \xrightarrow{R'|F'} q'$:

$$P \cup R = R' \;\Leftrightarrow\; F = F' \text{ and } q = q'.$$

Equation 5.1 has an important implication: *all transitions from a state of the form* (s, P)*, for arbitrary* s *and* P*, are determined by the transitions from the state* $(s, \emptyset)$.

Equation 5.1 and property ⑤ imply an additional property that characterises the transition function δ for automata models of $\mathcal{R}$eo connectors present in the literature:

$$❷\quad \delta_{(s,P)}(R) = \begin{cases} \delta_{(s,\emptyset)}(R \cup P) & \text{if } R \cap P = \emptyset \\ \emptyset & \text{otherwise} \end{cases}$$

Property ⑤ states that for an arbitrary $\mathcal{R}$eo automaton transition we have that $R \cap P = \emptyset$. Hence the side conditions of ❷.

Here is an example. The automaton model for the *Sync* channel of Figure 38(a) is an example of a $\mathcal{R}$eo connector present in the literature. Obviously ❷ holds: $\delta_{(s,\{A\})}(\{B\}) = \delta_{(s,\emptyset)}(\{B\} \cup \{A\}) = (\{A,B\},(s,\emptyset))$ and $\delta_{(s,\{B\})}(\{A\}) = \delta_{(s,\emptyset)}(\{A\} \cup \{B\}) = (\{A,B\},(s,\emptyset))$.

Consider a firing-set that involves multiple ports. By property ❶, the environment must perform requests on these ports before they can fire. Property ❷ says that the firing is produced independently of the order in which the environment executes the requests on the ports that subsequently fire. Therefore we can identify the different ordering possibilities as a single one and reduce the complexity of the model.

The $\mathcal{R}$eo connectors from the literature are expressive and constitute an interesting class to study in its own. These connectors enjoy the properties ❶, ❷, and ⑥. We proceed to define a particular class of non-deterministic intentional automata models by restricting the general definition with the properties ❶, ❷, and ⑥. As discussed, the class of intentional automata models that have properties ❶, ⑤, and ⑥ is expressive enough to provide semantics for $\mathcal{R}$eo connectors[2]. Note that property ❷ holds whenever ⑤ holds but not vice versa. Property ⑤ holds only if for a given request-set $\mathcal{R}$ and a configuration (s,P) an evaluation step is indeed defined, $\delta_{(s,P)} \Downarrow (R)$. By considering ❷ instead of ⑤ we further restrict the general class of intentional automata models and focus on models that provide semantics to the $\mathcal{R}$eo connectors defined in the literature. We call this class *$\mathcal{R}$eo automata*.

Definition 5.2.3 ($\mathcal{R}$eo automata model)**.** Consider a connector C with ports Σ, internal configurations S, and a set of initial configurations $I \subseteq S \times \mathcal{P}(\Sigma)$. A *$\mathcal{R}$eo automaton* for C is a non-deterministic intentional automaton $\mathcal{A}_C = (Q,\Sigma,\delta,I)$ with states $Q = S \times \mathcal{P}(\Sigma)$ and transition function $\delta : Q \to \mathcal{P}(\mathcal{F} \times Q)^{\mathcal{R}}$ that associates with every state $q = (s,P)$ a function $\delta_q : \mathcal{R} \longrightarrow \mathcal{P}(\mathcal{F} \times Q)$ such that:

❶⑥ $$\langle F,(s',P')\rangle \in \delta_{(s,P)}(R) \implies P \cup R = F \cup P' \text{ and } F \cap P' = \emptyset$$

❷ $$\delta_{(s,P)}(R) = \begin{cases} \delta_{(s,\emptyset)}(R \cup P) & \text{if } R \cap P = \emptyset \\ \emptyset & \text{otherwise} \end{cases}$$

Notation 5.2.4. We use $C = (S,\Sigma,\mathcal{A}_C)$ to denote a connector with name C, ports Σ, internal configurations S, and semantics given by a $\mathcal{R}$eo automaton $\mathcal{A}_C = (Q,\Sigma,\delta,I)$ where $Q = S \times \mathcal{P}\Sigma$.

In a $\mathcal{R}$eo automaton, it follows from properties ❶⑥ and ❷ that for each internal (memory) configuration s the transitions from the state $(s,\emptyset)$ are enough to characterise the transitions of all states of the form (s,P). Furthermore, for any transition of the form $(s,\emptyset) \xrightarrow{R|F} (s,P)$ we have that $R = F \cup P$ and $F \cap P = \emptyset$.

[2] Strictly speaking a class of intentional automata models that have properties ❶, ⑤, and ⑥ does not provide semantics to $\mathcal{R}$eo connectors where a pending request on a port can *time-out*.

Properties ❶⑥ and ❷ make it possible to turn a partial intentional automaton that defines only the transition function for the states of the form $(s, \emptyset)$ into a fully specified intentional automaton.

To represent a partial intentional automaton that specifies the transition function only for states of the form $(s, \emptyset)$ we define a concise tabular representation called *configuration table*.

Definition 5.2.5 (Configuration table)**.** A *configuration table* for a $\mathcal{R}$eo connector $C = (S, \Sigma, \mathcal{A}_C)$, denoted by θ_C, is a table such that:

- for each request-set $R \subseteq \Sigma$ there is one row labelled by R;
- for each configuration $s \in S$ there is a column labelled by s;
- and each cell of the table at the intersection of row R with column s contains the set $\delta_{(s,\emptyset)}(R) \subseteq \mathcal{P}(\mathcal{F} \times Q)$.

If $S = \{s_0, \ldots, s_m\}$, $\mathcal{P}(\Sigma) = \{R_0, \ldots, R_n\}$, with $m, n \in \mathbb{N}$ then:

	s_0	$\cdots$	s_m
R_0	$\delta_{(s_0,\emptyset)}(R_0)$	$\cdots$	$\delta_{(s_m,\emptyset)}(R_0)$
$\vdots$	$\vdots$	$\ddots$	$\vdots$
R_n	$\delta_{(s_0,\emptyset)}(R_n)$	$\cdots$	$\delta_{(s_m,\emptyset)}(R_n)$

Proposition 5.2.6. *Consider a $\mathcal{R}$eo connector $C = (S, \Sigma, \mathcal{A}_C)$. The configuration table θ_C fully determines the transition function of $\mathcal{R}$eo automaton $\mathcal{A}_C$.*

Proof. Follows directly by Definition 5.2.5 of configuration table and from property ❷ in Definition 5.2.3. □

Example 5.2.7 ($\mathcal{R}$eo automata model for *Sync*)**.** Consider the $\mathcal{R}$eo connector

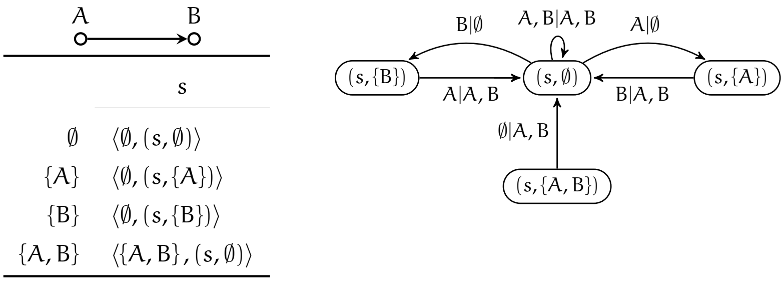
$$\mathit{Sync} = (\{s\}, \{A, B\}, \mathcal{A}_{\mathit{Sync}}).$$

The configuration table θ_{Sync} depicted in Figure 39, on the left, defines the transition function of the $\mathcal{R}$eo automaton $\mathcal{A}_{\mathit{Sync}}$. The corresponding labelled transition diagram is depicted on the right.

Notation 5.2.8. The set labelling each cell of a configuration table is depicted without the surrounding brackets.

Consider for instance $\delta_{(s,\{A\})}(\{B\})$. By the *if* condition of ❷, we know that:

$$\delta_{(s,\{A\})}(\{B\}) = \delta_{(s,\emptyset)}(\{A\} \cup \{B\}).$$



	s
$\emptyset$	$\langle\emptyset,(s,\emptyset)\rangle$
$\{A\}$	$\langle\emptyset,(s,\{A\})\rangle$
$\{B\}$	$\langle\emptyset,(s,\{B\})\rangle$
$\{A,B\}$	$\langle\{A,B\},(s,\emptyset)\rangle$

Figure 39: θ_{Sync} and the labelled transition diagram of $\mathcal{A}_{Sync}$.

Looking-up in the configuration table, in the intersection of column s with the row $\{A,B\}$, we know that $\delta_{(s,\emptyset)}(\{A,B\}) = \langle\{A,B\},(s,\emptyset)\rangle$. Therefore we have that $\delta_{(s,\{A\})}(\{B\}) = \langle\{A,B\},(s,\emptyset)\rangle$. Hence we have in the corresponding transition diagram on the right, a transition from $(s,\{A\})$ to $(s,\emptyset)$ labelled by the request-set $\{B\}$ and firing-set $\{A,B\}$. If we now consider for instance $\delta_{(s,\{B\})}(\{B\})$. By the *otherwise* condition of ❷ we have that $\delta_{(s,\{B\})}(\{B\}) = \emptyset$. Therefore in the transition diagram we have that $\delta_{(s,\{B\})} \Uparrow (\{B\})$, explaining why the transition does not exist.

Note that the automaton has 3 states but its transitions are completely defined by those of the single state $(s,\emptyset)$.

Definition 5.2.5 defines the cells of a configuration table for a given $\mathcal{R}$eo connector C in terms of the $\mathcal{R}$eo automaton $\mathcal{A}_C$. Next, we generalise, and define a configuration table exclusively in terms of an abstract set of internal configurations, and an abstract set of ports. Proposition 5.2.10 relates an abstract configuration table with a $\mathcal{R}$eo automaton model.

Definition 5.2.9 (Abstract configuration table)**.** An *abstract configuration table* over a set of internal configurations S and a set of ports Σ, denoted by $\theta(S,\Sigma)$, is a table such that:

- for each $s \in S$, there is one column labelled by s;
- for each $R \subseteq \Sigma$, there is one row labelled by R;
- at each cell of the table at the intersection of row R with column s we have a set, denoted $\theta\langle s,R\rangle$, such that $\theta\langle s,R\rangle \subseteq \mathcal{P}(\Sigma) \times (S \times \mathcal{P}(\Sigma))$, and for all $\langle F,(s',P')\rangle \in \theta\langle s,R\rangle$, we have $R = F \cup P', F \cap P' = \emptyset$.

If $S = \{s_0,\ldots,s_m\}$, $\mathcal{P}(\Sigma) = \{R_0,\ldots,R_n\}$, $m,n \in \mathbb{N}$ then $\theta(S,\Sigma)$ has the form:

	s_0	$\cdots$	s_m
R_0	$\theta\langle s_0, R_0\rangle$	$\cdots$	$\theta\langle s_m, R_0\rangle$
$\vdots$	$\vdots$	$\ddots$	$\vdots$
R_n	$\theta\langle s_0, R_n\rangle$	$\cdots$	$\theta\langle s_m, R_n\rangle$

We denote the family of all abstract configuration tables over an arbitrary S and Σ by $\Theta(S, \Sigma)$.

Proposition 5.2.10. *An abstract configuration table $\theta(S, \Sigma)$ defines a $\mathcal{R}$eo automaton (Q, Σ, δ) where:*

- $Q = S \times \mathcal{P}(\Sigma)$,
- $\delta_{(s,P)}(R) = \begin{cases} \theta\langle s, P \cup R\rangle & \textit{if } P \cap R = \emptyset \\ \emptyset & \textit{otherwise} \end{cases}$

and vice versa, (S, Σ, δ) *defines an abstract configuration table* $\theta(S, \Sigma)$.

Proof.
$(\Longrightarrow)$ Consider an abstract configuration table $\theta(S, \Sigma)$, $s \in S$ and $P, R \subseteq \Sigma$. We want to prove that the automaton $\mathcal{A} = (Q, \Sigma, \delta, I)$ obtained by the proposition above is a $\mathcal{R}$eo automaton. We have to show that the transition function δ is according to the definition of a $\mathcal{R}$eo automaton:

1) Show that $\delta_{(s,P)}(R) \subseteq \mathcal{P}(\mathcal{F} \times (S \times \mathcal{P}(\Sigma)))$.

   If $P \cap R \neq \emptyset$ then $\delta_{(s,P)}(R) = \emptyset$, and $\emptyset \subseteq \mathcal{P}(\mathcal{F} \times (S \times \mathcal{P}(\Sigma)))$.

   If $P \cap R = \emptyset$ then $\delta_{(s,P)}(R) = \theta\langle s, P \cup R\rangle$. $\theta\langle s, P \cup R\rangle \subseteq \mathcal{P}(\Sigma) \times (S \times \mathcal{P}(\Sigma))$, and by Definition 4.1.2 we have that $\mathcal{F} = \mathcal{P}(\Sigma)$, hence $\theta\langle s, P \cup R\rangle \subseteq \mathcal{P}(\mathcal{F} \times (S \times \mathcal{P}(\Sigma)))$.

2) Show that δ is such that property ❶⑥ holds.

   If $P \cap R \neq \emptyset$ then there is no transition defined, hence the premise of ❶⑥ is absurd and ❶⑥ trivially holds.

   If $P \cap R = \emptyset$ then $\delta_{(s,P)}(R) = \theta\langle s, P \cup R\rangle$.

   $$\begin{aligned} \langle F, (s', P')\rangle \in \delta_{(s,P)}(R) &\Rightarrow \langle F, (s', P')\rangle \in \theta\langle s, P \cup R\rangle \\ &\Rightarrow P \cup R = F \cup R' \text{ and } F \cap P' = \emptyset \end{aligned}$$

   Hence ❶⑥ holds.

3) Show that δ is such that property ❷ holds.

   If $P \cap R \neq \emptyset$ then $\delta_{(s,P)}(R) = \emptyset$, and ❷ holds.

   If $P \cap R = \emptyset$ then $\delta_{(s,P)}(R) = \theta\langle s, R \cup P\rangle$. By definition, $\theta\langle s, R \cup P\rangle = \delta_{(s,\emptyset)}(R \cup P)$, and ❷ holds.

Therefore δ is defined according to Definition 5.2.3. Thus $\mathcal{A} = (Q, \Sigma, \delta, I)$ is a $\mathcal{R}$eo automaton.
($\Longleftarrow$) Follows similarly. □

The proposition justifies the slogan:

> *Abstract configuration tables provide a* definition principle *for $\mathcal{R}$eo connector models—an abstract configuration table $\theta(S, \Sigma)$ defines a $\mathcal{R}$eo automaton model for a $\mathcal{R}$eo connector with a set of ports Σ and a set of memory configurations S.*

Notation 5.2.11. Given a configuration table $\theta(S, \Sigma)$ (a $\mathcal{R}$eo automaton (S, Σ, δ)) we denote by $[\![\theta(S, \Sigma)]\!]_R$ ($[\![(S, \Sigma, \delta)]\!]_T$) the $\mathcal{R}$eo automaton defined by $\theta(S, \Sigma)$ (the configuration table defined by (S, Σ, δ)) according to the proposition above. We sometimes use the term $\mathcal{R}$eo model to refer to the semantics of a connector defined by either a $\mathcal{R}$eo automaton or a configuration table.

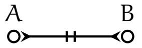
Next, we define the semantics of the CONLANG primitives using configuration tables, just like we did in Example 5.2.7 for the *Sync*. To illustrate how succinct the configuration tables are compared with the equivalent $\mathcal{R}$eo automata, we also depict the labelled transition diagrams for the $\mathcal{R}$eo automata denoted by the configuration tables.

5.3 $\mathcal{R}$EO PRIMITIVES

SYNCHRONOUS CHANNELS The two synchronous channels *SyncDrain* and *SyncSpout* have a $\mathcal{R}$eo model identical to the *Sync* channel because contrary to connector colouring models, the models in this chapter abstract from the polarity of ports and dataflow directions.

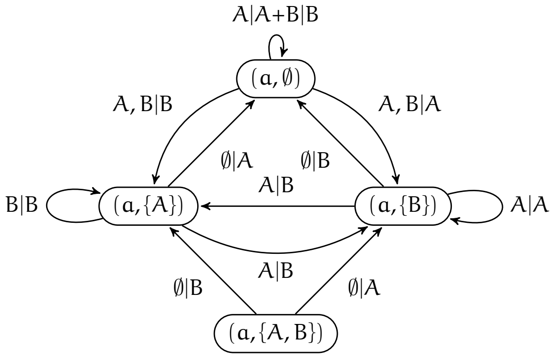
ASYNCHRONOUS CHANNELS We have three asynchronous channels: *AsyncDrain*, *AsyncSpout*, and *FIFO*$_1$. *AsyncDrain* and *AsyncSpout* have identical models. We present the $\mathcal{R}$eo model for the *AsyncDrain* and the *FIFO*$_1$. The intentional automaton model for the *AsyncDrain* is discussed in detail in Example 4.1.10. The *AsyncDrain* has a single internal configuration denoted by a, whereas the *FIFO*$_1$ has two, e and f denoting respectively the internal configuration in which the buffer cell is *empty* and the internal configuration in which the buffer cell is *full*.

- Figure 40 (a) contains $\theta_{AsyncDrain}(\{a\}, \{A, B\})$ and Figure 40 (b) contains the labelled transition diagram for the $\mathcal{R}$eo automaton $\mathcal{A}_{AsyncDrain}(\{a\}, \{A, B\}, \delta)$ denoted by $[\![\theta_{AsyncDrain}(\{a\}, \{A, B\})]\!]_R$.
- Figure 41 (a) contains $\theta_{FIFO_1}(\{e, f\}, \{A, B\})$, and Figure 41 (b) contains the labelled transition diagram for the $\mathcal{R}$eo automaton $\mathcal{A}_{FIFO_1}(\{e, f\}, \{A, B\}, \delta)$ denoted by $[\![\theta_{FIFO_1}(\{e, f\}, \{A, B\})]\!]_R$.



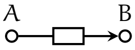
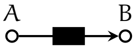
	a
$\emptyset$	$\langle \emptyset, (a, \emptyset) \rangle$
$\{A\}$	$\langle \{A\}, (a, \emptyset) \rangle$
$\{B\}$	$\langle \{B\}, (a, \emptyset) \rangle$
$\{A, B\}$	$\langle \{A\}, (a, \{B\}) \rangle +$ $\langle \{B\}, (a, \{A\}) \rangle$

(a) $\theta_{AsyncDrain}(\{a\}, \{A, B\})$.

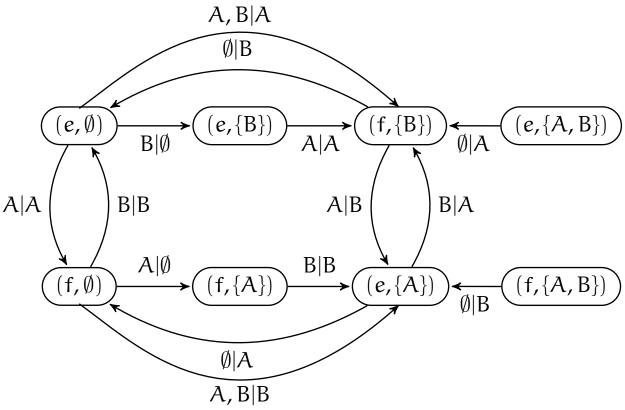


(b) The labeled transition diagram of the $\mathcal{R}$eo automaton $\mathcal{A}_{AsyncDrain}(\{a\}, \{A, B\}, \delta, \{(a, \emptyset)\})$.

Figure 40: $\theta_{AsyncDrain}(\{a\}, \{A, B\})$ and the labeled transition diagram for the $\mathcal{R}$eo automaton $\mathcal{A}_{AsyncDrain}(\{a\}, \{A, B\}, \delta)$.

		
	e	f
$\emptyset$	$\langle\emptyset,(e,\emptyset)\rangle$	$\langle\emptyset,(f,\emptyset)\rangle$
$\{A\}$	$\langle\{A\},(f,\emptyset)\rangle$	$\langle\emptyset,(f,\{A\})\rangle$
$\{B\}$	$\langle\emptyset,(e,\{B\})\rangle$	$\langle\{B\},(e,\emptyset)\rangle$
$\{A,B\}$	$\langle\{A\},(f,\{B\})\rangle$	$\langle\{B\},(e,\{A\})\rangle$

(a) $\theta_{FIFO_1}(\{e,f\},\{A,B\})$.



(b) The labelled transition diagram for the $\mathcal{R}$eo automaton $\mathcal{A}_{FIFO_1}(\{e,f\},\{A,B\},\delta)$.

Figure 41: $\theta_{FIFO_1}(\{e,f\},\{A,B\})$ and the labelled transition diagram for the $\mathcal{R}$eo automaton $\mathcal{A}_{FIFO_1}(\{e,f\},\{A,B\},\delta)$.

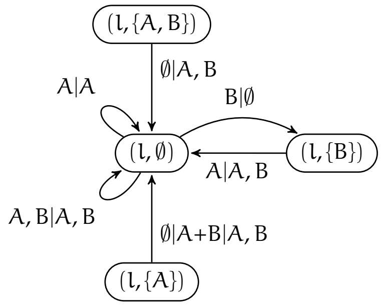
Notation 5.3.1. In case the set depicted in a cell of a configuration table has multiple elements we use the symbol '+' to separate its elements. Take for example the cell at intersection of row $\{A, B\}$ with column a in the configuration table $\theta_{AsyncDrain}$.

LOSSY CHANNELS The *LossySync* is the representative of lossy channels in the primitives we consider in CONLANG. In Section 5.4, we revisit this channel and discuss in more details its context-dependent behaviour.

- Figure 42 (a) contains $\theta_{LossySync}(\{l\}, \{A, B\})$, and Figure 42 (b) contains the labelled transition diagram of the $\mathcal{R}$eo automaton $\mathcal{A}_{LossySync}(\{l\}, \{A, B\}, \delta)$ denoted by $[\![\theta_{LossySync}(\{l\}, \{A, B\})]\!]_R$.

	l
$\emptyset$	$\langle \emptyset, (l, \emptyset)\rangle$
$\{A\}$	$\langle \{A\}, (l, \emptyset)\rangle$
$\{B\}$	$\langle \emptyset, (l, \{B\})\rangle$
$\{A, B\}$	$\langle \{A, B\}, (l, \emptyset)\rangle$

(a) $\theta_{LossySync}(\{l\}, \{A, B\})$.



(b) The labeled transition diagram for the $\mathcal{R}$eo automaton $\mathcal{A}_{LossySync}(\{l\}, \{A, B\}, \delta)$.

Figure 42: $\theta_{LossySync}(\{l\}, \{A, B\})$ and the labeled transition diagram for the $\mathcal{R}$eo automaton $\mathcal{A}_{LossySync}(\{l\}, \{A, B\}, \delta)$.

The transition $(l, \emptyset) \xrightarrow{A|A} (l, \emptyset)$ captures the lossy behaviour. The port A of the *LossySync* receives a request and fires in the same evaluation step but the data received in A is neither stored in a buffer cell (the outgoing state of the transition has the same internal configuration as the ingoing state) nor is transmitted to a sink port (no other port fires). Intuitively the data received in port A is lost.

MERGER AND REPLICATOR Finally we have the *Merger* and the *Replicator* connectors which give semantics to $\mathcal{R}$eo nodes. These are the primitives with three ports. The merger is the only non-deterministic primitive that $\mathcal{R}$eo offers. Using the *Merger* the user can define other non-deterministic channels, like the *AsyncDrain* that we have chosen also to include in the primitives of CONLANG.

- Figure 43 (a) contains $\theta_{Merger}(\{m\},\{A,B,C\})$, and Figure 43 (b) contains the labelled transition diagram for the $\mathcal{R}$eo automaton $\mathcal{A}_{Merger}(\{m\},\{A,B,C\},\delta,\{(m,\emptyset)\})$ denoted by $[\![\theta_{Merger}(\{m\},\{A,B,C\})]\!]_R$.
- Figure 44 (a) contains $\theta_{Replicator}(\{r\},\{A,B,C\})$, and Figure 44 (b) contains the labelled transition diagram for the $\mathcal{R}$eo automaton $\mathcal{A}_{Replicator}(\{r\},\{A,B,C\},\delta,\{(r,\emptyset)\})$ denoted by $[\![\theta_{Replicator}(\{r\},\{A,B,C\})]\!]_R$.

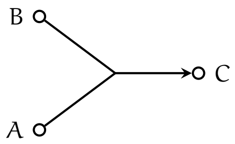
The configuration tables of the primitives resemble the colouring tables we defined, also for the primitives, when we discussed the connector colouring model. We will come back to this point briefly in Section 5.7.

5.4 CONTEXT-DEPENDENT CHANNELS

The context-dependent channels pose a problem in formalising the semantics of $\mathcal{R}$eo. This problem was first identified by Arbab et al. when using constraint automata to model $\mathcal{R}$eo connectors and $\mathcal{R}$eo operations [13]. In this paper the authors provide the model for various channels, including the *LossySync* and the *FIFO*$_1$, and define an operation to calculate the product of two automata to model the $\mathcal{R}$eo join operation. When calculating the product of the two automata models of the *LossySync* and *FIFO*$_1$, the resulting product automaton does not provide the intended semantics. Namely, there is one transition in the product automaton that allows a *write* operation on the *LossySync* to fire and the data to be lost despite the fact that the *FIFO*$_1$ buffer cell is empty. The original explanation [13] is quoted below (context-sensitive is used in place of context-dependent):

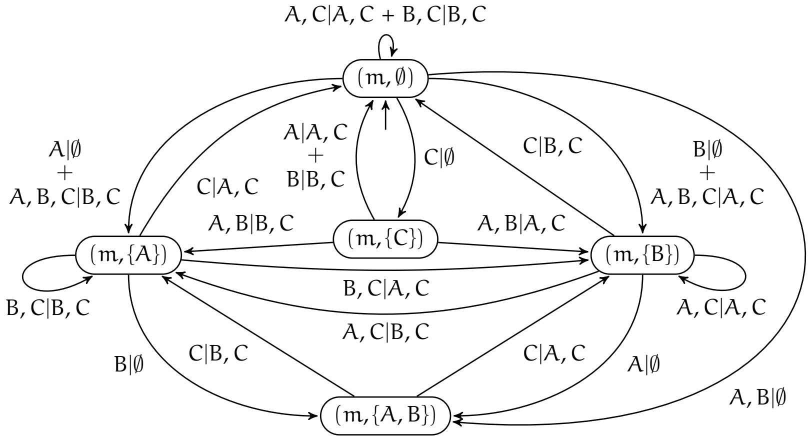
> "The specification of the behavior of the lossy synchronous channel requires it not to lose the data item written to its source end, if this data item can be consumed at its sink end. This type of context-sensitive behavior can be dealt with in constraint automata by introducing the notion of priorities for their transitions."

$\mathcal{R}$eo automata and configuration tables do not use any notion of priority in the transitions. $\mathcal{R}$eo automata encode in each transition the context in which a firing



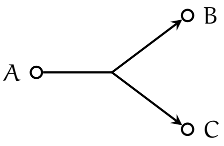
	$\mathfrak{m}$
$\emptyset$	$\langle \emptyset, (\mathfrak{m}, \emptyset) \rangle$
$\{A\}$	$\langle \emptyset, (\mathfrak{m}, \{A\}) \rangle$
$\{B\}$	$\langle \emptyset, (\mathfrak{m}, \{B\}) \rangle$
$\{A, B\}$	$\langle \emptyset, (\mathfrak{m}, \{A, B\}) \rangle$
$\{C\}$	$\langle \emptyset, (\mathfrak{m}, \{C\}) \rangle$
$\{A, C\}$	$\langle \{A, C\}, (\mathfrak{m}, \emptyset) \rangle$
$\{B, C\}$	$\langle \{B, C\}, (\mathfrak{m}, \emptyset) \rangle$
$\{A, B, C\}$	$\langle \{A, C\}, (\mathfrak{m}, \{B\}) \rangle + \langle \{B, C\}, (\mathfrak{m}, \{A\}) \rangle$

(a) $\theta_{Merger}(\{\mathfrak{m}\}, \{A, B, C\})$.



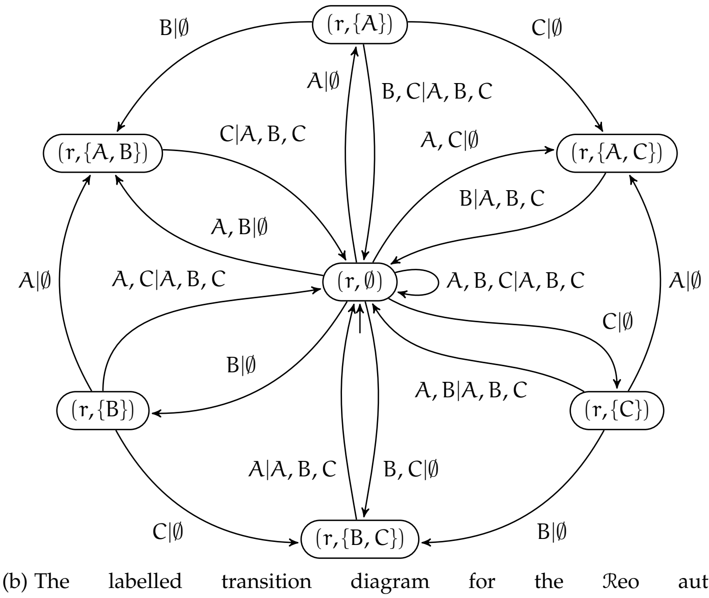
(b) The labelled transition diagram for the $\mathcal{R}$eo automaton $\mathcal{A}_{Merger}(\{\mathfrak{m}\}, \{A, B, C\}, \delta, \{(\mathfrak{m}, \emptyset)\})$.

Figure 43: $\theta_{Merger}(\{\mathfrak{m}\}, \{A, B, C\})$ and the labelled transition diagram for the $\mathcal{R}$eo automaton $\mathcal{A}_{Merger}(\{\mathfrak{m}\}, \{A, B, C\}, \delta, \{(\mathfrak{m}, \emptyset)\})$. (Note that we have omitted the unreachable states $(\mathfrak{m}, \{A, B, C\})$, $(\mathfrak{m}, \{A, C\})$, $(\mathfrak{m}, \{B, C\})$).



	r
$\emptyset$	$\langle\emptyset,(r,\emptyset)\rangle$
$\{A\}$	$\langle\emptyset,(r,\{A\})\rangle$
$\{B\}$	$\langle\emptyset,(r,\{B\})\rangle$
$\{A,B\}$	$\langle\emptyset,(r,\{A,B\})\rangle$
$\{C\}$	$\langle\emptyset,(r,\{C\})\rangle$
$\{A,C\}$	$\langle\emptyset,(r,\{A,C\})\rangle$
$\{B,C\}$	$\langle\emptyset,(r,\{B,C\})\rangle$
$\{A,B,C\}$	$\langle\{A,B,C\},(r,\emptyset)\rangle$

(a) $\theta_{\mathit{Replicator}}(\{r\},\{A,B,C\})$.



(b) The labelled transition diagram for the $\mathcal{R}$eo automaton $\mathcal{A}_{\mathit{Replicator}}(\{r\},\{A,B,C\},\delta,\{(r,\emptyset)\})$.

Figure 44: $\theta_{\mathit{Replicator}}(\{r\},\{A,B,C\})$ and the labelled transition diagram for the $\mathcal{R}$eo automaton $\mathcal{A}_{\mathit{Replicator}}(\{r\},\{A,B,C\},\delta,\{(r,\emptyset)\})$. (Note that we have omitted the unreachable state $(r,\{A,B,C\})$).

occurs. The context being the current configuration (state) of the connector and the request-set that triggers the firing. Having this extra expressiveness in the model we can define and precisely characterise context-dependent connectors.

Definition 5.4.1 (Context-dependent connector)**.** Consider a $\mathcal{R}$eo connector $C = (S, \Sigma, \mathcal{A})$. For each state (s, P) in $\mathcal{A} = (Q, \Sigma, \delta)$ we calculate the set:

$$\Phi_{(s,P)} = \{F \mid \exists R, q \text{ s.t. } (s, P) \xrightarrow{R|F}_{\delta} q, F \neq \emptyset\}.$$

C is a *context-dependent* connector if for some $s \in S$, there is a pair of different states (s, P_1), and (s, P_2) such that $\Phi_{(s,P_1)} \neq \Phi_{(s,P_2)}$.

Consider the *LossySync* $\mathcal{R}$eo automata model depicted in Figure 42 we have $\Phi_{(\iota,\emptyset)} = \{\{A\}, \{A, B\}\}$ and $\Phi_{(\iota,\{B\})} = \{\{A, B\}\}$. Hence, because $\Phi_{(\iota,\emptyset)} \neq \Phi_{(\iota,\{B\})}$, by definition 5.4.1 *LossySync* is a context-dependent connector. None of the other $\mathcal{R}$eo primitives considered in this thesis are context-dependent.

5.5 COMPOSITE CONNECTORS

Next, we take a particular look at composite $\mathcal{R}$eo connectors and define operations on configuration tables to provide the semantics of the $\mathcal{R}$eo operations join and hide. The configuration table operations permit to calculate models for composite $\mathcal{R}$eo connectors compositionally out of the models from primitive $\mathcal{R}$eo connectors we have defined in the previous section. The important reason to define the operations on configuration tables instead of considering the already defined intentional automata operations on $\mathcal{R}$eo automata is because the configuration tables are considerably smaller, therefore the operations on the configuration table models have lower computational cost compared to the operations on the equivalent $\mathcal{R}$eo automata models.

5.5.1 *Operations On Configuration Tables*

The product operation on intentional automata ignores the state structure and treats each state as an abstract state. The resulting product automaton is not of the type $\mathcal{R}$eo automaton. Indeed calculating the product of two $\mathcal{R}$eo automata using the intentional automata operation $\_\times\_$, from here on denoted by $\_\times_I\_$, yields an intentional automaton with states of type $(S \times \mathcal{P}\Sigma) \times (S \times \mathcal{P}\Sigma)$ which differs from the type that states of a $\mathcal{R}$eo automaton have, which is $(S \times \mathcal{P}\Sigma)$. The diagram in Figure 45 contains the type signature of each of the model representations.

To remedy the situation we define a product operation on configuration tables $\_\times_T\_$ that is *behaviourally faithful* to the product operation on intentional automata as we make precise shortly.

$$\begin{tikzcd}
\begin{array}{c}\text{Abstract}\\ \text{configuration table}\\ \theta(S,\Sigma)\end{array} & \begin{array}{c}\mathcal{R}\text{eo}\\ \text{automaton}\\ ((S\times\mathcal{P}\Sigma),\Sigma,\delta)\end{array} & \begin{array}{c}\text{Intentional}\\ \text{automaton}\\ (Q,\Sigma,\delta)\end{array} \\
S \arrow[d] \arrow[r, phantom, "\cong", "\text{Prop. 5.2.10}"'] & (S\times\mathcal{P}\Sigma) \arrow[d] \arrow[r, phantom, "\subsetneq", "\text{Def. 5.2.3}"'] & Q \arrow[d] \\
\mathcal{P}(\mathcal{P}\Sigma\times(S\times\mathcal{P}\Sigma))^{\mathcal{P}\Sigma} & \mathcal{P}(\mathcal{P}\Sigma\times(S\times\mathcal{P}\Sigma))^{\mathcal{P}\Sigma} & \mathcal{P}(\mathcal{P}\Sigma\times Q)^{\mathcal{P}\Sigma}
\end{tikzcd}$$

Figure 45: Every abstract configuration table defines a $\mathcal{R}$eo automaton, and vice versa. A $\mathcal{R}$eo automaton is by definition an intentional automaton, but not every intentional automaton is a $\mathcal{R}$eo automaton.

Definition 5.5.1 (product on abstract configuration tables)**.** Consider two abstract configuration tables $\theta(S_1,\Sigma_1)$ and $\theta(S_2,\Sigma_2)$.

$$\theta(S_1,\Sigma_1)\times_T\theta(S_2,\Sigma_2)=\theta(S_1\times S_2,\Sigma_1\cup\Sigma_2)$$

where each cell of the table is given by:

$$\begin{aligned}
&\theta\langle(s_1,s_2),R\rangle=\\
&\quad\{\,\langle F_1\cup F_2,\langle(s_1',s_2'),R\setminus(F_1\cup F_2)\rangle\rangle \mid \langle F_1,\langle s_1',P_1\rangle\rangle\in\theta\langle s_1,R_1\rangle,\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \langle F_2,\langle s_2',P_2\rangle\rangle\in\theta\langle s_2,R_2\rangle, \quad (a)\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad R=R_1\cup R_2,\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad F_1\cap\Sigma_2=F_2\cap\Sigma_1\,\}\\
&\quad\cup\{\,\langle F_1,\langle(s_1',s_2),R\setminus F_1\rangle\rangle \mid \langle F_1,\langle s_1',P_1\rangle\rangle\in\theta\langle s_1,R\rangle,\\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad F_1\cap\Sigma_2=\emptyset, R\subseteq\Sigma_1\,\} \qquad (b)\\
&\quad\cup\{\,\langle F_2,\langle(s_1,s_2'),R\setminus F_2\rangle\rangle \mid \langle F_2,\langle s_2',P_2\rangle\rangle\in\theta\langle s_2,R\rangle,\\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad F_2\cap\Sigma_1=\emptyset, R\subseteq\Sigma_2\,\} \qquad (c)
\end{aligned}$$

Proposition 5.5.2. *Consider two abstract configuration tables* $\theta(S_1,\Sigma_1)$ *and* $\theta(S_2,\Sigma_2)$. $\theta(S_1,\Sigma_1)\times_T\theta(S_2,\Sigma_2)$ *is an abstract configuration table.*

Proof. We have to show that for all $\langle F,\langle(s_1,s_2),P\rangle\rangle\in\theta\langle(s_1,s_2),R\rangle$ we have that $R=F\cup P$ and $F\cap P=\emptyset$. We have three particular cases to prove.

First case, we have to prove that for all $\langle F_1\cup F_2,\langle(s_1',s_2'),R\setminus(F_1\cup F_2)\rangle\rangle\in$ (a) we have that $R=(F_1\cup F_2)\cup(R\setminus(F_1\cup F_2))$ and $(F_1\cup F_2)\cap(R\setminus(F_1\cup F_2))=\emptyset$. And both hold trivially.

The remaining two cases also follow trivially from the definition. □

$$\begin{tikzcd}
\begin{matrix}(S_1 \times \mathcal{P}\Sigma_1, \Sigma_1, \delta_1)\\ \times \\ (S_2 \times \mathcal{P}\Sigma_2, \Sigma_2, \delta_2)\end{matrix} \arrow[r, "\_ \times_I \_"] & ((S_1 \times \mathcal{P}\Sigma_1) \times (S_2 \times \mathcal{P}\Sigma_2), \Sigma_1 \cup \Sigma_2, \delta') \arrow[d, dash, "\sim"] \\
 & ((S_1 \times S_2) \times \mathcal{P}(\Sigma_1 \cup \Sigma_2), \Sigma_1 \cup \Sigma_2, \delta) \\
\theta(S_1, \Sigma_1) \times \theta(S_2, \Sigma_2) \arrow[uu, "\llbracket \_ \rrbracket_R \times \llbracket \_ \rrbracket_R"] \arrow[r, "\_ \times_T \_"] & \theta(S_1 \times S_2, \Sigma_1 \cup \Sigma_2) \arrow[u, "\llbracket \_ \rrbracket_R"']
\end{tikzcd}$$

Figure 46: The commuting diagram expressing the faithful relationship between the operation $\_ \times_T \_$ and the operation $\_ \times_I \_$.

Theorem 5.5.3. *Consider two abstract configuration tables* $\theta(S_1, \Sigma_1)$, $\theta(S_2, \Sigma_2)$.

$$\llbracket \theta(S_1, \Sigma_1) \rrbracket_R \times_I \llbracket \theta(S_2, \Sigma_2) \rrbracket_R \sim \llbracket \theta(S_1, \Sigma_1) \times_T \theta(S_2, \Sigma_2) \rrbracket_R.$$

proof sketch. We have to show that the diagram in Figure 46 commutes. For that we must prove that:

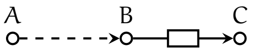
$$(\underbrace{(S_1 \times \mathcal{P}\Sigma_1)}_{Q_1} \times \underbrace{(S_2 \times \mathcal{P}\Sigma_2)}_{Q_2}, \underbrace{\Sigma_1 \cup \Sigma_2}_{\Sigma}, \delta') \sim (\underbrace{(S_1 \times S_2) \times \mathcal{P}(\Sigma_1 \cup \Sigma_2)}_{Q}, \underbrace{\Sigma_1 \cup \Sigma_2}_{\Sigma}, \delta)$$

Which means that we can define a relation $\mathcal{Z} \subseteq (Q_1 \times Q_2) \times Q$ which is a bisimulation. Consider

$$\begin{aligned}\mathcal{Z} = &\big\{ \big(\langle \langle s_1, P_1 \rangle, \langle s_2, P_2 \rangle \rangle, \langle (s_1, s_2), P_1 \cup P_2 \rangle \big) \mid P_1 \cap \Sigma_2 = P_2 \cap \Sigma_1 \big\} \cup \\ &\big\{ \big(\langle \langle s_1, P_1 \rangle, \langle s_2, P_2 \rangle \rangle, \langle (s_1, s_2), (P_1 \cup P_2) \setminus (P_1 \cap P_2) \rangle \big) \mid P_1 \cap \Sigma_2 \neq P_2 \cap \Sigma_1 \big\}.\end{aligned}$$

The proof that $\mathcal{Z}$ is a bisimulation can be done by performing a case analysis, according to the definitions of the product operations on intentional automata and on abstract configuration tables, for the two different types of pairs present in $\mathcal{Z}$. □

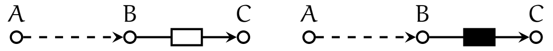
Example 5.5.4. To illustrate the product operation on configuration tables we calculate the configuration table for the composite $\mathcal{R}$eo connector:



We calculate the product of the configuration tables:

$$\theta_{LossySync}(\{l\}, \{A, B\}) \times_T \theta_{FIFO_1}(\{e, f\}, \{B, C\}) = \theta(\{(l, e), (l, f)\}, \{A, B, C\})$$

Figure 47 contains the table $\theta(\{(l, e), (l, f)\}, \{A, B, C\})$. We write le as a shorthand for (l, e), and lf as a shorthand for (l, f).



	le	lf
$\emptyset$	$\langle\emptyset,(le,\emptyset)\rangle$	$\langle\emptyset,(lf,\emptyset)\rangle$
$\{A\}$	$\langle\{A\},(le,\emptyset)\rangle$	$\langle\{A\},(lf,\emptyset)\rangle$
$\{B\}$	$\langle\emptyset,(le,\{B\})\rangle$	$\langle\emptyset,(lf,\{B\})\rangle$
$\{A,B\}$	$\langle\{A,B\},(lf,\emptyset)\rangle$	$\langle\{A\},(lf,\{B\})\rangle$
$\{C\}$	$\langle\emptyset,(le,\{C\})\rangle$	$\langle\{C\},(le,\emptyset)\rangle$
$\{A,C\}$	$\langle\{A\},(le,\{C\})\rangle$	$\langle\{A,C\},(le,\emptyset)\rangle$
$\{B,C\}$	$\langle\emptyset,(le,\{B,C\})\rangle$	$\langle\{C\},(le,\{B\})\rangle$
$\{A,B,C\}$	$\langle\{A,B\},(lf,\{C\})\rangle$	$\langle\{A,C\},(le,\{B\})\rangle$

Figure 47: $\theta_{LossySync}(\{l\},\{A,B\}) \times_T \theta_{FIFO_1}(\{e,f\},\{B,C\})$.

Definition 5.5.5 (hiding on abstract configuration tables)**.** Consider an abstract configuration table $\theta(S,\Sigma)$ and a port $h \in \Sigma$.

$\exists_T[h]\theta(S,\Sigma) = \theta[h](S,\Sigma \setminus \{h\})$ where:

$$\theta[h]\langle s,R\rangle = \begin{cases} \{\langle F\backslash\{h\},q\rangle \mid \langle F,q\rangle \in \theta\langle s,R\cup\{h\}\rangle, h \in F\} & \text{if this set is non-empty} \\ \theta\langle s,R\rangle & \text{otherwise} \end{cases}$$

Proposition 5.5.6. *Consider an abstract configuration table* $\theta(S,\Sigma)$*, and a port* $h \in \Sigma$*,* $\exists_T[h]\theta(S,\Sigma)$ *is an abstract configuration table.*

Proof. We take $\langle F',\langle s,P'\rangle\rangle \in \theta[h]\langle s,R\rangle$ and we have to show that $R = F' \cup P'$, and $F' \cap P' = \emptyset$. (Notice that $R \subseteq \Sigma\backslash\{h\}$)

By definition, if $\langle F',\langle s,P'\rangle\rangle \in \theta[h]\langle s,R\rangle$ we have two possible cases:

CASE 1: $\langle F,\langle s,P\rangle\rangle \in \theta\langle s,R\cup\{h\}\rangle$, AND $h \in F$. In this case $F' = F\backslash\{h\}$ and $P' = P$. By definition of configuration table we have that $R\cup\{h\} = F \cup P$, and $F \cap P = \emptyset$. Then $h \in F$ implies $h \notin P$ because $F \cap P = \emptyset$. Therefore $(R\cup\{h\})\backslash\{h\} = F\backslash\{h\} \cup P\backslash\{h\} \Leftrightarrow R = F' \cup P \Leftrightarrow R = F' \cup P'$. Furthermore $F \cap P = \emptyset \Leftrightarrow F\backslash\{h\} \cap P\backslash\{h\} = \emptyset \Leftrightarrow F' \cap P = \emptyset \Leftrightarrow F' \cap P' = \emptyset$. Hence $R = F' \cup P'$, and $F' \cap P' = \emptyset$ as required.

CASE 2: $\langle F,\langle s,P\rangle\rangle \in \theta\langle s,R\rangle$. In this case $F' = F$ and $P' = P$. By definition of configuration table we have that $R = F \cup P$, and $F \cap P = \emptyset$. Hence $R = F' \cup P'$, and $F' \cap P' = \emptyset$ as required.

For all $\langle F,\langle s,P\rangle\rangle \in \theta[h]\langle s,R\rangle$ we have that $R = F' \cup P'$, and $F' \cap P' = \emptyset$ as required. □

$$\begin{tikzcd}
\Sigma \times (S \times \mathcal{P}\Sigma, \Sigma, \delta) \arrow[r, "\exists_I[h]\_"] & (S \times \mathcal{P}\Sigma, \Sigma \setminus \{h\}, \delta') \arrow[d, dash, "\sim"] \\
 & (S \times \mathcal{P}(\Sigma \setminus \{h\}), \Sigma \setminus \{h\}, \delta[h]) \\
\Sigma \times \theta(S, \Sigma) \arrow[uu, "\mathtt{id} \times \llbracket \_ \rrbracket_R"] \arrow[r, "\exists_T[h]\_"] & \theta[h](S, \Sigma \setminus \{h\}) \arrow[u, "\llbracket \_ \rrbracket_R"']
\end{tikzcd}$$

Figure 48: The commuting diagram characterising the relationship between the operation $\exists_T[\_]\_$ and the operation $\exists_I[\_]\_$.

From the definition of $\exists_T[\_]\_$ it follows directly that the operation is commutative.

Proposition 5.5.7 (commutativity). *Consider an abstract configuration table* $\theta\langle S, \Sigma\rangle$ *and* $h_1, h_2 \in \Sigma$. *We have that* $\exists_T[h_2](\exists_T[h_1](S, \Sigma)) = \exists_T[h_1](\exists_T[h_2](S, \Sigma))$.

Notation 5.5.8. Given $h_1, \ldots, h_n \in \Sigma$, we denote by $\exists_T\{h_1, \ldots, h_n\}(S, \Sigma)$ the abstract colouring table obtained by hiding the ports in $\{h_1, \ldots, h_n\}$ one at a time subsequently.

The operation $\exists_T[\_]\_$ is *behaviourally faithful* to the operation $\exists[\_]\_$ on intentional automata (from now on denoted $\exists_I[\_]\_$). The commuting diagram in Figure 48 makes precise what we mean by *behaviourally faithful*.

Theorem 5.5.9. *Consider the abstract configuration table* $\theta(S, \Sigma)$ *and* $h \in \Sigma$.

$$\exists_I[h]\llbracket \theta(S, \Sigma) \rrbracket_R \sim \llbracket \exists_T[h]\theta(S, \Sigma) \rrbracket_R.$$

Proof. We must show that the diagram in Figure 48 indeed commutes. For that we must prove that:

$$(\underbrace{S \times \mathcal{P}\Sigma}_{Q}, \Sigma \setminus \{h\}, \delta') \sim (\underbrace{S \times \mathcal{P}(\Sigma \setminus \{h\})}_{Q'}, \Sigma \setminus \{h\}, \delta[h])$$

Which means that we can define a relation $\mathcal{Z} \subseteq Q \times Q'$ which is a bisimulation. We consider

$$\begin{aligned} \mathcal{Z} = & \left\{ \langle \langle s, P \rangle, \langle s, P \rangle \rangle \mid P \subseteq \Sigma \setminus \{h\} \right\} \cup \left\{ \langle \langle s, P \cup \{h\} \rangle, \langle s, P \rangle \rangle \mid P \subseteq \Sigma \setminus \{h\} \right\} \\ \subseteq & \ Q \times Q' \end{aligned}$$

and prove that $\mathcal{Z}$ is a bisimulation.

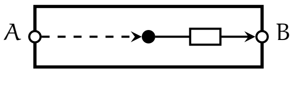
CASE ($\langle s, P\rangle \mathcal{Z} \langle s, P\rangle$ AND $P \subseteq \Sigma \setminus \{h\}$). If $\langle s, P\rangle \xrightarrow{R|F}_{\delta'} \langle s', P'\rangle$ by Definition 4.3.3 of $\exists_I[\_]\_$ it follows that either: (i) $\langle F', \langle s', P'\rangle\rangle \in \delta_{\langle s,P\rangle}(R)$ and $F = F' \setminus \{h\}$; or (ii) $\langle F', \langle s', P'\rangle\rangle \in \delta_{\langle s,P\rangle}(R \cup \{h\})$, $F = F' \setminus \{h\}$ and $h \in F'$. In case (i): by property ❶ we know that $R \cup P = F' \cup P'$ and because $R \subseteq \Sigma \setminus \{h\}, P \subseteq \Sigma \setminus \{h\}$ it follows that $F' \cup P' \subseteq \Sigma \setminus \{h\}$, in particular $F' \subseteq \Sigma \setminus \{h\}$. By property ❷ it follows $\langle F', \langle s', P'\rangle\rangle \in \delta_{\langle s,\emptyset\rangle}(R \cup P)$. By Proposition 5.2.10 $\langle F', \langle s', P'\rangle\rangle \in \theta\langle s, R \cup P\rangle$. By Definition 5.5.5 of $\exists_T[\_]\_$ and because $F' \subseteq \Sigma \setminus \{h\}$ follows that $\langle F', \langle s', P'\rangle\rangle \in \theta[h]\langle s, R \cup P\rangle$. By $[\![\_]\!]_R$, property ❷, and because $R \cap P = \emptyset$ we have $\langle F', \langle s', P'\rangle\rangle \in \delta[h]_{\langle s,P\rangle}(R)$. Hence $\langle s, P\rangle \xrightarrow{R|F}_{\delta[h]} \langle s', P'\rangle$ as required, and by definition of $\mathcal{Z}$ it follows that $\langle s', P'\rangle \mathcal{Z} \langle s', P'\rangle$ as required. In case (ii): by property ❷ and Proposition 5.2.10 follows that $\langle F', \langle s', P'\rangle\rangle \in \theta\langle s, P \cup R \cup \{h\}\rangle$. By definition of $\exists_T[\_]\_$ and because $h \in F'$ it follows that $\langle F, \langle s', P'\rangle\rangle \in \theta[h]\langle s, P \cup R\rangle$. By $[\![\_]\!]_R$, property ❷, and because $P \cap R = \emptyset$ we have $\langle F, \langle s', P'\rangle\rangle \in \delta[h]_{\langle s,P\rangle}(R)$. Hence $\langle s, P\rangle \xrightarrow{R|F}_{\delta[h]} \langle s', P'\rangle$ as required, and by definition of $\mathcal{Z}$ it follows that $\langle s', P'\rangle \mathcal{Z} \langle s', P'\rangle$ as required.

If $\langle s, P\rangle \xrightarrow{R|F}_{\delta[h]} \langle s', P'\rangle$ the proof follows the same arguments used above.

CASE ($\langle s, P \cup \{h\}\rangle \mathcal{Z} \langle s, P\rangle$). If $\langle s, P \cup \{h\}\rangle \xrightarrow{R|F}_{\delta'} \langle s', P'\rangle$ by Definition 4.3.3 of $\exists_I[\_]\_$ it follows that either: (i) $\langle F', \langle s', P'\rangle\rangle \in \delta_{\langle s,P\cup\{h\}\rangle}(R)$ and $F = F' \setminus \{h\}$; or (ii) $\langle F', \langle s', P'\rangle\rangle \in \delta_{\langle s,P\cup\{h\}\rangle}(R \cup \{h\})$, $h \in F'$ and $F = F' \setminus \{h\}$. Case (ii) is absurd since $(R \cup \{h\}) \cap (P \cup \{h\}) \neq \emptyset$ violates property ❶ of $\mathcal{R}$eo automaton. Therefore such a transition cannot exist. In case (i): by ❶ we know that $R \cup (P \cup \{h\}) = F' \cup P'$ and since $P' \subseteq \Sigma \setminus \{h\}$ it follows that $h \in F'$. By ❷ we have that $\langle F', \langle s', P'\rangle\rangle \in \delta_{\langle s,\emptyset\rangle}(R \cup P \cup \{h\})$. By Proposition 5.2.10 it follows $\langle F', \langle s', P'\rangle\rangle \in \theta\langle s, R \cup P \cup \{h\}\rangle$. By definition of $\exists_T[\_]\_$ and because $h \in F'$, we have $\langle F' \setminus \{h\}, \langle s', P'\rangle\rangle \in \theta[h]\langle s, R \cup P\rangle$. By $[\![\_]\!]_R$ and because $R \cap P = \emptyset$, $F = F' \setminus \{h\}$ follows $\langle F, \langle s', P'\rangle\rangle \in \delta[h]_{\langle s,P\rangle}(R)$. Hence $\langle s, P\rangle \xrightarrow{R|F}_{\delta[h]} \langle s', P'\rangle$ as required, and by definition of $\mathcal{Z}$ it follows that $\langle s', P'\rangle \mathcal{Z} \langle s', P'\rangle$ as required.

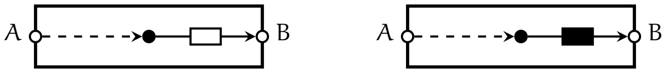
If $\langle s, P \cup \{h\}\rangle \xrightarrow{R|F}_{\delta[h]} \langle s', P'\rangle$ the proof follows the same arguments used above. □

Example 5.5.10. To illustrate the hiding operation on abstract configuration tables we calculate the abstract configuration table for:



We take the abstract configuration table $\theta_{LossySync}(\{l\}, \{A, B\}) \times_T \theta_{FIFO_1}(\{e, f\}, \{B, C\})$ calculated in the Example 5.5.4 and we calculate the hiding of port B:

$$\exists_T[B](\theta_{LossySync}(\{l\}, \{A, B\}) \times_T \theta_{FIFO_1}(\{e, f\}, \{B, C\})).$$



	le	lf
$\emptyset$	$\langle\emptyset,(le,\emptyset)\rangle$	$\langle\emptyset,(lf,\emptyset)\rangle$
$\{A\}$	$\langle\{A\},(lf,\emptyset)\rangle$	$\langle\{A\},(lf,\emptyset)\rangle$
$\{C\}$	$\langle\emptyset,(le,\{C\})\rangle$	$\langle\{C\},(le,\emptyset)\rangle$
$\{A,C\}$	$\langle\{A\},(lf,\{C\})\rangle$	$\langle\{A,C\},(le,\emptyset)\rangle$

Figure 49: $\exists_T[B](\theta_{LossySync}(\{l\},\{A,B\}) \times_T \theta_{FIFO_1}(\{e,f\},\{B,C\}))$

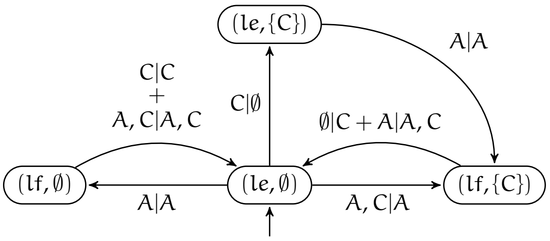


Figure 50: The transition diagram of $[\![\exists_T[B](\theta_{LossySync}(\{l\},\{A,B\}) \times_T \theta_{FIFO_1}(\{e,f\},\{B,C\}))]\!]_R$. We consider $(le,\emptyset)$ the single intial state and we do not depict the unreachable states.

The resulting table is depicted in Figure 49. The transition diagram of the $\mathcal{R}$eo automaton $[\![\exists_T[B](\theta_{LossySync}(\{l\},\{A,B\}) \times_T \theta_{FIFO_1}(\{e,f\},\{B,C\}))]\!]_R$ is depicted in Figure 50.

To illustrate how the operation hiding handles connectors with memory cells we continue with another example.

Example 5.5.11. In this example we construct a connector $FIFO_2$ out of two $FIFO_1$ using the product and hiding operations on configuration tables. We compose $\theta_{FIFO_1}(\{e,f\},\{A,C\})$ and $\theta_{FIFO_1}(\{e,f\},\{C,B\})$ using the product operation. After, we calculate the hiding of port C:

$$\exists_T[C](\theta_{FIFO_1}(\{e,f\},\{A,C\}) \times_T \theta_{FIFO_1}(\{e,f\},\{C,B\})).$$

Figure 51 shows $\theta_{FIFO_1}(\{e,f\},\{A,C\})) \times_T \theta_{FIFO_1}(\{e,f\},\{C,B\}))$, and Figure 52 shows $\exists_T[C](\theta_{FIFO_1}(\{e,f\},\{A,C\})) \times_T \theta_{FIFO_1}(\{e,f\},\{C,B\})))$. The transition marked with the

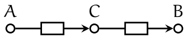
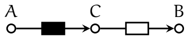
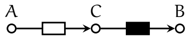
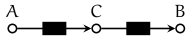
				
	ee	fe	ef	ff
$\emptyset$	$\langle\emptyset,(ee,\emptyset)\rangle$	$\langle\emptyset,(fe,\emptyset)\rangle$	$\langle\emptyset,(ef,\emptyset)\rangle$	$\langle\emptyset,(ff,\emptyset)\rangle$
$\{A\}$	$\langle\{A\},(fe,\emptyset)\rangle$	$\langle\emptyset,(fe,\{A\})\rangle$	$\langle\{A\},(ff,\emptyset)\rangle$	$\langle\emptyset,(ff,\{A\})\rangle$
$\{C\}$	$\langle\emptyset,(ee,\{C\})\rangle$	☞ $\langle\{C\},(ef,\emptyset)\rangle$	$\langle\emptyset,(ef,\{C\})\rangle$	$\langle\emptyset,(ff,\{C\})\rangle$
$\{A,C\}$	$\langle\{A\},(fe,\{C\})\rangle$	$\langle\{C\},(ef,\{A\})\rangle$	$\langle\{A\},(ff,\{C\})\rangle$	$\langle\emptyset,(ff,\{A,C\})\rangle$
$\{B\}$	$\langle\emptyset,(ee,\{B\})\rangle$	$\langle\emptyset,(fe,\{B\})\rangle$	$\langle\{B\},(ee,\emptyset)\rangle$	$\langle\{B\},(fe,\emptyset)\rangle$
$\{A,B\}$	$\langle\{A\},(fe,\{B\})\rangle$	$\langle\emptyset,(fe,\{A,B\})\rangle$	$\langle\{A,B\},(fe,\emptyset)\rangle$	$\langle\{B\},(fe,\{A\})\rangle$
$\{C,B\}$	$\langle\emptyset,(ee,\{C,B\})\rangle$	$\langle\{C\},(ef,\{B\})\rangle$	$\langle\{B\},(ee,\{C\})\rangle$	$\langle\{B\},(fe,\{C\})\rangle$
$\{A,C,B\}$	$\langle\{A\},(fe,\{C,B\})\rangle$	$\langle\{C\},(ef,\{A,B\})\rangle$	$\langle\{A,B\},(fe,\{C\})\rangle$	$\langle\{B\},(fe,\{A,C\})\rangle$

Figure 51: $\theta_{FIFO_1(A,C)} \times_T \theta_{FIFO_1(C,B)}$

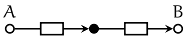
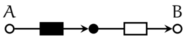
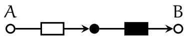
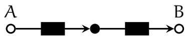
				
	ee	fe	ef	ff
$\emptyset$	$\langle\emptyset,(ee,\emptyset)\rangle$	☞ $\langle\emptyset,(ef,\emptyset)\rangle$	$\langle\emptyset,(ef,\emptyset)\rangle$	$\langle\emptyset,(ff,\emptyset)\rangle$
$\{A\}$	$\langle\{A\},(fe,\emptyset)\rangle$	$\langle\emptyset,(ef,\{A\})\rangle$	$\langle\{A\},(ff,\emptyset)\rangle$	$\langle\emptyset,(ff,\{A\})\rangle$
$\{B\}$	$\langle\emptyset,(ee,\{B\})\rangle$	$\langle\emptyset,(ef,\{B\})\rangle$	$\langle\{B\},(ee,\emptyset)\rangle$	$\langle\{B\},(fe,\emptyset)\rangle$
$\{A,B\}$	$\langle\{A\},(fe,\{B\})\rangle$	$\langle\emptyset,(ef,\{A,B\})\rangle$	$\langle\{A,B\},(fe,\emptyset)\rangle$	$\langle\{B\},(fe,\{A\})\rangle$

Figure 52: $\exists_T[C](\theta_{FIFO_1(A,C)} \times_T \theta_{FIFO_1(C,B)})$

symbol ☞ in Figure 51: $(fe,\emptyset) \xrightarrow{C|C} (ef,\emptyset)$, corresponds to a transition that changes the connector from the internal configuration fe to ef. Note that C is not internalised by the operation product therefore the connector requires a request on C to fire C. After hiding C we obtain the configuration table depicted in Figure 52 and the transition that is responsible for taking the connector from configuration fe to ef is now an internal transition $(fe,\emptyset) \xrightarrow{\emptyset|\emptyset} (ef,\emptyset)$ (marked with ☞). Meaning that the connector now controls when the connector transits from the configuration fe to the configuration ef.

We defer for future work the definition of the operation to eliminate internal transitions on abstract configuration tables. However, just as an example, consider $\exists_T[C](\theta_{FIFO_1}(\{e,f\},\{A,C\}) \times_T \theta_{FIFO_1}(\{e,f\},\{C,B\}))$ derived above (Example 5.5.11). Applying proposition 5.2.10 we calculate its equivalent $\mathcal{R}$eo automaton, denoted by $[\![\exists_T[C](\theta_{FIFO_1}(\{e,f\},\{A,C\}) \times_T \theta_{FIFO_1}(\{e,f\},\{C,B\}))]\!]_R$. We have now a $\mathcal{R}$eo automaton, therefore we can apply the operation on intentional automata to eliminate internal transitions and calculate $\big([\![\exists_T[C](\theta_{FIFO_1}(\{e,f\},\{A,C\}) \times_T \theta_{FIFO_1}(\{e,f\},\{C,B\}))]\!]_R\big)_{/\Rightarrow}$. The labelled transition diagram of the intentional automaton obtained

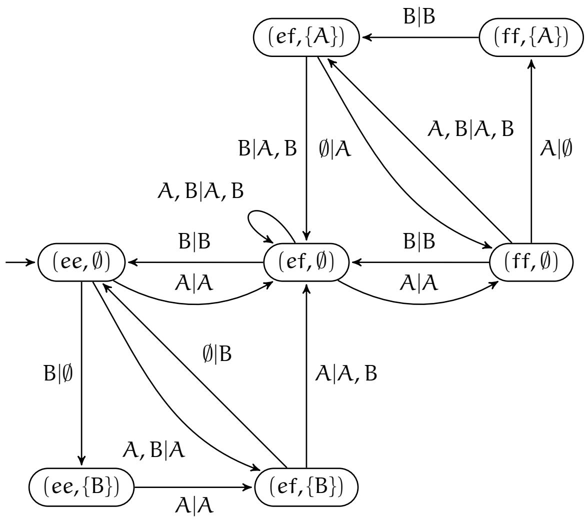


Figure 53: The transition diagram for $(\llbracket\exists_T[C](\theta_{FIFO_1}(\{e,f\},\{A,C\}) \times_T \theta_{FIFO_1}(\{e,f\},\{C,B\}))\rrbracket_R)_{/\Rightarrow}$. The state $(ee,\emptyset)$ is considered the initial state and the states unreachable from it are not depicted.

is shown in Figure 53. Important to note that, in fact, the resulting automaton is a $\mathcal{R}$eo automaton, which suggests that we can define an operation to eliminate internal transitions on abstract configuration tables that is faithful to the operation on intentional automata, i. e., such that the diagram on Figure 54 commutes.

5.6 OTHER INTENTIONAL AUTOMATA MODELS FOR CONNECTORS

In this section we briefly consider possible intentional automata classes other than $\mathcal{R}$eo automata that we think are worth mentioning in the context of software connectors.

CONNECTORS WITHOUT PROPERTY ❷ In the context of $\mathcal{R}$eo, we can consider a larger class of definable $\mathcal{R}$eo connectors if we drop property ❷. We are unaware of any such connectors in the present literature, therefore, we introduce one. We call it the *alternating drain* channel, and we denote it by *AlterDrain*. It is a memoryless channel, therefore, has two ports, and a singleton internal configuration set $S = \{t\}$. This channel behaves like a *SyncDrain* as long as *write* operations are performed simultaneously on both of its ports. It stops behaving like a *SyncDrain* the moment that a *write* operation is performed on one of its ports without a *write* operation on the other port. The channel keeps the unmatched port with the *write* operation

$$\begin{tikzcd}
(S,\Sigma,\delta) \arrow[r, "-_{/\Rightarrow}"] & (S,\Sigma,\delta_{/\Rightarrow}) \arrow[d, dash, "\sim"] \\
 & (S,\Sigma,\delta'_{/\Rightarrow}) \\
\theta(S,\Sigma) \arrow[uu, "{\llbracket \_ \rrbracket_R}"] \arrow[r, "?"] & \theta_{/\Rightarrow}(S,\Sigma) \arrow[u, "{\llbracket \_ \rrbracket_R}"']
\end{tikzcd}$$

Figure 54: A diagram describing the open question with respect to the relationship between an operation to eliminate internal transitions on abstract configuration tables and the operation $-_{/\Rightarrow}$ on intentional automata.

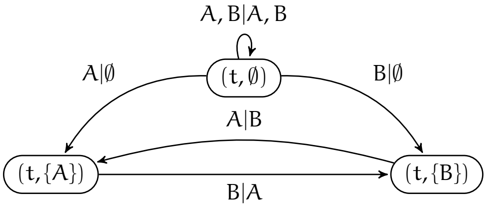


Figure 55: The intentional automaton model of the *AlterDrain*.

pending until a *write* is performed on the other port. When the other port receives a *write* operation the pending *write* succeeds and the channel drains the data from that port. Leaving the other port that just received the write operation pending. Thus the first single write ends the phase in which this channel behaves as a SyncDrain, after which it behaves as an alternator with a delay.

The labelled transition diagram of the *AlterDrain* is depicted in Figure 55. We have, for instance

$$\delta_{(t,\{A\})}(\{B\}) = \{A\} \neq \{A, B\} = \delta_{(t,\emptyset)}(\{A, B\}).$$

Therefore $\mathcal{A}_{AlterDrain}$ does not comply with ❷. Property ❷ is not explicitly stated in the literature. It is more evident when we consider a formalism that also models requests to describe the formal semantics of $\mathcal{R}$eo connectors.

In a formalism that abstracts from operation requests, a Reo connector like the *AlterDrain* can be represented by a model where the distinction between pending requests and just performed requests is disguised under some form of non-determinism. Such a model abstracts away whether a port is pending or has just received a request. The model of such a connector would turn out to be inconveniently uninformative, explaining, perhaps, why we do not find $\mathcal{R}$eo connectors like the *AlterDrain* in the present literature.

We have not investigated the potential benefits and advantages of connectors such as the *AlterDrain*. They fit the original description of a $\mathcal{R}$eo connector though. Intentional automata allow to give the intended semantics to this class of connectors.

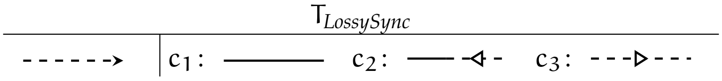
CONNECTORS WITH PENDING REQUESTS TIME-OUT Property ➊ requires that for a transition $(s, P) \xrightarrow{R|F} (s', P')$ we have that $P \cup R = F \cup P'$. With this property the possibility that a pending *request* on a port can time-out is not allowed: *every pending request on a port eventually has a corresponding firing on that port*. A request time-out (*cf.* Definition 5.2.2) would correspond to a transition in which a pending request on a port $p \in P$ would not fire ($p \notin F$) and would not remain pending ($p \notin P'$). A class of automata models where time-outs on pending requests are allowed would have to relax property ➊, and consider transitions with the property: $F \cup P' \subseteq P \cup R$.

PRO-ACTIVE CONNECTORS If property ➊ or the timeout variation discussed in the previous paragraph is dropped we can define models in which we have a transition $(s, P) \xrightarrow{R|F} (s', P')$ that fires a port $p \in F$ without requiring port port p to be pending or requested $p \notin P$ and $p \notin R$. The connector fires a port without having registered a request on the port beforehand. This is pro-active behaviour from the connector towards the environment. This type of behaviour becomes particularly relevant when certain ports of the connector are connected to passive components or services. In which case the firing of a port could model a method call or a service invocation.

CONTEXT-INDEPENDENT CONNECTORS Context-independent connectors constitute an important class of connectors. In fact constraint automata provide models for this class of connectors. From the Definition 5.4.1 we can easily define the intentional automata models for context-independent connectors. The class of intentional automata models corresponding to context-independent connectors includes only the $\mathcal{R}$eo automata with the property that for all $s \in S$ and for all distinct states (s, P_1), (s, P_2) we have that $\Phi_{(s,P_1)} = \Phi_{(s,P_2)}$. For this class of models the behaviour of each state of the automata does not depend on the pending ports, it depends only on s. Exactly as it happens with the constraint automata models that abstract away from pending ports.

5.7 DISCUSSION

We have presented $\mathcal{R}$eo automata, a particular class of intentional automata models for context dependent $\mathcal{R}$eo connectors. Our main motivation was to use properties



(a) 3-colouring table for the *LossySync*.

	l
$\emptyset$	$\langle\emptyset,(l,\emptyset)\rangle$
$\{A\}$	$\langle\{A\},(l,\emptyset)\rangle$
$\{B\}$	$\langle\emptyset,(l,\{B\})\rangle$
$\{A,B\}$	$\langle\{A,B\},(l,\emptyset)\rangle$

(b) $\theta_{LossySync}(\{l\},\{A,B\})$.

Figure 56: 3-colouring table and configuration table for *LossySync*.

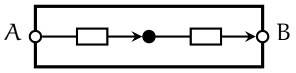


Figure 57: A *FIFO*$_2$ connector built using two *FIFO*$_1$.

specific to $\mathcal{R}$eo connectors to restrict the class of intentional automata models to those that can be interpreted as $\mathcal{R}$eo connectors. We have shown that the $\mathcal{R}$eo automata admit a concise representation as configuration tables. Configuration tables are indeed isomorphic to $\mathcal{R}$eo automata. $\mathcal{R}$eo automata can be large compared to constraint automata models. The configuration tables representation however considerably improves in this respect, by reducing the model to a size comparable to a 3-colouring table model. As an example, Figure 56 contains the 3-colouring table model and the configuration table model for *LossySync*. As we can see the colouring table has three entries, whereas the configuration table has four entries. However, c_3 encodes a fourth entry, that is derived using the flip-rule, c_4 : -▷--◁-· . An important line of future work is to make the comparison between the 3-colouring tables and the configuration tables more clear and precise.

The $\mathcal{R}$eo class of guarded automata models defined by Bonsange *et al.* [25] are elegant and succinct, and close to the $\mathcal{R}$eo automata class. The main difference between the $\mathcal{R}$eo class of guarded automata models and the $\mathcal{R}$eo class of intentional automata models is that the former does not permit to express the intended semantics of the *FIFO*$_2$ connector depicted in Figure 57. In more general terms, the difference seems to be that the guarded automata semantics defines no notion of internal behaviour. The behaviour of an automaton is described in terms of the accepted guarded strings, i. e., the language it accepts, and it includes, for example, behaviour that in our $\mathcal{R}$eo automata model, is identified as internal behaviour. We leave a formal comparison of these two classes of automata as future work.

6

CONNECTOR SIMULATION AND ANIMATION

In this chapter we present a framework, called Connector Animation, to simulate the dataflow behaviour of $\mathcal{R}$eo connectors by means of visual animations. The visual animations are based on the connector colouring semantics presented in Chapter 3. The dataflow behaviour prescribed by a colouring is refined into an *animation specification*. An animation specification consists of a collection of basic *dataflow actions* that respect the *data dependencies* dictated by the colouring. The animation specifications for a connector can be obtained compositionally out of the animation specifications of its primitive constituents. The dataflow actions and the data dependencies that govern their composition are defined by an *animation specification language*. The animation specifications can be compiled into generic executable *animation descriptions*. An animation description consists of an abstract animation code tailored for $\mathcal{R}$eo connectors and suitable to be mapped into standard animation languages, such as Flash© [91]. Connector Animation enables one to synthesise insightful rich multimedia animations that are easy to comprehend by non $\mathcal{R}$eo experts, and serves as a valuable tool for $\mathcal{R}$eo connector developers. Currently, two implementations of Connector Animation exist, both of which generate Flash animations.

6.1 INTRODUCTION: WHAT DO WE MEAN BY CONNECTOR ANIMATION?

Before delving into the details of Connector Animation we present a series of examples with which we illustrate what we mean by animations of the dataflow behaviour of connectors. The examples introduce progressively the visual elements that are used to represent relevant aspects present in the model of a connector, and illustrate the actions used to describe the flow of data tokens throughout the connector.

6.1.1 *Example 1*

Consider a simple connector consisting of a *SyncDrain* with a component connected to each of its boundary nodes performing I/O operations:

$$\begin{aligned}\mathcal{C}on_{SyncDrain} = \big\langle &\{A,B\},\{A,B\},\big\{(A^{o})_{I/O},(A^{i},B^{i})_{SyncDrain},(B^{o})_{I/O}\big\}\,,\\ &\{T_{\square},T_{\blacksquare}\}\cdot\big\{T_{SyncDrain}\big\}\cdot\{T_{\square},T_{\blacksquare}\},\\ &\eta_{I/O}\times\eta_{SyncDrain}\times\eta_{I/O}\big\rangle.\end{aligned}$$

$\mathcal{C}on_{SyncDrain}$ is graphically depicted in Figure 58a. To visually represent the connector we use the channel's visual syntax, and the visual representation of generic components: *Writer* and *Taker*. Recall that a *Writer* models a component connected to a source node, and capable of performing *write* operations; a *Taker* models a component connected to a sink node, and capable of performing *take* operations. Hence, a connector's environment is modelled by connecting either a *Writer* or a *Taker* to each boundary node of the connector.

Remark 6.1.1. In Connector Animation, nodes are depicted as original $\mathcal{R}$eo nodes, which abstract away their internal *Merger* and *Replicator* primitives. A node is represented as a circle that can have more then two connected edges. For connector animation we can use this more compact node depiction without compromising the correct understanding of the dataflow behaviour. In Example 6.1.3 we have a connector involving nodes with more than two edges connected.

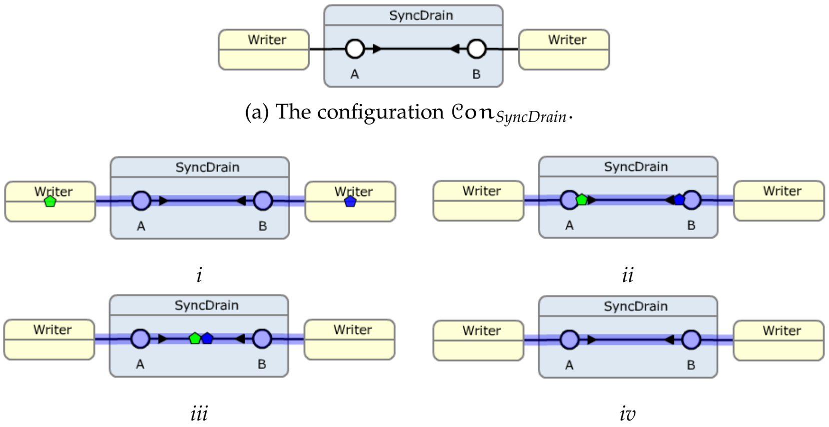
Consider in particular the colouring table $T_{\blacksquare}\cdot T_{SyncDrain}\cdot T_{\blacksquare}$. This table contains only the colouring $c_0 = \{A \mapsto$ ——, $B \mapsto$ ——$\}$. Performing c_2 the connector evolves to the configuration given by:

$$(\eta_{I/O}\times\eta_{SyncDrain}\times\eta_{I/O})(c_0) = T_{\square}\cdot T_{SyncDrain}\cdot T_{\square}.$$

The colouring c_0 models the step behaviour that takes $\mathcal{C}on_{SyncDrain}$ from configuration $T_{\blacksquare}\cdot T_{SyncDrain}\cdot T_{\blacksquare}$ to configuration $T_{\square}\cdot T_{SyncDrain}\cdot T_{\square}$. The basic idea underlying Connector Animation is to capture visually step behaviours like this. In order to accomplish that, we use *visual representations* for *configurations* and *colourings*.

The configuration $T_{\blacksquare}\cdot T_{SyncDrain}\cdot T_{\blacksquare}$ is depicted in Figure 58b*i*. For the representation of colourings, Connector Animation introduces new visual elements. The parts of the connector that a colouring assigns the *dataflow* colour (——) are overlaid with a thick translucent blue bar[1]. Hence, parts of the connector highlighted in blue correspond to places where data flow. Colouring c_0 assigns the dataflow colour to both nodes, depicting the flow of data through the *SyncDrain*. Therefore, *SyncDrain* is highlighted in blue. To provide the visual experience of data flowing, data tokens

[1] We postpone until the next example the discussion about the representation of parts of the connector that a colouring assigns *no-dataflow* colours.



(a) The configuration $\mathfrak{Con}_{SyncDrain}$.

(b) sequence of dataflow actions executing colouring c_0.

Figure 58: Animation of the colouring c_0 of the $\mathfrak{Con}_{SyncDrain}$.

represented as pentagons ⬠ are used, depicting the data written by a *write* operation. Tokens can be *created*, *copied*, *deleted*, and *moved*. In Figure 58b*i* the creation of two data tokens is depicted, one by each *write* operation, and in Figure 58b*ii* the data tokens move from the *Writer*s through the nodes A and B. The creation of a data token and the subsequent movement of the token through the node models the intuitive idea that when a *write* operation succeeds it sends data through the node to which it is connected. In Figure 58b*iii* each data token moves from the *Writer* in the direction of the centre of the *SyncDrain*, and in Figure 58b*iv*, the two data tokens are deleted. The movement and subsequent deletion of the data tokens along the channel fit the intuition that a *SyncDrain* consumes simultaneously data from both its channel-ends.

After the execution of the dataflow actions corresponding to a colouring, the connector representation is that of the next configuration that the connector evolves into. In our example, after the execution of the dataflow actions associated with colouring c_0 the connector representation is that of the configuration $T_{\square} \cdot T_{SyncDrain} \cdot T_{\square}$ depicted in Figure 58a.

We argue that the depictions of data tokens create a visual experience that describes intuitively and in detail the dataflow behaviour of the *SyncDrain*.

6.1.2 *Example 2*

Consider a connector similar to the previous one. We just substitute the *SyncDrain* channel with a *FIFO*$_1$ channel.

$$\begin{aligned}\mathcal{C}on_{FIFO_1} = \big\{&\{A,B\},\{A,B\},\big\{(A^{o})_{I/O},(A^{i},B^{o})_{FIFO_1},(B^{i})_{I/O}\big\},\\ &\{T_{\square},T_{\blacksquare}\}\cdot\big\{T_{empty},T_{full}\big\}\cdot\{T_{\square},T_{\blacksquare}\},\ \eta_{I/O}\times\eta_{FIFO_1}\times\eta_{I/O}\big\}.\end{aligned}$$

Particularly, we look at the configuration $T_{\blacksquare}\cdot T_{empty}\cdot T_{\blacksquare}$. In this configuration the connector has only one possible behaviour given by the colouring $c_1=\{A\mapsto$ ——, $B\mapsto$ -▷-} depicted in Figure 59a. Parts of the connector where c_1 assigns the *flow* colour are highlighted in blue. A data token is created in the *Writer* connected at node A, which subsequently moves along the *FIFO*$_1$ channel until it reaches the empty buffer cell where it stops and remains still (frames *i-iv*). This sequence of dataflow actions models the storage of the data token in the buffer cell of the *FIFO*$_1$. The parts of the connector where a colouring assigns a *no-dataflow* colour, are overlaid with red triangles aligned according to the triangles of the *no-dataflow* colour. Consequently, the red triangles ▷ point away from the source of dataflow-exclusion, and in the direction of the excluded source of dataflow. In parts of the connector where the connector assigns the *no-dataflow* colour, data does not flow. Therefore potentially existing tokens in this part of a connector are not subject to any dataflow actions. In frame *i*, we can see that where colouring c_1 assigns a *no-dataflow* colour, the connector representation is overlaid with red triangles. The vertex of the triangles point away from the empty buffer cell of the *FIFO*$_1$ (source of the exclusion) in the direction of the *Taker* connected at node B (disallowing any potential *take* operation to succeed).

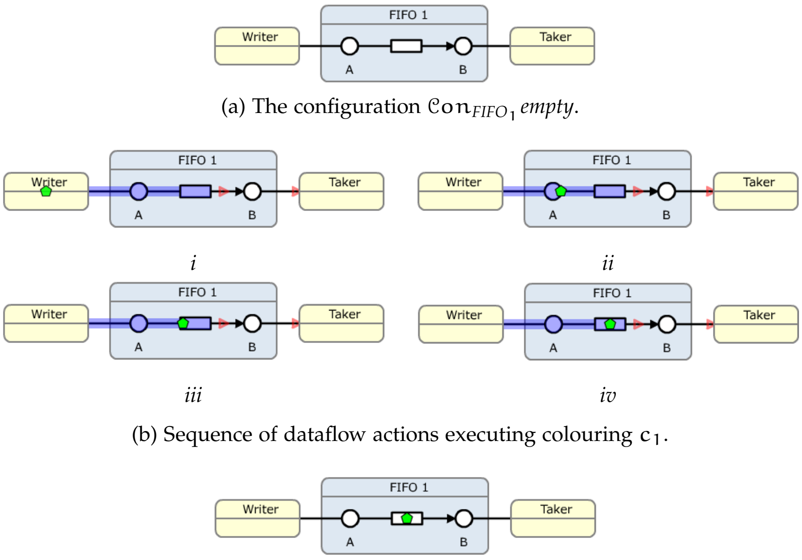
Performing c_1, the connector evolves into the configuration depicted in Figure 59c:

$$(\eta_{I/O}\times\eta_{FIFO_1}\times\eta_{I/O})(c_1)=T_{\square}\cdot T_{full}\cdot T_{\blacksquare}.$$

The buffer cell is now filled with the data token that came from A. In this configuration, the connector admits as possible behaviour the colouring $c_2=\{A\mapsto$ -◁-, $B\mapsto$ ——} depicted in Figure 59e. Colouring c_2 takes the connector $\mathcal{C}on_{FIFO_1}$ to the configuration:

$$(\eta_{I/O}\times\eta_{FIFO_1}\times\eta_{I/O})(c_2)=T_{\square}\cdot T_{empty}\cdot T_{\square}.$$

Colouring c_2 moves the data token stored in the buffer cell through the output channel-end of the *FIFO*$_1$ channel (frames *i,ii*), until it reaches the *Taker* (frame *iii*). Once there, the data token is deleted (frame *iv*), modelling the intuitive idea that a *take* operation consumes data from the node to which it is connected. The parts of the connector where c_2 assigns *no-dataflow* are represented according to the same rules we discussed for colouring c_1. Two ▷ overlaying the representation of the connector point away from the filled buffer cell (source of the exclusion), and in direction of the boundary node A.

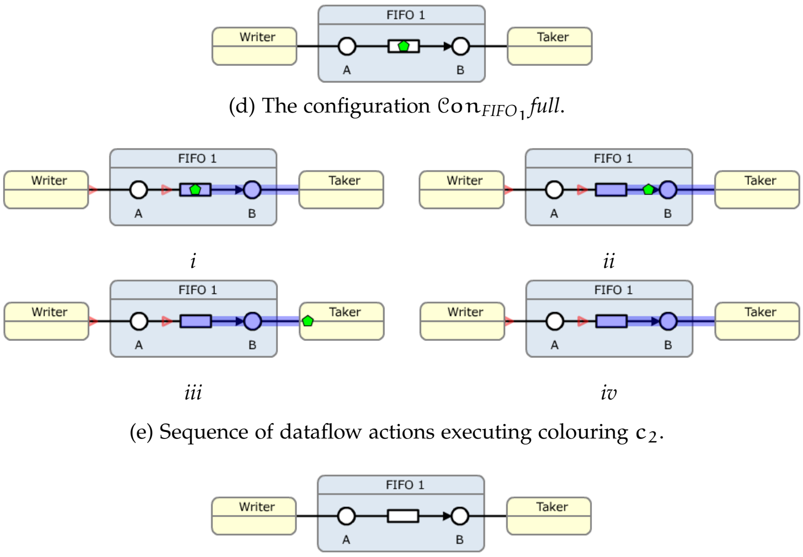


(a) The configuration $\mathfrak{Con}_{FIFO_1}$ *empty*.

(b) Sequence of dataflow actions executing colouring c_1.

(c) The configuration $\mathfrak{Con}_{FIFO_1}$ *full*.

Figure 59: Animation of the colouring c_1 of the $\mathfrak{Con}_{FIFO_1}$ (Cont.).



(d) The configuration $\mathfrak{Con}_{FIFO_1}$ *full*.

(e) Sequence of dataflow actions executing colouring c_2.

(f) The configuration $\mathfrak{Con}_{FIFO_1}$ *empty*.

Figure 59: Animation of the colouring c_2 of the $\mathfrak{Con}_{FIFO_1}$.

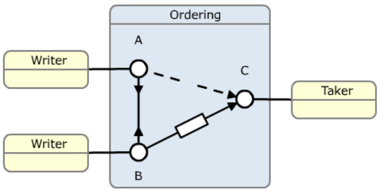


Figure 60: The configuration $\mathcal{C}on_{Ordering}$ *empty*.

6.1.3 *Example 3*

In this example we introduce the *Ordering* connector [7] depicted in the blue rectangle of Figure 60. The *Ordering* connector is built out of 3 channels: $(A^i, C^o)_{LossySync}$, $(A^i, B^i)_{SyncDrain}$, and $(B^i, C^o)_{FIFO_1}$. Nodes A and B are *Replicator*s, and node C is a *Merger*. This connector imposes an ordering on the flow of the data coming from the source nodes A and B. The *SyncDrain* connecting nodes A and B enforces that data flows through A and B synchronously. In the initial configuration of the connector, the buffer cell is empty, and the *FIFO*$_1$ together with the *LossySync* guarantee that the data obtained from B is stored in the *FIFO*$_1$ buffer, whereas the data obtained from A is delivered to C. When the buffer of the *FIFO*$_1$ is full, data cannot flow through neither A nor B. However, C can obtain the data stored in the buffer. The buffer is then empty again and the connector is back to its initial configuration.

Ordering is an example of a connector where a small number of channels and nodes constrain in a non-trivial manner the possible ways data can flow through the connector. The synchronisation and mutual exclusion constraints imposed by each channel and node propagate across the entire connector. The global dataflow behaviour that emerges is rather complex to explain textually, yielding verbose and somewhat confusing explanations, as in the previous paragraph. We argue that a more suitable way to convey the behaviour of the connector is by means of visual representation and animation.

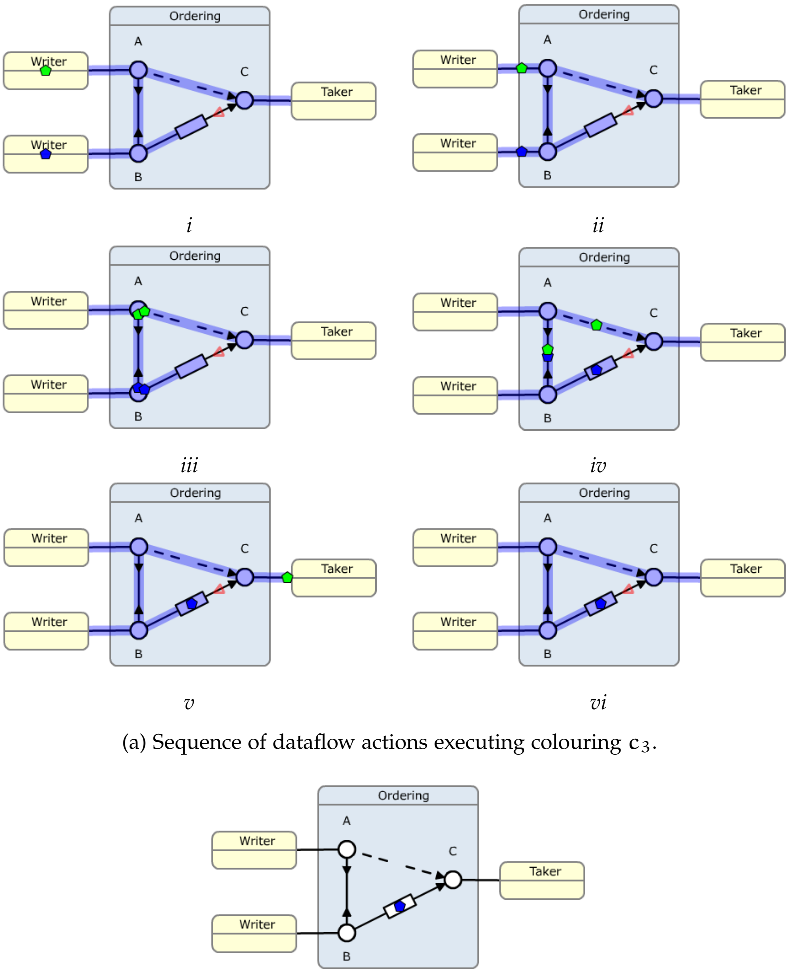
Consider the connector $\mathcal{C}on_{Ordering}$ in the following configuration:

$$T_{\blacksquare}(A) \cdot T_{\blacksquare}(B) \cdot T_{oempty}(A, B, C) \cdot T_{\blacksquare}(C).$$

The buffer cell of the *FIFO*$_1$ is empty and the *Writer*s and *Taker* perform I/O operations. In this configuration, the dataflow behaviour is prescribed by colouring c_3, depicted in Figure 61a*i*.

The sequence of frames *ii-vi* in Figure 61a come from the animation of colouring c_3. We can see in frame *ii* that the data tokens produced by the *Writer*s at nodes A and B flow through their respective nodes. Source nodes, such as nodes A and B, are replicators, hence they create copies of their incoming data tokens.

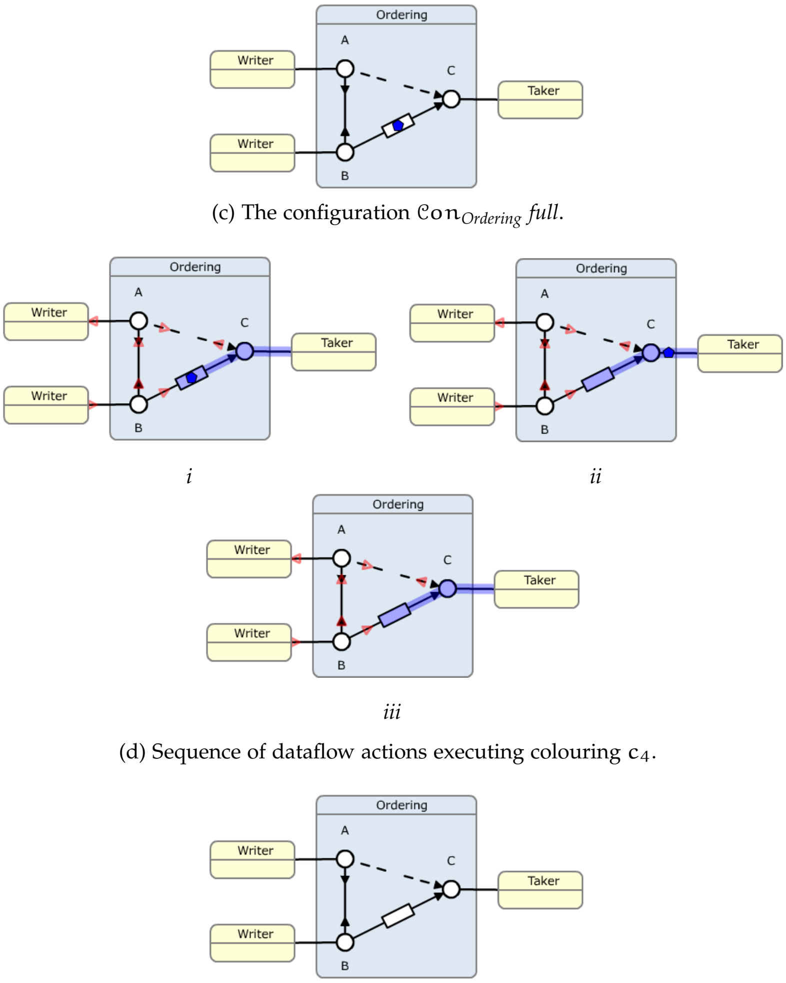
Remark 6.1.2. Data tokens with different contents are represented with different colours, but tokens with different colour can have the same data content. The latter



(a) Sequence of dataflow actions executing colouring c_3.

(b) The configuration $\mathcal{C}on_{\mathit{Ordering}}$ *full*.

Figure 61: Animation of the colouring c_3 of the ordering connector (Cont.).



(c) The configuration $\mathcal{C}on_{Ordering}$ *full*.

i

ii

iii

(d) Sequence of dataflow actions executing colouring c_4.

(e) The configuration $\mathcal{C}on_{Ordering}$ *empty*.

Figure 61: Animation of the colouring c_4 of the ordering connector.

case arises simply because the information about actual data contents is not available. Copies of the same data token have the same colour. We assume that different *Writer* components produce different data tokens, unless it is stated otherwise.

At node A the green data token is replicated into two copies (frame *iii*). Note that we now have two green data tokens, denoting that they carry the same data. One copy is routed through the *SyncDrain* whereas the other flows through the *LossySync*. Concurrently, at node B the blue data token is replicated into two copies. One blue token is routed through the *SyncDrain*, and the other through the $FIFO_1$. The *SyncDrain* consumes the incoming green and blue tokens (frames *iii-iv*) in the same fashion we described in Example 1. The other green token flows through *LossySync*, and node C until it reaches the *Taker* component where it is consumed (frames *iv-vi*); the other blue token is stored in the $FIFO_1$ buffer cell in the same fashion we described in Example 2. Upon completion of colouring c_3 the connector reaches the configuration depicted in Figure 61b:

$$T_{\square}(A) \times T_{\square}(B) \cdot T_{\mathit{ofull}}(A, B, C) \cdot T_{\square}(C).$$

We will return to this example later, as we use it as a running example throughout this chapter.

6.2 connector animation framework

Connector Animation can be described as a three-level modelling framework for $\mathcal{R}$eo connectors. At the top level of abstraction we have the connector colouring semantics to describe the abstract dataflow behaviour of connectors. At the lowest level of abstraction Connector Animation provides the animation description language—a small generic[2] animation language that is tailored to describe in detail the structure and dataflow behaviour of $\mathcal{R}$eo connectors. At its intermediate level of abstraction we have the animation specification language. This is a necessary intermediate model to bridge the gap between the specification model (Connector Colouring) and the executable dataflow model (Connector Description). The dataflow behaviour of connectors is simulated by executing its animation description model; the simulation is visualised using some specific animation language, which in this thesis is Flash©.

We proceed by presenting the Connector Animation framework in detail.

6.2.1 *Animation Specification Model*

Consider a finite set of locations $\mathcal{L}oc$. Each *location* in $\mathcal{L}oc$ has a name of the form l_i. Variables $\alpha, \beta, \gamma, \delta \ldots$ range over the set $\mathcal{L}oc$. A location is a placeholder for a single

[2] The animation description language is generic because its syntax is not tied to any production animation tool.

data-token in $\mathcal{Data}$, where $\mathcal{Data}$ is an enumerable set of data-tokens. A data-token, or simply *token*," is an abstract representation of a unit of information flowing through a connector. We use the terms *full* and *vacant* to qualify, respectively, a location that holds one token, and a location that is vacant.

Tokens flow through a connector according to specified dataflow actions prescribed by an animation specification. There are four basic dataflow actions for *moving*, *copying*, *creating*, and *deleting* tokens:

- The *move* action, denoted by $\alpha \blacktriangleright \beta$, reads as *move the token in location* α *to location* β. $\alpha \blacktriangleright \beta$ executes if and only if location α is full and location β is vacant. When it executes, it moves the token in location α to the location β.

- The *copy* action, denoted by $\alpha\,!\,\beta$, reads as *copy token from location* α *to location* β. $\alpha\,!\,\beta$ executes if and only if location α is full and location β is vacant. When it executes, it creates a replica in location β of the token in location α.

- The *create* action, denoted by $\bigstar\,\alpha$, reads as *create a token in location* α. $\bigstar\,\alpha$ executes if and only if location α is vacant. When it executes, it creates a token in location α.

- The *delete* action, denoted by $\boxtimes\,\alpha$, reads as *delete the token from location* α. $\boxtimes\,\alpha$ executes if and only if location α is full. When it executes, it deletes the token in location α.

An *animation specification* consists of a *finite set of dataflow actions*. As such, an animation specification consists of a finite unordered collection of actions, where the same action can appear multiple times if it has different arguments.

An animation specification can be empty, denoted by 0; can contain a single dataflow action; or a set of many dataflow actions.

The composition of two animation specifications as_1 and as_2 is an animation specification given by the union of the two animation specifications, denoted as $as_1 \cup as_2$.

Definition 6.2.1 (Animation Specification Language (ASL))**.** The animation specification language over $\mathcal{Loc}$ is defined by the following grammar:

$$as \ ::= \ \alpha \blacktriangleright \beta \ \mid \ \alpha\,!\,\beta \ \mid \ \bigstar\,\alpha \ \mid \ \boxtimes\,\alpha \ \mid \ 0 \ \mid \ as_1 \cup as_2$$

Notation 6.2.2. To avoid the use of parenthesis, we assume that the composition operator $\cup$ has the lowest precedence, and the other operators all have the same precedence. Instead of writing $(\alpha \blacktriangleright \beta) \cup (\beta\,!\,\gamma)$, we simply write $\alpha \blacktriangleright \beta \cup \beta\,!\,\gamma$.

Example 6.2.3 (animation specifications)**.** To illustrate the syntax of the animation specifications and provide some informal intuition on their semantics, we present a series of examples:

1) $l_1 \blacktriangleright l_2 \cup \boxtimes l_2$ — an animation specification composed of two actions that specify that a token moves from location l_1 to location l_2 and that a token is deleted from location l_2. Nothing is stated about the order of execution of the two actions, or whether they can be executed in parallel. Nevertheless, in this example, the specification imply an execution order: for the action $l_1 \blacktriangleright l_2$ to execute, location l_2 must be vacant, and in contrast, the action $\boxtimes l_2$ requires location l_2 to be full to be able to execute; hence, since a location is either vacant or full we can already say that the two actions cannot execute in parallel.

2) $\bigstar l_4 \cup l_6 \,!\, l_5$ — an animation specification composed of two actions that specify that a token is created in location l_4 and a replica of the token in location l_6 is created in location l_5. The actions refer to different locations, suggesting that they can execute in any order or in parallel.

3) $\boxtimes l_3 \cup l_3 \blacktriangleright l_7$ — an animation specification composed of two actions that specify that the token in location l_3 must be deleted and that the token in location l_3 moves to location l_7. Location l_3 is required to be full by one action and vacant by the other action, which is a requirement that a location cannot fulfil, suggesting that the animation specification cannot execute both of its actions.

4) $\boxtimes l_3 \cup l_3 \blacktriangleright l_7 \cup \bigstar l_3$ — an animation specification composed out of the animation specification of the previous example with the third action $\bigstar l_3$. Additionally to what is specified by the previous example, the resulting animation specification specifies that a token is created in location l_3. With the addition of the action $\bigstar l_3$ the animation specification can now interleave the execution of the actions $\boxtimes l_3$, and $l_3 \blacktriangleright l_7$ with the execution of $\bigstar l_3$. For example, after the execution of $\boxtimes l_3$ which requires location l_3 to be empty after its execution, action $\bigstar l_3$ can execute, filling location l_3, and making it possible for $l_3 \blacktriangleright l_7$ to execute.

The domain of an animation specification $as \in \text{ASL}$ is the set of locations that are referred to in *as*.

Definition 6.2.4 (Domain of an animation specification)**.** The domain of an animation specification $as \in \text{ASL}$, denoted by $\mathtt{dom}(as)$, is a subset of $\mathcal{Loc}$ inductively defined by the following equations:

$$\begin{array}{ll}
\mathtt{dom}(\alpha \blacktriangleright \beta) = \{\alpha, \beta\} & \mathtt{dom}(\boxtimes \alpha) = \{\alpha\} \\
\mathtt{dom}(\alpha \,!\, \beta) = \{\alpha, \beta\} & \mathtt{dom}(0) = \emptyset \\
\mathtt{dom}(\bigstar \alpha) = \{\alpha\} & \mathtt{dom}(as_1 \cup as_2) = \mathtt{dom}(as_1) \cup \mathtt{dom}(as_2)
\end{array}$$

Remark 6.2.1*.* The domain of an animation specification abstracts from the number of times the same location is referred to.

The semantics of ASL is defined in terms of *frames*. A *frame* is a predicate on $\mathcal{Loc}$ that indicates whether a location is full.

Definition 6.2.5 (frame)**.** The frame of an animation specification *as*, is the subset of all full locations in $\mathtt{dom}(as)$. Given a frame Γ, as a shorthand, we write Γ^{as} to denote the frame $\Gamma \cap \mathtt{dom}(as)$ of the animation specification *as*.

The main idea underlying the semantics of ASL is to determine whether, in a given frame, an action (or multiple actions) of an animation specification can execute (in parallel), and consequently produce a new frame. For an action to execute, the data and exclusion dependencies implied by the action must be met without violating the invariant that a location can only hold at most one token. The syntax of an animation specification does not determine any particular order of execution of the dataflow actions, neither does it determine whether dataflow actions can be executed in parallel. It is the semantics of ASL that determines whether there exists an order in which all the actions of an animation specification can be executed, possibly in parallel. The execution of an animation specification with multiple actions results in a sequence of frames. The semantics of ASL is given in the relational style of structured operational semantics, also called natural semantics [60, 78]. For an animation specification $as \in$ ASL the *evaluation relation* $\mathcal{E} \subseteq \mathcal{P}\mathcal{L}oc \times \text{ASL} \times \mathcal{P}\mathcal{L}oc$ is defined inductively by the rules in Table 21. For convenience, we write $\Gamma \vdash as \rightarrow \Gamma'$ to denote the triple $(\Gamma, as, \Gamma') \in \mathcal{E}$, which reads as: *the animation specification as executes in frame* Γ *and produces a new frame* Γ'.

$$\textit{move}\ \frac{}{\Gamma \vdash \alpha \blacktriangleright \beta \rightarrow (\Gamma \setminus \{\alpha\}) \cup \{\beta\}}\ \mathsf{if}\ \alpha \in \Gamma,\ \beta \notin \Gamma$$

$$\textit{copy}\ \frac{}{\Gamma \vdash \alpha\,!\,\beta \rightarrow \Gamma \cup \{\beta\}}\ \mathsf{if}\ \alpha \in \Gamma,\ \beta \notin \Gamma \qquad \textit{skip}\ \frac{}{\Gamma \vdash 0 \rightarrow \Gamma}$$

$$\textit{create}\ \frac{}{\Gamma \vdash \bigstar\, \alpha \rightarrow \Gamma \cup \{\alpha\}}\ \mathsf{if}\ \alpha \notin \Gamma \qquad \textit{delete}\ \frac{}{\Gamma \vdash \boxtimes\, \alpha \rightarrow \Gamma \setminus \{\alpha\}}\ \mathsf{if}\ \alpha \in \Gamma$$

(a) Action axioms.

$$\textit{seq}_1\ \frac{\Gamma \vdash as_1 \rightarrow \Gamma_1 \qquad \Gamma_1 \vdash as_2 \rightarrow \Gamma'}{\Gamma \vdash as_1 \ \cup\ as_2 \rightarrow \Gamma'} \qquad \textit{seq}_2\ \frac{\Gamma \vdash as_2 \rightarrow \Gamma_2 \qquad \Gamma_2 \vdash as_1 \rightarrow \Gamma'}{\Gamma \vdash as_1 \ \cup\ as_2 \rightarrow \Gamma'}$$

$$\textit{par}\ \frac{\Gamma^{as_1} \vdash as_1 \rightarrow \Gamma_1 \qquad \Gamma^{as_2} \vdash as_2 \rightarrow \Gamma_2}{\Gamma \vdash as_1 \ \cup\ as_2 \rightarrow (\Gamma_1 \cup \Gamma_2) \cup (\Gamma \setminus \Gamma^{as_1 \ \cup \ as_2})}\ \mathsf{if}\ \mathtt{dom}(as_1) \cap \mathtt{dom}(as_2) = \emptyset$$

(b) Inference rules.

Table 21: Natural semantics of ASL.

The action axioms in Table 21(a) are guarded by data dependency and exclusion conditions imposed by the invariants of a location. The semantics guarantees that the execution of the actions of an animation specification *as* preserves two invariants: *as* does not *move*, *create*, or *copy* a token into a full location; *as* does not *delete* or *move* a token from an empty location. For example, to execute the dataflow action $l_1 \blacktriangleright l_2$ in a *frame* Γ, rule *move* requires location l_1 to be full ($l_1 \in \Gamma$) and location l_2 to be vacant ($l_2 \notin \Gamma$).

The inference rules of ASL semantics are depicted in table 21(b). We have three inference rules each accounting for a different execution ordering. The inferential reading of rule seq_1 says that if we can evaluate as_1 in Γ obtaining Γ', and as_2 in Γ' obtaining L'', then we can evaluate $as_1 \cup as_2$ in Γ and obtain Γ''. The inference rule seq_2 is similar to rule seq_1, except that it inverts the order in which as_1 and as_2 execute. The inferential reading of rule seq_2 says that if we can evaluate as_2 in Γ obtaining Γ', and as_1 in Γ' obtaining L'', then we can evaluate $as_1 \cup as_2$ in Γ and obtain Γ''. Rules seq_1 and seq_2 apply whenever there is data dependency between as_1 and as_2. More specifically seq_1 applies when the execution of as_2 depends on the execution of as_1; and vice-versa, seq_2 applies when the execution of as_1 depends on the execution of as_2.

The inferential reading of rule *par* says that if as_1 and as_2 are such that their domains are disjoint, and both can evaluate in parallel, as_1 in Γ^{as_1} and as_2 in Γ^{as_2}, producing the frames Γ_1 and Γ_2, respectively, then we can evaluate $as_1 \cup as_2$ in Γ and obtain the frame resulting from the union of Γ_1, Γ_2, and the locations in the initial frame Γ that are left unchanged because they are not in the domain of the animation $as_1 \cup as_2$.

These inference and the axiom rules allow us to construct derivation trees for animation specifications. A derivation tree encodes an ordering of the actions that guarantees their execution. Furthermore, at the completion of each execution step (inference or axiom rule) we have its corresponding subsequent frame. The sequence of frames encodes the dynamic aspect (tokens flowing) of an animation.

Definition 6.2.6 (Derivation tree)**.** For a given frame Γ, and an animation specification *as*, a *derivation tree* of *as* in Γ is a tree of instances of the inference and axiom rules of table 21, where all the leaves are action axiom instances, and the root node is of the form $\Gamma \vdash as \rightarrow \Gamma'$. We denote the set of derivation trees as root $\Gamma \vdash as \rightarrow \Gamma'$ by $\mathcal{D}(\Gamma \vdash as \rightarrow \Gamma')$.

Example 6.2.7. We construct derivation trees for the animation specifications of Example 6.2.3 to illustrate the use of the semantics of ASL:

1) For $\Gamma = \{l_1\}$ we can construct the following derivation tree:

$$seq_1\,\dfrac{move\,\dfrac{}{\{l_1\} \vdash l_1 \blacktriangleright l_2 \rightarrow \{l_2\}} \qquad delete\,\dfrac{}{\{l_2\} \vdash \boxtimes\, l_2 \rightarrow \emptyset}}{\{l_1\} \vdash l_1 \blacktriangleright l_2 \cup \boxtimes\, l_2 \rightarrow \emptyset}$$

2) For $\Gamma = \{l_6\}$ we can construct the following derivation tree:

$$\textit{par}\;\dfrac{\textit{create}\;\dfrac{}{\{l_6\} \vdash \bigstar\, l_4 \rightarrow \{l_4, l_6\}} \qquad \textit{copy}\;\dfrac{}{\{l_6\} \vdash l_6 \,!\, l_5 \rightarrow \{l_6, l_5\}}}{\{l_6\} \vdash \bigstar\, l_4 \;\cup\; l_6 \,!\, l_5 \rightarrow \{l_4, l_6, l_5\}}$$

3) $\boxtimes\, l_3 \;\cup\; l_3 \blacktriangleright l_7$ does not admit any derivation tree. The animation specification requires two tokens from location l_3. Even a frame wherein l_3 is full is insufficient to fulfil the token requirements implied by this animation specification.

4) For $\Gamma = \{l_3\}$ we can construct the following derivation tree:

$$\textit{seq}_1\;\dfrac{\textit{delete}\;\dfrac{}{\{l_3\} \vdash \boxtimes\, l_3 \rightarrow \emptyset} \qquad \textit{seq}_2\;\dfrac{\textit{create}\;\dfrac{}{\emptyset \vdash \bigstar\, l_3 \rightarrow \{l_3\}} \qquad \textit{move}\;\dfrac{}{\{l_3\} \vdash l_3 \blacktriangleright l_7 \rightarrow \{l_7\}}}{\emptyset \vdash l_3 \blacktriangleright l_7 \;\cup\; \bigstar\, l_3 \rightarrow \{l_7\}}}{\{l_3\} \vdash \boxtimes\, l_3 \;\cup\; l_3 \blacktriangleright l_7 \;\cup\; \bigstar\, l_3 \rightarrow \{l_7\}}$$

6.2.2 *Reo Animation Specification*

We now present how we can use animation specifications to refine the dataflow behaviour defined by a constructive 3-colouring (Definition 3.2.1). The nodes of a 3-colouring that are coloured with the flow colour are denoted with locations. Data flows through those locations according to dataflow actions prescribed by an animation specification. A node that is coloured with a flow colour is in that way refined into dataflow actions that detail how the data item (token) flows.

However, not every animation specification constitutes a possible refinement of a particular colouring. A colouring enforces dataflow constraints that an animation specification must satisfy to be considered as a possible candidate to refine the colouring:

a) We consider a violation of the dataflow behaviour encoded by a colouring if: the flow colour of a node is refined into an empty set of dataflow actions; and dually, we consider it a violation if the no-dataflow colour is refined into a non-empty set of dataflow actions. With that in mind, every node coloured with the flow colour must be associated with a location contained in the domain of the animation specification that refines the colouring. On the other hand, the nodes coloured with a no-dataflow colour must not be associated with a location contained in the domain of the animation specification. Observe that it is enough to talk in terms of locations in the domain of the animation specification, because by definition, the domain contains the locations where the dataflow actions take place.

b) The separation that $\mathcal{R}$eo nodes model must be respected by the animation specification that refines a colouring. Different nodes where, according to the colouring, data flow must be associated with different locations contained in the domain of the animation specification.

c) The dataflow actions that refine the flow colour assigned by a colouring must respect the causal relation of the colouring. Consider a colouring c that assigns the flow colour to two nodes n_1 and n_2, and has a causal relation that contains the ordered pair (n_1, n_2). The ordered pair indicates that the dataflow observed in n_2 *is caused by* the dataflow observed in n_1, or in simple terms, the data that flow to n_2 come from n_1. Naturally, the dataflow causality must be preserved by the animation specification that refines the dataflow assigned to nodes n_1 and n_2. An animation specification that refines a colouring c must include a *move* or a *copy* action from the location associated with node n_1, to the location associated with node n_2, *if* the ordered pair (n_1, n_2) is in the causal relation of c. And vice-versa, colouring c must include the pair (n_1, n_2) in its causal relation if the animation specification contains a *move* or a *copy* action from the location associated with node n_1, to the location associated with node n_2. This way we rule-out the possibility of an animation specification introducing a dataflow causality that is not defined by the colouring, and vice versa.

d) The animation specification that refines a colouring must admit a derivation tree in the same conditions that the colouring applies:

- the animation specification must admit a derivation tree in the frame containing the locations associated with the input nodes that are coloured with the flow colour, because when the colouring applies, an input node has data, which means that the associated locations are full.
- The derivation tree of the animation specification must yield a frame that contains the locations associated with the output nodes that are assigned the flow colour by the colouring. These nodes receive data as a result of the dataflow prescribed by the colouring, which means that the locations must be full after the dataflow actions of the animation specification execute.

Note that there can be additional locations in either frames. For instance, each frame can contain locations that model the buffer cells of a connector. Those locations are not associated with either input or output nodes and yet can be full in the frame in which the animation specification admits a derivation tree or in the frame yielded by the animation specification.

Definition 6.2.8 (Refinement map)**.** Consider a constructive 3-colouring (c, R) over the set of nodes N, the set of nodes coloured with the flow colour $\overline{N}_c = \{n \mid n \in N, c(n) = \text{———}\}$, and an animation specification *as*. The animation specification *as refines* the dataflow specified by the colouring (c, R) if there exists a *refinement map*:

$$\overline{N}_c \xrightarrow{r} \mathtt{dom}(as)$$

such that:

1. r is total and injective;

2. $(n_1, n_2) \in R$ and $n_1, n_2 \in \overline{N}_c \iff (\alpha \blacktriangleright \beta$ or $\alpha \,!\, \beta) \in$ *as* and $r(n_1) = \alpha$ and $r(n_2) = \beta$;

3. *as* admits a derivation tree $t \in \mathcal{D}(\Gamma \vdash \mathit{as} \to \Gamma')$ where:

   $\Gamma \supseteq \left\{ r(n) \mid n \in \overline{N}_c,\ n \text{ is an input node} \right\}$, and
   $\Gamma' \supseteq \left\{ r(n) \mid n \in \overline{N}_c,\ n \text{ is an output node} \right\}$.

Remark 6.2.2. The properties of a refinement map r follow from the conditions listed before the definition. 1. captures a) and b): the domain of r is given by the set $\overline{N}_c$ because we do not want to assign locations of the animation specification to the nodes coloured by c with the no-dataflow colour; r must be total because all the nodes coloured by c with the flow colour must be associated with a location in the animation specification; r is injective because different nodes must correspond to different locations. 2. captures c), and 3. captures d).

We call the structure resulting from a colouring, an animation specification, and a refinement map between the two, a $\mathcal{R}$eo animation specification.

Definition 6.2.9 ($\mathcal{R}$eo animation specification)**.** A $\mathcal{R}$eo animation specification over nodes $N \subseteq \mathcal{N}ode$, is a tuple $\langle (c, R), \mathit{as}, r \rangle$ where:

- (c, R) is a constructive 3-colouring over N;
- *as* is an animation specification;
- r is a refinement map between (c, R) and *as*.

To collect the $\mathcal{R}$eo animation specifications corresponding to the *animated* dataflow behaviour of a $\mathcal{R}$eo connector we introduce animation tables.

Definition 6.2.10 (Animation table)**.** Consider a set of nodes $N \subseteq \mathcal{N}ode$. An animation table AT, over N is a set of $\mathcal{R}$eo animation specifications over N. We write $AT \sqsubseteq T$ to denote that the animation table AT refines the colouring table $T = \{(c, R) \mid \langle (c, R), \mathit{as}, r \rangle \in AT\}$.

Example 6.2.11. Table 22 depicts the animation table of the *Sync* channel. On the left of the animation table we show the *Sync* channel connecting two boundary nodes a, and b. Each line of the animation table contains one $\mathcal{R}$eo animation specification of the animation table. The first column contains the constructive 3-colouring, the causal relation appears just below each 3-colouring. The second column contains the animation specification. The third column contains the refinement map between the constructive 3-colouring and the animation specification. Consider the first entry. We can verify that it indeed complies with the definition of $\mathcal{R}$eo animation specification, and more interestingly, that the map $\{a \mapsto \alpha,\, b \mapsto \beta\}$ is a refinement map. According to the 3-colouring we have that indeed $\overline{N}_c = \{a, b\}$, and $\mathtt{dom}(\alpha \blacktriangleright \beta) = \{\alpha, \beta\}$;

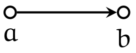
	AT_{Sync}		
	——— $\{(a,b)\}$	$\alpha \blacktriangleright \beta$	$\{a \mapsto \alpha,$ $b \mapsto \beta\}$
	- - -▷- - - $\{(a,b)\}$	0	$\emptyset$
	- - -◁- - - $\{(b,a)\}$	0	$\emptyset$
	- ▷- - ◁- · $\emptyset$	0	$\emptyset$

Table 22: Animation table of *Sync*

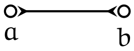
hence r is total and injective as required. We have (a, b) in the causal relation, and we have that the animation specification contains $\alpha \blacktriangleright \beta$ as required. Finally, $\alpha \blacktriangleright \beta$ admits a derivation tree $\{\alpha\} \vdash \alpha \blacktriangleright \beta \rightarrow \{\beta\}$ as required. The other entries, because they correspond to colourings that assign only the no-dataflow colours, are trivial: the animation specification is 0 and the refinement map is the empty refinement map denoted by $\emptyset$.

Remark 6.2.3. From now on, we only partially display the animation tables, namely, only the $\mathcal{R}$eo animation specification entries that correspond to a colouring that assigns the flow colour to some of the nodes of the connector. The other entries, because they correspond to colourings that assign only the no-dataflow colours, are trivial: the animation specification is 0 and the refinement map is the empty refinement map denoted by $\emptyset$. We leave those out from the animation tables for brevity. Although not interesting, these entries are part of the animation tables and are important to represent the no-dataflow colouring in the animations accurately, as we have seen in Example 6.1.2.

Animation tables of other primitives

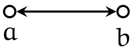
Next, we define the animation tables for other primitive connectors by refining their colouring tables that we defined in Chapter 3. In Table 23 we have the animation tables for some of the other channels in Conlang. We highlight some of their important aspects in the animation tables.

We explain the animation table $AT_{SyncDrain}$, in particular, the colouring that assigns the flow colour to both nodes a and b. The colouring is refined into the animation specification $\alpha \blacktriangleright \gamma \cup \boxtimes\gamma \cup \beta \blacktriangleright \delta \cup \boxtimes\delta$. According to the refinement map the node a is refined into location α and node b into location β. The two other additional locations γ and δ are used to model the locations where the data tokens



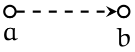
$\mathsf{AT}_{\mathit{SyncDrain}}$		
—— $\emptyset$	$\alpha \blacktriangleright \gamma \cup \boxtimes\gamma \cup \beta \blacktriangleright \delta \cup \boxtimes\delta$	$\{a \mapsto \alpha,$ $b \mapsto \beta\}$

(a) Animation table of *SyncDrain*



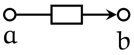
$\mathsf{AT}_{\mathit{SyncSpout}}$		
—— $\emptyset$	$\bigstar\gamma \cup \gamma \blacktriangleright \alpha \cup \bigstar\delta \cup \delta \blacktriangleright \beta$	$\{a \mapsto \alpha,$ $b \mapsto \beta\}$

(b) Animation table of *SyncSpout*



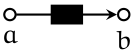
$\mathsf{AT}_{\mathit{LossySync}}$		
—— $\{(a,b)\}$	$\alpha \blacktriangleright \beta$	$\{a \mapsto \alpha,$ $b \mapsto \beta\}$
—— -◁ - $\emptyset$	$\alpha \blacktriangleright \gamma \cup \boxtimes\gamma$	$\{a \mapsto \alpha\}$

(c) Animation table of *LossySync*



$\mathsf{AT}_{\mathit{FIFO}_1}$		
—— -▷ - $\emptyset$	$\alpha \blacktriangleright \gamma$	$\{a \mapsto \alpha\}$

(d) Animation table of *FIFO*$_1$



$\mathsf{AT}_{\mathit{FIFO}_1(x)}$		
- ◁ —— $\emptyset$	$\gamma \blacktriangleright \beta$	$\{b \mapsto \beta\}$

(e) Animation table of *FIFO*$_1$*(x)*

Table 23: Animation tables of some CONLANG channels.

that come from locations α and β, respectively, flow to. Once arrived in locations γ and δ the tokens are deleted $\boxtimes\gamma, \boxtimes\delta$, thus modelling the intuitive behaviour that the data that flows through nodes a and b is consumed by the channel.

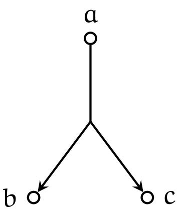
In general the *additional*[3] locations considered in an animation specification provide a means to describe the dataflow of a colouring in more detail and closer to the intended behaviour. For instance, in the *SyncSpout* channel the locations γ and δ correspond to the locations where the data tokens that the channel produces appear by $\bigstar\gamma$, $\bigstar\delta$. From these locations the data tokens flow to the locations α and β, associated by the refinement map to the nodes a and b, where they are dispensed, as intended. The animation specifications for the *LossySync*, *FIFO*$_1$ and *FIFO*$_1$*(x)* use similar modelling.

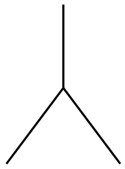
The animation tables for the *Replicator* and *Merger* are presented in Table 24. What is particular with the animation specifications of these two connectors is that there is no movement of data tokens. Observe that the dataflow action *move* ($\_ \blacktriangleright \_$) is not used. The reason for this is that the nodes of a connector in the animation are not represented in their expanded representation using explicit *Merger*s and *Replicator*s. Instead, they are represented as boundary $\circ$ or internal nodes $\bullet$. Hence, the animation specifications of *Replicator* and *Merger* are used solely to guarantee that the data tokens are copied and deleted according to the semantics of the *Replicator* and the *Merger*. In case of the *Replicator* we have that the data token received in location α associated with the source node a, is copied to the locations β and γ associated with the sink nodes b and c, respectively, and subsequently deleted. Thus we have that the causality dependencies (a, b) and (a, c) are refined into $\alpha\,!\,\beta$ and $\alpha\,!\,\gamma$, fulfilling the requirement for the refinement of causality dependencies. Additionally, since the locations β and γ are associated with the output nodes b and c, we are required by the definition of refinement map to guarantee that the frame resulting from the derivation tree of the animation specification is given by the set containing these two locations. By the semantics of the *copy* dataflow action ($\_ \,!\, \_$), we have that, indeed, locations β and γ are full after executing the animation specification, as required.

The animation specifications of the *Merger* are similar to the animation specification of the *Replicator*.

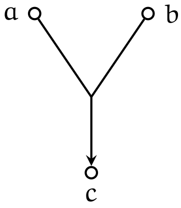
The animation tables of the I/O operations are depicted in Table 25. The *take* operation has one colouring that assigns the flow colour to the boundary node a. This colouring is refined in the animation specification $\alpha \blacktriangleright \beta \cup \boxtimes\beta$. The refinement maps node a to location α. Location β is the location in the animation where the data token that flows from location α is deleted $\boxtimes\beta$, modelling the intended behaviour that the data token received in node a is consumed by the *take* I/O operation. The animation specification for the *write* operation should be easy to understand. It follows a rationale similar to that of the *take* I/O operation.

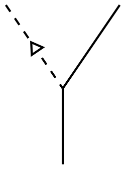
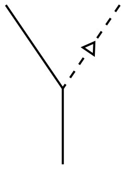
[3] The locations that are not in the codomain of the refinement map.



$\mathsf{AT}_{\mathit{Replicator}}$		
 $\{(a,b),(a,c)\}$	$\alpha\,!\,\beta \cup \alpha\,!\,\gamma \cup \boxtimes\,\alpha$	$\{a \mapsto \alpha,$ $b \mapsto \beta,$ $c \mapsto \gamma\}$

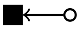
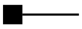
(a) Animation table of *Replicator*



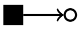
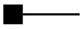
$\mathsf{AT}_{\mathit{Merger}}$		
 $\{(b,c)\}$	$\alpha\,!\,\beta \cup \boxtimes\,\alpha$	$\{b \mapsto \alpha,$ $c \mapsto \beta\}$
 $\{(a,c)\}$	$\alpha\,!\,\beta \cup \boxtimes\,\alpha$	$\{a \mapsto \alpha,$ $c \mapsto \beta\}$

(b) Animation table of *Merger*

Table 24: Animation tables of *Replicator* and *Merger*

	$AT_{I/O\ take}$		
 a	 $\emptyset$	$\alpha \blacktriangleright \beta \cup \boxtimes \beta$	$\{a \mapsto \alpha\}$

(a) Animation table of the *take* I/O operation.

	$AT_{I/O\ write}$		
 a	 $\emptyset$	$\bigstar \alpha \cup \alpha \blacktriangleright \beta$	$\{a \mapsto \beta\}$

(b) Animation table of the *write* I/O operation.

Table 25: Animation tables of the *take* and *write* I/O operations.

Compositionality of Animation Tables

Like colouring tables, animation tables can also be composed. Two $\mathcal{R}$eo animation specifications compose when the conditions to compose their underlying colourings apply and the locations common to the domains of both animation specifications map to nodes common to both colourings. This condition ensures that only the nodes and the locations that refine those nodes share the same names.

Theorem 6.2.12 (Composition of $\mathcal{R}$eo animation specifications)**.** *Consider two $\mathcal{R}$eo animation specifications $\langle (c_1, R_1), as_1, r_1 \rangle$ over nodes N_1 and $\langle (c_2, R_2), as_2, r_2 \rangle$ over nodes N_2 such that:*

1. $n \in (N_1 \cap N_2) \Rightarrow c_1(n) = c_2(n)$;
2. $(R_1 \cup R_2)^+$ *is a strict partial order;*
3. $\alpha \in (\mathit{dom}(as_1) \cap \mathit{dom}(as_2)) \iff \exists n \in (\mathrm{dom}(r_1) \cap \mathrm{dom}(r_2))$ *such that* $r_1(n) = r_2(n) = \alpha$.

We have that $\langle (c_1 \cup c_2, R_1 \cup R_2), as_1 \ \cup \ as_2, r_1 \cup r_2 \rangle$ is a $\mathcal{R}$eo animation specification over nodes $N_1 \cup N_2$, where $r_1 \cup r_2$ is a function: $(r_1 \cup r_2)(n) \begin{cases} r_1(n) & n \in \overline{N_1}_{c_1} \\ r_2(n) & n \in \overline{N_2}_{c_2} \end{cases}$ which is well defined due to condition 3.

Proof. It follows from 1. and 2. that $(c_1 \cup c_2, R_1 \cup R_2)$ is a constructive 3-colouring. Given that as_1 and as_2 are animation specifications, then by Definition 6.2.1 $as_1 \ \cup \ as_2$ is also an animation specification. We are left to check that $r_1 \cup r_2$ defines a refinement map between the colouring $(c_1 \cup c_2, R_1 \cup R_2)$ and the animation specification $as_1 \ \cup \ as_2$.

- We have to show that $(r_1 \cup r_2)(n)$ is total, and hence defined for every $n \in \overline{N_1 \cup N_2}_{(c_1 \cup c_2)}$. Observe that $\overline{N_1 \cup N_2}_{(c_1 \cup c_2)} = \overline{N_1}_{c_1} \cup \overline{N_2}_{c_2}$ follows from 1. By applying the union of the graphs of two functions, and because r_1 and r_2 are total, it follows that for $n \in \overline{N_1}_{c_1}$ and $n \notin \overline{N_2}_{c_2}$, $(r_1 \cup r_2)(n) = r_1(n)$; for $n \in \overline{N_2}_{c_2}$ and $n \notin \overline{N_1}_{c_1}$, $(r_1 \cup r_2)(n) = r_2(n)$; for $n \in \overline{N_1}_{c_1}$ and $n \in \overline{N_2}_{c_2}$, it follows from 3. that $(r_1 \cup r_2)(n) = r_1(n) = r_2(n)$. Hence $r_1 \cup r_2$ is total, as required.

- We have to show that $(r_1 \cup r_2)(n)$ is injective. That is $(r_1 \cup r_2)(n) = (r_1 \cup r_2)(n') \Rightarrow n = n'$. From 3. we know that $r_1(n) = r_2(n)$ if and only if $n \in \mathrm{dom}(r_1) \cap \mathrm{dom}(r_2)$, that is $n \in \overline{N_1}_{c_1} \cap \overline{N_2}_{c_2}$. For $n \in \overline{N_1}_{c_1}$ and $n \notin \overline{N_2}_{c_2}$, we have that $(r_1 \cup r_2)(n) = (r_1 \cup r_2)(n')$ implies $(r_1)(n) = r_1(n')$, and because r_1 is injective, it follows that $n = n'$. For $n \notin \overline{N_1}_{c_1}$ and $n \in \overline{N_2}_{c_2}$, from 3., we have $(r_1 \cup r_2)(n) = (r_1 \cup r_2)(n')$ implies $(r_2)(n) = r_2(n')$, and because r_2 is injective, it follows that $n = n'$. For $n \in \overline{N_1}_{c_1}$ and $n \in \overline{N_2}_{c_2}$, from 3., it follows that $r_1(n) = r_2(n)$, hence by the definition of the union of graphs of two functions $(r_1 \cup r_2)(n) = r_1(n) = r_2(n)$ because r_1 and r_2 are injective, it follows that $(r_1 \cup r_2)(n) = (r_1 \cup r_2)(n')$ implies $n = n'$. Hence $(r_1 \cup r_2)$ is injective.

- We have to show that for all $n, n' \in \overline{N_1 \cup N_2}_{(c_1 \cup c_2)}$ such that $(n, n') \in R_1 \cup R_2$ we have that: (1) $(\alpha \blacktriangleright \beta \text{ or } \alpha\,!\,\beta) \in as_1 \cup as_2$; and (2) $(r_1 \cup r_2)(n) = \alpha$ and $(r_1 \cup r_2)(n') = \beta$.

  * Proof of (1): by the definition of $\cup$ of sets, $(n, n') \in R_1 \cup R_2$ implies that $(n, n') \in R_1$ or $(n, n') \in R_2$. From Definition 6.2.8 of refinement maps it follows that $(\alpha \blacktriangleright \beta \text{ or } \alpha\,!\,\beta) \in as_1$ or $(\alpha \blacktriangleright \beta \text{ or } \alpha\,!\,\beta) \in as_2$. Hence by Definition 6.2.1 of ASL, finally we have that $(\alpha \blacktriangleright \beta \text{ or } \alpha\,!\,\beta) \in as_1 \cup as_2$.

  * Proof of (2): if $(\alpha \blacktriangleright \beta \text{ or } \alpha\,!\,\beta) \in as_1$ then because r_1 is a refinement map we have that $r_1(n) = \alpha$ and $r_1(n') = \beta$. If $(\alpha \blacktriangleright \beta \text{ or } \alpha\,!\,\beta) \in as_2$ then because r_2 is a refinement map we have that $r_2(n) = \alpha$ and $r_2(n') = \beta$. Finally by the definition of the union of graphs of two functions, and condition 3., we have that $(r_1 \cup r_2)(n) = \alpha$ and $(r_1 \cup r_2)(n') = \beta$.

- By definition, $as_{i,\ i \in \{1,2\}}$ admits a derivation tree $t_i \in \mathcal{D}(\Gamma_i \vdash as_i \to \Gamma'_i)$ where:

  $\Gamma_i \supseteq \{\alpha \mid r_i(n) = \alpha, n \in \overline{N_i}_{c_i}, n \text{ is an input node}\}$, and
  $\Gamma'_i \supseteq \{\beta \mid r_i(n) = \beta, n \in \overline{N_i}_{c_i}, n \text{ is an output node}\}$.

  Recall that from $t_1 \in \mathcal{D}(\Gamma_1 \vdash as_1 \to \Gamma'_1)$ it follows that t_1 is a derivation tree whose root is equal to $\Gamma_1 \vdash as_1 \to \Gamma'_1$ which denotes the triple $(\Gamma_1, as_1, \Gamma'_1) \in \mathcal{E}$. Likewise, we have a similar result for t_2. Through the proof we use t_1 and t_2 as parts of larger derivation trees, and we use the root elements of these two derivation trees $(\Gamma_1, as_1, \Gamma'_1)$ and $(\Gamma_2, as_2, \Gamma'_2)$ when convenient.

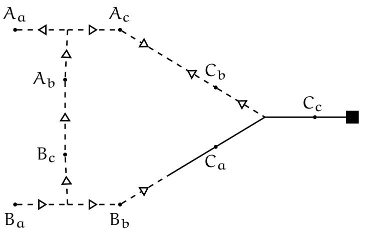


Figure 62: Colouring c_4 of the *Ordering* (Figure 61).

We have to show that $as_1 \ \cup \ as_2$ admits a derivation tree $t \in \mathcal{D}(\Gamma \vdash as_1 \ \cup \ as_2 \to \Gamma')$, where:

$$\Gamma \supseteq \left\{\gamma \mid (r_1 \cup r_2)(n) = \gamma, n \in \overline{N_1 \cup N_2}_{c_1 \cup c_2}, n \text{ is an input node}\right\} \text{ and}$$
$$\Gamma' \supseteq \left\{\gamma \mid (r_1 \cup r_2)(n) = \gamma, n \in \overline{N_1 \cup N_2}_{c_1 \cup c_2}, n \text{ is an output node}\right\}.$$

* In case $(\texttt{dom}(as_1) \cap \texttt{dom}(as_2) = \emptyset)$, we have $\Gamma = \Gamma_1 \cup \Gamma_2$, $\Gamma' = \Gamma_1' \cup \Gamma_2'$, and the following derivation tree:

$$\textit{par}\,\dfrac{t_1 \quad t_2}{\Gamma_1 \cup \Gamma_2 \vdash as_1 \ \cup \ as_2 \to \Gamma_1' \cup \Gamma_2'}$$

is in $\mathcal{D}(\Gamma \vdash as_1 \ \cup \ as_2 \to \Gamma')$ as required.

* In case $I = \texttt{dom}(as_1) \cap \texttt{dom}(as_2) \neq \emptyset$, we need to inspect I.

> Case $I \cap \Gamma_1 = \emptyset$, we have $\Gamma = \Gamma_1 \cup (\Gamma_2 \setminus I)$, $\Gamma' = (\Gamma_1' \setminus I) \cup \Gamma_2'$, and the following derivation tree:

$$\textit{seq}_1\,\dfrac{\textit{par}\,\dfrac{t_1 \qquad \textit{skip}\,\dfrac{}{\Gamma_2 \setminus I \vdash 0 \to \Gamma_2 \setminus I}}{\Gamma_1 \cup (\Gamma_2 \setminus I) \vdash as_1 \ \cup \ 0 \to \Gamma_1' \cup (\Gamma_2 \setminus I)} \qquad \textit{par}\,\dfrac{\textit{skip}\,\dfrac{}{\Gamma_1' \setminus I \vdash 0 \to \Gamma_1' \setminus I} \qquad t_2}{\Gamma_1' \cup (\Gamma_2 \setminus I) \vdash 0 \ \cup \ as_2 \to (\Gamma_1' \setminus I) \cup \Gamma_2'}}{\Gamma_1 \cup (\Gamma_2 \setminus I) \vdash as_1 \ \cup \ as_2 \to (\Gamma_1' \setminus I) \cup \Gamma_2'}$$

is in $\mathcal{D}(\Gamma \vdash as_1 \ \cup \ as_2 \to \Gamma')$ as required.

> Case $I \cap \Gamma_2 = \emptyset$ is similar to the previous cases, using the inference rule $\textit{seq}_2$ instead of $\textit{seq}_1$.

> Case $I \cap \Gamma_1 \neq \emptyset$ and $I \cap \Gamma_2 \neq \emptyset$. The colouring $c_1 \cup c_2$ has a causality loop, meaning that $(R_1 \cup R_2)^+$ is not anti-reflexive, violating 2. Therefore, this case cannot occur.

□

Example 6.2.13 (Ordering connector (revisit))**.** In this example we recall the *Ordering* connector of Example 6.1.3. We proceed calculating the $\mathcal{R}$eo animation specification that refines the colouring c_4 illustrated in Figure 61. The colouring c_4 with the nodes expanded into their respective merger and replicator representations is depicted in Figure 62. The $\mathcal{R}$eo animation specification that refines the colouring c_4 is given by the composition of the $\mathcal{R}$eo animation specifications:

- *FIFO*$_1(B_b, C_a)$ (Table 23e):

$$\langle(\{B_b \mapsto \text{‘-}\!\triangleright\!\text{-’}, C_a \mapsto \text{‘——’}\}, \emptyset),\ \gamma \blacktriangleright \alpha,\ \{C_a \mapsto \alpha\}\rangle;$$

- *Merger*(C_a, C_b, C_c) (Table 24b):

$$\langle(\{C_a \mapsto \text{‘——’},\ C_b \mapsto \text{‘-}\!\triangleleft\!\text{-’}, C_c \mapsto \text{‘——’}\}, \{(C_a, C_c)\}),\\ \alpha\,!\,\beta\ \cup\ \boxtimes\alpha,\ \{C_a \mapsto \alpha, C_c \mapsto \beta\}\rangle;$$

- I/O take(C_c) (Table 25a):

$$\langle(\{C_c \mapsto \text{‘——’}\}, \emptyset),\ \beta \blacktriangleright \delta\ \cup\ \boxtimes\delta,\ \{C_c \mapsto \beta\}\rangle;$$

Plus the respective no-dataflow entries from each of the colouring tables of the *write* I/O operation, the *LossySync* and the *SyncDrain*. The no-dataflow entries have the trivial animation specification 0 and the empty refinement map $\emptyset$. The $\mathcal{R}$eo animation specification that refines colouring c_4 is given by:

$$\langle(\{B_b \mapsto \text{‘-}\!\triangleright\!\text{-’}, C_a \mapsto \text{‘——’}, C_b \mapsto \text{‘-}\!\triangleleft\!\text{-’}, C_c \mapsto \text{‘——’}, \ldots\}, \{(C_a, C_c), \ldots\}),\\ \gamma \blacktriangleright \alpha\ \cup\ \alpha\,!\,\beta\ \cup\ \boxtimes\alpha\ \cup\ \beta \blacktriangleright \delta\ \cup\ \boxtimes\delta\ \cup\ 0,\ \{C_a \mapsto \alpha, C_c \mapsto \beta\}\rangle;$$

And it admits the derivation tree:

$$\mathit{seq}_1\,\dfrac{\dfrac{}{\{\gamma\} \vdash \gamma \blacktriangleright \alpha \to \{\alpha\}} \qquad \mathit{seq}_1\,\dfrac{\mathit{seq}_1\,\dfrac{\dfrac{}{\{\alpha\} \vdash \alpha\,!\,\beta \to \{\alpha, \beta\}} \qquad \dfrac{}{\{\alpha, \beta\} \vdash \boxtimes\alpha \to \{\beta\}}}{\{\alpha\} \vdash \alpha\,!\,\beta\ \cup\ \boxtimes\alpha \to \{\beta\}} \qquad t}{\alpha \vdash \alpha\,!\,\beta\ \cup\ \boxtimes\alpha\ \cup\ \beta \blacktriangleright \delta\ \cup\ \boxtimes\delta\ \cup\ 0 \to \emptyset}}{\{\gamma\} \vdash \gamma \blacktriangleright \alpha\ \cup\ \alpha\,!\,\beta\ \cup\ \boxtimes\alpha\ \cup\ \beta \blacktriangleright \delta\ \cup\ \boxtimes\delta\ \cup\ 0 \to \emptyset}$$

$$t = \mathit{seq}_1\,\dfrac{\dfrac{}{\{\beta\} \vdash \beta \blacktriangleright \delta \to \{\delta\}} \qquad \mathit{par}\,\dfrac{\dfrac{}{\{\delta\} \vdash \boxtimes\delta \to \emptyset} \qquad \dfrac{}{\emptyset \vdash 0 \to \emptyset}}{\{\delta\} \vdash \boxtimes\delta\ \cup\ 0 \to \emptyset}}{\{\beta\} \vdash \beta \blacktriangleright \delta\ \cup\ \boxtimes\delta\ \cup\ 0 \to \emptyset}$$

Composite connectors

For a non-primitive connector, similar to its colouring table, its animation table is obtained compositionally out of the animation tables of its constituents primitives. A *connector animator* is a model of a connector in which the dataflow behaviour prescribed by the colouring table and the animation table that refines that colouring table are both obtained compositionally out of the colouring tables and the animation tables of its primitive constituents.

Definition 6.2.14 (Connector animator)**.** A connector animator for a connector $C = (N, B, E, S, \eta)$ is defined by a tuple $(N, B, E, S, \zeta, \eta)$, where $\zeta = \{AT \mid T \in S, AT \sqsubseteq T\}$ is a set of animation tables which contains one animation table AT for each colouring table T in S.

6.2.3 *Animation Descriptions*

A $\mathcal{R}$eo animation specification refines the dataflow prescribed by a colouring into dataflow actions. Our next step is to simulate the dataflow behaviour by executing those dataflow actions to produce animations as the ones we described in the examples at the beginning of this chapter.

A derivation tree provides the ordering to execute (if possible in parallel) the dataflow actions. The execution of each action produces a frame. The idea is to visually represent each frame, and display the sequence of frames. A $\mathcal{R}$eo animation is the result of displaying the connector circuit, overlaid with its respective colouring, where the movement of the data tokens results from displaying at a certain rate the sequence of frames generated by the dataflow actions. The resulting visual experience simulates the dataflow through the connector. This process is very similar to a conventional animation that results from displaying a sequence of pictures at a certain rate.

To visually represent a frame, we overlay the graphical representation of the $\mathcal{R}$eo circuit with its colouring, and the visual elements introduced when discussing the examples in Section 6.1. However, the $\mathcal{R}$eo animation specifications do not provide all the information necessary to perform the overlay. Additionally, we use the information from the layout of the $\mathcal{R}$eo circuit that we obtain from the $\mathcal{R}$eo editor. Using the information from the layout of the $\mathcal{R}$eo circuit, we map $\mathcal{R}$eo animation specifications to *animation descriptions*. An animation description is a concrete executable list of instructions tailored for the dataflow behaviour of $\mathcal{R}$eo connectors. Animation descriptions are very similar to those used in standard animation languages, thus facilitating their compilation into a mainstream animation language, such as Flash©. Compared to animation specifications, animation descriptions introduce information that is specific to the layout of the connector circuit, and necessary to perform the overlay of the visual elements that we use for the dataflow animation. In animation descriptions each location has a coordinate attribute that indicates its position in the connector layout. The coordinate of a location can coincide, for example, with the position of a $\mathcal{R}$eo node, or a buffer cell, or an I/O operation in the connector layout. Tokens are represented as a triple $\langle \mathtt{id}, \mathtt{colour}, \mathtt{time} \rangle$ where $\mathtt{id}$ is the unique identifier of the token, $\mathtt{colour}$ is the colour to use to display the token, $\mathtt{time}$ indicates the time at which the token must leave its current location, either by moving or disappearing. The trajectories of the tokens that flow through the connector coincide with the geometries of its channels. Because the $\mathcal{R}$eo nodes in the animations are not expanded into their respective *Replicator* and *Merger* representations, multiple locations may have the same coordinate. For example a node that corresponds to a *Replicator* will be displayed in the layout as a circle with one incoming and two outgoing edges. The locations associated with the input node and the two output node of the *Replicator* will then have the same coordinate and coincide in the layout.

6.3 CONNECTOR ANIMATION IMPLEMENTATIONS

Currently, there are two implementations of Connector Animation: a Haskell implementation, by José Proença—*ReoFlash* [84], and an Eclipse Plugin implementation, by Christian Krause — part of the *Eclipse Coordination Tools (ECT)* [42].

6.3.1 *ReoFlash*

ReoFlash came to exist as a proof-of-concept implementation of Connector Animation. *ReoFlash* provides libraries to write Haskell definitions for $\mathcal{R}$eo circuits, colourings, and animation specifications. Given these definitions, *ReoFlash* derives the animation description automatically. The definitions for the most common primitive $\mathcal{R}$eo connectors are part of *ReoFlash*. To obtain animation descriptions of composite $\mathcal{R}$eo connectors that use already defined primitives, the only definition required is that of the composite $\mathcal{R}$eo circuit. The definition for the colourings and the animation specifications are obtained compositionally, and together with the given circuit definition, *ReoFlash* automatically generates the animation description definitions.

To visualise the animations, *ReoFlash* translates animation descriptions to the *script file format* [87]. The *script file format* is part of *SWFTools* [90]. *SWFTools* is a collection of utilities for working with Adobe© Flash© files (*SWF* files). The tool collection includes programs for reading *SWF* files, combining them, and creating them from other content (like images, sound files, videos or source code). Once in the *script file format* the animation descriptions are compiled into *SWF* files and ready to be visualised by a Flash Player.

ReoFlash is currently used to write circuit definitions of $\mathcal{R}$eo connectors that express relevant interaction protocols. These are collected on a website [84] that constitutes a repository of $\mathcal{R}$eo connectors and interaction patterns. The behaviour of each connector is explained by means of a textual description and accompanied by an animation obtained with *ReoFlash* that illustrates and simulates the dataflow behaviour of the connector.

6.3.2 *Connector Animation Eclipse Plugin*

The Eclipse Plugin implementation of connector animation is part of the *Eclipse Coordination Tools* (ECT) framework [42]. *ECT* is a framework for developing component-based software using $\mathcal{R}$eo. The framework consists of a set of integrated tools which are implemented as plugins for the Eclipse© platform [41]. *ECT* provides the functionality for design, verification and execution of component-based applications. $\mathcal{R}$eo is used to define the connectors that are responsible for expressing the interaction among the components that constitute the system.

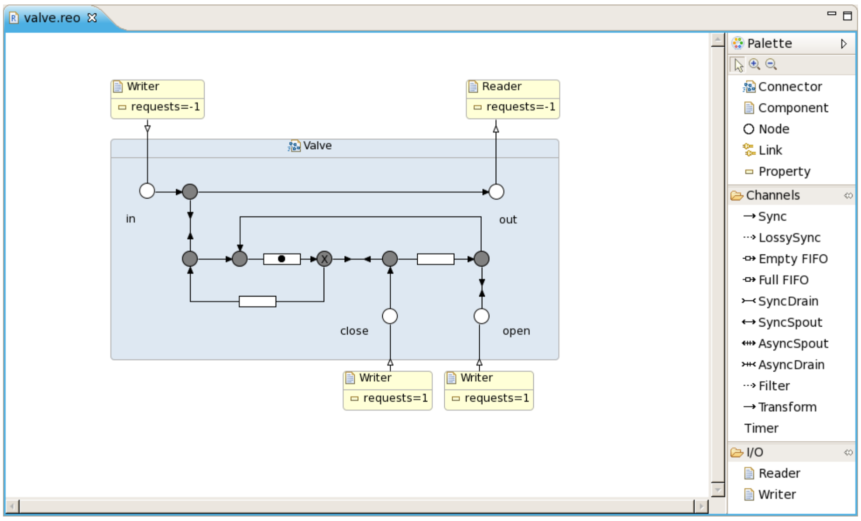


Figure 63: Screenshot of the graphical Reo editor. The main panel constitutes the canvas where circuits can be edited. The palette on the right-hand side contains a number of structural elements for constructing Reo circuits.

Reo editor

For specifying connectors and components, ECT includes a graphical Reo editor. Figure 63 shows a screenshot of the graphical Reo editor. Elements from the palette on the right-hand side can be dragged and dropped into the editor and constitute the building blocks for constructing connectors, describing components, and connecting components to connectors.

The palette has three sections. The first section contains the basic elements that are normally present in a component-based system:

- The element *Connector* serve as a container for nodes and channels.
- The element *Component* is a generic black-box components.
- The element *Node* correspond to Reo nodes.
- The element *Link* serves to connect components to source/sink nodes of connectors.
- The element *Property* can be assigned to components and used to describe a generic black-box component.

The second section in the palette contains a list of common Reo channels, like the ones described in Conlang. Channels can be dragged and dropped to connect *Node* elements.

The third and last section contains *Writer* and *Taker* (called *Reader*) elements, which are special instances of components.

The ℛeo editor plugin is associated with what in the Eclipse platform is called a *View*. A *View* is typically used to provide, navigate, or display information about the active selection in the active editor. In this case, the ℛeo editor provides a *View* called Animation. The Animation *View* uses the Connector Animation plugin to create animations of the connector selected in the editor panel. The Animation *View* listens to selection events in the editor. When the animation view is open and enabled, one can trigger the animation generation by selecting a connector or a component. Figure 64 contains a screenshot of the Animation *View*. The Animation *View* is an important asset for ℛeo connector developers, making their task simpler and less error prone, and their designs more reliable. When composing several basic connectors to build more complex connectors, interpreting the connector colouring models of the dataflow requires some level of expertise that cannot be expected from the developers. The Animation *View* provides a means to analyse the dataflow behaviour of the connector that is being built. The developer has a certain dataflow in mind for the resulting connector and as the connector is built, the dataflow behaviour can be analysed in an intuitive manner. Figure 64 shows the *Ordering* connector under construction and in the Animation *View* an animation of the dataflow behaviour of this connector is played. In the Animation *View* the possible next steps are displayed on the left under the heading *Next step* heading. The compositional semantics of Connector Animation is exploited to perform a gradual, incremental synthesis of the animation while the connector is being built.

Vereofy

Vereofy [95] is a formal verification tool for checking the operational correctness of component-based systems. *Vereofy* uses constraint automata as the formal semantics for the behaviour of components as well as for the ℛeo connectors. *Vereofy* supports linear and branching time model checking [63, 64] adapted to ℛeo and constraint automata. *Vereofy* can be used as a standalone model-checking tool or as an *ECT* Plugin that uses the Animation *View*. One of the key features of *Vereofy* is that if it finds a dataflow property that is violated, *Vereofy* provides a counterexample witnessing the violation. Traditionally, verification tools that provide counterexamples do so by providing a trace of the execution that violates the property being checked. Such traces are typically cumbersome to comprehend. In contrast, *Vereofy* uses the Animation *View* in *ECT* to provide a visualisation of the dataflow execution that violates the property being checked. Figure 65 shows a screenshot of the *Vereofy* plugin.

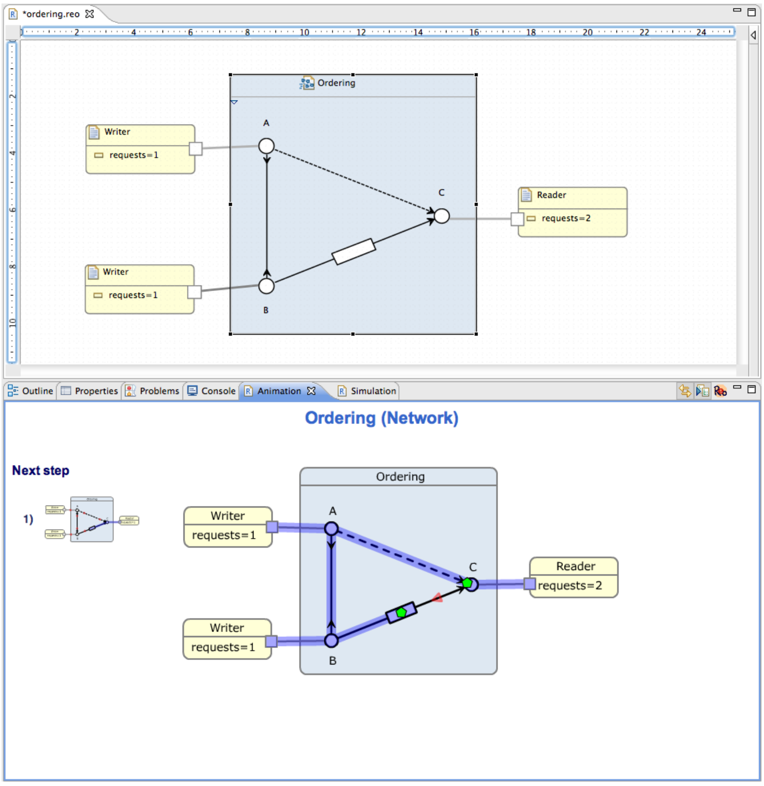
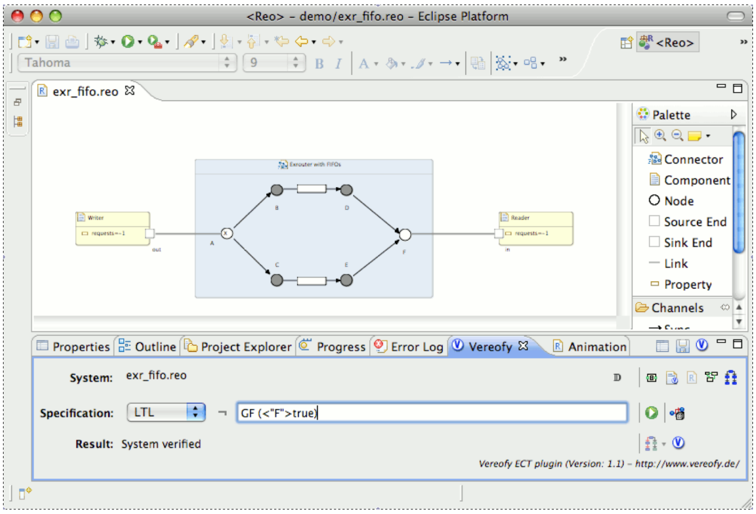


Figure 64: Screenshot of the ℛeo Editor (top) with the Animation *View* (bottom).



System: exr_fifo.reo

Specification: LTL ¬ GF (<"F">true)

Result: System verified

Vereofy ECT plugin (Version: 1.1) – http://www.vereofy.de/

Figure 65: Screenshot of the *Vereofy* plugin.

6.4 RELATED WORK

The importance of visual formalisms to model systems more intuitively yet accurately is witnessed by the vast amount of work on visual languages like Petri Nets, Sequence Charts and Statecharts [49, 50, 48]. Existing tools like Statemate [52] and Rhapsody [51] are able to generate running code for Statecharts models, and use simple animations to facilitate the process of model development, specification and analysis. Such animations use different colours to model different aspects in the statechart diagrams, which may evolve through time. Similarly, Connector Animation facilitates the implementation of tools that support developers in the design and analysis of $\mathcal{R}$eo connectors by means of animations. Motivated by the importance of representing real world animations, Harel et al. proposed an architecture where the executable model is separated from the animations, called Reactive Animations [53, 43]. They presented examples of the Rhapsody tool communicating with Flash animations, where the animations are based on the current state of the executable model. The Eclipse Coordination Tools suite uses the Reactive Animation principle, and its animations are generated and updated dynamically every time a connector is changed, but independently of the calculation of the semantics of the connector.

The benefits of using animations is also advocated by the workflow patterns community. Notably, van der Aalst et al. [94] use it to compare different workflow languages. To explain the differences between the workflow patterns, they have hand-programmed insightful Flash animations and offer them together with additional information on a website. The appearance of our animations is largely influenced by their work. Our approach to obtain the animations differs, however: we generate the animations automatically, and according to the formal semantics of our models, while they create the animations manually, according to the textual description of their intended workflow semantics.

Animation of formal models has proven to facilitate the communication between developers and stakeholders. H.T. Van et al. present animations for Goal-Oriented Requirements in [93]. They propose to animate UML state diagrams to visualise and simulate a specific model in a scenario very close to the real scenario in which the system is intended to be deployed. Multiple users can interact with an execution of the model, and property violations are monitored at animation time. Westeergaard and Lassen present the BRITNeY suite Animation tool [96], a tool to create visualisations of formal models, especially of coloured Petri nets (CPN), which is already integrated in the CPN Tools.

Our experience with Connector Animation is that it greatly facilitates the task of $\mathcal{R}$eo developers. Connector Animation instantaneously produces quality animations that comply with the formal semantics of $\mathcal{R}$eo, allowing $\mathcal{R}$eo developers to focus entirely on the design of new connectors. On the other hand, approaches as in [93, 73] force developers to construct both a model for the system and an additional model for its animation.

7

CONCLUSIONS

In this chapter we discuss and answer the research questions of the thesis, before summarising the more relevant directions for future research.

7.1 ANSWERS TO THE RESEARCH QUESTIONS

The research presented in this thesis builds upon a large body of existing knowledge from the field of coordination languages and models, and contributes to a prominent line of research followed in recent years, that of developing compositional models for component connectors. In particular, it contributes to the understanding of compositional models for context-dependent connectors. In Chapter 1 we explained how the models from 1) the coinductive calculus, and 2) the constraint automata are unable to capture context-dependent behaviour. We then formulated the following research questions:

i *How to introduce the context information reflecting the pending* I/O *operations in the model*?
ii *Can we express context-dependent behaviour and yet keep the model intuitive, transparent and compositional like models 1) and 2)*?
iii *Can we devise context-dependent models that can serve as implementation specifications, unlike the original informal semantics?*
iv *Having an answer for iii, can we refine implementation specifications into fully executable models that permit the automatic synthesis of efficient code in main stream programming languages*?

We now discuss and answer each of these questions according to the work presented in the bulk of the thesis.

ANSWERS TO QUESTION *i* We have two answers for question *i*. The connector colouring model, in Chapter 3, considers two extra primitives as part of the model, two abstract components: *Writer* and *Taker* which model general components that can perform I/O operations on a port. The 2-colouring and 3-colouring tables of these two components encode the context information concerning *pending* I/O operations. However, only the 3-colouring tables propagate the context information through composition and permit to capture the intended semantics of context-dependent connectors compositionally. The intentional automata model, in Chapter 4, labels the transitions of the automata with the information about the requests pending on the ports of the connector as well as the ports that fire as a result of taking the transition.

ANSWERS TO QUESTION *ii* The 3-colouring models and the intentional automata models capture context-dependent behaviour, and provide a means to reason about context-dependent connectors compositionally.

The colouring metaphor confers a degree of transparency to the model that makes the 3-colouring models easy to understand. The circuit representation overlaid with the dataflow colours provide insights into how each of the individual primitives of the circuit contributes to the overall dataflow in the circuit. We claim that the 3-colouring model is intuitive. Our claim is supported by the preference in using 3-colouring models to design ℛeo connectors in the Eclipse Coordination Tools [42].

The transitions of intentional automata models are intuitive and easy to understand. However, an automaton model for a connector tends to be large when compared with its 3-colouring model, or for example with its constraint automata model. That observation has led to the work on the ℛeo automata model described in Chapter 5.

The ℛeo automata models are a particular class of intentional automata models that we have identified by considering extra properties on the transitions. Those properties as we showed follow from the axiomatisation of the dataflow behaviour of ℛeo connector ports. An important observation about the ℛeo automata models is that they accommodate a more succinct representation than the intentional automata in general. Indeed, configuration tables are more succinct than the ℛeo automata, and we showed how configuration tables constitute a definition principle for ℛeo automata models. Furthermore, we defined operations on configuration table models that are faithful with respect to the operations on intentional automata. In simple terms, this means, that the compositional manipulation of connector models can be done using either the configuration table representation and the operations on configuration tables, or the ℛeo automaton representation and the operations on intentional automata. In the end, the resulting configuration table defines uniquely the resulting ℛeo automaton obtained, and vice versa.

Even through we have not formally shown how close configuration tables are to 3-colouring tables, we have given enough evidence to suggest that they are indeed *very similar*, thus we claim that the ℛeo automata models are intuitive and easy to comprehend as much as the 3-colouring models are. However, the 3-colouring models are in a way more informative because they contain the information about the topology of the circuit of the connector.

ANSWERS TO QUESTION *iii* In chapter 3 we presented and discussed algorithms that illustrate how the 3-colouring model can be used as a basis for the implementations of $\mathcal{R}$eo. The *Reolite* and Dreams implementations of $\mathcal{R}$eo support our claim that the 3 colouring-model can be used as a basis for (distributed) implementations. The *Reolite* non-distributed implementation supports several dataflow features in the language, including asynchrony, multi-party synchronisation, mutual exclusion, and propagation of context information. The Dreams framework uses the 3-colouring model to encode the dataflow behaviour of connectors. The colouring tables are encoded using propositional logic and at each execution step of the connector the overall dataflow behaviour is calculated according to the 3-colouring semantics, using SAT solving techniques. Our work, in Chapter 6 on the connector animation framework corroborates the idea that 3-colouring models can be used to formally obtain implementation specifications. Indeed, we show that 3-colouring models can be formally refined into animation specifications, and from animation specifications we obtain implementation specifications.

ANSWERS TO QUESTION *iv* In Chapter 6 we discussed how the connector animation framework provides a means to formally derive fully executable models. The animations obtained are compositional and therefore prove to be very interesting from an implementation point of view—we can reason about executable models modularly. The animation language is very simple and yields implementations that are lean and easy to compile using mainstream commercial programming languages, such as Java. The implementation of the connector animation framework in the Eclipse Coordination Tools support our claims.

7.2 DIRECTIONS FOR FUTURE WORK

We have identified two research topics, that we wish to pursue further in the future:

ALGEBRAIC PROPERTIES OF THE OPERATIONS ON INTENTIONAL AUTOMATA We would like to study the algebraic properties of the operations of intentional automata. We have limited our studies to the congruence result with respect to weak-equivalence. The study of other properties, for instance how the product and the hiding operation distribute, remains to be done. Those properties would be the bases for a classification of the intentional automata as a connector calculi. Questions about how this calculus compares with well established calculi could yield interesting results.

EVALUATE THE DIFFERENT MODELS FOR CONTEXT-DEPENDENT CONNECTORS In addition to the compositional models for component connectors proposed in this thesis, other models have been recently proposed [55, 57, 15, 25, 23], as well. Some comparison between the models is present in the literature, however a rigorous comparison of the different models is due in our opinion.

BIBLIOGRAPHY

[1] Luca Aceto, Anna Ingólfsdóttir, Kim G. Larsen, and Jiri Srba. *Reactive Systems: Modelling, Specification and Verification*. Cambridge University Press, 2007. (Cited on page 72.)

[2] Gul Agha. *Actors: a model of concurrent computation in distributed systems*. MIT Press, 1986. (Cited on page 66.)

[3] Kenneth Appel, Wolfgang Haken, and John Koch. Every planar map is four colorable. *Journal of Mathematics*, 21(0):439–567, December 1977. (Cited on page 69.)

[4] Farhad Arbab. The IWIM model for coordination of concurrent activities. In Paolo Ciancarini and Chris Hankin, editors, *COORDINATION*, volume 1061 of *LNCS*, pages 34–56. Springer, 1996. (Cited on page 5.)

[5] Farhad Arbab. What do you mean, coordination? In *Bulletin of the Dutch Association for Theoretical Computer Science*, NVTI, pages 11–22, 1998. (Cited on page 3.)

[6] Farhad Arbab. A Channel-Based Coordination Model For Component Composition. CWI report SEN-R 0203, CWI, 2002. (Cited on pages 5 and 14.)

[7] Farhad Arbab. Reo: a channel-based coordination model for component composition. *Mathematical Structures in Computer Science*, 14(3):329–366, 2004. (Cited on pages 5, 14, 20, 24, 25, 60, 62, 67, 85, 102, and 132.)

[8] Farhad Arbab. Abstract behavior types: a foundation model for components and their composition. *SCP*, 55:3–52, March 2005. (Cited on pages 23 and 67.)

[9] Farhad Arbab and Jan J. M. M. Rutten. A coinductive calculus of component connectors. In Martin Wirsing, Dirk Pattinson, and Rolf Hennicker, editors, *WADT*, volume 2755 of *LNCS*, pages 34–55. Springer, 2002. (Cited on pages 4, 5, 21, 36, and 67.)

[10] Farhad Arbab, Ivan Herman, and Per Spilling. MANIFOLD: Concepts and implementation. In Luc Bougé, Michel Cosnard, Yves Robert, and Denis Trystram, editors, *CONPAR*, volume 634 of *LNCS*, pages 793–794. Springer, 1992. (Cited on page 5.)

[11] Farhad Arbab, Ivan Herman, and Per Spilling. An overview of MANIFOLD and its implementation. *Concurrency - Practice and Experience*, 5(1):23–70, 1993. (Cited on page 5.)

[12] Farhad Arbab, Frank S. de Boer, Juan Guillen Scholten, and Marcello M. Bonsangue. MoCha: A Middleware Based on Mobile Channels. In *COMPSAC*, pages 667–673. IEEE, IEEE Computer Society, 2002. (Cited on pages 5, 58, 62, 66, and 67.)

[13] Farhad Arbab, Christel Baier, Jan J. M. M. Rutten, and Marjan Sirjani. Modeling Component Connectors in Reo by Constraint Automata (Extended Abstract). *ENTCS*, 97:25–46, 2004. (Cited on pages 6, 21, 23, 36, 67, 68, and 111.)

[14] Farhad Arbab, Christel Baier, Frank S. de Boer, and Jan J. M. M. Rutten. Models and temporal logical specifications for timed component connectors. *Software and System Modeling*, 6(1):59–82, 2007. (Cited on page 22.)

[15] Farhad Arbab, Roberto Bruni, Dave Clarke, Ivan Lanese, and Ugo Montanari. Tiles for Reo. In *Recent Trends in Algebraic Development Techniques*, volume 5486 of *LNCS*, pages 37–55. Springer, 2009. (Cited on pages 7, 68, and 161.)

[16] Farhad Arbab, Tom Chothia, Rob van der Mei, Sun Meng, Young-Joo Moon, and Chrétien Verhoef. From coordination to stochastic models of QoS. In Field and Vasconcelos [45], pages 268–287. (Cited on pages 22 and 71.)

[17] Christel Baier. Probabilistic models for Reo connector circuits. *J. UCS*, 11(10): 1718–1748, 2005. (Cited on page 22.)

[18] Christel Baier and Verena Wolf. Stochastic reasoning about channel-based component connectors. In Paolo Ciancarini and Herbert Wiklicky, editors, *COORDINATION*, volume 4038 of *LNCS*, pages 1–15. Springer, 2006. (Cited on page 22.)

[19] Christel Baier, Marjan Sirjani, Farhad Arbab, and Jan Rutten. Modeling component connectors in Reo by Constraint Automata. *SCP*, 61(2):75–113, 2006. (Cited on pages 4, 6, 80, and 95.)

[20] Christel Baier, Tobias Blechmann, Joachim Klein, and Sascha Klüppelholz. A uniform framework for modeling and verifying components and connectors. In Field and Vasconcelos [45], pages 247–267. (Cited on pages 6 and 22.)

[21] Richard Banach, Farhad Arbab, George A. Papadopoulos, and John R. W. Glauert. A multiply hierarchical automaton semantics for the IWIM coordination model. *J. UCS*, 9(1):2–33, 2003. (Cited on page 5.)

[22] Jean-Pierre Banâtre and Daniel Le Métayer. The gamma model and its discipline of programming. *SCP*, 15(1):55–77, 1990. (Cited on page 67.)

[23] Marco A. Barbosa and Luis S. Barbosa. A perspective on service orchestration. *SCP*, 74(9):671–687, 2009. (Cited on pages 7 and 161.)

[24] Gérard Berry. The foundations of Esterel. In *Proof, Language and Interaction: Essays in Honour of Robin Milner*. MIT Press, 2000. (Cited on page 46.)

[25] Marcello Bonsangue, Dave Clarke, and Alexandra Silva. Automata for context-dependent connectors. In Field and Vasconcelos [45], pages 184–203. (Cited on pages 7, 125, and 161.)

[26] Marcello M. Bonsangue, Farhad Arbab, Jaco W. de Bakker, Jan J. M. M. Rutten, Adriano Scutella, and Gianluigi Zavattaro. A transition system semantics for the control-driven coordination language MANIFOLD. *TCS*, 240(1):3–47, 2000. (Cited on page 67.)

[27] Robert Bringhurst. *The Elements of Typographic Style*. Version 2.5. Hartley & Marks, Publishers, 2002. (Cited on page 181.)

[28] J. Dean Brock and William B. Ackerman. Scenarios: A model of non-determinate computation. In *Proceedings of the International Colloquium on Formalization of Programming Concepts*, pages 252–259. Springer-Verlag, 1981. (Cited on page 6.)

[29] Antonio Brogi and Jean-Marie Jacquet. On the expressiveness of Linda-like concurrent languages. In *EXPRESS '98, Fifth International Workshop on Expressiveness in Concurrency (Satellite Workshop of CONCUR '98)*, volume 16, pages 75–96. ENTCS, 1998. (Cited on page 67.)

[30] Stephen D. Brookes, Charles A. R. Hoare, and Andrew W. Roscoe. A Theory of Communicating Sequential Processes. *J. ACM*, 31(3):560–599, 1984. (Cited on page 95.)

[31] Roberto Bruni, Ivan Lanese, and Ugo Montanari. Complete axioms for stateless connectors. In *Algebra and Coalgebra in Computer Science*, volume 3629 of *LNCS*, pages 98–113. Springer Berlin / Heidelberg, 2005. (Cited on page 4.)

[32] Roberto Bruni, Ivan Lanese, and Ugo Montanari. A basic algebra of stateless connectors. *TCS*, 366(1-2):98–120, 2006. (Cited on pages 4 and 68.)

[33] Dave Clarke. Reolite Implementation. `http://www.cwi.nl/~dave/reolite`, 2005. (Cited on page 60.)

[34] Dave Clarke, David Costa, and Farhad Arbab. Modelling coordination in biological systems. In Tiziana Margaria and Bernhard Steffen, editors, *ISoLA*, volume 4313 of *LNCS*, pages 9–25. Springer, 2004. (Cited on page 22.)

[35] Dave Clarke, David Costa, and Farhad Arbab. Connector Colouring I: Synchronization and Context Dependency. *ENTCS*, 154(1):101–119, 2006. (Cited on pages 4 and 23.)

[36] Dave Clarke, David Costa, and Farhad Arbab. Connector Colouring I: Synchronisation and Context Dependency. *SCP*, 66(3):205–225, 2007. (Cited on pages 4 and 23.)

[37] Dave Clarke, José M. P. Proença, Alexander Lazovik, and Farhad Arbab. Deconstructing Reo. In *FOCLASA*. Elsevier, 2008. (Cited on page 66.)

[38] Juan Carlos Cruz and Stéphane Ducasse. A group based approach for coordinating active objects. In Paolo Ciancarini and Alexander L. Wolf, editors, *COORDINATION*, volume 1594 of *LNCS*, pages 355–370. Springer, 1999. (Cited on page 67.)

[39] Robert de Simone and Annie Ressouche. Compositional semantics of ESTEREL and verification by compositional reductions. In David L. Dill, editor, *CAV*, volume 818 of *LNCS*, pages 441–454. Springer, 1994. (Cited on page 46.)

[40] Volker Diekert. *The Book of Traces*. World Scientific Publishing Co., Inc., 1995. (Cited on page 77.)

[41] Eclispe. Eclipse open development framework. `http://www.eclipse.org`, 2010. (Cited on page 152.)

[42] ECT. Eclipse Coordination Tools. `http://reo.project.cwi.nl/cgi-bin/trac.cgi/reo/wiki/Tools`, 2010. (Cited on pages 152 and 160.)

[43] Sol Efroni, David Harel, and Irun R. Cohen. Reactive Animation: Realistic modelling of complex dynamics systems. *Computer*, 38(1):38–47, 2005. (Cited on page 156.)

[44] Kees Everaars, David Costa, Nikolay Diakov, and Farhad Arbab. A distributed computational model for Reo. Tech. Report SEN-E0601, CWI, February 2006. (Cited on pages 5, 24, 58, 60, and 67.)

[45] John Field and Vasco Thudichum Vasconcelos, editors. *Coordination Models and Languages, 11th International Conference, COORDINATION 2009, Lisboa, Portugal, June 9-12, 2009. Proceedings*, volume 5521 of *LNCS*, 2009. Springer. (Cited on pages 164 and 165.)

[46] Fabio Gadducci and Ugo Montanari. The Tile Model. In *Proof, Language and Interaction: Essays in Honour of Robin Milner*, pages 133–165. MIT Press, 2000. (Cited on page 68.)

[47] David Gelernter and Nicholas Carriero. Coordination languages and their significance. *Commun. ACM*, 35(2):97–107, 1992. (Cited on page 1.)

[48] Radu Grosu, Gheorghe Stefanescu, and Manfred Broy. Visual formalisms revisited. In *CSD '98: Proceedings of the 1998 International Conference on Application of Concurrency to System Design*, page 41. IEEE Computer Society, 1998. (Cited on page 156.)

[49] David Harel. Statecharts: A visual formalism for complex systems. *SCP*, 8(3): 231–274, June 1987. (Cited on page 156.)

[50] David Harel. On visual formalisms. *Commun. ACM*, 31(5):514–530, 1988. (Cited on page 156.)

[51] David Harel and Hillel Kugler. The Rhapsody Semantics of Statecharts (or, On the Executable Core of the UML). In *Integration of Software Specification Techniques for Applications in Engineering*, LNCS, pages 325–354. Springer, 2004. (Cited on page 156.)

[52] David Harel and Amnon Naamad. The statemate semantics of statecharts. *ACM Trans. Softw. Eng. Methodol.*, 5(4):293–333, 1996. (Cited on page 156.)

[53] David Harel, Sol Efroni, and Irun R. Cohen. Reactive Animation. In *FMCO 2002*, volume 2852 of *LNCS*, pages 13–153. Springer Verlag, 2003. (Cited on page 156.)

[54] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., 2006. (Cited on page 91.)

[55] Mohammad Izadi and Marcello M. Bonsangue. Recasting constraint automata into büchi automata. In John S. Fitzgerald, Anne Elisabeth Haxthausen, and Hüsnü Yenigün, editors, *ICTAC*, volume 5160 of *LNCS*, pages 156–170. Springer, 2008. (Cited on pages 7 and 161.)

[56] Mohammad Izadi, Ali Movaghar, and Farhad Arbab. Model checking of component connectors. *Computer Software and Applications Conference, Annual International*, 1:673–675, 2007. (Cited on page 22.)

[57] Mohammad Izadi, Marcello M. Bonsangue, and Dave Clarke. Modeling component connectors: Synchronisation and context-dependency. In Antonio Cerone and Stefan Gruner, editors, *SEFM*, pages 303–312. IEEE Computer Society, 2008. (Cited on pages 7 and 161.)

[58] Kurt Jensen. *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use. Volume 1*, volume 1 of *Monographs in Theoretical Computer Science*. Springer-Verlag, 2 edition, 1997. (Cited on page 69.)

[59] Kurt Jensen. *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use. Volume 2*, volume 2 of *TCS*. Springer-Verlag, 2 edition, 1997. (Cited on page 69.)

[60] Gilles Kahn. Natural semantics. In Franz-Josef Brandenburg, Guy Vidal-Naquet, and Martin Wirsing, editors, *Proceedings of the Symposium on Theoretical Aspects of Computer Science (STACS)*, volume 247 of *LNCS*, pages 22–39. Springer-Verlag, 1987. (Cited on page 138.)

[61] Oscar Kanters. QoS analysis by simulation in Reo. Master thesis, Vrije Universiteit, March 2009. (Cited on page 22.)

[62] Stephanie Kemper. SAT-based verification for timed component connectors. *ENTCS*, 255:103–118, 2009. (Cited on page 22.)

[63] Sascha Klüppelholz and Christel Baier. Symbolic model checking for channel-based component connectors. *ENTCS*, 175(2):19–37, 2007. (Cited on pages 22 and 154.)

[64] Sascha Klüppelholz and Christel Baier. Alternating-time stream logic for multi-agent systems. In Doug Lea and Gianluigi Zavattaro, editors, *COORDINATION*, volume 5052 of *LNCS*, pages 184–198. Springer, 2008. (Cited on page 154.)

[65] Donald E. Knuth. Computer Programming as an Art. *Communications of the ACM*, 17(12):667–673, December 1974. (Cited on page 171.)

[66] Christian Koehler, Farhad Arbab, and Erik P. de Vink. Reconfiguring distributed Reo connectors. In Andrea Corradini and Ugo Montanari, editors, *WADT*, volume 5486, pages 221–235. Springer-Verlag, 2008. (Cited on page 22.)

[67] Christian Koehler, David Costa, José Proença, and Farhad Arbab. Reconfiguration of Reo connectors triggered by dataflow. *ECEASST*, 10, 2008. (Cited on page 22.)

[68] Christian Koehler, Alexander Lazovik, and Farhad Arbab. Connector rewriting with high-level replacement systems. *ENTCS*, 194(4):77–92, 2008. (Cited on page 22.)

[69] Ivan Lanese and Emilio Tuosto. Synchronized hyperedge replacement for heterogeneous systems. In Jean-Marie Jacquet and Gian Pietro Picco, editors, *COORDINATION*, volume 3454 of *LNCS*, pages 220–235. Springer, 2005. (Cited on page 68.)

[70] Edward A. Lee, Haiyang Zheng, and Ye Zhou. Causality interfaces and compositional causality analysis. Invited paper in Foundations of Interface Technologies (FIT), Satellite to CONCUR 2005, 2009. (Cited on page 46.)

[71] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufman Publishers, Inc., 1996. (Cited on page 60.)

[72] Nancy A. Lynch and Mark R. Tuttle. An introduction to input/output automata. *CWI Quarterly*, 2:219–246, 1989. (Cited on page 6.)

[73] Jeff Magee, Nat Pryce, Dimitra Giannakopoulou, and Jeff Kramer. Graphical animation of behavior models. In *International Conference on Software Engineering*, pages 499–508. IEEE Computer Society, 2000. (Cited on page 157.)

[74] Nikunj R. Mehta, Nenad Medvidovic, and Sandeep Phadke. Towards a taxonomy of software connectors. In Carlo Chairman-Ghezzi, Mehdi Chairman-Jazayeri, and Alexander L. Chairman-Wolf, editors, *ICSE*, pages 178–187. ACM, 2000. (Cited on page 67.)

[75] Sun Microsystems. Java programming language. `http://java.sun.com/`, 2010. (Cited on page 60.)

[76] Robin Milner. Calculi for Synchrony and Asynchrony. *TCS*, 25:267–310, 1983. (Cited on page 68.)

[77] Robin Milner. *Communication and Concurrency*. Prentice-Hall, Inc., 1989. (Cited on page 78.)

[78] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT, Aug 1990. (Cited on page 138.)

[79] Robert N. Moll, Michael A. Arbib, and Assaf J. Kfoury. *An introduction to formal language theory*. Springer-Verlag, 1988. (Cited on page 77.)

[80] George A. Papadopoulos and Farhad Arbab. Coordination of distributed and parallel activities in the IWIM model. *International Journal of High Speed Computing*, 9(2):127–160, 1997. (Cited on page 5.)

[81] George A. Papadopoulos and Farhad Arbab. Coordination. In *M. Zelkowitz (Ed.), The engineering of Large Systems*, volume 46 of *Advances in Computers*, pages 329–400. Academic Press, 1998. (Cited on page 67.)

[82] David Park. Concurrency and Automata on Infinite Sequences. In *Proceedings of the 5th GI-Conference on Theoretical Computer Science*, pages 167–183. Springer-Verlag, 1981. (Cited on page 78.)

[83] José Proença. Dreams webpage. `http://reo.project.cwi.nl/cgi-bin/trac.cgi/reo/wiki/Tools/DistributedReoEngine`, 2009. (Cited on page 66.)

[84] Reo Repository. Reo connectors repository. `http://homepages.cwi.nl/~proenca/webreo/home.htm`, 2010. (Cited on page 152.)

[85] Grzegorz Rozenberg and Arto Salomaa, editors. *Handbook of formal languages, vol. 1: word, language, grammar*. Springer-Verlag, 1997. (Cited on page 77.)

[86] Jan J. M. M. Rutten. Universal coalgebra: a theory of systems. *TCS*, 249(1):3–80, 2000. (Cited on page 5.)

[87] SC file format. Script file format .sc. `http://wiki.swftools.org/index.php/Sc_file_format`, 2009. (Cited on page 152.)

[88] Juan Guillen Scholten, Farhad Arbab, Frank S. de Boer, and Marcello M. Bonsangue. MoCha-pi, an exogenous coordination calculus based on mobile channels. In Hisham Haddad, Lorie M. Liebrock, Andrea Omicini, and Roger L. Wainwright, editors, *SAC*, pages 436–442. ACM, 2005. (Cited on page 5.)

[89] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. *MPI: The Complete Reference (Vol. 1)*, volume Volume 1 - The MPI Core of *Scientific and Engineering Computation*. The MIT Press, 1 edition, September 1998. (Cited on page 60.)

[90] SWFTools. SWFTools. `http://www.swftools.org`, 2010. (Cited on page 152.)

[91] Adobe Systems. Adobe flash. `http://www.adobe.com`, 2006. (Cited on page 127.)

[92] Robert Endre Tarjan. *Data structures and network algorithms*. Society for Industrial and Applied Mathematics, 1983. (Cited on page 58.)

[93] Hung T. Van, Axel van Lamsweerde, Philippe Massonet, and Christophe Ponsard. Goal-oriented requirements animation. In *RE*, pages 218–228, 2004. (Cited on page 157.)

[94] Wil M. P. van der Aalst, Alistair P. Barros, Arthur H. M. ter Hofstede, and Bartek Kiepuszewski. Advanced workflow patterns. In *CoopIS*, pages 18–29, 2000. (Cited on page 156.)

[95] Vereofy. Vereofy. `http://www.vereofy.de/`, 2010. (Cited on page 154.)

[96] Michael Westergaard and Kristian B. Lassen. The BRITNeY Suite Animation Tool. In Susanna Donatelli and P. S. Thiagarajan, editors, *ICATPN*, volume 4024 of *LNCS*, pages 431–440. Springer, 2006. (Cited on page 157.)

[97] Gary R. Wright and William R. Stevens. *TCP/IP Illustrated, Volume 2: The Implementation.* Computing Series. Addison-Wesley Professional, 1 edition, January 1995. (Cited on page 64.)

We have seen that computer programming is an art, because it applies accumulated knowledge to the world, because it requires skill and ingenuity, and especially because it produces objects of beauty.
— Donald E. Knuth [65]

SUMMARY

The corner stone of all engineering disciplines is the principle of constructing complex systems out of building blocks according to well defined rules. In the paradigm of hardware design, for example, complex systems are obtained by composing interconnected, inherently parallel components, which can represent transistors, logic gates, functional components such as adders, or architectural components such as a processor. In this thesis we study *computer systems* according to the same principle. We study computer systems and their construction according to a component-based paradigm. In particular, we consider paradigms for component-based systems that provide a clean conceptual separation between *computation* and *interaction*, and as a result favour the reusability of individual (heterogenous) components and the dynamic interchangeability of components.

The research presented in this thesis builds upon a large body of existing knowledge from the field of coordination languages and models. In this field components are classified into two distinct classes: components that provide systems-specific functionality, and components that provide systems-independent interaction protocols, called *connectors*.

The main contributions of this thesis advance a prominent line of research pursued in recent years: that of developing compositional models for component connectors. In particular, it contributes by proposing two new compositional models for context-dependent connectors: *connector colouring* and *intentional automata*. The behaviour of context-dependent connectors permits to express notions of *dataflow priority* and *dataflow blocking* which are traditionally hard to model compositionally. Connector colouring and intentional automata models capture the behaviour of context dependent connectors compositionally.

Additionally, this thesis contributes a simulation and animation framework for connectors, called *connector animation*. The dataflow behaviour of a connector is simulated by means of visual animations. The visual animations are based on the connector colouring semantics. We define a formal refinement map from the dataflow behaviour prescribed by a colouring into an animation specification. Our main result here is that the composition of colourings carries over to refinement maps, meaning that whenever two colourings compose, we can compose the animation specifications that refine each colouring and obtain an animation specification that refines the composed colouring.

SAMENVATTING

De hoeksteen van alle technische disciplines is het beginsel dat complexe systemen volgens wel-gedefinieerde regels worden opgebouwd uit bouwstenen. Bijvoorbeeld, in het kader van hardware design worden complexe systemen verkregen door aan elkaar verbonden en inherent parallelle componenten, zoals transistors, logische poorten, functionele componenten als adders en/of architectonische componenten als een processor, samen te voegen. In dit proefschrift nemen we hetzelfde beginsel als uitgangspunt om *computersystemen* te bestuderen. We bestuderen computersystemen en hun constructie op grond van een op component gebaseerd paradigma. We behandelen met name de paradigma's voor de component gebaseerde systemen die een zuiver conceptuele scheiding bieden tussen *computatie* en *interactie* en daardoor de herbruikbaarheid van individuele (heterogene) componenten en de dynamische uitwisselbaarheid van componenten bevoordelen.

Het onderzoek dat in dit proefschrift gepresenteerd wordt bouwt voort op een grote hoeveelheid reeds bestaande kennis op het gebied van coördinatie-talen en -modellen. In dit kader worden componenten in twee groepen onderverdeeld: componenten die systeemspecifieke functionaliteit verschaffen en componenten die systeemonafhankelijke interactie protocollen verschaffen, genaamd *connectors*.

De voornaamste bijdrages in dit proefschrift bevorderen de prominente lijn die de afgelopen jaren in de wetenschap werd nagestreefd: het ontwikkelen van compositionele modellen voor component connectors. Hierbij wordt in het bijzonder gewezen op de twee nieuwe compositionele modellen die worden voorgesteld voor de context afhankelijke connectors: *connector colouring* en *intentional automata*. Het gedrag van context afhankelijke connectors maakt het mogelijk de *dataflow priority* en *dataflow blocking* uit te drukken, iets dat doorgaans moeilijk is om compositioneel te modelleren. Connector colouring en intentional automata modellen leggen het gedrag van context afhankelijke connectors compositioneel vast.

Verder biedt dit proefschrift een simulatie- en animatieframework voor connectors, genaamd *connector animation*. Het dataflow gedrag van een connector wordt gesimuleerd door middel van visuele animatie. De visuele animaties zijn gebaseerd op de connector colouring semantiek. We definiëren een formele *refinement map* van het dataflow gedrag, van kleur-toekenningen naar animatie specificaties. Het voornaamste resultaat is dat de compositie van kleur-toekenningen goed overgedragen wordt naar de refinement maps. Dit betekent dat wanneer twee kleur-toekenningen samen te stellen zijn, de refinement van het resultaat ook te verkrijgen is door de refinement van beide originele kleur-toekenningen samen te stellen.

Titles in the IPA Dissertation Series since 2005

E. Ábrahám. *An Assertional Proof System for Multithreaded Java -Theory and Tool Support-* . Faculty of Mathematics and Natural Sciences, UL. 2005-01

R. Ruimerman. *Modeling and Remodeling in Bone Tissue*. Faculty of Biomedical Engineering, TU/e. 2005-02

C.N. Chong. *Experiments in Rights Control - Expression and Enforcement*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-03

H. Gao. *Design and Verification of Lock-free Parallel Algorithms*. Faculty of Mathematics and Computing Sciences, RUG. 2005-04

H.M.A. van Beek. *Specification and Analysis of Internet Applications*. Faculty of Mathematics and Computer Science, TU/e. 2005-05

M.T. Ionita. *Scenario-Based System Architecting - A Systematic Approach to Developing Future-Proof System Architectures*. Faculty of Mathematics and Computing Sciences, TU/e. 2005-06

G. Lenzini. *Integration of Analysis Techniques in Security and Fault-Tolerance*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-07

I. Kurtev. *Adaptability of Model Transformations*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-08

T. Wolle. *Computational Aspects of Treewidth - Lower Bounds and Network Reliability*. Faculty of Science, UU. 2005-09

O. Tveretina. *Decision Procedures for Equality Logic with Uninterpreted Functions*. Faculty of Mathematics and Computer Science, TU/e. 2005-10

A.M.L. Liekens. *Evolution of Finite Populations in Dynamic Environments*. Faculty of Biomedical Engineering, TU/e. 2005-11

J. Eggermont. *Data Mining using Genetic Programming: Classification and Symbolic Regression*. Faculty of Mathematics and Natural Sciences, UL. 2005-12

B.J. Heeren. *Top Quality Type Error Messages*. Faculty of Science, UU. 2005-13

G.F. Frehse. *Compositional Verification of Hybrid Systems using Simulation Relations*. Faculty of Science, Mathematics and Computer Science, RU. 2005-14

M.R. Mousavi. *Structuring Structural Operational Semantics*. Faculty of Mathematics and Computer Science, TU/e. 2005-15

A. Sokolova. *Coalgebraic Analysis of Probabilistic Systems*. Faculty of Mathematics and Computer Science, TU/e. 2005-16

T. Gelsema. *Effective Models for the Structure of pi-Calculus Processes with Replication*. Faculty of Mathematics and Natural Sciences, UL. 2005-17

P. Zoeteweij. *Composing Constraint Solvers*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2005-18

J.J. Vinju. *Analysis and Transformation of Source Code by Parsing and Rewriting*.

Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2005-19

M.Valero Espada. *Modal Abstraction and Replication of Processes with Data*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2005-20

A. Dijkstra. *Stepping through Haskell*. Faculty of Science, UU. 2005-21

Y.W. Law. *Key management and link-layer security of wireless sensor networks: energy-efficient attack and defense*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-22

E. Dolstra. *The Purely Functional Software Deployment Model*. Faculty of Science, UU. 2006-01

R.J. Corin. *Analysis Models for Security Protocols*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-02

P.R.A. Verbaan. *The Computational Complexity of Evolving Systems*. Faculty of Science, UU. 2006-03

K.L. Man and R.R.H. Schiffelers. *Formal Specification and Analysis of Hybrid Systems*. Faculty of Mathematics and Computer Science and Faculty of Mechanical Engineering, TU/e. 2006-04

M. Kyas. *Verifying OCL Specifications of UML Models: Tool Support and Compositionality*. Faculty of Mathematics and Natural Sciences, UL. 2006-05

M. Hendriks. *Model Checking Timed Automata - Techniques and Applications*. Faculty of Science, Mathematics and Computer Science, RU. 2006-06

J. Ketema. *Böhm-Like Trees for Rewriting*. Faculty of Sciences, VUA. 2006-07

C.-B. Breunesse. *On JML: topics in tool-assisted verification of JML programs*. Faculty of Science, Mathematics and Computer Science, RU. 2006-08

B. Markvoort. *Towards Hybrid Molecular Simulations*. Faculty of Biomedical Engineering, TU/e. 2006-09

S.G.R. Nijssen. *Mining Structured Data*. Faculty of Mathematics and Natural Sciences, UL. 2006-10

G. Russello. *Separation and Adaptation of Concerns in a Shared Data Space*. Faculty of Mathematics and Computer Science, TU/e. 2006-11

L. Cheung. *Reconciling Nondeterministic and Probabilistic Choices*. Faculty of Science, Mathematics and Computer Science, RU. 2006-12

B. Badban. *Verification techniques for Extensions of Equality Logic*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2006-13

A.J. Mooij. *Constructive formal methods and protocol standardization*. Faculty of Mathematics and Computer Science, TU/e. 2006-14

T. Krilavicius. *Hybrid Techniques for Hybrid Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-15

M.E. Warnier. *Language Based Security for Java and JML*. Faculty of Science, Mathematics and Computer Science, RU. 2006-16

V. Sundramoorthy. *At Home In Service Discovery*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-17

B. Gebremichael. *Expressivity of Timed Automata Models*. Faculty of Science, Mathematics and Computer Science, RU. 2006-18

L.C.M. van Gool. *Formalising Interface Specifications*. Faculty of Mathematics and Computer Science, TU/e. 2006-19

C.J.F. Cremers. *Scyther - Semantics and Verification of Security Protocols*. Faculty of Mathematics and Computer Science, TU/e. 2006-20

J.V. Guillen Scholten. *Mobile Channels for Exogenous Coordination of Distributed Systems: Semantics, Implementation and Composition*. Faculty of Mathematics and Natural Sciences, UL. 2006-21

H.A. de Jong. *Flexible Heterogeneous Software Systems*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2007-01

N.K. Kavaldjiev. *A run-time reconfigurable Network-on-Chip for streaming DSP applications*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-02

M. van Veelen. *Considerations on Modeling for Early Detection of Abnormalities in Locally Autonomous Distributed Systems*. Faculty of Mathematics and Computing Sciences, RUG. 2007-03

T.D. Vu. *Semantics and Applications of Process and Program Algebra*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2007-04

L. Brandán Briones. *Theories for Model-based Testing: Real-time and Coverage*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-05

I. Loeb. *Natural Deduction: Sharing by Presentation*. Faculty of Science, Mathematics and Computer Science, RU. 2007-06

M.W.A. Streppel. *Multifunctional Geometric Data Structures*. Faculty of Mathematics and Computer Science, TU/e. 2007-07

N. Trčka. *Silent Steps in Transition Systems and Markov Chains*. Faculty of Mathematics and Computer Science, TU/e. 2007-08

R. Brinkman. *Searching in encrypted data*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-09

A. van Weelden. *Putting types to good use*. Faculty of Science, Mathematics and Computer Science, RU. 2007-10

J.A.R. Noppen. *Imperfect Information in Software Development Processes*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-11

R. Boumen. *Integration and Test plans for Complex Manufacturing Systems*. Faculty of Mechanical Engineering, TU/e. 2007-12

A.J. Wijs. *What to do Next?: Analysing and Optimising System Behaviour in Time*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2007-13

C.F.J. Lange. *Assessing and Improving the Quality of Modeling: A Series of Empirical*

Studies about the UML. Faculty of Mathematics and Computer Science, TU/e. 2007-14

T. van der Storm. *Component-based Configuration, Integration and Delivery*. Faculty of Natural Sciences, Mathematics, and Computer Science,UvA. 2007-15

B.S. Graaf. *Model-Driven Evolution of Software Architectures*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2007-16

A.H.J. Mathijssen. *Logical Calculi for Reasoning with Binding*. Faculty of Mathematics and Computer Science, TU/e. 2007-17

D. Jarnikov. *QoS framework for Video Streaming in Home Networks*. Faculty of Mathematics and Computer Science, TU/e. 2007-18

M. A. Abam. *New Data Structures and Algorithms for Mobile Data*. Faculty of Mathematics and Computer Science, TU/e. 2007-19

W. Pieters. *La Volonté Machinale: Understanding the Electronic Voting Controversy*. Faculty of Science, Mathematics and Computer Science, RU. 2008-01

A.L. de Groot. *Practical Automaton Proofs in PVS*. Faculty of Science, Mathematics and Computer Science, RU. 2008-02

M. Bruntink. *Renovation of Idiomatic Crosscutting Concerns in Embedded Systems*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2008-03

A.M. Marin. *An Integrated System to Manage Crosscutting Concerns in Source Code*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2008-04

N.C.W.M. Braspenning. *Model-based Integration and Testing of High-tech Multidisciplinary Systems*. Faculty of Mechanical Engineering, TU/e. 2008-05

M. Bravenboer. *Exercises in Free Syntax: Syntax Definition, Parsing, and Assimilation of Language Conglomerates*. Faculty of Science, UU. 2008-06

M. Torabi Dashti. *Keeping Fairness Alive: Design and Formal Verification of Optimistic Fair Exchange Protocols*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2008-07

I.S.M. de Jong. *Integration and Test Strategies for Complex Manufacturing Machines*. Faculty of Mechanical Engineering, TU/e. 2008-08

I. Hasuo. *Tracing Anonymity with Coalgebras*. Faculty of Science, Mathematics and Computer Science, RU. 2008-09

L.G.W.A. Cleophas. *Tree Algorithms: Two Taxonomies and a Toolkit*. Faculty of Mathematics and Computer Science, TU/e. 2008-10

I.S. Zapreev. *Model Checking Markov Chains: Techniques and Tools*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-11

M. Farshi. *A Theoretical and Experimental Study of Geometric Networks*. Faculty of Mathematics and Computer Science, TU/e. 2008-12

G. Gulesir. *Evolvable Behavior Specifications Using Context-Sensitive Wildcards*.

Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-13

F.D. Garcia. *Formal and Computational Cryptography: Protocols, Hashes and Commitments*. Faculty of Science, Mathematics and Computer Science, RU. 2008-14

P. E. A. Dürr. *Resource-based Verification for Robust Composition of Aspects*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-15

E.M. Bortnik. *Formal Methods in Support of SMC Design*. Faculty of Mechanical Engineering, TU/e. 2008-16

R.H. Mak. *Design and Performance Analysis of Data-Independent Stream Processing Systems*. Faculty of Mathematics and Computer Science, TU/e. 2008-17

M. van der Horst. *Scalable Block Processing Algorithms*. Faculty of Mathematics and Computer Science, TU/e. 2008-18

C.M. Gray. *Algorithms for Fat Objects: Decompositions and Applications*. Faculty of Mathematics and Computer Science, TU/e. 2008-19

J.R. Calamé. *Testing Reactive Systems with Data - Enumerative Methods and Constraint Solving*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-20

E. Mumford. *Drawing Graphs for Cartographic Applications*. Faculty of Mathematics and Computer Science, TU/e. 2008-21

E.H. de Graaf. *Mining Semi-structured Data, Theoretical and Experimental Aspects of Pattern Evaluation*. Faculty of Mathematics and Natural Sciences, UL. 2008-22

R. Brijder. *Models of Natural Computation: Gene Assembly and Membrane Systems*. Faculty of Mathematics and Natural Sciences, UL. 2008-23

A. Koprowski. *Termination of Rewriting and Its Certification*. Faculty of Mathematics and Computer Science, TU/e. 2008-24

U. Khadim. *Process Algebras for Hybrid Systems: Comparison and Development*. Faculty of Mathematics and Computer Science, TU/e. 2008-25

J. Markovski. *Real and Stochastic Time in Process Algebras for Performance Evaluation*. Faculty of Mathematics and Computer Science, TU/e. 2008-26

H. Kastenberg. *Graph-Based Software Specification and Verification*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-27

I.R. Buhan. *Cryptographic Keys from Noisy Data Theory and Applications*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-28

R.S. Marin-Perianu. *Wireless Sensor Networks in Motion: Clustering Algorithms for Service Discovery and Provisioning*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-29

M.H.G. Verhoef. *Modeling and Validating Distributed Embedded Real-Time Control Systems*. Faculty of Science, Mathematics and Computer Science, RU. 2009-01

M. de Mol. *Reasoning about Functional Programs: Sparkle, a proof assistant for*

Clean. Faculty of Science, Mathematics and Computer Science, RU. 2009-02

M. Lormans. *Managing Requirements Evolution*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2009-03

M.P.W.J. van Osch. *Automated Model-based Testing of Hybrid Systems*. Faculty of Mathematics and Computer Science, TU/e. 2009-04

H. Sozer. *Architecting Fault-Tolerant Software Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-05

M.J. van Weerdenburg. *Efficient Rewriting Techniques*. Faculty of Mathematics and Computer Science, TU/e. 2009-06

H.H. Hansen. *Coalgebraic Modelling: Applications in Automata Theory and Modal Logic*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2009-07

A. Mesbah. *Analysis and Testing of Ajax-based Single-page Web Applications*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2009-08

A.L. Rodriguez Yakushev. *Towards Getting Generic Programming Ready for Prime Time*. Faculty of Science, UU. 2009-9

K.R. Olmos Joffré. *Strategies for Context Sensitive Program Transformation*. Faculty of Science, UU. 2009-10

J.A.G.M. van den Berg. *Reasoning about Java programs in PVS using JML*. Faculty of Science, Mathematics and Computer Science, RU. 2009-11

M.G. Khatib. *MEMS-Based Storage Devices. Integration in Energy-Constrained Mobile Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-12

S.G.M. Cornelissen. *Evaluating Dynamic Analysis Techniques for Program Comprehension*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2009-13

D. Bolzoni. *Revisiting Anomaly-based Network Intrusion Detection Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-14

H.L. Jonker. *Security Matters: Privacy in Voting and Fairness in Digital Exchange*. Faculty of Mathematics and Computer Science, TU/e. 2009-15

M.R. Czenko. *TuLiP - Reshaping Trust Management*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-16

T. Chen. *Clocks, Dice and Processes*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2009-17

C. Kaliszyk. *Correctness and Availability: Building Computer Algebra on top of Proof Assistants and making Proof Assistants available over the Web*. Faculty of Science, Mathematics and Computer Science, RU. 2009-18

R.S.S. O'Connor. *Incompleteness & Completeness: Formalizing Logic and Analysis in Type Theory*. Faculty of Science, Mathematics and Computer Science, RU. 2009-19

B. Ploeger. *Improved Verification Methods for Concurrent Systems*. Faculty of Mathematics and Computer Science, TU/e. 2009-20

T. Han. *Diagnosis, Synthesis and Analysis of Probabilistic Models*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-21

R. Li. *Mixed-Integer Evolution Strategies for Parameter Optimization and Their Applications to Medical Image Analysis*. Faculty of Mathematics and Natural Sciences, UL. 2009-22

J.H.P. Kwisthout. *The Computational Complexity of Probabilistic Networks*. Faculty of Science, UU. 2009-23

T.K. Cocx. *Algorithmic Tools for Data-Oriented Law Enforcement*. Faculty of Mathematics and Natural Sciences, UL. 2009-24

A.I. Baars. *Embedded Compilers*. Faculty of Science, UU. 2009-25

M.A.C. Dekker. *Flexible Access Control for Dynamic Collaborative Environments*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-26

J.F.J. Laros. *Metrics and Visualisation for Crime Analysis and Genomics*. Faculty of Mathematics and Natural Sciences, UL. 2009-27

C.J. Boogerd. *Focusing Automatic Code Inspections*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2010-01

M.R. Neuhäußer. *Model Checking Nondeterministic and Randomly Timed Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2010-02

J. Endrullis. *Termination and Productivity*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2010-03

T. Staijen. *Graph-Based Specification and Verification for Aspect-Oriented Languages*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2010-04

Y. Wang. *Epistemic Modelling and Protocol Dynamics*. Faculty of Science, UvA. 2010-05

J.K. Berendsen. *Abstraction, Prices and Probability in Model Checking Timed Automata*. Faculty of Science, Mathematics and Computer Science, RU. 2010-06

A. Nugroho. *The Effects of UML Modeling on the Quality of Software*. Faculty of Mathematics and Natural Sciences, UL. 2010-07

A. Silva. *Kleene Coalgebra*. Faculty of Science, Mathematics and Computer Science, RU. 2010-08

J.S. de Bruin. *Service-Oriented Discovery of Knowledge - Foundations, Implementations and Applications*. Faculty of Mathematics and Natural Sciences, UL. 2010-09

D. Costa. *Formal Models for Component Connectors*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2010-10

COLOPHON

This thesis was typeset with $\LaTeX\,2_\varepsilon$ using Hermann Zapf's *Palatino* and *Euler* type faces (Type 1 PostScript fonts *URW Palladio L* and *FPL* were used).
The typographic style was inspired by Bringhurst's genius as presented in *The Elements of Typographic Style* [27]. It is available for LaTeX via CTAN as "`classicthesis`".
The cover was designed using Adobe Illustrator CS4.