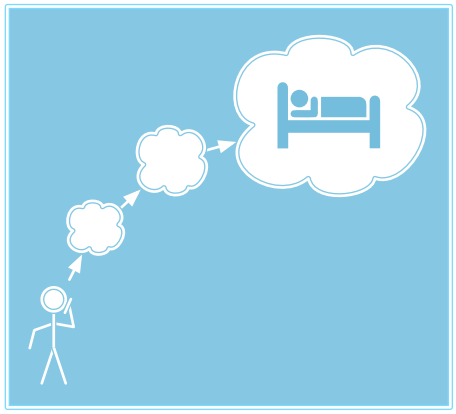
Synchronous coordination of distributed components

Proença, J.

Synchronous Coordination of Distributed Components



José Proença

Synchronous Coordination of Distributed Components

Synchronous Coordination of Distributed Components

PROEFSCHRIFT

ter verkrijging van
de graad van Doctor aan de Universiteit Leiden
op gezag van de Rector Magnificus prof. mr. P.F. van der Heijden,
volgens besluit van het College voor Promoties
te verdigen op woensdag 11 mei 2011
klokke 13.45 uur

door

José Miguel Paiva Proença

geboren te Porto, Portugal, in 1982.

promotor:	prof.dr. F. Arbab
copromotor:	dr. D. Clarke, *Katholieke Universiteit Leuven, Belgium*
copromotor:	dr. E. de Vink, *Technische Universiteit Eindhoven*
overige leden:	prof.dr. T. Bäck
	prof.dr. F.S. de Boer
	dr. M.M. Bonsangue
	prof.dr. J.-M. Jacquet, *University of Namur, Belgium*
	prof.dr. J.N. Kok
	prof.dr. J.J.M.M. Rutten, *Radboud University Nijmegen*

The work in this thesis was supported by the portuguese foundation FCT (Funcação para a Ciência e Tecnologia), grant 22485 – 2005, and has been carried out under the auspices of the research school IPA (Institute for Programming research and Algorithmics).

Cover design by Kseniya Rogova.
Printed and published by Boxpress BV || Proefschriftmaken.nl.

ISBN: 978-90-8891-265-8
IPA Dissertation Series 2011-05

Acknowledgments

This thesis is the result of four years of work carried out at CWI, and completed while employed at the Katholieke Universteit Leuven, in Belgium. During this time I faced many obstacles, which I was able to surpass only with the support of all my friends and family. To all of you, a warm "thank you"!

I would like to thank my advisors: Farhad Arbab, Dave Clarke, and Erik de Vink, for their friendship during these years. Farhad always found time in his busy schedule to discuss any topic, and gave the best and wisest personal advises. Furthermore, he is an incredible storyteller and prepares delicious Iranian food. Dave is a good friend and a very creative person, with whom I enjoyed many conversations, and who later offered me my current position as a researcher in Leuven. I specially enjoyed our few mountain biking trips and our dinners at Domus. Erik is a patient and dedicated person, who helped me with the balance between private and working life. I got acquainted with the great research environment provided by CWI through David, also a PhD student by that time, and Luís Barbosa, a brilliant writer and researcher at the University of Minho. David introduced me to Farhad Arbab and Jan Rutten. I am grateful to them, among many others, for their personal commitment and enthusiasm that led me into this fantastic four years journey.

David and Stephanie, both members of SEN3 and paranymphs of my defense, deserve a special gratitude. David is a good friend since my undergraduate times, with whom I shared my office and who deserves my trust and admiration. Not only did we collaborate in the animation work, included in this thesis, but we also shared amazing moments, such as our trips to Cambridge, Switzerland, and Croatia. Stephanie started her PhD at around the same time as I did. She showed me how to appreciate German precision and organisation. I recall with great pleasure our waterski adventures, the fun game sessions that she organised, and the challenging mathematical puzzles that she knows.

I am also thankful for all the moments and discussions with the remaining

members of SEN3, who made my days at work both interesting and exciting. A big "thank you" to Alexander, Alexandra, Behnaz, Christian, Clemens, Delphine, Filippo, Frank, Helle, Helen, Immo, Jan, Lara, Mahdi, Marcello, Milad, Natallia, Sun Meng, Tom, Young-Joo, and Ziyan. I want to emphasise the particular good time I had when *killing monsters* with Immo, eating Korean food with Young-Joo, camping with Lara and Behnaz, swimming in Budapest's thermal water with Christian, and climbing with Natallia. All these episodes gave a sweet touch to my PhD student life. I also had the great opportunity to attend several IPA events, mainly organised by Tijn Borghuis, where I met Paul, Rena, Jörg, José Pedro Magalhães, Sonja, Mark, among others. Furthermore, I also want to thank Einar Johnsen and Martin Steffen for the short time I spent in Oslo, and for all the discussions we had in different places around the globe.

Within CWI I met incredible people, with whom I spent moments that I will never forget. These include Bas, Bikkie, Christian and Sonja, Chrétien, Domenico, Eike and Marta, Enav and Sigal (and Shai), Fernando, Hanna, Ishan, Jana and Petr, Jurjen, Katja, Krzysztof, Li Chao, Lorenzo, Marjan, Mathieu, Rodrigo, Rómulo, Susanne, Yanjing, and Yunus. Life in Amsterdam was not centred only around research. I would also like to thank for the great time spent with other new friendships made during my stay in Amsterdam. A big thanks to Daniel and Stephanie (and Julian), Peter and Wendy, Nikolay, Ronny and Rocio, Dagmar, Sunny, Nils, Nynke, Ovideo, Jin, Joost and Dina, Tiago and Andreia, Miguel and Claudia, Zé Pedro Correia, Ivo, Levi, Luciano, and Jorge Laranjo. I recall the long lazy afternoons at the IJ brewery, enchanting tango salons, energetic salsa evenings, exciting climbing sessions, numerous home parties, interesting house moves, and revitalising trips away from Amsterdam.

Before coming to Amsterdam I lived in Braga, Portugal, and studied at the University of Minho. There I met a group of great researchers who introduced me to the scientific world, and to whom I owe part of my interest in research. In particular, I wish to thank Jorge Sousa Pinto, Alcino Cunha, José Nuno Oliveira, Luís Barbosa, Manuel Barbosa, João Saraiva, Luís Pinto, José João Almeida, Pedro Henriques, and José Manuel Valença, among others. I am also immensely grateful to all my friends back in Braga, who make me feel at home every time I visit my cosy country, including Jácome and Carina, Tércio e Marta, Gonçalo, Zé Marques, João Paulo, Paulo Silva, Nuno, Daniel, Ana Ferreira, Miguel and Ricardo Vilaça, and many, many others. They all contributed to my mental sanity ever since I met them.

Living for one year in Leuven, Belgium, I also want to thank all those who played a special role in my daily life, such as Dave, Ilya, Radu, and Dimiter. In particular, I want to thank Kseniya. She gave me all the support and attention I needed, and more importantly, she showed me how beautiful life can be. For that and so much more, a tender thankful *potzelui*!

Last, but not the least, I wish to thank the dearest and most important people to me: my beloved family. You have encouraged and supported me in all possible and imaginative ways. I miss you all... A huge "thank you" to *minha Mãe* Fernanda, to *meu Pai* Alberto, to *minha maninha* Miana, to *minhas queridas avós* Teresinha and Ruth, to *minha tia* São, to *meu tio* Paulo, to *meus priminhos* Ricardo, Pedro, and Rita, and to *minha tiazinha* Sensa.

Leuven José Proença
March, 2011.

Contents

Chapter 1

Introduction

Coordination is a relatively recent field, considerably inspired by concurrency theory [83, 85, 91]. Coordination languages and models [90] are based on the philosophy that an application or a system should be divided into the parts that perform computations, typically components or services, and the parts that coordinate the results and resources required to perform the computations. The coordination aspect focuses on the latter, describing how the components or services are connected. In this thesis we study a specific class of coordination models, namely *synchronous*, *exogenous*, and *composable* models, and we exploit implementation techniques for such models in distributed environments. Our work concentrates on the $\mathcal{R}$eo coordination model [8] as the main representative of this class of coordination models.

We motivate our work using a simple example of a web-based system, depicted in Figure 1.1. In this example a phone- and an Internet-based service can be used to book two distinct hotels. The layer performing coordination connects these services, represented by the mesh of blue clouds, and describes when each service is allowed to communicate and how the data should be transferred. A

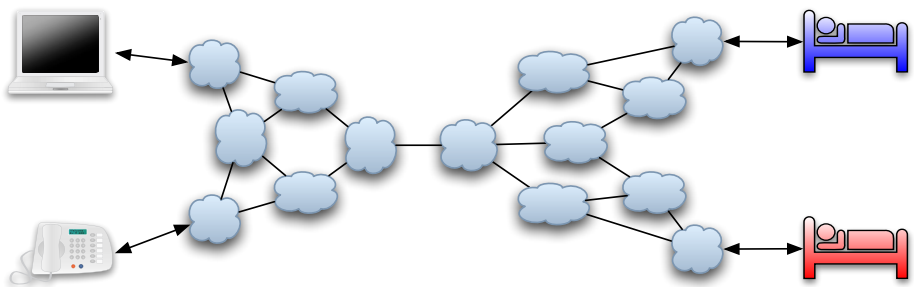


Figure 1.1: Decentralised coordination of web services.

possible incarnation of the coordination layer is to allow either either the phone- or the Internet-based services to book a hotel at any given time, giving priority to the phone when both services try to book a hotel simultaneously, and to alternate the hotel that is booked. Many other alternative behaviours exist for our booking example.

In this thesis we target coordination models that are: *(i) synchronous* in the sense that the communication between elements is performed atomically in a per-round fashion, *(ii) exogenous* because the components or services are not aware of the coordination, which is described orthogonally, and *(iii) composable* since the behaviour of the coordination layer can be completely described from its composition of smaller building blocks. In our booking scenario, synchrony allows us to express, for example, that a request for booking a hotel using the phone-based service can only be sent if one of the hotels can receive this request. In Figure 1.1 we emphasise the composability aspect using multiple clouds, each providing its own contribution for the global behaviour. Furthermore, the existence of edges between the clouds reflects our choice of a concurrent model for execution, where each cloud is regarded as an independent thread of computation that communicates only with its connected clouds.

In this thesis the $\mathcal{R}$eo coordination model is studied in detail. $\mathcal{R}$eo is a channel-based coordination language with a graphical notation, introduced by Arbab in 2001 [7], wherein complex connectors are built out of a simple set of primitive connectors. Compliant with our target, $\mathcal{R}$eo is synchronous, exogenous, and composable, yielding an expressive and intuitive coordination model. The $\mathcal{R}$eo model is currently being used to specify coordination patterns and concurrent systems. More specifically, $\mathcal{R}$eo has been used in a variety of different areas, such as systems biology [36], service oriented computing [82], mashups [79], business process modelling [17, 103], model driven development [20], and multi-agent systems [12]. $\mathcal{R}$eo has also been extended to include, for example, timed behaviour [13], probabilistic and stochastic models [21, 23, 88], quality of service [15], resource bounds [81], and reconfiguration [35, 72, 71, 77, 76]. Several tools have been developed to edit, verify, simulate, and execute $\mathcal{R}$eo systems [16, 41, 75, 37]. However, little effort has been spent so far on its (distributed) *implementation* aspects.

Current engines that execute $\mathcal{R}$eo [37, 16] allow the coordination layer to run only in a single thread of execution, although the components can execute in parallel or on a distributed platform. Furthermore, due to the synchrony aspect these engines only support small systems, and do not scale. To address these limitations, our approach to implement $\mathcal{R}$eo-like models makes a tradeoff between precompiling the possible behaviour and calculating it at runtime. Furthermore, we exploit the fact that different parts of a connector can execute concurrently, and identify parts of the system that can execute independently of each other.

Implementations of most models of concurrency and coordination typically involve synchronisation constructs. These constructs are either explicit, allowing the users of a model to specify their own tailor-made synchrony, or implicit. This thesis supports the development of implementations that provide implicit synchronisation constructs, as well as the ones that, like $\mathcal{R}$eo, support user-defined synchrony. Our work contributes to the field of coordination, in particular to $\mathcal{R}$eo, by improving existing approaches to execute synchronisation models in three major ways:

1. by supporting *decoupled execution* and lightweight *reconfiguration*;
2. by increasing *performance* using *constraint satisfaction* techniques; and
3. by improving *scalability* by identifying *synchronous regions*.

We explain each of these three contributions in more detail below. Throughout this thesis we support our statements both in theory and in practice. We give formal arguments that show the correctness of our approach with respect to existing models, and present tools and benchmarks that confirm our claims.

Decoupled execution and lightweight reconfiguration

In this thesis we present the Dreams framework, a distributed framework for compositional synchronous coordination models. The Dreams framework is based on the *actor model* [1] and creates an actor for each building block of the coordination model. Each actor consists of a concurrent thread of execution that communicates asynchronously with other actors. We introduce a distributed protocol that allows actors to reach consensus about data exchange, and performs the actual communication of data. We developed a prototype Dreams engine to test this protocol, using an actor library for the Scala language [57].

Reconfiguring an instance of a coordination pattern consists of changing some of its parts. The Dreams framework assumes not only that the underlying coordination model is compositional but also that it evolves in a stepwise manner. The stepwise development combined with the decoupled execution of actors provide the necessary conditions for inexpensive reconfiguration, allowing systems that are expected to be reconfigured frequently to do so in an incremental way, without requiring the full system to be changed. Reconfiguration of a small part of the system is independent of the execution or behaviour of unrelated parts of the same system.

Coordination via constraint satisfaction

Computation within the Dreams framework evolves in a stepwise manner. In each round, descriptions of the behaviour of all building blocks are combined and a

coordination pattern for the current round is chosen. In the case of the $\mathcal{R}$eo coordination language, its present models do not come with an efficient (distributed) implementation technique. Hence, we develope a new semantic and executable model for $\mathcal{R}$eo based on constraint satisfaction. A $\mathcal{R}$eo connector is seen as a set of constraints, based on the way the primitives are connected and on their current state.

We identify the four main concepts that characterise coordination in $\mathcal{R}$eo, viz. synchrony, data-awareness, state, and context dependency, and describe these concepts using logical constraints. Our approach is shown to be consistent with existing $\mathcal{R}$eo models. We apply available constraint satisfaction techniques to derive a more efficient implementation of $\mathcal{R}$eo. Specifically, we developed an initial implementation using a SAT-solver to search for possible solutions, and compared its performance with an existing $\mathcal{R}$eo engine. The results strongly support the idea that constraint satisfaction offers an appealing approach to implementing coordination languages.

Scalability

Our stepwise approach to coordination is the first factor to contribute to the scalability of the Dreams framework. Implementations that require the knowledge of all future actions, such as those based on a precomputed automaton, do not scale, since finding all possible behaviour for all possible states of a concurrent system, even when possible, can be very expensive and space inefficient. For example, the number of states generally doubles for every buffer in a connector, assuming all states are reachable. In our approach infinite state spaces are excluded.

The coordination mechanism should be able to scale up to coordinate a large number of entities, possibly by exploiting multiple CPU cores or multiple computers across a network. Considering the behaviour for each round at a time is not enough to achieve this level of scalability, because of the complexity of combining the behaviour of all entities. We improve scalability by identifying regions that can execute independently, thus achieving truly decoupled execution of connectors.

As mentioned above, we create an actor for each building block involved. The resulting actors are organised in a graph structure, where edges represent communication links. We identify independent regions of the graph of actors, referred to as *synchronous regions*. Actors from each synchronous region can agree on the coordination pattern to be executed at each round without considering the behaviour of the actors outside of this region. Consequently, the constraint problem representing the behaviour at each round is smaller and more easily solved. We identify these synchronous regions by recognising that some primitive connectors have asynchronous behaviour, according to our formal characterisation of asynchronous behaviour.

The prototype Dreams engine that executes $\mathcal{R}$eo connectors is based on constraint satisfaction and exploits the existence of synchronous regions. We compared compilation and execution times of our engine against a centralised engine for $\mathcal{R}$eo, obtaining promising results for Dreams. The separation of a single connector into independently executing sub-connectors also allows for a more flexible framework for development and concurrent reconfiguration of larger coordination specifications.

Organisation of the thesis

This thesis describes a series of developments that culminated in the Dreams framework, using the $\mathcal{R}$eo coordination model as the main case study. We start by motivating the need for our distributed approach, and by providing context in the coordination field. We then develop more efficient implementation techniques using constraint satisfaction techniques, and introduce a distributed protocol that can make local decisions to advance the coordination of larger systems.

Chapter 2 – Dataflow-oriented coordination models. We describe two coordination languages, $\mathcal{R}$eo and Linda, and present some of their models. We give an overview of existing formalisms used in the field that can be used in our distributed framework. The $\mathcal{R}$eo coordination language, described in this chapter, is explored in more detail throughout the thesis.

Chapter 3 – A stepwise coordination model. The stepwise coordination model focuses on aspects of coordination relevant for the Dreams framework. The coordination behaviour is described using a state-based formalism that we call *behavioural automata*. In these automata, labels represent atomic actions, and their composition is based on the composition of atomic actions. We also encode the models presented in Chapter 2 as behavioural automata.

Chapter 4 – Connector colouring & animation. We describe the recent connector colouring (CC) semantics of $\mathcal{R}$eo, which we consider better suited for distribution than other semantic models. The CC semantics motivated the development of the Dreams framework and is described in this thesis as an instance of behavioural automata, providing insight for some of the choices made in Chapter 3. We also introduce connector animation as an extension of connector colouring that is used to visualise the dataflow of distributed $\mathcal{R}$eo connectors.

Chapter 5 – Constraint-based models for $\mathcal{R}$eo. In this chapter we represent coordination using constraints to develop an efficient implementation of the $\mathcal{R}$eo coordination model using existing SAT-solving and constraint satisfaction techniques. Constraints represent possible coordination patterns, composition of $\mathcal{R}$eo connectors is achieved by adding conjunctively their constraints, and solutions

to the constraints represent atomic actions that the system can perform. A comparison between a prototype implementation using a SAT-solver and another existing $\mathcal{R}$eo engine based on connector colouring indicates the advantage of using constraint-solving techniques for coordination. We also present correctness proofs with respect to previous $\mathcal{R}$eo models, and make an extensive comparison of existing $\mathcal{R}$eo models and implementation approaches.

Chapter 6 – Dreams framework. The Dreams chapter describes coordination as an activity in a system of actors, where each actor is associated with a behavioural automaton, and actors communicate with their neighbour actors using asynchronous messages. We exploit the combination of the actor model with the behavioural automata to partition the system of actors into *synchronous regions*. Actors within a synchronous region reach consensus before communicating data values, while data exchanged between synchronous regions can be sent asynchronously. We formalise the correctness of this approach, and illustrate the intuition behind our approach using $\mathcal{R}$eo.

Chapter 7 – Implementing Dreams This chapter describes the implementation details of the Dreams framework. We define how actors communicate with each other, and describe the distributed protocol introduced in Dreams. To illustrate the protocol, we show traces of the execution of some $\mathcal{R}$eo connectors in our distributed framework. We also show how to use Dreams to deploy a connector in a distributed network, and we compare its performance against the performance of a centralised $\mathcal{R}$eo engine. The Dreams engine is integrated within an existing $\mathcal{R}$eo toolkit [16].

The main contributions of this thesis result from two main observations regarding the implementation of synchronous coordination models in a distributed environment. First, there is no need to pre-calculate all future behaviour at compile-time: shifting part of this computation to run-time increases scalability and eases reconfiguration. Second, not all communication in a synchronous system is performed synchronously. By identifying parts with asynchronous communication we can restrict the synchronisation process to smaller parts of the coordination layer, thereby improving the overall performance of the coordination engine.

Chapter 2

Dataflow-oriented coordination models

2.1 Introduction

In a survey of coordination languages [10], Arbab distinguishes three different approaches to coordination: a data-oriented, a control-oriented, and a dataflow-oriented approach. The main goal of most data-oriented approaches is to provide a mechanism to guarantee consistency among shared data. The code for coordination and computation can be combined, without losing the separation of these two concerns. In control-driven approaches the separation of coordination and computation is more explicit. The main focus of control-driven approaches is the processing or flow of control, and often the notion of a data value is not even required. Finally, dataflow-oriented approaches sit between data- and control-oriented approaches, managing *who* can communicate, *where* data flow, and *what* data values are sent.

The most relevant work in this thesis results from an effort to implement a distributed engine for the $\mathcal{R}$eo [8, 9] coordination language. The coordination survey mentioned above refers to $\mathcal{R}$eo as a dataflow-driven coordination model. An earlier survey by Arbab and Papadopoulos [90] also classifies Manifold [28], a predecessor of $\mathcal{R}$eo, as a dataflow-driven coordination model. The implementation approach taken in this thesis fits within a general dataflow-driven view of coordination languages.

In this chapter we describe different coordination languages and explain dataflow-driven aspects later in this thesis. We start by describing two important semantic models for $\mathcal{R}$eo followed by two similar semantic models for Linda. In Chapter 3 we will introduce the so-called *stepwise coordination model* to capture our view of dataflow-driven models, and we will present encodings of each of the coordination models described in this chapter into the stepwise coordination model.

In this thesis, the $\mathcal{R}$eo coordination language plays a more relevant role than

the other concrete models, as we use it as our main case-study for our distributed implementation. We emphasise the $\mathcal{R}$eo coordination language, since it is the main motivator of the work developed in this thesis. We describe $\mathcal{R}$eo in §2.2. We describe the Linda coordination language in §2.3, be it less detailed Linda [54] is probably the first and definitely the best known coordination model, categorised in the surveys mentioned above as a data-oriented coordination model.

As Arbab and Papadopoulos indicate, the separation of data-driven vs. control-driven coordination is not a clear cut one. For example, data-driven coordination languages can be used in application domains where the data control the execution of the components, and vice-versa. Dataflow-driven approaches sit between control- and data-driven approaches, hence their separation is also not very clear. We continue to explore the dataflow-oriented aspects of Linda in Chapter 3 by presenting an encoding of Linda into the stepwise model, which follows a typical dataflow-driven approach.

Contribution This chapter takes the first step toward the main goal of this thesis: an efficient distributed implementation of a synchronous coordination model. The implemented model, called the stepwise coordination model, is described in the next chapter. In this chapter we present existing coordination models that can be encoded into the stepwise model. Furthermore, the stepwise model leaves some aspects partially unspecified, which depend on the concrete model being encoded into the stepwise model.

2.2 $\mathcal{R}$eo

$\mathcal{R}$eo [8, 9] is presented as a channel-based coordination language wherein component connectors are compositionally constructed out of an open set of *primitive connectors*, also simply called primitives. Channels are special primitives with two ends. In this thesis we use a fixed set of primitives to illustrate the $\mathcal{R}$eo language, although user defined primitives are also possible. Being able to compose connectors out of smaller primitives is one of the strengths of $\mathcal{R}$eo. It allows, for example, multi-party synchronisation to be expressed as a composition of simple channels. Ends of primitives are regarded as ports. In addition, $\mathcal{R}$eo has a graphical notation that helps to bring some intuition about the behaviour of a connector, particularly in conjunction with animation tools, which we will cover in more detail in §4.6.

Most $\mathcal{R}$eo-related tools are being integrated in a common framework known as Eclipse Coordination Tools (ECT) [16]. The tools included in the ECT framework comprise a $\mathcal{R}$eo editor, an animation generator, model checkers, editors of $\mathcal{R}$eo-specific automata, QoS modelling and analysis tools, and a code generator.

The behaviour of connectors is described in terms of dataflow through the channels and the nodes connecting them, along with the synchronisation and mu-

tual exclusion constraints they that impose. Components attached at the boundary of a connector either attempt to write data to or read data from the ends of the channels that they are connected to. The connector coordinates the components by determining when the writes and takes succeed, often by synchronising a collection of such actions. Data flow from an end of a primitive to an end of another primitive to which it is connected, thus synchronising the two ends. A primitive decides, possibly non-deterministically, whether data is accepted or offered on an end based on the dataflow on its other ends and its state. In principle, data continue to flow like this through the connector, with primitives routing the data based on their internal behavioural constraints and the possibilities offered by the surrounding context. Primitives are executed by locally synchronising actions or by excluding the possibility of actions occurring synchronously. These 'constraints' are propagated through the connector, under the restriction that the only communication between entities occurs through the channels. Consequently, the behaviour of a system depends upon the combined choices of primitives and what possibilities the components offer, none of which is known locally to the primitives.

This section discusses the $\mathcal{R}$eo language as follows. We start by giving a general description of $\mathcal{R}$eo in §2.2.1 to introduce the main concepts, the visual notation, and some motivating examples. We follow this general description by two automata models that give a precise and formal semantics to $\mathcal{R}$eo. In §2.2.2 we present the constraint automata model, which emphasises how the value of data can affect the dataflow. In §2.2.3 we present a more recent model that focuses on how the availability of dataflow can affect the behaviour, known in the $\mathcal{R}$eo community as context dependency. Later in Chapters 4 and 5 we present two more approaches to describe $\mathcal{R}$eo.

2.2.1 General description

$\mathcal{R}$eo connectors are constructed by composing more primitive connectors. Each primitive offers a variety of behavioural policies regarding synchronisation, buffering, lossiness, and even the direction of dataflow. Communication with a primitive occurs through its ports, called *ends*: primitives consume data through their *source ends*, and produce data through their *sink ends*. Source and sink ends correspond to the notion of source and sink in directed graphs, although the names *input* and *output* ends are sometimes used instead. Primitives are not only a means for communication, but they also impose relational constraints, such as synchronisation or mutual exclusion, on the timing of dataflow on their ends. The behaviour of such primitives is limited only by the model underlying a given $\mathcal{R}$eo implementation. For the purpose of this thesis, we do not distinguish between primitives such as channels used for coordination and the components being coordinated. Typically, the 'coordinator' has more control over the choice of the behaviour of

primitives, whereas component behaviour is externally determined.

We present a description of $\mathcal{R}$eo's semantics in terms of general constraints. The first thing to note is that the behaviour of each primitive depends upon its current state.[1] The semantics of a connector is described as a collection of possible steps for each state, and we call the change of state of the connector triggered by one of these steps a round. Dataflow on a primitive's end occurs when a single datum is passed through that end. Within any round dataflow may occur on some number of ends. The semantics of a connector is defined in terms of two kinds of constraints:

Synchronisation constraints describe the sets of ends that can be synchronised in a particular step. For example, synchronous channel types typically permit dataflow either on both of their ends or on neither end, and asynchronous channel types typically permit dataflow on at most one of their two ends.

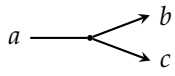
Dataflow constraints describe the data flowing on the ends that synchronise. For example, such a constraint may say that the data item flowing on the source end of a synchronous channel is the same as the data item flowing on its sink end; or that there is no constraint on the dataflow, such as for a drain which simply discards its data; or it may say that the data satisfies a particular predicate, as in the case of a filter channel.

Connectors are formed by plugging the ends of primitives together in a one-to-one fashion to form *nodes*. A node is a logical place consisting of a sink end, a source end, or both a sink and a source end.[2] We call nodes with a single end *boundary* nodes, represented by ○, and we call nodes with a sink end and a source end mixed nodes, represented by ●. Data flow through a connector from primitive to primitive through nodes, subject to the constraint that nodes cannot buffer data. This means that the two ends in a node are synchronised and have the same dataflow—behaviourally, they are equal. Nodes can be handled transparently by using the same name for the two ends on the node, as the synchronisation and dataflow at the two ends is identical.

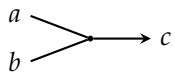
We now give an informal description of some of the most commonly used $\mathcal{R}$eo primitives. Note that for all of these primitives, no dataflow is one of the behavioural possibilities.

[1] Note that most $\mathcal{R}$eo primitives presented here have a single state.

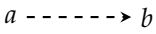
[2] Generalised nodes with multiple sink and source ends can be represented using binary mergers and replicators [22, 37].



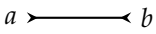
Replicator It replicates data synchronously from a to b and c. Thus, data flow either at all ends or nowhere, and the values of the data at the ends b and c are the same as the value at the end a. An n-replicator behaves similarly, replicating data synchronously from its source end to all of its n sink ends.



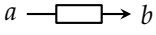
Merger It copies data synchronously from a or b to c, but not from both. Thus data flow on the ends a and c (and not on the end b) or on the ends a and c (and not on the end b), where the values of the data are equal on both ends where data flow. When both alternatives are possible, the decision of which dataflow alternative occurs is made non-deterministically at each round.



LossySync Data flow from a to b, if possible and data flow at a is always enabled. If data flow at a, dataflow at b is also possible, in which case the data that flow at ends a and b are equal.



SyncDrain It acts purely to synchronise ends a and b, thus data flow at the end a if and only if data flow at the end b, and there is no constraint on the values of the data.



FIFO$_1$ It has two possible states: empty or full. When the state is empty it can receive a data item from a, changing its state to full. When the state is full it cannot receive any more data, but it can send the data item received previously through b, changing its state back to empty.

We now present two examples of connectors to illustrate $\mathcal{R}$eo's semantics, using the primitives introduced above. We start with a simple example of an exclusive router, and afterwards present a more complex example that coordinates the control flow of two components.

2.2.1. EXAMPLE. The connector in Figure 2.1 is an exclusive router built by composing two LossySync channels (b-e and d-g), one SyncDrain (c-f), one Merger (h-i-f), and three Replicators (a-b-c-d, e-j-h and g-i-k). The constraints of these primitives can be combined to give the following two behavioural possibilities (plus the no-flow-everywhere possibility):

- ends $\{a, b, c, d, e, f, h, j\}$ synchronise and a data item flows from a to j,
- ends $\{a, b, c, d, f, g, i, k\}$ synchronise and a data item flows from a to k. $\Diamond$

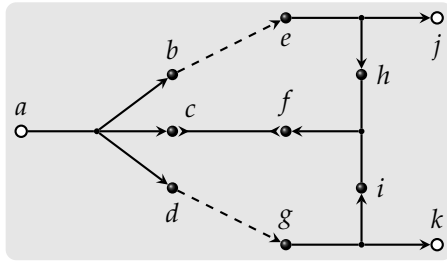


Figure 2.1: Exclusive router connector.

The merger makes a non-deterministic choice whenever both behaviours are possible. Data can never flow from a to both j and k, as this is excluded by the behavioural constraints of the Merger h-i-f.

For our second example of a $\mathcal{R}$eo connector, we denote the exclusive router using a node notation $\otimes$, depicted in Figure 2.1. Each exclusive router node needs to have exactly one channel pointing towards the node, which will be connected to the end a in the figure. It also needs to have two channel ends pointing away from the node, that will be connected to the ends j and k in the figure. Furthermore, we may define the exclusive router node to have more than two channel ends pointing away from the node. We omit the details of the generalisation of the connector here, which will be presented later in Figure 5.5. We also represent nodes with multiple sink and source ends, that can be constructed using binary mergers and replicators [22, 37]. The second example is one of the workflow patterns defined by Van der Aalst [99], where the execution of two components is synchronised by what he describes to be a synchronising merger. The $\mathcal{R}$eo connector can be seen as a precise formalisation of the textual description by Van der Aalst.

2.2.2. EXAMPLE. The *synchronising merge* connector controls the execution of two components A and B such that either A executes, or B executes, or both execute and the connector synchronises on the completion of A and B. The components A and B are represented by boxes with a source end on the left side, and a sink end on the right side. We assume that each of these components receives a signal that triggers its execution via its source end, and that it returns a signal on its sink end after its execution is complete.

One of the main differences between this connector and the exclusive router connector is that the synchronising merge connector contains stateful channels, namely four FIFO_1 channels. The behaviour of the synchronising merge connector depends on the state of its FIFO_1 buffers. $\Diamond$

The expected behaviour of the connector, involving the flow of data on ends i, o, and on the ends of A and B, is as follows. Initially, only the source end i can have

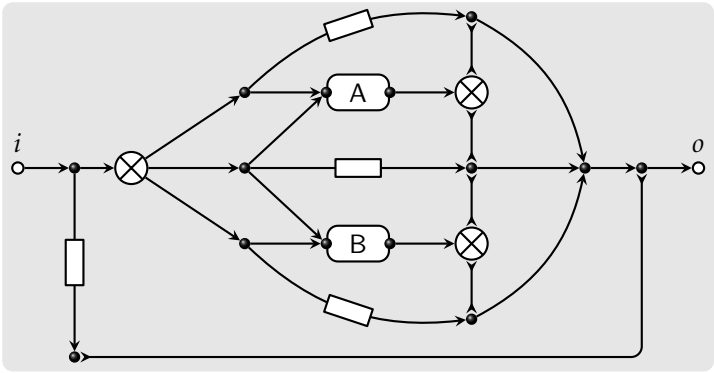


Figure 2.2: Synchronising merge connector.

dataflow, causing one or both of the components to start executing, and changing the state of two of the $FIFO_1$ buffers to full. After this, the only possible behaviour is to wait for the components that started to execute to finish and output a signal each. In the same step the sink end will have dataflow, and the two $FIFO_1$ buffers will become empty. The two exclusive routers on the right forward the data from the components A and B to exactly one of the connected synchronous drains. A simple node would not work in place of either of these exclusive routers because only one of the three central $FIFO_1$ channels can have data at any given time, thus a substitute node can never replicate the data into both connected drains. An animation for this connector can be found online.[3]

Consider, for instance, the following possible scenario. Initially all the $FIFO_1$ channels are empty, and data is available at the node *i*. The component A is ready to receive data, but not the component B. Data can then flow from *i* to the leftmost buffer, to the top exit of the exclusive router node, to the component A, and also to the uppermost buffer. Data cannot flow to any of the other two central $FIFO_1$ channels because this would require B to receive data, which contradicts our original assumption regarding B. After this step the top and the leftmost $FIFO_1$ channels become full. In the next step the component A can output a value, the uppermost SyncDrain receives the data from A and the upper $FIFO_1$, the data from the upper $FIFO_1$ also flow through node *o*. Finally, the long SyncDrain at the bottom receives data from both ends. After this round all $FIFO_1$ channels become empty again.

[3] http://reo.project.cwi.nl/webreo/generated/syncmerge/frameset.htm

2.2.2 Constraint automata

Constraint automata (CA) formalise the behaviour and the dataflow in a $\mathcal{R}$eo connector that controls the interaction of a set of anonymous components. Baier *et al.* [22] show that constraint automata can serve as a computational model for $\mathcal{R}$eo, using a coalgebraic semantics for $\mathcal{R}$eo connectors that assigns to a connector a relation over infinite timed data streams.

In this section we describe constraint automata and their composition. The composition will be used later in this thesis in two ways. First, the next section defines the encoding of the constraint automata model in our behavioural automata model. The composition of the behavioural automata thus obtained from constraint automata is based on the composition of constraint automata. Second, in Chapter 5 we use constraint automata to show compositionality of the constraint-based model for $\mathcal{R}$eo.

Constraint automata use a finite set of port names $\mathcal{N} = \{x_1, \ldots, x_n\}$, where x_i is the i-th port of a connector. When clear from the context, we write xyz instead of $\{x, y, z\}$ to enhance readability. We write $\widehat{x_i}$ to represent the data value flowing through the port x_i, and use $\widehat{\mathcal{N}}$ to denote the set of data variables $\{\widehat{x}_1, \ldots, \widehat{x}_n\}$, for each $x_i \in \mathcal{N}$. We define DC_X for each $X \subseteq \mathcal{N}$ to be a set of data constraints over the variables in $\widehat{X}$, where the underlying data domain is a finite set $\mathbb{D}$. Data constraints in $DC_{\mathcal{N}}$ can be viewed as a symbolic representation of sets of data-assignments, and are generated by the following grammar:

$$g \;::=\; \mathtt{tt} \;\mid\; \widehat{x} = d \;\mid\; g_1 \vee g_2 \;\mid\; \neg g$$

where $x \in \mathcal{N}$ and $d \in \mathbb{D}$. The other logical connectives can be encoded as usual. We use the notation $\widehat{a} = \widehat{b}$ as a shorthand for the constraint

$$(\widehat{a} = d_1 \wedge \widehat{b} = d_1) \vee \ldots \vee (\widehat{a} = d_n \wedge \widehat{b} = d_n),$$

with $\mathbb{D} = \{d_1, \ldots, d_n\}$.

2.2.3. DEFINITION (CONSTRAINT AUTOMATON [22]).
A *constraint automaton* (over the finite data domain $\mathbb{D}$) is a tuple $\mathcal{A} = \langle Q, \mathcal{N}, \rightarrow, Q_0 \rangle$, where Q is a set of states, $\mathcal{N}$ is a finite set of port names, $\rightarrow$ is a subset of $Q \times \mathbf{2}^{\mathcal{N}} \times DC_{\mathcal{N}} \times Q$, called the transition relation of $\mathcal{A}$, and $Q_0 \subseteq Q$ is the set of initial states. $\lhd$

We write $q \xrightarrow{X|g} p$ instead of $(q, X, g, p) \in \rightarrow$. For every transition $q \xrightarrow{X|g} p$, we require that g, the guard, is a DC_X-constraint. For every state $q \in Q$, there is a transition $q \xrightarrow{\emptyset|\mathtt{tt}} q$.

We define $\mathsf{CAS} \subseteq \mathbf{2}^{\mathcal{N}} \times DC_{\mathcal{N}}$ to be the set of solutions for all possible labels of the transitions of constraint automata. That is, $X|g \in \mathsf{CAS}$ if $X = \{x_1, \ldots, x_n\}$,

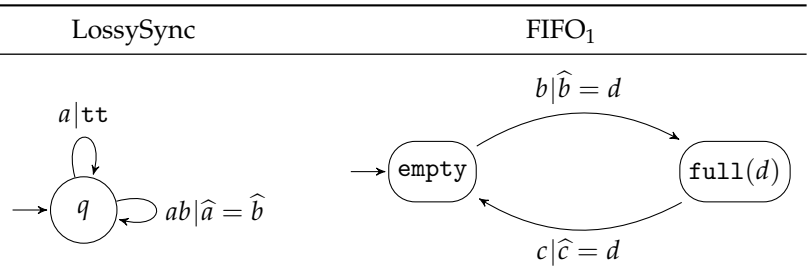


Figure 2.3: Constraint automata for the LossySync and the FIFO_1 channels.

$g = \bigwedge \widehat{x_i} = v_i$, and there is a transition $q \xrightarrow{X|g'} q'$ such that g satisfies g'. We call each $s \in \mathsf{CAS}$ a constraint automaton step. We interpret each transition $q \xrightarrow{X|g} p$ as follows. When the automaton is in state q, it is possible to have dataflow at all the ports in X, while excluding flow at ports in $\mathcal{N} \setminus X$. The data flowing through the ports X must satisfy the constraint g, and the automaton evolves to state p. Thus, in constraint automata synchronisation is described by the set X and dataflow is described by the constraint g. For every state $q \in Q$, the empty transition $q \xrightarrow{\emptyset|\mathtt{tt}} q$ is present for technical reasons, namely, to simplify the definition of the product, below. For clarity, we normally do not display the empty transitions when depicting constraint automata in our figures.

2.2.4. EXAMPLE. Most primitives presented in §2.2.1 are stateless, which means each of their corresponding constraint automata has a single state. The LossySync channel is formalised by the automaton $\mathcal{A}_L = \langle Q, \mathcal{N}, \to, Q_0 \rangle$ with one state and three transitions, depicted in Figure 2.3, where:

$$Q = \{q\}\,, \quad \mathcal{N} = \{a, b\}\,, \quad Q_0 = \{q\}$$
$$\to = \left\{ \langle q, a, \mathtt{tt}, q\rangle\,, \langle q, ab, \widehat{a} = \widehat{b}, q\rangle, \langle q, \emptyset, \mathtt{tt}, q\rangle \right\} \qquad \lozenge$$

2.2.5. EXAMPLE. The FIFO_1 channel, depicted as a constraint automaton in Figure 2.3, has, besides the `empty` state, multiple states $\mathtt{full}(d)$. The FIFO_1 channel is formalised by the automaton $\mathcal{A}_F = \langle Q, \mathcal{N}, \to, Q_0 \rangle$ where:

$$Q = \{\mathtt{empty}\} \cup \{\mathtt{full}(d) \mid d \in \mathbb{D}\} \quad \to = \{\langle \mathtt{empty}, b, \widehat{b} = d, \mathtt{full}(d)\rangle \mid d \in \mathbb{D}\}$$
$$\mathcal{N} = \{b, c\} \qquad \cup \{\langle \mathtt{full}(d), c, \widehat{c} = d, \mathtt{empty}\rangle \mid d \in \mathbb{D}\}$$
$$Q_0 = \{\mathtt{empty}\} \qquad \cup \{\langle q, \emptyset, \mathtt{tt}, q\rangle \mid q \in Q\} \qquad \lozenge$$

Composition

Common port names correspond to the places where connectors are joined. Note that $\mathcal{R}$eo connectors can only be composed by connecting source ports to sink ports, such that each port is connected to at most one other port. Therefore, when composing two constraint automata $\mathcal{A}_1$ and $\mathcal{A}_2$ representing two $\mathcal{R}$eo connectors, we require that common ports must be source ports in one connector and sink ports in the other. However, the constraint automata model does not distinguish source and sink ends, and the restriction is only required for $\mathcal{R}$eo connectors.

2.2.6. DEFINITION (PRODUCT OF CONSTRAINT AUTOMATA [22]).
Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be two constraint automata, where $\mathcal{A}_i = \langle Q_i, \mathcal{N}_i, \rightarrow_i, Q_{0,i} \rangle$ for $i \in \{1, 2\}$. The composition of $\mathcal{A}_1$ and $\mathcal{A}_2$ yields the constraint automaton

$$\mathcal{A}_1 \bowtie \mathcal{A}_2 = \langle Q_1 \times Q_2, \mathcal{N}_1 \cup \mathcal{N}_2, \rightarrow, Q_{0,1} \times Q_{0,2} \rangle \,,$$

where the transition relation $\rightarrow$ is given by the condition below.

$$(q_1, q_2) \xrightarrow{X_1 \cup X_2 | g_1 \wedge g_2} (p_1, p_2) \;\; \textit{iff} \\ q_1 \xrightarrow{X_1|g_1}_1 p_1, q_2 \xrightarrow{X_2|g_2}_2 p_2, \text{ and } X_1 \cap \mathcal{N}_2 = X_2 \cap \mathcal{N}_1 \qquad \lhd$$

The requirement $X_1 \cap \mathcal{N}_2 = X_2 \cap \mathcal{N}_1$ states that the steps $X_1|g_1$ and $X_2|g_2$ can only be combined into a new transition in the automaton $\mathcal{A}_1 \bowtie \mathcal{A}_2$ when they agree on the firing of all of their common ports in these transitions. I.e., for all ports $x \in \mathcal{N}_1 \cap \mathcal{N}_2$ it holds that $x \in X_1 \Leftrightarrow x \in X_2$.

2.2.7. EXAMPLE. Recall the constraint automata of the LossySync and the FIFO_1 channels presented in Example 2.2.4 and in Example 2.2.5, respectively. Observe that the two automata share exactly one port, the port b, which is a source port in the LossySync channel and a sink port in the FIFO_1 channel. The composition of these two constraint automata yields $\mathcal{A}_L \bowtie \mathcal{A}_F$, depicted in Figure 2.4. The transitions with labels $a|\texttt{tt}$ and $c|\widehat{c} = d$ result from the combination of the transition $q' \xrightarrow{\emptyset|\texttt{tt}} q'$, for some state $q' \in \{q, \texttt{empty}\} \cup \{\texttt{full}(d) \mid d \in \mathbb{D}\}$, of one of the automata with a transition $t = q \xrightarrow{a|\texttt{tt}} q$ or $t = \texttt{full}(d) \xrightarrow{c|\widehat{c}=d} \texttt{empty}$ of the other automaton. This reflects that t can be performed 'independently' of the other automaton. On the other hand, the two other transitions correspond to the combination of a transition with dataflow from each automaton, which can be understood as "the two transitions must synchronise". $\Diamond$

The automaton $\mathcal{A}_L \bowtie \mathcal{A}_F$ presented in Example 2.2.7 exhibits a property that is often undesired in $\mathcal{R}$eo. The self transition $a|\texttt{tt}$ of the state $(q, \texttt{empty})$ is a transition where data is lost by the LossySync channel, disregarding the fact that the FIFO_1

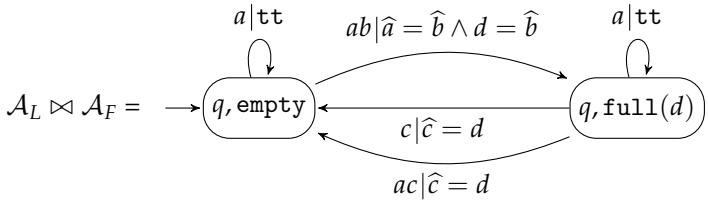


Figure 2.4: Composition of the constraint automata $\mathcal{A}_L$ and $\mathcal{A}_F$.

channel is empty and can receive data. The constraint automata model is not expressive enough to take into account the availability of the FIFO_1 channel, and to remove the undesired transition. When a model takes into account the availability of data flow in each port, we say the model captures *context dependency*. The $\mathcal{R}$eo automata model described in the next subsection is an example of a model that captures such context dependency.

2.2.3 Normalised $\mathcal{R}$eo automata

We now describe a simplified variation of the $\mathcal{R}$eo automata model [29], introduced by Bonsangue *et al.* around four years after the CA model, with the main purpose of capturing *context dependency*, a feature missing in the constraint automata model. Later in this thesis we show how this model can be used by our distributed framework, by encoding it into our behavioural automata model. The main difference between Bonsangue *et al.* and our presentation is that they use boolean algebra over port names to describe constraints over the context, while we use sets for the same purpose. Hence we refer to our model as *normalised* $\mathcal{R}$eo *automata* (RA). Furthermore, Bonsangue *et al.* assume that the product of automata is defined for automata with disjoint alphabets, achieving synchronisation via an explicit operation that 'combines' two ports. We will assume that products share the names of the ports that synchronise, to make it easier to relate to the constraint automata model.

The notion of context dependency was initially introduced in $\mathcal{R}$eo at its inception, but was formalised first by the connector colouring model, which precedes the $\mathcal{R}$eo automata model. We describe the connector colouring model in this thesis in more detail in Chapter 4. A more exhaustive overview of context dependency in $\mathcal{R}$eo can be found in [37], and in the thesis of Costa [41], where he presents a similar automata model that captures context dependency for $\mathcal{R}$eo. Costa introduces the intentional automata model to support the ideas behind the connector colouring model, which he also introduces in his thesis. Similarly to constraint automata, in intentional automata ports that synchronise are not removed after composition, hence it requires a larger state space than the presentation of $\mathcal{R}$eo

automata by Bonsangue *et al.* where these ports are discarded. Costa also gives a precise formalisation of the hiding operation of ports as intended initially [19]. We do not consider this in this thesis because it does not influence the distributed implementation of connectors.

Recall the constraint automaton $\mathcal{A}_L \bowtie \mathcal{A}_F$ from Example 2.2.7. To avoid the undesired behaviour where data is lost when the FIFO_1 buffer is empty the constraint automata model is extended to capture context dependency, by adding explicitly the context information to the transitions. Two important examples of $\mathcal{R}$eo primitives that could not be represented in the CA model are:

Context-dependent LossySync This channel loses data written to its source only if the surrounding context is unable to accept the data through its sink; otherwise the data flow through the channel. This corresponds to the initial intention of the LossySync channel [8].

Priority merger This is a special variant of a merger that favours one of its sink ports: if dataflow is possible at both sink ports, it prefers one port over the other.

The $\mathcal{R}$eo automata model, when compared to the constraint automata model, abstracts away from data and introduces guards to capture the context in which a $\mathcal{R}$eo primitive is evaluated. In normalised $\mathcal{R}$eo automata we assume that the product of automata synchronises and hides ports with shared names, following the convention of the $\mathcal{R}$eo automata model, and the context is described by sets of literals. When possible, we keep the same notation as in the $\mathcal{R}$eo automata paper [29], while trying to preserve the naming notation of the constraint automata model.

Normalised $\mathcal{R}$eo automata use a finite set of port names $\mathcal{N} = \{x_1, ..., x_n\}$ as in constraint automata. A *guard* g is a set $\{a_1, \ldots, a_k\}$ of literals derived from $\mathcal{N}$, where $a_i \in \{x_i, \overline{x}_i\}$ and $x_i \in \mathcal{N}$. We denote the set of all literals of $\mathcal{N}$ as $\mathbf{Lt}_{\mathcal{N}}$, and $\overline{X} = \{\overline{x}_1, \ldots, \overline{x}_m\}$, where $X = \{x_1, \ldots, x_m\}$ is a set of ports. Observe that $g \in \mathcal{N} \cup \overline{\mathcal{N}}$. A guard represents which ports have data available to flow, and which ports cannot have data available. The extra knowledge about the (im)possibility of dataflow expressed by the guards characterises the context dependency modelled by the $\mathcal{R}$eo automata model. As for constraint automata, we often write $a_1 \ldots a_k$ instead of $\{a_1, \ldots, a_k\}$ and $g_1 g_2$ instead of $g_1 \cup g_2$.

2.2.8. DEFINITION (NORMALISED $\mathcal{R}$eo AUTOMATON [29]).
A normalised $\mathcal{R}$eo automaton over an alphabet $\mathcal{N}$ is a triple $\mathcal{A} = \langle Q, \mathcal{N}, \rightarrow \rangle$ where Q is a finite set of states, and $\rightarrow$ is a subset of $Q \times \mathbf{Lt}_{\mathcal{N}} \times \mathbf{2}^{\mathcal{N}} \times Q$ called the transition relation of $\mathcal{A}$, such that for each $\langle q, g, X, q' \rangle \in \rightarrow$ the context property $X \subseteq g$ holds.[4] ◁

[4]The original definition of the $\mathcal{R}$eo automata model by Bonsangue *et al.* refers to the context property as *reactivity property*, and also includes a *uniformity property*, which they use to distinguish between

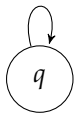
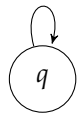
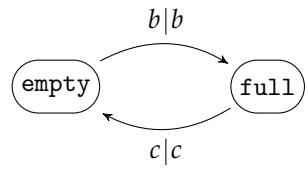
Context dependent LossySync	PriorityMerger	$FIFO_1$
$ab\|ab$ $a\overline{b}\|a$ 	$ac\|ac$ $\overline{a}bc\|bc$ 	

Figure 2.5: Normalised $\mathcal{R}$eo automata of the context dependent LossySync, the priority merger, and the $FIFO_1$ primitives.

We write $q \xrightarrow{g|X} q'$ as a shorthand for $\langle q, g, X, q' \rangle \in \rightarrow$. Informally, a normalised $\mathcal{R}$eo automaton over an alphabet $\mathcal{N}$ is a non-deterministic automaton with transition labels $\mathsf{RAS} = \mathbf{Lt}_{\mathcal{N}} \times \mathbf{2}^{\mathcal{N}}$ ($\mathcal{R}$eo automata steps) that obey the context property. The intuition is that for each label $g|X \in \mathsf{RAS}$ the ports in X have dataflow when the context respects g. For example, the transition $q \xrightarrow{a\overline{b}|a} q'$ can be read as "the automaton can evolve from q to q' by having dataflow on a when the context is ready to have dataflow on a and the context refuses dataflow on b (for the current round)". The context property reflects the fact that a port can have dataflow only when the context is ready to accommodate it.

2.2.9. EXAMPLE. In Figure 2.5 we present the normalised $\mathcal{R}$eo automata $\mathcal{A}'_C$, $\mathcal{A}'_P$, and $\mathcal{A}'_F$, of the context dependent LossySync channel, the priority merger, and the $FIFO_1$ channel, respectively. The normalised $\mathcal{R}$eo automata of all $\mathcal{R}$eo primitives presented in this thesis possess the context property. Furthermore, Bonsangue *et al.* show that this property is preserved by composition as defined below. ◊

Composition

The composition of normalised $\mathcal{R}$eo automata that we present here is slightly different then what Bonsangue *et al.* define [29], because we do not require that the alphabets of the two automata are disjoint, and we require all guards to be normalised. Compared to their approach, our definition of the product of $\mathcal{R}$eo automata also captures the synchronisation of shared variables and the normalisation of transitions, since we consider normalised transitions only. We introduce two auxiliary concepts used in the definition of the composition of normalised

$\mathcal{R}$eo automata and a more general automata, and to show some theoretical results such as the fact that the Sync channel behaves as the identity.

$\mathcal{R}$eo automata: (1) the set of unsatisfiable guards for a state q, denoted by $q^{\#}$, and the (2) compatibility of two guards g_1 and g_2, denoted by $g_1 \frown g_2$.

Given a state q of a normalised $\mathcal{R}$eo automaton $\mathcal{A} = \langle Q, \mathcal{N}, \rightarrow \rangle$, we define the set of satisfiable guards of q as follows:

$$guards(q) = \{g \mid q \xrightarrow{g|X} q'\}. \tag{2.1}$$

We now define the set $q^{\#}$ of all unsatisfiable guards based on $guards(q)$:

$$q^{\#} = \{\overline{a}_1 \cdots \overline{a}_n \mid guards(q) = \{g_1, \ldots, g_n\}, \forall_{i \in 1..n} \cdot a_i \in g_i\}, \tag{2.2}$$

where for any $x \in \mathcal{N}$, $\overline{\overline{x}} = x$. Therefore the set of all unsatisfiable guards consists of all possible combinations of the negations of literals from each reachable transition.

A remark is in order regarding the correctness of our definition of $q^{\#}$ with respect to the definition of $q^{\#}$ presented in the original paper by Bonsangue *et al.* [29]. Let $(\cdot)^{\circ}$ be a function that maps guards to logical formulæ in disjunctive normal form:

$$\{a_{11} \cdots a_{1m_1}, \ldots, a_{n1} \cdots a_{nm_i}\}^{\circ} = (a_{11} \wedge \cdots \wedge a_{1m_1}) \vee \ldots \vee (a_{nm_1} \wedge \cdots \wedge a_{nm_i}),$$

where each a_{ik} is a literal and $n, m_i, i \in \mathbb{N}$. Let $q^{\dagger} = \neg(guards(q)^{\circ})$ be the definition of $q^{\#}$ in the original formulation of $\mathcal{R}$eo automata [29]. We prove that the two definitions of $q^{\#}$, i.e., $q^{\#}$ as defined above and $q^{\dagger}$, are equivalent. Before relating $q^{\dagger}$ to $q^{\#}$ we establish an auxiliary result.

2.2.10. LEMMA. *Let* $\{ a_{ij} \mid 1 \leq i \leq n, 1 \leq j \leq m_i \}$ *be a set of literals. Then*

$$\bigwedge_{i=1}^{n} \left(\bigvee_{j=1}^{m_i} a_{ij} \right) \quad = \quad \bigvee \left\{ \bigwedge_{i=1}^{n} a_{ij} \;\middle|\; 1 \leq j \leq m_i \right\}.$$

Proof. We proceed by induction on n. The base case when $n = 1$ is trivial. As to the induction step, using the distributive laws for $\wedge$ and $\vee$:

$$\begin{aligned}
\textstyle\bigwedge_{i=1}^{n+1} \left(\bigvee_{j=1}^{m_i} a_{ij} \right) &= \textstyle\bigwedge_{i=1}^{n} \left(\bigvee_{j=1}^{m_i} a_{ij} \right) \wedge \left(\bigvee_{j=1}^{m_{n+1}} a_{n+1,j} \right) \\
&= \textstyle\bigvee \left\{ \bigwedge_{i=1}^{n} a_{ij} \mid 1 \leq j \leq m_i \right\} \wedge \left(\bigvee_{k=1}^{m_{n+1}} a_{n+1,k} \right) \\
&= \textstyle\bigvee_{k=1}^{m_{n+1}} \left(\bigvee \left\{ \bigwedge_{i=1}^{n} a_{ij} \mid 1 \leq j \leq m_i \right\} \wedge a_{n+1,k} \right) \\
&= \textstyle\bigvee_{k=1}^{m_{n+1}} \left(\bigvee \left\{ \bigwedge_{i=1}^{n} a_{ij} \wedge a_{n+1,k} \mid 1 \leq j \leq m_i \right\} \right) \\
&= \textstyle\bigvee \left\{ \bigwedge_{i=1}^{n} a_{ij} \wedge a_{n+1,k} \mid 1 \leq j \leq m_i, 1 \leq k \leq m_{n+1} \right\} \\
&= \textstyle\bigvee \left\{ \bigwedge_{i=1}^{n+1} a_{ij} \mid 1 \leq j \leq m_i \right\}.
\end{aligned}$$

□

Based on Lemma 2.2.10 we have the following.

2.2.11. PROPERTY. $q^\dagger = (q^\#)^\circ$.

Proof. Suppose $guards(q) = \{g_1, \ldots, g_n\}$, and let $g_i = a_{i1} \cdots a_{im_i}$, for $i \in 1..n$. On the one hand, by the definitions and the De Morgan laws we have:

$$q^\dagger \;=\; \neg(guards(q)^\circ) \;=\; \neg \bigvee_{i=1}^{n} \bigwedge_{j=1}^{m_i} a_{ij} \;=\; \bigwedge_{i=1}^{n} \bigvee_{j=1}^{m_i} \overline{a}_{ij}.$$

On the other hand, by the definition of $(\cdot)^\circ$ we have

$$(q^\#)^\circ \;=\; \bigvee \left\{ \bigwedge_{i=1}^{n} \overline{a}_i \;\middle|\; a_i \in g_i \right\} \;=\; \bigvee \left\{ \bigwedge_{i=1}^{n} \overline{a}_{ij} \;\middle|\; 1 \le j \le m_i \right\}$$

because $a_i \in g_i$ if and only if $a_i = a_{ij}$ for some j where $1 \le j \le m_i$. Now the property follows directly from Lemma 2.2.10. □

Finally, we define the compatibility of two guards g_1 and g_2 in $\mathcal{N} \cup \overline{\mathcal{N}}$, written as $g_1 \frown g_2$, as follows.

$$g_1 \frown g_2 \iff g_1 \cap g_2 \subseteq \mathcal{N} \;\wedge\; \forall a \in g_1 g_2 \cdot \overline{a} \notin g_1 g_2.$$

The first part of the definition of $g_1 \frown g_2$ states that g_1 and g_2 cannot expect the context to negate the same port, i.e., $\overline{x}$ cannot occur in both guards, for any $x \in \mathcal{N}$. When $\overline{x}$ appears in both guards then they cannot rely on each other to give 'reason' for the absence of dataflow on the port x, an issue explained in more detail in Chapter 4. The second part simply states that if a literal and its negation both occur in g_1 and g_2 they cannot be compatible. We use $q^\#$ and $\frown$ in the definition of the product of normalised $\mathcal{R}$eo automata, presented as Definition 2.2.12.

2.2.12. DEFINITION (PRODUCT OF NORMALISED $\mathcal{R}$eo AUTOMATA).
Let $\mathcal{A}_i = \langle Q_i, \mathcal{N}_i, \to_i \rangle$, where $i \in \{1, 2\}$, be two normalised $\mathcal{R}$eo automata, and let $s = \mathcal{N}_1 \cap \mathcal{N}_2$ the set of shared ports over which $\mathcal{A}_1$ and $\mathcal{A}_2$ must synchronise. The product of the two automata is

$$\mathcal{A}_1 \bowtie \mathcal{A}_2 = \langle Q_1 \times Q_2, \mathcal{N}_1 \cup \mathcal{N}_2, \to \rangle,$$

and the transition relation $\to$ is given by the following conditions, where we write $g \setminus s$ to denote the guard g after removing all the literals in s.

$$(q_1, q_2) \xrightarrow{g_1 g_2 \setminus s\overline{s} \mid X_1 X_2 \setminus s} (p_1, p_2) \text{ if}$$
$$q_1 \xrightarrow{g_1 \mid X_1}_1 p_1,\ q_2 \xrightarrow{g_2 \mid X_2}_2 p_2,\ g_1 \frown g_2, \text{ and } X_1 \cap \mathcal{N}_2 = X_2 \cap \mathcal{N}_1 \tag{2.3}$$
$$(q, p) \xrightarrow{g g' \mid X} (q', p) \text{ if}$$
$$q \xrightarrow{g \mid X}_1 q', g' \in q^{\#}, g \frown g', \text{ and } X \cap \mathcal{N}_2 = \emptyset \tag{2.4}$$
$$(q, p) \xrightarrow{g g' \mid X} (q, p') \text{ if}$$
$$p \xrightarrow{g \mid X}_2 p', g' \in p^{\#}, g \frown g', \text{ and } X \cap \mathcal{N}_1 = \emptyset \tag{2.5}$$

◁

Condition 2.3 in Definition 2.2.12 is very similar to the composition of constraint automata in Definition 2.2.6. The main differences are (1) the shared ports s are removed from the label, (2) the guards must be compatible by not sharing any atom of the form $\overline{x}$, and (3) the data constraints do not play any role. The $\mathcal{R}$eo automata model disregards data constraints to focus on context dependency, but the data constraints can be added orthogonally. The conditions (2.4) and (2.5) represent the transitions build from only one of the original automata. In the constraint automata model the same goal was achieved by assuming an 'empty' transition $q \xrightarrow{\emptyset, \mathtt{tt}} q$ for every state q of every constraint automaton. In the normalised $\mathcal{R}$eo automata model we add conjunctively to each guard g another guard $g' \in q^{\#}$ that confirms that no transition of the other automaton is ignored.

2.2.13. EXAMPLE. We now present the product of the $\mathcal{R}$eo automata $\mathcal{A}'_L$ and $\mathcal{A}'_F$, of the context dependent LossySync and the FIFO_1 channels depicted in Figure 2.5. The product of these two automata is given by $\mathcal{A}'_L \bowtie \mathcal{A}'_F$, presented in Figure 2.6. When compared to the constraint automata of the LossySync and the FIFO_1 channels, presented in Example 2.2.7, we find two main differences. First, the port b is hidden in the final system because it is a shared port of $\mathcal{A}'_L$ and $\mathcal{A}'_F$. Second, and most importantly, the behaviour changes in the sense that there is no transition that loops around the state $(q, \mathtt{empty})$. In the constraint automata model of this connector, this self transition corresponds to the possibility of data flowing on a and being lost by the LossySync channel. Note that $\mathtt{empty}^{\#} = \{\overline{b}\}$. Therefore the transition $q \xrightarrow{a\overline{b} \mid a} q$ of the context dependent LossySync cannot be included in the resulting automaton because for the guard $\overline{b}$ we do not have $a\overline{b} \frown \overline{b}$, more specifically $a\overline{b} \cap \overline{b} \not\subseteq \mathcal{N}$, as required by the condition (2.3) of Definition 2.2.12. That is, the automaton $\mathcal{A}'_F$, while empty, requires that the port b must have dataflow when data is available.

◊

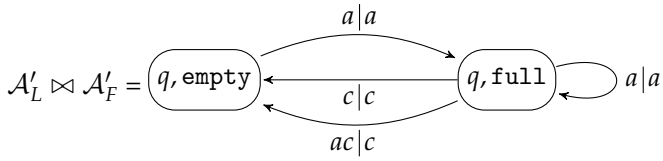


Figure 2.6: Composition of the $\mathcal{R}$eo automata $\mathcal{A}'_L$ and $\mathcal{A}'_F$.

2.3 Linda

Linda, introduced by Gelernter [54], is seen by many as the first coordination language. We describe it using two different semantics, and show in the next chapter how it can be modelled in our distributed framework. Linda is based on the *generative communication* paradigm, which describes how different processes in a distributed environment exchange data. In Linda, data objects are referred to as *tuples*, and multiple processes can communicate data using a *shared tuple-space*, where they can write or read tuples, as depicted in Figure 2.7.

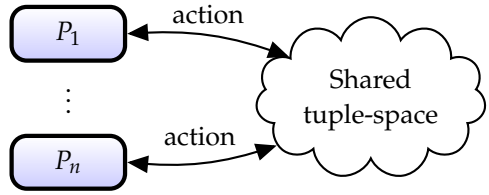


Figure 2.7: Communication between processes and a tuple-space in Linda.

Communication between processes and the tuple-space is done by actions executed by processes over the tuple-space. In general, these actions can occur only atomically, that is, the shared tuple-space can accept and execute an action from only one of the processes at a time. There are four possible actions, $\mathbf{out}(t)$, $\mathbf{in}(s)$, $\mathbf{rd}(s)$, and $\mathbf{eval}(P)$, explained below.

- $\mathbf{out}(t)$ – Denotes the output of a tuple (data value) t from a process, to be stored in the shared tuple-space. Note that t can be stored multiple times, that is, the shared tuple-space is modelled by a multi-set.

- $\mathbf{in}(s)$ – Denotes the removal of a tuple t from the tuple-space and the sending of t to the process, where s and t are related by a binary *match* relation. We describe the *match* relation in more detail below.

- $\mathbf{rd}(s)$ – Similar to the $\mathbf{in}(s)$ action, except that the tuple t is not removed from the tuple-space.

- **eval**(P) – Denotes the creation of a new process P that will run in parallel with the other processes. In the literature [54, 90, 33] each process P is also referred to as an *active tuple*, as opposed to passive tuples that represent data values.

More specifically, we write s and t to range over tuples, and we define each tuple to be a sequence of parameters, generated by the following grammar, where X ranges over a set of variables.

$$t \ ::= \ v \in \mathbb{D} \ \mid \ X \ \mid \ t;t$$

We denote by *Tuple* the set of all tuples generated by the above grammar. Each parameter can be a data value v from a domain $\mathbb{D}$ (an actual parameter), or a variable x (a formal parameter). The interaction between processes and the tuple-space is based on a pattern-matching relation between tuples. We say t *matches* s if t has only $\mathbb{D}$ values, and there is a substitution γ whose domain is the set of free variables of s, such that $t = s[\gamma]$, where $s[\gamma]$ denotes the tuple s after substituting variables in s according to γ. We write t γ-matches s when t matches s and $t = s[\gamma]$.

Several variations of Linda were introduced later, such as Java's popular implementation JavaSpace of Jini [51], and the Klaim language [26], which considers multiple distributed tuple-spaces. Other implementations of Linda can also be found in widespread programming languages such as Prolog [97], Ruby (Rinda),[5] Python (PyLinda),[6] C++ (CppLINDA),[7] Smalltalk [95], and Lisp [47]. Individual tuple operations in Linda-like languages are atomic, but they do not provide the global synchronisation supported by $\mathcal{R}$eo. In the remaining of this section we formalise Linda using the Linda-Calculus[43], and give both course-grained and fine-grained operational semantics for the Linda-Calculus.

The Linda-Calculus

We use the Linda-Calculus model, described by Goubault [43], to give a formal description of Linda, studied also by Ciancarini *et al.* [33] and others. The Linda-Calculus abstracts away from the local behaviour of processes, and focuses on the communication primitives between a *store* and a set of *processes*. Processes P are generated by the following grammar.

$$P ::= Act.P \ \mid \ X \ \mid \ \mathbf{rec}X.P \ \mid \ P \,[]\, P \ \mid \ \mathbf{end} \tag{2.6}$$

$$Act ::= \mathbf{out}(t) \ \mid \ \mathbf{in}(s) \ \mid \ \mathbf{rd}(s) \ \mid \ \mathbf{eval}(P) \tag{2.7}$$

[5] http://ruby-doc.org/stdlib/libdoc/rinda/rdoc/index.html
[6] http://code.google.com/p/pylinda/
[7] http://sourceforge.net/projects/cpplinda/

The first case $Act.P$ represents the execution of a Linda action. The productions X and $\mathbf{rec}X.P$ are used to model recursive processes, where X ranges over a set of variables, and $P \,[\!]\, P$ is used to model local non-deterministic choice. We assume that Linda processes do not have free variables, i.e., every X is bound by a corresponding $\mathbf{rec}X$. Finally **end** represents termination.

We model a Linda store as a multi-set of tuples from *Tuple*. We use the $\oplus$ operator to denote multi-set construction and multi-set union. For example, we write $M = t \oplus t = \{|t, t|\}$ and $M \oplus M = \{|t, t, t, t|\}$, where t is a tuple and $\{|s, t|\}$ denotes a multi-set with the elements s and t.

A *tuple-space term* M is a multi-set of processes and tuples, generated by the grammar $M ::= P \mid t \mid M \oplus M$. We adopt the approach of Goubault and provide an operational semantics for the Linda-Calculus.

We present a set of inference rules that give the operational semantics of the Linda-Calculus. The relation $match \in Tuple \times Tuple$, described in the beginning of this section, represents the matching of two tuples, and $P[\gamma]$ denotes the process P after replacing all of its the free variables according to the substitution γ. We also write $\gamma = P'/x$ to denote the substitution of x by the process P'.

2.3.1. DEFINITION (ITS-LINDA). The *interleaved transition system* for Linda is defined by the inference rules below.

$$M \oplus \mathbf{eval}(P).P' \longrightarrow M \oplus P \oplus P' \qquad \text{(eval)}$$

$$M \oplus \mathbf{out}(t).P \longrightarrow M \oplus P \oplus t \qquad \text{(out)}$$

$$M \oplus \mathbf{rd}(s).P \oplus t \longrightarrow M \oplus P[\gamma] \oplus t \quad \text{if } t \ \gamma\text{-matches } s \qquad \text{(rd)}$$

$$M \oplus \mathbf{in}(s).P \oplus t \longrightarrow M \oplus P[\gamma] \qquad \text{if } t \ \gamma\text{-matches } s \qquad \text{(in)}$$

$$\frac{M \oplus P[\mathbf{rec}X.P/X] \longrightarrow M \oplus P'}{M \oplus \mathbf{rec}X.P \longrightarrow M \oplus P'} \qquad \text{(rec)}$$

$$M \oplus P \,[\!]\, P' \longrightarrow M \oplus P \qquad \text{(left)}$$

$$M \oplus P \,[\!]\, P' \longrightarrow M \oplus P' \qquad \text{(right)}$$

$$M \oplus \mathbf{end} \longrightarrow M \qquad \text{(end)}$$

We use ITS to denote the above transition system. ◁

The semantics described by the interleaved transition system for Linda follows a *coarse grained* approach, and assumes that every two actions must occur in an interleaved way, but never simultaneously. A *fine grained* approach assumes that multiple actions can occur at the same time. We augment the interleaved transition system for Linda with an additional inference rule that captures the fine grained alternative as follows.

2.3.2. DEFINITION (MTS-LINDA). The *multistep transition system* for Linda is described by the ITS-Linda semantics augmented with the following inference rule.

$$\frac{M_1 \longrightarrow M_1' \qquad M_2 \longrightarrow M_2'}{M_1 \oplus M_2 \longrightarrow M_1' \oplus M_2'} \tag{par}$$

We use MTS to denote this new transition system. ◁

Consider the following sequence of transitions of a tuple-space term in the Linda-Calculus, illustrating the sending of data between two processes.

$$\begin{aligned}
&\mathbf{rd}(42,x).P(x) \oplus \mathbf{out}(42,43).\mathbf{end} \oplus \mathbf{in}(42,x).P'(x) \\
&\to\ \mathbf{rd}(42,x).P(x) \oplus \mathbf{end} \oplus \mathbf{in}(42,x).P'(x) \oplus \langle 42,43\rangle && \text{(out)} \\
&\to\ \mathbf{rd}(42,x).P(x) \oplus \mathbf{in}(42,x).P'(x) \oplus \langle 42,43\rangle && \text{(end)} \\
&\to\ P(43) \oplus \mathbf{in}(42,x).P'(x) \oplus \langle 42,43\rangle && \text{(rd)} \\
&\to\ P(43) \oplus P'(43) && \text{(in)}
\end{aligned}$$

This simple example illustrates how the communication of processes is modelled in Linda. We use the notation $P(x)$ as syntactic sugar to denote a process P where the variable x occurs freely. In the first step the process $\mathbf{out}(42,43).\mathbf{end}$ writes a tuple $\langle 42,43\rangle$ into the tuple-space, and in the second step the process **end** is dropped. In the third step the process $\mathbf{rd}(42,x).P(x)$ requests a tuple whose first element is 42, and reads the tuple $\langle 42,43\rangle$. The use of the action **rd** instead of the action **in** results in a non-destructive read, that is, the tuple $\langle 42,43\rangle$ is not removed from the tuple-space. The use of the destructive read **in** is performed in the forth step, removing the tuple $\langle 42,43\rangle$ from the tuple space.

In this chapter we presented existing models for coordination, namely for $\mathcal{R}$eo and Linda, focusing in synchronisation and in dataflow constraints. To avoid describing our approach to distribute all of these models, we will introduce in the next chapter a more general model, the stepwise coordination model. We will show how to instantiate each of the models described in this chapter into the stepwise model, which captures only the relevant concepts of our distributed approach.

Chapter 3

A stepwise coordination model

3.1 Introduction

In this thesis we study the distributed implementation for a class of coordination languages, in particular the $\mathcal{R}$eo coordination language. Having presented some concrete coordination models in Chapter 2, we take a step back and present a more abstract model that focuses on the aspects of coordination relevant for distributed implementation. We call this model the *stepwise coordination model*, where the coordination behaviour is described by a state-based formalism which we call *behavioural automata*. The goal of the stepwise coordination model is to justify the assumptions required by the Dreams framework, which is largely independent of all the specific semantic models mentioned for $\mathcal{R}$eo. The Dreams framework is described later in Chapters 6 and 7. Also, we want to accommodate several existing concrete models that can be implemented in the Dreams framework. For example, the constraint automata model for $\mathcal{R}$eo can naturally be formulated in the stepwise coordination model, by considering each transition in the stepwise coordination model to correspond to a specific dataflow in a $\mathcal{R}$eo connector. Our model can also capture other aspects of coordination, such as context sensitivity, and does not require the assumption of a finite state space.

The stepwise coordination model serves as the basis for an implementation. The presentation of the model borrows ideas from the Tile model [53, 14], distinguishing evolution in time (execution of the coordination system) and evolution in space (composition of coordination systems). The key aspects of the stepwise coordination model are *composability* of the coordination process and the *atomicity* of the execution of actions.

Composability We say a model is composed if it results from the composition of smaller building blocks. However, not every composed model possesses the property of composability. The composability property holds for a model based on a set of *attributes of interest*: a model is composed if it allows the attributes

of interest of a composed model to be derived as a composition of the attributes of interest of its building blocks. In the present setting, the attributes of interest concern where and which data flow.

Atomicity Coordination occurs in discrete steps, called *rounds*. Each round requires the consensus of a number of building blocks to be performed atomically.

We postpone the discussion of how the notions of composability and atomicity relate to similar notions in the literature until §3.7.

We start by briefly describing the main concepts underlying the stepwise coordination model, and how the model will be used. The stepwise model is an exogenous coordination model, that is, we distinguish the elements being composed—the components—from the part coordinating the communication of these—the coordinator. See Figure 3.1(a). In the stepwise model, the behaviour of a system is described by behavioural automata. We take a bottom-up approach and start by describing *atomic steps* and *concurrency predicates*, the core ingredients of behavioural automata, before presenting the full definition of behavioural automata and its composition operator.

The stepwise model is *composed* because the behavioural automaton of the coordinator can be composed of more primitive behavioural automata. We represent this composition by the binary operator $\cdot \bowtie \cdot$. In Figure 3.1(b) we give a graphical representation for a possible definition of the coordinator as the behavioural automaton b, resulting from the product of simpler behavioural automata, denoted by $b_1 \bowtie b_2 \bowtie b_3$.

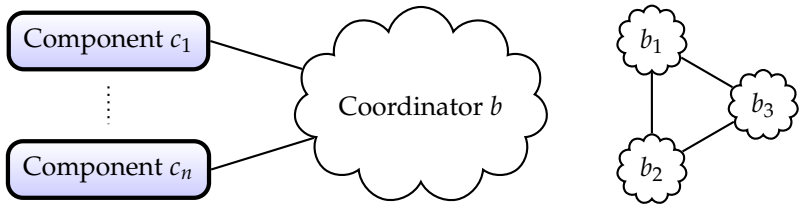


Figure 3.1: Diagrams representing (a) a system where n components are attached to a single coordinator with behavioural automaton b, and (b) a more fine grained representation of the coordinator, whose behavioural automaton is given by the product $b_1 \bowtie b_2 \bowtie b_3$.

A behavioural automaton consists of a labelled transition system where labels are atomic steps, and each state has an associated concurrent predicate that guides the composition of atomic steps. More precisely, we say a behavioural automaton is a tuple

$$\langle Q, \mathsf{L}, \rightarrow, \mathcal{O} \rangle ,$$

where Q is a set of states, L is a set of labels, $\rightarrow \subseteq Q \times \mathsf{L} \times Q$ is a transition relation, and $\mathcal{O} : Q \rightarrow \mathsf{CP}$ maps each state to its observation in CP, which stands for 'concurrency predicates' over labels and will be used to guide the composition of behavioural automata. We will define atomic steps and concurrency predicates based on labels of transition systems from L. The *atomicity* aspect is captured by the labels of behavioural automata transitions. A label represents a step in the system that has to be performed atomically. The composition of behavioural automata is based on the composition of labels and on concurrency predicates, as we explain later.

Organisation of the chapter We provide some preliminaries in §3.2 where we introduce our notation and recall the definition of labelled transition systems. We define labels, atomic steps, and concurrency predicates in §3.3, and present behavioural automata in §3.4. We discuss some practical consequences of our model in §3.5, namely locality properties and how to group similar atomic steps. In §3.6 we describe the instantiation process of concrete coordination languages as behavioural automata, including the encoding of constraint automata in §3.6.1, the encoding of normalised $\mathcal{R}$eo automata in §3.6.2, and the encoding of the two semantics for Linda-Calculus in §3.6.3. We wrap up with some discussion and conclusions in §3.7 and §3.8, respectively.

3.2 Preliminaries

We consider the coordination of a system as a set of constraints over the exchanges of data observed on a given set of ports and their flow between these ports. A *port* is an abstraction of a variable or an object that produces, consumes, or transports data, used to communicate values between elements of a coordination system. We assume a global set of ports, written as $\mathbb{P}$, and for each $x \in \mathbb{P}$ we write $\widehat{x}$ to denote the data value flowing through the port x in a specific round.[1] We write $\mathbb{D}$ to represent a global (possibly infinite) data domain. Given a set of ports $P \subseteq \mathbb{P}$, we define $\widehat{P} = \{\widehat{x} \mid x \in P\}$ to be the set of values $\widehat{x} \in \mathbb{D}$ that flow through the ports in P. A set of ports is often referred to as an alphabet.

Notation $\mathbf{2}^A$ denotes the powerset of A. $f : A \rightarrow B$ is a function with domain A and codomain B. $A \rightharpoonup B$ is a partial mapping from A to B, and is a shorthand for $A \rightarrow (\mathbf{1} + B)$ where $\mathbf{1} = \{\bot\}$. We write $\{a_1 \mapsto b_1, \ldots, a_n \mapsto b_n\}$ to denote a partial map m in $A \rightharpoonup B$, where $a_i \in A$, $b_i \in B$, $m(a_i) = b_i$, and $m(a) = \bot$ for $a \in A \setminus \{a_1, \ldots, a_n\}$. We also use the shorthands $\{a_1, \ldots, a_n \mapsto b_1, \ldots, b_n\}$ and

[1] In the setting of automata models for $\mathcal{R}$eo one can interpret the set $\mathbb{P}$ as the set $\mathcal{N}$ of port names, presented in Sections 2.2.2 and 2.2.3.

$\{a_i \mapsto b_i\}_{i=1}^{n}$ to denote the partial mapping $\{a_1 \mapsto b_1, \ldots, a_n \mapsto b_n\}$. For a partial mapping $m : A \rightharpoonup B$, we write $dom(m)$ to denote the elements in A for which m is defined, and $codom(m)$ to denote the set B. For a pair of mappings m_1, m_2, we write $m_1 \frown m_2$ to denote that $\forall x \in dom(m_1) \cap \mathrm{dom}(m_2) \cdot m_1(x) = m_2(x)$, and say m_1 and m_2 are compatible.

We represent the evolution over time of a coordination system by a labelled transition system (LTS) with an observation function of states, where the labels of the transition relation range over a set Lb, and its states have observables in a set Obs. The labels denote steps performed by the coordination system, while observations denote properties of each state that we use later to compose behavioural automata.

3.2.1. DEFINITION (LTS). A labelled transition system over the labels Lb and the observations Obs is a tuple $\langle Q, Lb, Obs, \rightarrow, \mathcal{O} \rangle$ where:

- Q is a set of states,
- Lb is a set of labels,
- Obs is a set of observations of states,
- $\mathcal{O} : Q \to Obs$ is an observation function, and
- $\rightarrow \subseteq Q \times Lb \times Q$ is a transition relation. $\lhd$

An example of a labelled transition system is the classical non-deterministic automaton, where the labels of each transition are taken from some alphabet Lb, and each state has an observable value of *true* or *false*, indicating whether the state is final or not. The stepwise coordination model we propose is a refinement of an LTS, where labels correspond to possible atomic executions of the system, and observations are conditions required to compose LTS's, represented by concurrency predicates, both notions defined below.

3.2.2. EXAMPLE (ALTERNATING COORDINATOR). We present a toy example of an *alternating coordinator* to illustrate the concepts introduced so far. The alternating coordinator is represented by AC in Figure 3.2(a). It receives data from two different data writers W_1 and W_2, and sends data to a reader R. The components W_1, W_2 and R are connected, respectively, to the ports a, b and c of the alternating coordinator. The alternating coordinator describes how data can flow between the connectors, and coordination is specified by the labelled transition system depicted in Figure 3.2(b). Each transition of this labelled transition system represents a possible step in time of the coordinator AC, describing how the ports a, b, and c can have dataflow. Initially, the coordinator is in state q_0, where only one step can be performed. This step consists of reading a value w from W_1 through a and

sending it to the reader R through c, while reading and buffering a value v sent by W_2 through b. Note that if only one of the writers can produce data, the step cannot be taken, and the system cannot evolve. In the next state, q_1, also only one step is possible. In this step, the value v is sent to the reader R, and the coordinator returns to state q_0. The arrows between states (⟶) represent the transition relation $\rightarrow$. In both states there is the possibility of allowing the concurrent execution of other automata, provided that this execution does not interfere with the current behaviour. The conditions of when other automata can execute concurrently are captured by the observation function $\mathcal{O}$, represented in the diagram by squiggly arrows (⇝) from the corresponding state. We will continue the example later. ◊

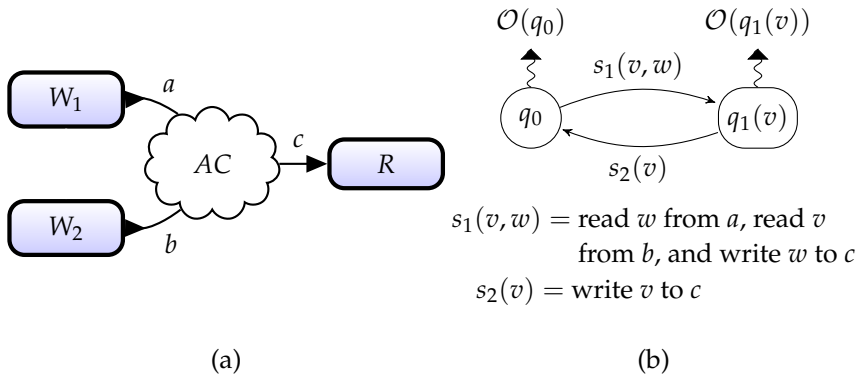


Figure 3.2: Alternating coordinator (a), and its behavioural automaton (b).

3.3 Atomic steps and concurrency predicates

The key ingredients of behavioural automata are atomic steps and concurrency predicates. Each label of a behavioural automata has an associated atomic step in the set AS, which captures the aspects such as the ports that have flow and the data flowing though them, and each state has an associated concurrency predicate, which describes which labels from other systems executing in parallel can be performed independently. We start by describing atomic steps and then introduce concurrency predicates.

3.3.1 Labels and atomic steps

An *atomic step* is an abstraction of a basic action that a system can perform. The notion of an atomic step plays a fundamental role in the definition of a behavioural

automaton. We assume a global set of labels L for behavioural automata, a global set of atomic steps AS, and a global function $\alpha : \mathsf{L} \to \mathsf{AS}$ that associates each label to an atomic step. Recall also that $\mathbb{P}$ is a global set of ports and $\mathbb{D}$ is a global set of data values. We characterise the composition of labels based on their atomic steps, and define the composition of behavioural automata based only on the composition of atomic steps and on concurrency predicates. An atomic step is defined over a set of ports $P \subseteq \mathbb{P}$, and describes which ports can have dataflow, and how data on different ports relate to each other.

3.3.1. DEFINITION (ATOMIC STEP). An *atomic step* over the alphabet P is a tuple $\langle P, F, IP, OP, data \rangle$ where:

- $P \subseteq \mathbb{P}$ is a set of ports;
- $F \subseteq P$ is the flow set;
- $IP \subseteq F$ is a set of relevant input ports;
- $OP \subseteq F$ is a set of relevant output ports;
- $IP \cap OP = \emptyset$ relevant input and output ports are disjoint; and
- $data : (IP \cup OP) \to \mathbb{D}$ is a data function. $\triangleleft$

The flow set F is a set of ports that *synchronise*, i.e., that have data flowing in the same atomic step. The sets IP and OP represent the input and output ports of the atomic step that have dataflow, and whose values are considered to be relevant when performing a step. Ports in F but not in IP or OP are ports with dataflow, but whose data values are not relevant, that is, they are used only for imposing synchronisation constraints. The data values that flow through the relevant ports are given by the data function *data*. We distinguish IP and OP to capture the notion of data dependencies that we use in the Dreams framework, and we do not relate these sets with source or sink ports. The intuition is that input ports are matched against output ports from elsewhere, and every input port must be given a value by a matched output port or an explicit environment function. Later in this chapter we explore the use of input ports and output ports to capture data dependency in more detail.

Notation We write $\mathsf{AS}[P]$ to denote the set of all atomic steps over the ports in P, and $\mathsf{L}[P]$ to denote the set of all labels ℓ such that $\alpha(\ell) \subseteq \mathsf{AS}[P]$. We use AS and L as shorthands for $\mathsf{AS}[\mathbb{P}]$ and $\mathsf{L}[\mathbb{P}]$, respectively. For simplicity, we write $a_1 \ldots a_n$ instead of $\{a_1, \ldots, a_n\}$ in the definition of atomic steps when the intention is clear from the context.

Consider the example of a set of labels with flow on the ports a, b, c and x, y, whose intended behaviour is to receive data, natural numbers say, from a, b and c,

and to produce data through x and y. Furthermore, the data value produced on x should be the maximum of the data values obtained through a and b, if available, and data should flow from c to y directly. If x does not produce data values then there is no dataflow. The relevant input and output ports are therefore defined as $IP = \{a, b, c\}$ and $OP = \{x, y\}$, respectively. We choose the flow set to be $F = IP \cup OP$, and assume that P is a set of ports containing at least F. Finally, we define the data domain $\mathbb{D}$ to be $\mathbb{N}$, and the data function $data[v_1, v_2, v_3]$ as follows:

$$data[v_1, v_2, v_3] = \{a \mapsto v_1, b \mapsto v_2, c \mapsto v_3, x \mapsto max(v_1, v_2), y \mapsto v_3\} \tag{3.1}$$

where *max* is the standard maximum function on $\mathbb{N}$. We characterise these labels by the set S of their associated atomic steps as follows.

$$S = \{\langle P, F, IP, OP, data[v_1, v_2, v_3]\rangle \in \mathsf{AS} \mid v_1, v_2, v_3 \in \mathbb{D}\}$$

The set S reflects that data flow from a and b to x, and from c to y. Using the notation $\widehat{z}$ for the dataflow on a port z we can represent the set of atomic steps over a set of ports P as

$$\widehat{x} := max(\widehat{a}, \widehat{b}) \quad \text{and} \quad \widehat{y} := \widehat{c}$$

concisely describing the choice of the data values, $max(\widehat{a}, \widehat{b})$ and $\widehat{c}$, flowing on x and y in terms of the dataflow on a, b, and c.

Composition of labels

Let $\alpha : \mathsf{L} \to \mathsf{AS}$ be a mapping from labels in L to atomic steps in AS, and let $\ell_1, \ell_2 \in \mathsf{L}$ be two labels. We assume to be given a binary composition operator $\otimes$ on the set of labels L, such that certain properties hold for $\alpha(\ell_1 \otimes \ell_2)$, described below. We also write $\alpha(\ell_1) \otimes \alpha(\ell_2)$ to denote the atomic step $\alpha(\ell_1 \otimes \ell_2)$.

Composing ℓ_1 and ℓ_2 can be intuitively understood as "what is the step for a system where part of it can perform ℓ_1 and another part can perform ℓ_2". Such a composition may not always be possible. Hence, $\otimes : \mathsf{L}^2 \rightharpoonup \mathsf{L}$ is a partial mapping. Moreover, for technical convenience, we require $\otimes$ to be associative and to have an identity element, denoted by id_{L}. Thus, $(\mathsf{L}, \otimes)$ is assumed to be a *commutative partial monoid* $\langle \mathsf{L}, \otimes\rangle$.[2] Below we will require for two labels ℓ_1 and ℓ_2 in L with atomic steps $\alpha(\ell_1) = \langle P_1, F_1, IP_1, OP_1, data_1\rangle$, $\alpha(\ell_2) = \langle P_2, F_2, IP_2, OP_2, data_2\rangle$, the composition $\ell_1 \otimes \ell_2$ to be defined (but not necessarily exclusively) if the alphabets P_1 and P_2 are disjoint. We say ℓ_1 and ℓ_2 are *incompatible* if $\ell_1 \otimes \ell_2$ is undefined, which means that ℓ_1 and ℓ_2 cannot be performed simultaneously. If ℓ_1 and ℓ_2 are compatible, then $\alpha(\ell_1 \otimes \ell_2) = \langle P, F, IP, OP, data\rangle$ satisfies:

[2]Not to be confused with partially commutative monoids, or partially trace monoids, where *commutativity* is partial [80].

- $P \subseteq P_1 \cup P_2$;
- $F \subseteq F_1 \cup F_2$;
- $IP \subseteq (IP_1 \cup IP_2) \backslash (OP_1 \cup OP_2)$;
- $OP \subseteq OP_1 \cup OP_2$; and
- $data_1 \frown data_2$;
- $data = data_1 \cup data_2$.

Based on these conditions we require the identity to be id_{L} to be such that $\alpha(id_{\mathsf{L}}) = \langle \emptyset, \emptyset, \emptyset, \emptyset, \emptyset \rangle$. We introduced the notation $m_1 \frown m_2$ in §3.2 to represent that the values of the common domain of mappings m_1 and m_2 match. The requirements on the sets IP and OP reflect that when composing two atomic steps, the input ports that have an associated output port are no longer treated as input ports (since the dependencies have been met), and the output ports are combined.

3.3.2. EXAMPLE. Consider three labels ℓ_1, ℓ_2 and ℓ_3 with atomic steps $\alpha(\ell_i) = s_i$ defined below, where $\mathbb{D} = \mathbb{N}$ and where we write abc instead of $\{a, b, c\}$, etc.

$$\begin{aligned} s_1 &= \langle ab, ab,\ a, b, \{a, b \mapsto 3, 5\} \rangle \\ s_2 &= \langle bc, bc,\ b, c, \{b, c \mapsto 5, 7\} \rangle \\ s_3 &= \langle cd,\ \ c, \emptyset, c, \{c \mapsto 6\} \rangle \end{aligned}$$

It is possible to define $\otimes$ such that $s_1 \otimes s_2 = \langle abc, abc, a, bc, \{a, b, c \mapsto 3, 5, 7\} \rangle$. An alternative definition can also yield $s_1 \otimes s_2 = \langle abc, ac, a, c, \{a, c \mapsto 3, 7\} \rangle$. In the alternative definition we ignore the flow of data on b after the composition, because it has become internal. However, $s_2 \otimes s_3$ has to be undefined, because their data functions are not compatible: $\{b, c \mapsto 5, 7\} \not\frown \{c \mapsto 6\}$. The opposite holds for the composition $s_1 \otimes s_3$, which has to be always defined because $ab \cap cd = \emptyset$. ◊

3.3.2 Concurrency predicates

We introduced atomic steps associated to labels of behavioural automata by a function α. We now introduce *concurrency predicates* to represent observations of states by a function $\mathcal{O}$. The intuition is that labels from two systems can be executed together (composing their steps using $\otimes$), but they can also be executed in parallel, independent of each other, using the concurrency predicates to control when a label from a system can be interleaved with those from the other system. Recall that we assume a global set of ports $\mathbb{P}$, a global set of labels $\mathsf{L} = \mathsf{L}[\mathbb{P}]$, and a global set of atomic steps $\mathsf{AS} = \mathsf{AS}[\mathbb{P}]$.

3.3.3. DEFINITION (CONCURRENCY PREDICATE). A *concurrency predicate* is a set of labels in L. We write CP to denote the set $\mathbf{2}^{\mathsf{L}}$ of all concurrency predicates. For $C \in \mathsf{CP}$ and $\ell \in \mathsf{L}$, we write $C(\ell)$ to denote $\ell \in C$. ◁

For a given state q, an observation function $\mathcal{O}$, and a label ℓ, the concurrency predicate $\mathcal{O}(q)(\ell)$ can be understood as "the label ℓ from another behavioural automata can be executed". The name "concurrency predicate" reflects the main goal of these observations: concurrency of systems. More specifically, the concurrency predicate describes when two systems can execute independently, by listing for each state the set of labels that do not affect the choice of the next transition. The composition of concurrency predicates is simply the intersection of sets, i.e., the composition of two concurrency predicates C_1 and C_2 is $C_1 \cap C_2$. Unlike the composition of labels, the composition of concurrency predicates is total, i.e., any two concurrency predicates can be composed.

The motivation for introducing concurrency predicates is twofold. First, it captures a similar notion from various $\mathcal{R}$eo models: the constraint automata model assumes that every state has the transition $\emptyset, \mathtt{tt}$ (Definition 2.2.3), and the $\mathcal{R}$eo automata model uses the condition $q^{\#}$ when composing automata (Definition 2.2.12). We believe that this separation between steps and concurrency predicates is more natural than assuming the existence of specific steps to make the composition behave as expected. Second, by making explicit a condition that describes when steps can be executed concurrently, we do not need to globally evaluate the coordination constraints. Hence, certain labels can be executed locally in a *distributed environment* independently of other systems running in parallel.

Notation In most of the following examples it is enough to define the observation of a state based on the atomic steps of the labels of other systems. For these cases, we introduce a notation to quickly define a concurrency predicate consisting of all labels that do not have dataflow in a set of ports P_0. We write $cp(P_0)$ to denote the concurrency predicate

$$cp(P_0) \;=\; \{\ell \mid \alpha(\ell) = \langle P, F, IP, OP, data\rangle \,, P_0 \cap F = \emptyset\}\,. \tag{3.2}$$

Example

Recall the running example of the alternating coordinator from §3.2 on page 42. In Figure 3.2(b) we described each label and observation informally. In Table 3.1 we define each of these textual labels as atomic steps and concurrency predicates. In all transitions, the alphabet P is always $\{a, b, c\}$, and the observations only allow steps where none of the known ports has flow.

When composing two behavioural automata, the composition of labels is used to combine all possible pairs of labels from the two automata, and the concurrency predicates are used to include all labels that can be executed independently. In the next section we study in more detail labelled transition systems where labels have associated atomic steps and states have associated concurrency predicates, assuming a possible definition for the composition of atomic steps.

Labels		Atomic steps
$s_1(v,w)$	–	$\langle P, abc, ab, c, \{a,b,c \mapsto w,v,w\}\rangle$
$s_2(v)$	–	$\langle P, \quad c, \emptyset, c, \{c \mapsto v\}\rangle$

States		CP
q_0	–	$cp(P)$
$q_1(v)$	–	$cp(P)$

Table 3.1: Atomic steps of the labels and concurrency predicates of the states of the alternating coordinator, where $v, w \in \mathbb{D}$.

3.4 Behavioural automata

Labels describe each round of the dynamic behaviour of a system. Recall that we assume a global set of ports $\mathbb{P}$ and a global data domain $\mathbb{D}$. We now assume also a global set of labels $\mathsf{L} = \mathsf{L}[\mathbb{P}]$ with atomic steps $\mathsf{AS} = \mathsf{AS}[\mathbb{P}]$, and with a composition operator $\otimes$. Let α be the function that associates labels to atomic steps. We refine the definition of LTS introduced in §3.2, using labels from L in the transition relation and concurrency predicates from $\mathbf{2}^{\mathsf{L}}$ as observations. We call the resulting labelled transition systems *behavioural automata*.

3.4.1. DEFINITION (BEHAVIOURAL AUTOMATA). A *behavioural automaton* of a system over a set of ports $P \subseteq \mathbb{P}$ is a labelled transition system

$$\langle Q, \mathsf{L}[P], \rightarrow, \mathcal{O}\rangle\,,$$

where $\mathsf{L}[P]$ is the set of labels over P, $\rightarrow \subseteq Q \times \mathsf{L}[P] \times Q$ is the transition relation, and $\mathcal{O} : Q \rightarrow \mathbf{2}^{\mathsf{L}}$ is the observation function that maps states to concurrency predicates. ◁

Note that we omit set of possible observations with respect to Definition 3.2.1 of labelled transition systems, because it is always $\mathsf{CP} = \mathbf{2}^{\mathsf{L}}$. The set CP is fixed because of the assumption of a global set of atomic steps AS. Similarly, we do not include the function $\alpha : \mathsf{L} \rightarrow \mathsf{AS}$. We also use the notation $q_1 \xrightarrow{\ell} q_2$ denoting $\langle q_1, \ell, q_2\rangle \in \rightarrow$. The observation function $\mathcal{O}$ returns a concurrency predicate for each state q, such that for any label ℓ from another behavioural automata, $\mathcal{O}(q)(\ell) = \mathit{true}$ means that $\ell \in \mathcal{O}(q)$, that is, ℓ can execute while the current behavioural automaton is in state q. We will investigate the local execution of a behavioural automaton further in this section, exploring properties of the observation function.

Note that no assumption is made regarding the cardinality of the state set Q, the set of ports P, and the transition relation $\rightarrow$. On one hand, assuming a finite world brings us to finite automata models, where we can formally verify a large class of properties. On the other hand, we can also assume a *black box* scenario, where the set of states, ports, or relations is not provided a priori, or even scenarios where these are calculated and given on demand as the system evolves. This perspective has the advantage of allowing the execution of systems that are

not completely specified. We choose to allow infinite sets of states, ports, and transitions. Composition can be performed in a stepwise fashion, that is, we can calculate the possible transitions from the current state resulting from composing a set of behavioural automata, considering only the transitions from the current state of each of the behavioural automata involved in the composition. We refer to a subpart of a behavioural automaton that contains only the states, transitions, and observations that involve a given state, called the *frontier*, defined as follows.

3.4.2. DEFINITION (FRONTIER). Let $b = \langle Q, \mathsf{L}[P], \rightarrow, \mathcal{O} \rangle$ be a behavioural automaton and $q \in Q$ a state of b. We define $b\rfloor_q \mathrel{\widehat{=}} \langle Q', \mathsf{L}[P], \rightarrow', \mathcal{O}' \rangle$ to be the behavioural automaton restricted to the state q, called the *frontier* of b at q, where

- $Q' = \{q\} \cup \{q' \mid \exists \ell \cdot q \xrightarrow{\ell} q'\}$
- $\rightarrow' = \{\langle q, \ell, q' \rangle \mid q \xrightarrow{\ell} q'\}$
- $\mathcal{O}' = \mathcal{O} \restriction Q'$. ◁

Note that the composition of frontiers, defined bellow, will be again a frontier. The operator $\restriction$ denotes the standard restriction of functions. It is possible to define a *trace* of a behavioural automaton as a sequence of atomic steps, to consider all traces of a behavioural automaton for a given state, yielding the *language* accepted by a behavioural automaton, and setting the stage for the notion of bisimulation. We will not do this at the abstract level of a behavioural automaton. Instead, we work out these details for more concrete models below, where more is known about the definition of atomic steps.

3.4.1 Product of behavioural automata

We now describe the composition of behavioural automata based on the composition $\otimes$ of labels and on the concurrency predicates associated to states via the observation function $\mathcal{O}$. This composition mimics the composition of existing $\mathcal{R}$eo models [22, 37, 29].

3.4.3. DEFINITION (PRODUCT OF BEHAVIOURAL AUTOMATA).
The product of two behavioural automata $b_1 = \langle Q_1, \mathsf{L}[P_1], \rightarrow_1, \mathcal{O}_1 \rangle$ and $b_2 = \langle Q_2, \mathsf{L}[P_2], \rightarrow_2, \mathcal{O}_2 \rangle$, denoted by $b_1 \bowtie b_2$, is the behavioural automaton $\langle Q_1 \times Q_2, \mathsf{L}[P_1 \cup P_2], \rightarrow, \mathcal{O} \rangle$, where $\rightarrow$ and $\mathcal{O}$ are defined as follows:

$$\rightarrow = \left\{ \langle (p,q), \ell, (p',q') \rangle \mid p \xrightarrow{\ell_1}_1 p', q \xrightarrow{\ell_2}_2 q', \ell = \ell_1 \otimes \ell_2, \ell \neq \bot \right\} \tag{3.3}$$

$$\cup \left\{ \langle (p,q), \ell, (p',q) \rangle \mid p \xrightarrow{\ell}_1 p', \ell \in \mathcal{O}_2(q) \right\} \tag{3.4}$$

$$\cup \left\{ \langle (p,q), \ell, (p',q) \rangle \mid q \xrightarrow{\ell}_2 q', \ell \in \mathcal{O}_1(p) \right\} \tag{3.5}$$

$$\mathcal{O}(p,q) = \mathcal{O}_1(p) \cap \mathcal{O}_2(q) \quad \text{for} \quad p \in Q_1, q \in Q_2 \tag{3.6}$$

◁

Case (3.3) composes all possible combinations of transitions from $\rightarrow_1$ and $\rightarrow_2$, reflecting both behavioural automata executing in parallel. Cases (3.4) and (3.5) cover the situation where one of the behavioural automata performs a step admitted by the concurrency predicate of the other. Finally, case (3.6) defines the composition of two concurrency predicates, given by their intersection.

The empty behavioural automaton $\langle \{q\}, \emptyset, \emptyset, \mathcal{O} \rangle$, where $\mathcal{O}(q) = \mathsf{L}$, is denoted by $\mathbf{0}_{\mathsf{BA}}$. It is an identity of the operator $\bowtie$ up to isomorphism, defined below. The isomorphism captures the fact that the state names can differ. We formalise the properties of $\mathbf{0}_{\mathsf{BA}}$ in Proposition 3.4.5.

3.4.4. DEFINITION (ISOMORPHISM OF BEHAVIOURAL AUTOMATA).
Given two behavioural automata $b_1 = \langle Q_1, \mathsf{L}[P_1], \rightarrow_1, \mathcal{O}_1 \rangle$ and $b_2 = \langle Q_2, \mathsf{L}[P_2], \rightarrow_2, \mathcal{O}_2 \rangle$ we say that b_1 and b_2 are isomorphic, written as $b_1 \simeq b_2$, *iff* $P_1 = P_2$ and there is a bijection $f : Q_1 \rightarrow Q_2$ such that:

- $q \xrightarrow{\ell}_1 q'$ if and only if $f(q) \xrightarrow{\ell}_2 f(q')$; and
- $\mathcal{O}_1(q) = \mathcal{O}_2(f(q))$. ◁

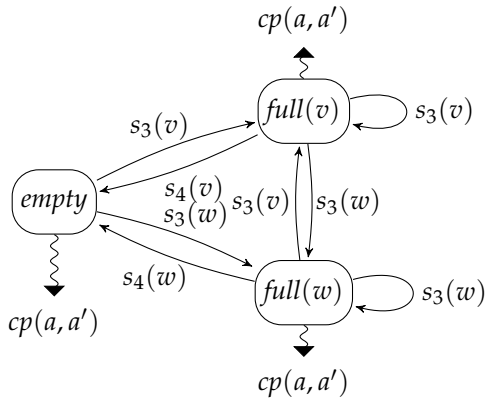
3.4.5. PROPOSITION. *We have that* $\mathbf{0}_{\mathsf{BA}} \bowtie b \simeq b$ *and* $b \bowtie \mathbf{0}_{\mathsf{BA}} \simeq b$, *for every behavioural automaton* b. □

3.4.2 Example: lossy alternator

Recall the behavioural automaton AC of our running example of the alternating coordinator, illustrated in Figure 3.2, whose atomic steps were formalised in Table 3.1. The alternating coordinator has two source ports, a and b, and a sink port c. Data is received always via ports a and b simultaneously, and sent via port c, alternating the values received from a and b. We now imagine the following scenario: the data on a becomes available always at a much faster rate than data on b. To adapt our alternating coordinator to this new scenario, we introduce a lossy-FIFO$_1$ connector LF and combine it with the alternating coordinator, yielding a variation of the alternating coordinator $LF \bowtie AC$.

Recall the definition of $cp : \mathbb{P} \rightarrow \mathsf{L}$ given by Equation (3.2), on page 47. The behavioural automaton for the lossy-FIFO$_1$ connector is depicted in Figure 3.3, and its atomics steps range over the set of ports $\{a, a'\}$, where a is an input port and a' is an output port.[3] Recall also that the alternating coordinator ranges over the set of ports $\{a, b, c\}$. We depict the interface of both of these connectors on top of Figure 3.4. After combining the behavioural automata of the two connectors, they

[3]This lossy-FIFO is also known in the literature [8] as the shift-lossy FIFO$_1$.



where:
$s_3(v) = \langle aa', a', a', \emptyset, \{a' \mapsto v\}\rangle$
$s_4(v) = \langle aa', a, \emptyset, a, \{a \mapsto v\}\rangle$

Figure 3.3: Behavioural automaton of the lossy-FIFO connector.

become connected via their shared port a. The new variation of the alternating coordinator can then be connected to writers and readers by using the ports a', b and c, as depicted at the bottom of Figure 3.4.

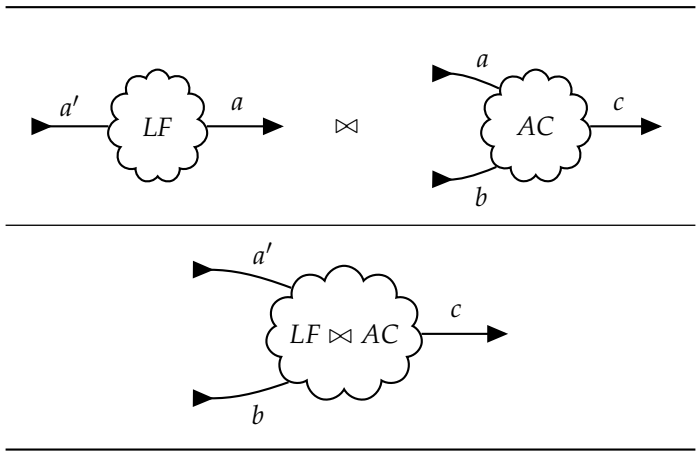


Figure 3.4: The sink and source ports of LF, AC, and their composition.

Intuitively, the lossy-FIFO connector receives data from the source port a' and buffers its value before sending it through port a. When the buffer is full it can also receive data from a', and the content of the buffer is replaced by the new received value. The connector resulting from the composition $LF \bowtie AC$ is formalised in Table 3.2 and in Figure 3.5. The flow sets of the atomic steps of the labels $s_1(v, w)$, $s_2(v)$, $s_3(v)$ and $s_4(v)$ are, respectively, abc, c, a', and $a'a$. The set of known ports in this example is always $P = \{a', a, b, c\}$. Let $\mathcal{O}_{LF}$ be the observation function of LF, and $\mathcal{O}_{AC}$ be the observation function of AC. The observation function $\mathcal{O}_{LF \bowtie AC}$

$\otimes$	$s_1(u,v)$	$s_2(w)$
$s_3(y)$	$\bot$	$\langle P, a'c, a', c, \{a', c \mapsto y, w\}\rangle$
$s_4(z)$	$\bot$ (for $z \neq v$)	$\bot$
$s_4(v)$	$\langle P, abc, ab, c, \{a, b, c \mapsto v, u, v\}\rangle$	$\bot$

LF	$\mathcal{O}_{LF}(empty)$	$\mathcal{O}_{LF}(full(v'))$
$s_1(v,w)$	*false*	*false*
$s_2(v)$	*true*	*true*

AC	$\mathcal{O}_{AC}(q_0)$	$\mathcal{O}_{AC}(q_1(v'))$
$s_3(v)$	*true*	*true*
$s_4(v)$	*false*	*false*

Table 3.2: Atomic steps of the composition of labels from LF and AC (top), and verification of the concurrency predicate for each label (below).

for $LF \bowtie AC$ results from the intersection of the observations of the states of each behavioural automaton, and corresponds precisely to the concurrency predicate $cp(a', a, b, c)$. Each state in $LF \bowtie AC$ uses two names, the first is the name of a state from LF, and the second is the name of a state from AC. For simplicity, we colour two of the states in grey in Figure 3.5 to represent that these two states that are closely related to their nearest state, and therefore not completely specified. The main difference between a grey state and the state right next to it is the value v in $full(v)$. We only draw a few transitions to these disregarded states, to help to understand the role of these states, without cluttering the diagram with several similar transitions.

From the diagram it is clear that some transitions originate only from the LF or the AC connector, while others result from the composition via the operator $\otimes$. One can see that the transitions $s_2(v)$ and $s_3(w)$ can be performed simultaneously or interleaved; simultaneously because $s_2(v) \otimes s_3(w)$ is defined, and interleaved because $\mathcal{O}_{LF}$ applied to any of the states of LF contains $s_2(v)$ and $\mathcal{O}_{AC}$ applied to any of the states of AC contains $s_3(w)$. The possible execution scenarios of these atomic steps follows our intuition that steps 'approved' by concurrency predicates can be performed independently. The steps $s_1(u,v)$ and $s_4(w)$ can be taken only when composed, and their composition exists only when the values v and w match.

3.5 Locality and grouping of atomic steps

We introduce the notion of locality as a property of behavioural automata that allows ports from different behavioural automata to have *always* dataflow through them. Recall that we assume the existence of universal sets of ports $\mathbb{P}$, labels L, and atomic steps AS.

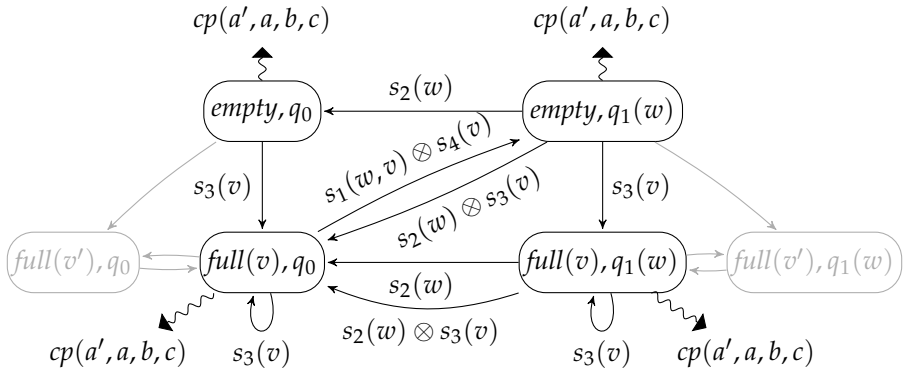


Figure 3.5: Behavioural automaton for the composition of LF and AC.

3.5.1. DEFINITION (LOCALITY OF BEHAVIOURAL AUTOMATA).
A behavioural automaton $b = \langle Q, \mathsf{L}[P], \rightarrow, \mathcal{O} \rangle$ obeys the *locality property* if, for any port set P' such that $P \cap P' = \emptyset$, the following condition holds.

$$\forall \ell \in \mathsf{L}[P'] \cdot \forall q \in Q \cdot \ell \in \mathcal{O}(q). \qquad \lhd$$

Any two behavioural automata with disjoint port sets that obey the locality property can therefore evolve in parallel in an interleaved fashion. Let $b = b_1 \bowtie b_2$ be a behavioural automaton, and ℓ a step from b_1. We say ℓ is a *local step* of b if $(q_1, q_2) \xrightarrow{\ell} (q_1', q_2')$ is a transition of b and either $q_1 \xrightarrow{\ell}_1 q_1'$ and $\ell \in \mathcal{O}_2(q_2)$, or $q_2 \xrightarrow{\ell}_2 q_2'$ and $\ell \in \mathcal{O}_1(q_1)$., In the behavioural automaton exemplified in Figure 3.5, the local steps are exactly the transitions labelled by the steps $s_2(w)$ and $s_3(v)$.

3.5.2. PROPOSITION. *Let $b = b_1 \bowtie b_2 \bowtie b_3$ be a behavioural automaton where $b_i = \langle Q_i, \mathsf{L}[P_i], \rightarrow_i, \mathcal{O}_i \rangle$, for $i \in 1..3$, and assume the locality property from Definition 3.5.1 holds for b_1, b_2 and b_3. Suppose $P_1 \cap P_3 = \emptyset$. Then, for any step $\ell_1 \in \mathsf{L}[P_1]$ performed by b_1 and $q_2 \in Q_2$ it holds that*

$$\ell_1 \in \mathcal{O}_2(q_2) \quad \Rightarrow \quad \ell_1 \textit{ is a local step of } b.$$

Proof. Observe that $\bowtie$ is associative, up to the state names, because the composition of labels $\otimes$ is associative. From $P_1 \cap P_3 = \emptyset$, $\ell_1 \in \mathsf{L}[P_1]$, and from the locality property in Definition 3.5.1 we conclude that $\forall q \in Q_3 \cdot \ell_1 \in \mathcal{O}_3(q)$. Therefore, for any state $q_3 \in Q_3$ and for a state $q_2 \in Q_2$ such that $\ell_1 \in \mathcal{O}_2(q_2)$, we have that $\ell_1 \in \mathcal{O}_2(q_2) \cap \mathcal{O}_3(q_3)$. We conclude that $\ell_1 \in \mathcal{O}'$, where $\mathcal{O}'$ is the observation function of $b_2 \bowtie b_3$, hence a local step of b. $\square$

Proposition 3.5.2 has a practical consequence. Consider a composed system $b_1 \bowtie \cdots \bowtie b_n$ where the locality property holds for each b_i. Whenever b_1 can

perform an atomic step it is sufficient to check the concurrency predicates of its *neighbour systems*, i.e., the behavioural automata with shared port sets.

The product of behavioural automata presented in §3.4.1 captures the data-flow on the relevant ports of a global behavioural automaton. However, when composing two behavioural automata b_1 and b_2 into $b = b_1 \bowtie b_2$ some information regarding internal ports of b might be abstracted away, since it is relevant to b_1 or b_2, but not for b. For a centralised implementation where only the global behaviour is important this abstraction is adequate. In our distributed implementation of Dreams we need to find a global behaviour b, and for each transition labelled by ℓ in b we need to find a corresponding transition labelled by ℓ_1 in b_1 and by ℓ_2 in b_2, such that $\ell_1 \otimes \ell_2 = \ell$. The underlying coordination models of our distributed implementation guarantee that we can always recover the atomic step of each component from a global atomic step. We now explore in more detail the process of recovering the atomic steps for each component.

Let $b = b_1 \bowtie \cdots \bowtie b_n$, where $b = \langle Q, \mathsf{L}[P], \rightarrow, \mathcal{O} \rangle$ and $b_i = \langle Q_i, \mathsf{L}[P_i], \rightarrow_i, \mathcal{O}_i \rangle$. A simple approach to recover the information about each of the original atomic steps is to assume the existence of a projection function that is built during the composition of $b_1, \ldots, b_n$, which maps each of the labels ℓ in the transitions of b to a set of atomic steps $\ell_1, \ldots, \ell_n$, such that each ℓ_i is a label of b_i and $\ell = \ell_1 \otimes \cdots \otimes \ell_n$. For the implementation we go further and, instead of calculating a transition ℓ for the current round, we calculate a group γ of labels that differ only on the data values being transferred, using the knowledge about which ports are input ports (IP).

In each round, our implementation calculates a group of labels γ with atomic steps with the same sets of known ports, flow ports, input ports, and output ports, which differ only on their data functions. This group is indexed by an assignment of data of the input ports, that is, the type of γ is

$$\gamma : (IP \to \mathbb{D}) \to \mathsf{L}[P] \tag{3.7}$$

where γ fulfils the condition

$$\forall m : IP \to \mathbb{D} \;\cdot\; \alpha(\gamma(m)) = \langle P, F, IP, OP, data \rangle\,, \quad \text{for some fixed } F, OP, \text{ and where } data \frown m.$$

The projection of γ to the behavioural automata $b_1, \ldots, b_n$ yields n groups $\gamma_1, \ldots, \gamma_n$ such that $\gamma_i : (IP_i \to \mathbb{D}) \to \mathsf{L}[P_i]$ is characterised as follows.

$$\forall m : IP \to \mathbb{D} \cdot \alpha(\gamma_i(m \upharpoonright P_i)) = \langle P_i, F \cap P_i, IP \cap P_i, OP \cap P_i, data \upharpoonright P_i \rangle \quad \text{and } \gamma(m) = \gamma_1(m \upharpoonright P_1) \otimes \cdots \otimes \gamma_n(m \upharpoonright P_n).$$

Recall the notation for the compatibility of maps introduced in the beginning of this chapter, where $m_1 \frown m_2$ holds when the values of m_1 and m_2 coincide for

the common domain. The intuition for these conditions is that each γ is a group of atomic steps that share the same ports, parameterised on the values flowing on their input ports. If the set $IP = \emptyset$, then γ can be regarded as a single label.

3.6 Concrete behavioural automata

The stepwise coordination model describes the coordination behaviour as a behavioural automaton. A behavioural automaton is an abstraction of concrete coordination models that focuses on relevant aspects to the execution of the coordination model. As we will argue, $\mathcal{R}$eo and Linda can be cast in our framework of behavioural automata. Therefore, both $\mathcal{R}$eo and Linda coordination models can be seen as specific instances of the stepwise model described above. For a concrete coordination model to fit into the stepwise model, we need to define:

1. labels in the concrete model;
2. the encoding α of labels into atomic steps;
3. composition of labels; and
4. concurrency predicates for states.

We start by encoding the constraint automata and the $\mathcal{R}$eo automata models as behavioural automata. Later, because of its relevance in the coordination community as one of the first coordination languages, we also encode Linda as a behavioural automaton.

3.6.1 Constraint automata as behavioural automata

Recall the constraint automata model (CA) for $\mathcal{R}$eo described in the previous chapter. A constraint automaton $\mathcal{A}$ is a tuple $\langle Q, \mathcal{N}, \rightarrow, Q_0 \rangle$ with a set of states Q, a set of port names $\mathcal{N}$, a transition relation $\rightarrow$, and a set of initial states Q_0. The transition relation is a set of triples in $Q \times \mathbf{2}^{\mathcal{N}} \times DC_{\mathcal{N}} \times Q$, where each transition label is a pair $X|dc$ that associate a set of port names X to a dataflow constraints dc. Each of these pairs $X|dc$ has a set of solutions in $\mathsf{CAS} = \mathbf{2}^{\mathcal{N}} \times DC_{\mathcal{N}}$, whose elements are written as $X|(\bigwedge_{i=1}^{n} \widehat{x_i} = d_i)$ such that $X = \{x_1, \ldots, x_n\}$ and $g = \bigwedge_{i=1}^{n} \widehat{x_i} = d_i$ is a data assignment that satisfies dc. The CA model assumes a finite data domain $\mathbb{D}$, and that data constraints such as $\mathtt{tt}$, $\widehat{a} \neq d$, or $\widehat{a} = \widehat{b}$ stand for simpler data constraints that use $\widehat{a} = d$ and the operators $\wedge$ and $\vee$.

The encoding of the constraint automaton $\mathcal{A} = \langle Q, \mathcal{N}, \rightarrow_{\mathsf{CA}}, Q_0 \rangle$ is the behavioural automaton

$$[\![\mathcal{A}]\!]_{\mathsf{CA}} = \langle Q, \mathsf{L}[\mathcal{N}], \rightarrow_{\mathsf{BA}}, \mathcal{O} \rangle$$

with $\mathsf{L}[\mathcal{N}]$, $\rightarrow_{\mathsf{BA}}$, $\mathcal{O}$, and the composition of labels as follows:

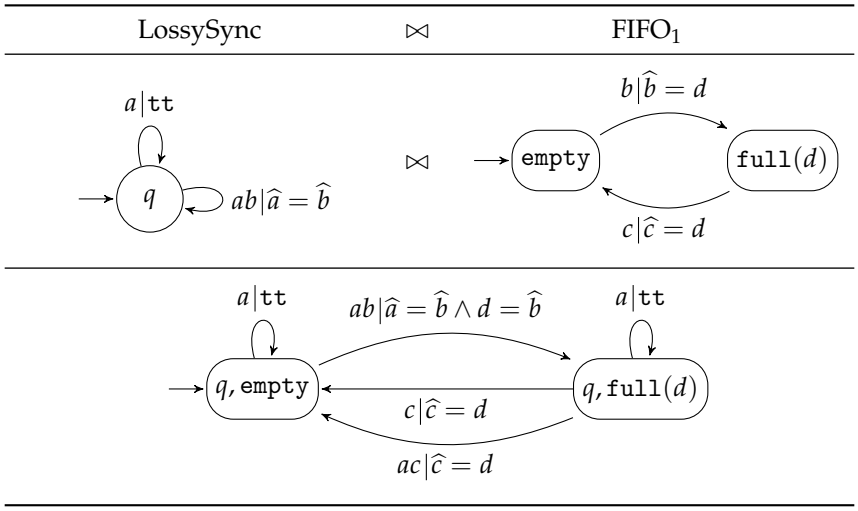


Figure 3.6: Product of the constraint automata for LossySync and FIFO_1.

- $\mathsf{L} = \mathsf{CAS}$, and α is defined as:

$$\alpha(X|\textstyle\bigwedge_{i=1}^{n} \widehat{x_i} = d_i) \;=\; \langle \mathcal{N}, X, \emptyset, X, \{x_i \mapsto d_i\}_{i=1}^{n} \rangle\,.$$

- We have $q \xrightarrow{X|g}_{\mathsf{BA}} q'$ for $X|g \in \mathsf{L}[\mathcal{N}]$ if $q \xrightarrow{X|g'}_{\mathsf{CA}} q'$ and g satisfies g'.

- Let $cas_i = X_i|g_i$ be a solution for a label from a constraint automaton with ports $\mathcal{N}_i$, where $i \in 1..2$. Then

$$cas_1 \otimes cas_2 = \begin{cases} (X_1 \cup X_2)|(g_1 \wedge g_2) & \text{if } X_1 \cap \mathcal{N}_2 = X_2 \cap \mathcal{N}_1 \;\wedge\; g_1 \frown g_2 \\ \bot & \text{otherwise} \end{cases}$$

  where $g_1 \frown g_2$ if for every shared port $x \in X_1 \cap X_2$ and for any $d \in \mathbb{D}$, $x = d$ satisfies g_1 iff $x = d$ satisfies g_2.

- $\mathcal{O}(q) = cp(\mathcal{N})$ for every $q \in Q$. Recall that $cp(\mathcal{N}) = \{\ell \mid \alpha(\ell) = \langle P, F, IP, OP, data\rangle, P_0 \cap F = \emptyset\}$, introduced in Equation (3.2).

3.6.1. EXAMPLE. We recall from Figure 3.6 the constraint automata $\mathcal{A}_L$ and $\mathcal{A}_F$ for the LossySync and the FIFO_1 channels presented in Example 2.2.4 and Example 2.2.5. Let $\mathcal{A}_L = \langle Q_L, \mathcal{N}_L, \rightarrow_1, Q_1\rangle$ and $\mathcal{A}_F = \langle Q_F, \mathcal{N}_F, \rightarrow_2, Q_2\rangle$. The encoding of $\mathcal{A}_L$ into behavioural automata is given below, and depicted in the top part of Figure 3.7.

$$[\![\mathcal{A}_L]\!]_{\mathsf{CA}} = \langle Q_L, \mathsf{L}[\mathcal{N}_L], \rightarrow_L, \mathcal{O}_L\rangle$$

where

$$\begin{aligned} Q_L &= \{q\} \qquad \mathcal{N}_L = \{a, b\} \quad \mathcal{O}_L(q) = cp(\mathcal{N}_L) \text{ for } q \in Q_L \\ s_1(v) &= ab|(\widehat{a} = v \wedge \widehat{b} = v) \qquad s_2(v) = a|(\widehat{a} = v) \\ \rightarrow_L &= \{\langle q, s_1(v), q\rangle \mid v \in \mathbb{D}\} \cup \{\langle q, s_2(v), q\rangle \mid v \in \mathbb{D}\}. \end{aligned}$$

Similarly, the encoding of $\mathcal{A}_F$ into behavioural automata is given below, and depicted on top of Figure 3.7.

$$[\![\mathcal{A}_F]\!]_{\mathsf{CA}} = \langle Q_F, \mathsf{L}[\mathcal{N}_F], \mathsf{CP}, \rightarrow_F, \mathcal{O}_F\rangle$$

where

$$\begin{aligned} Q_F &= \{\texttt{empty}\} \cup \{\texttt{full}(v) \mid v \in \mathbb{D}\} \quad \mathcal{O}_F(q) = cp(\mathcal{N}_F) \text{ for } q \in Q_F \\ \mathcal{N}_F &= \{b, c\} \qquad s_3(v) = b|(\widehat{b} = v) \quad s_4(v) = c|(\widehat{c} = v) \\ \rightarrow_F &= \{\langle \texttt{empty}, s_3(v), \texttt{full}(v)\rangle \mid v \in \mathbb{D}\} \cup \{\langle \texttt{full}(v), s_4(v), \texttt{empty}\rangle \mid v \in \mathbb{D}\} \end{aligned}$$

The product of the constraint automata for LossySync and FIFO_1 is presented on the bottom of Figure 3.6. The composition of $[\![\mathcal{A}_L]\!]_{\mathsf{CA}} \bowtie [\![\mathcal{A}_F]\!]_{\mathsf{CA}}$ yields the behavioural automaton depicted in the end of Figure 3.7, where

$$\begin{aligned} s_1(v) \otimes s_3(v) &= ab|(\widehat{a} = v \wedge \widehat{b} = v), \text{ and} \\ s_2(w) \otimes s_4(v) &= ac|(\widehat{a} = w \wedge \widehat{c} = v). \end{aligned}$$ ◊

As expected, the behavioural automaton of the composition depicted in Figure 3.7 is equivalent to the encoding of the constraint automata $\mathcal{A}_L \bowtie \mathcal{A}_F$, depicted in Figure 3.6, where each transition in constraint automata is split into several transitions in behavioural automata, one for each solution of the data constraints.

Discussion The product of the two behavioural automata presented in Example 3.6.1 is equivalent to the product of the two associated constraint automata, presented in Example 2.2.5 in the previous chapter, with respect to the atomic steps of the labels of the automata. We expect this equivalence to hold in general, but we do not give a formal proof. The atomic steps associated to the labels of behavioural automata in our encoding assume all ports with dataflow are output ports, since the constraint automata model does not capture the direction of dataflow. An existing implementation of the CA model [22, 16] uses a variation that adds memory to the state, and introduces the notion of dependencies between ports, which is similar in spirit to our use of input and output variables. In this implementation data constraints are not used when deciding the coordination patterns to be executed, and a notion of dependency between ports guides the decision of which data flows in the connector.

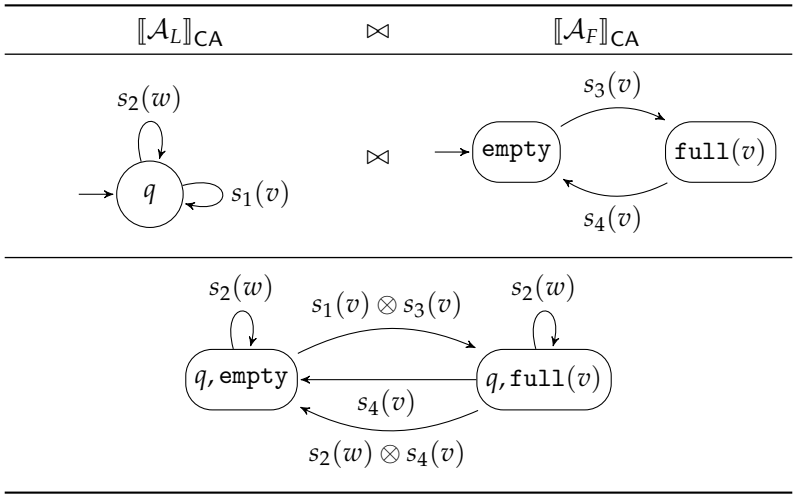


Figure 3.7: Composition of $[\![\mathcal{A}_L]\!]_{\mathsf{CA}}$ and $[\![\mathcal{A}_F]\!]_{\mathsf{CA}}$, for any $v, w \in \mathbb{D}$.

3.6.2 $\mathcal{R}$eo automata as behavioural automata

The encoding of normalised $\mathcal{R}$eo automata as behavioural automata follows the same reasoning as the formulation of constraint automata, presented in the previous subsection. Furthermore, labels in the encoded behavioural automata correspond to the labels of the original $\mathcal{R}$eo automaton, because we consider normalised transitions and there are no data constraints.

Recall the normalised $\mathcal{R}$eo automata model (RA) described in §2.2.3. A normalised $\mathcal{R}$eo automaton $\mathcal{A}$ is a triple $\langle Q, \mathcal{N}, \rightarrow \rangle$ with a set of states Q, a set of port names $\mathcal{N}$, and a transition relation $\rightarrow$. The transition is a set of triples in $Q \times \mathsf{RAS} \times Q$, where Q is the set of states and $\mathsf{RAS} = \mathbf{At}_{\mathcal{N}} \times \mathbf{2}^{\mathcal{N}}$ is a set of pairs consisting of guards written as sets of atoms, and sets of port names. A detailed definition of guards can be found in §2.2.3. We use the name RAS to refer to $\mathcal{R}$eo automata (atomic) steps.

We define an encoding $[\![\cdot]\!]_{\mathsf{RA}} : \mathsf{RA} \rightarrow \mathsf{BA}$ from $\mathcal{R}$eo automata to behavioural automata, and the composition of atomic steps such that the composition of the encoded behavioural automata coincides with the encoding of the composition of $\mathcal{R}$eo automata. Note that we do not show soundness or completeness of the encoding.

The encoding of the normalised $\mathcal{R}$eo automaton $\mathcal{A} = \langle Q, \mathcal{N}, \rightarrow_{\mathsf{RA}} \rangle$ is the behavioural automaton:

$$[\![\mathcal{A}]\!]_{\mathsf{RA}} = \langle Q, \mathsf{L}[\mathcal{N}], \rightarrow_{\mathsf{BA}}, \mathcal{O} \rangle$$

with $\mathsf{L}[\mathcal{N}]$, $\rightarrow_{\mathsf{BA}}$, and $\mathcal{O}$ as follows:

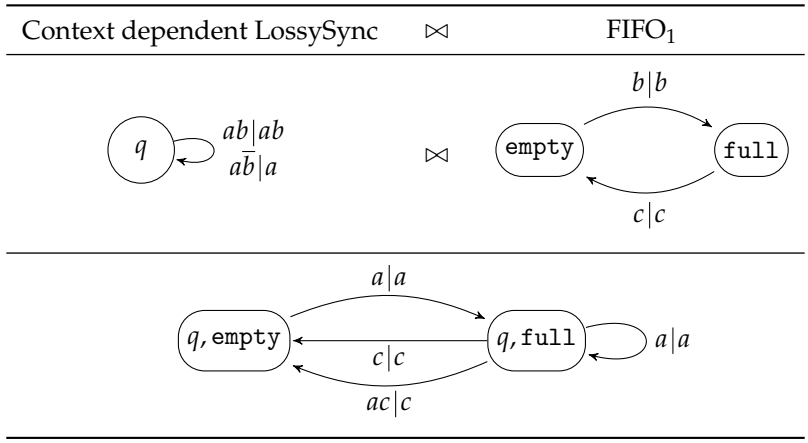


Figure 3.8: Product of the normalised $\mathcal{R}$eo automata of the context dependent LossySync and FIFO$_1$ channels.

- $\mathsf{L} = \mathsf{RAS}$, and α is defined as follows.

$$\alpha(g|X) = \langle \mathcal{N}, X, \emptyset, \emptyset, \emptyset \rangle$$

- We have $q \xrightarrow{g|X}_{\mathsf{BA}} q'$ if $q \xrightarrow{g|X}_{\mathsf{RA}} q'$.

- Let $ras_i = X_i|g_i$ be a label from a normalised $\mathcal{R}$eo automaton with ports $\mathcal{N}_i$, where $i \in 1..2$, and let $s = \mathcal{N}_1 \cap \mathcal{N}_2$. Then

$$ras_1 \otimes ras_2 = \begin{cases} g_1 g_2 \backslash s\bar{s} \mid X_1 X_2 \backslash s & \text{if } X_1 \cap \mathcal{N}_2 = X_2 \cap \mathcal{N}_1 \ \wedge \ g_1 \frown g_2 \\ \bot & \text{otherwise.} \end{cases}$$

  Recall that $g_1 \frown g_2$ *iff* $g_1 \cap g_2 \subseteq \mathcal{N} \ \wedge \ \forall a \in g_1 g_2 \cdot \bar{a} \notin g_1 g_2$.

- $\mathcal{O}(q) = \{g|X \in \mathsf{L} \mid X \cap \mathcal{N} = \emptyset, \exists g' \in q^{\#} \cdot g \frown g'\}$. Recall that from Equation (2.2) that

$$q^{\#} = \{\bar{a}_1 \cdots \bar{a}_n \mid \{g_1, \ldots, g_n\} = \{g \mid q \xrightarrow{g|X} q'\}, \forall_{i \in 1..n} \cdot a_i \in g_i\}.$$

We recall from Figure 3.8 the normalised $\mathcal{R}$eo automata $\mathcal{A}'_L$ and $\mathcal{A}_F$ for the context dependent LossySync and the FIFO$_1$ channels presented in Figure 2.5. Let $\mathcal{A}'_L = \langle Q_L, \mathcal{N}_L, \rightarrow_1 \rangle$ and $\mathcal{A}_F = \langle Q_F, \mathcal{N}_F, \rightarrow_2 \rangle$. The encoding of $\mathcal{A}'_L$ into behavioural automata is depicted in the top part of Figure 3.9, and given as follows.

$$[\![\mathcal{A}'_L]\!]_{\mathsf{CA}} = \langle Q_L, \mathsf{AS}[\mathcal{N}_L], \rightarrow_L, \mathcal{O}_L \rangle$$

where

$$\begin{aligned} Q_L &= \{q\} \qquad \mathcal{N}_L = \{a, b\} \quad \rightarrow_L = \{\langle q, s_1, q\rangle, \langle q, s_2, q\rangle\} \\ s_1 &= ab|ab \qquad s_2 = a\overline{b}|a \\ \mathcal{O}_L(q) &= \{g|X \in \mathsf{L} \mid X \cap \mathcal{N}_L = \emptyset, \exists g' \in q^{\#} \cdot g \frown g'\} \end{aligned}$$

Similarly, the encoding of $\mathcal{A}_F$ into behavioural automata is depicted also on the top part of Figure 3.9, and given by

$$[\![\mathcal{A}_F]\!]_{\mathsf{CA}} = \langle Q_F, \mathsf{AS}[\mathcal{N}_F], \rightarrow_F, \mathcal{O}_F\rangle$$

where

$$\begin{aligned} Q_F &= \{\texttt{full}, \texttt{empty}\} \quad \mathcal{N}_F = \{b, c\} \\ s_3 &= b|b \qquad\qquad\qquad s_4 = c|c \\ \rightarrow_F &= \{\langle \texttt{empty}, s_3, \texttt{full}\rangle, \langle \texttt{full}, s_4, \texttt{empty}\rangle\} \\ \mathcal{O}_L(q) &= \{g|X \in \mathsf{L} \mid X \cap \mathcal{N}_F = \emptyset, \exists g' \in q^{\#} \cdot g \frown g'\} \end{aligned}$$

The product of the normalised $\mathcal{R}$eo automata for the context dependent Lossy-Sync and FIFO_1 channels is depicted at the bottom of Figure 3.8. The composition $[\![\mathcal{A}'_L]\!]_{\mathsf{RA}} \bowtie [\![\mathcal{A}_F]\!]_{\mathsf{RA}}$ of the corresponding encodings into behavioural automata yields the behavioural automaton depicted at the bottom of Figure 3.9, where

$$\begin{aligned} s_1 \otimes s_3 &= a|a, \text{ and} \\ s_2 \otimes s_4 &= ac|ac \end{aligned}$$

Note that $\texttt{full}^{\#} = \overline{b}$, $s_2 = a|a$, and $\mathcal{N}_F = \{b, c\}$. Therefore $s_2 \in \mathcal{O}_F(\texttt{full})$ because $\{a\} \cap \{b, c\} = \emptyset$, and $a \frown \overline{b}$. Similarly, $s_4 \in \mathcal{O}(\texttt{full})$. As expected, the resulting behavioural automaton resembles to the encoding of the constraint automaton $\mathcal{A}'_L \bowtie \mathcal{A}_F$, depicted in Figure 2.6.

3.6.3 Linda as behavioural automata

Linda [54], described in §2.3, is a purely asynchronous language where coordination is performed by managing a single tuple-space, and components send requests to read or write data. We described two possible approaches for modelling Linda: (1) a *coarse grained approach* that assumes that actions can occur only in an interleaved fashion, and (2) a *fine grained approach* that considers that two components can read or write at the same time (in the same atomic step), subject to data availability and exclusion constraints. These two approaches correspond to the semantics given by the interleaved transition system (ITS) and the multistep transition system (MTS) for the Linda-Calculus described in §2.3.

We define two encoding functions, $[\![\cdot]\!]_{\mathsf{ITS}} : \mathsf{ITS} \rightarrow \mathsf{BA}$ and $[\![\cdot]\!]_{\mathsf{MTS}} : \mathsf{MTS} \rightarrow \mathsf{BA}$, from Linda tuple space terms under the semantics given by the interleaved transition system and the multistep transition system, respectively, to behavioural automata. Furthermore, we define the composition of atomic steps that preserve

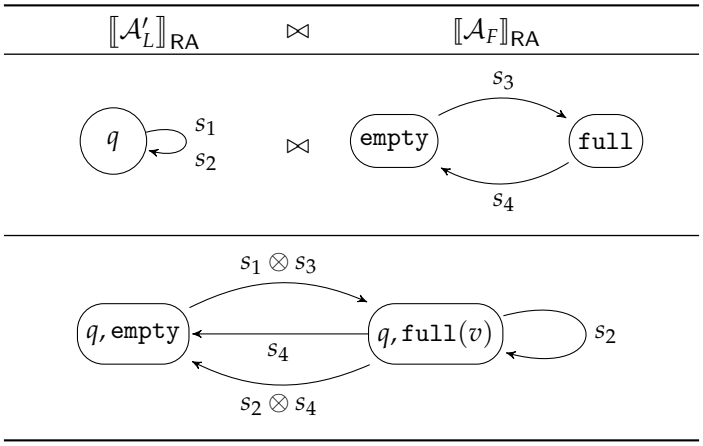


Figure 3.9: Composition of $[\![\mathcal{A}'_L]\!]_{\mathsf{RA}}$ and $[\![\mathcal{A}_F]\!]_{\mathsf{RA}}$.

the respective semantics. In both cases, we encode each Linda process P as a behavioural automaton, and we create a special behavioural automaton that describes the multi-set of available tuples. The diagram in Figure 3.10 depicts the set of behavioural automata resulting from the application of the encoding functions to a Linda tuple space term, where b_i is the behavioural automaton corresponding to a process P_i, and b_{store} is the behavioural automaton that represents the set of available tuples. The edge between each b_i and b_{store} reflects the synchronisation of ports caused by the actions of Linda over tuples.

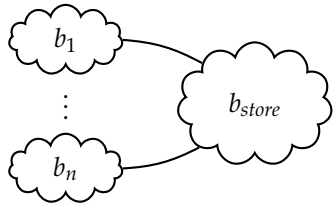


Figure 3.10: Outline of the behavioural automata for the tuple space term $P_1 \oplus \cdots \oplus P_n \oplus T$.

We do not address the action $\mathbf{eval}(P)$, for a Linda process P. The intuitive semantics of $\mathbf{eval}(P)$ is the creation of a new process P that runs in parallel. In our setting, this corresponds to have a system described by the behavioural automaton $b = b_{store} \bowtie b_1 \bowtie \cdots \bowtie b_n$ that evolves to $b \bowtie b_{n+1}$, where b_{n+1} is the behavioural automaton for the newly created process. However, we make the distinction between reconfiguration of a system, provided by the composition operator, and the evolution of a system, described by the execution of a labelled transition system. Behavioural automata describes only the evolution process, and the

reconfiguration is achieved via the composition operator $\bowtie$, which is dealt with orthogonally. Therefore, we consider only actions that change the state of the system in time, and we do not consider the $\mathbf{eval}(P)$ action because it is regarded as a reconfiguration of the system. We redefine the possible actions to be generated by the following grammar.

$$Act \;::=\; \mathtt{in}(t) \;\mid\; \mathtt{out}(t) \;\mid\; \mathtt{rd}(t) \tag{3.8}$$

Let $\overline{Act} = \{\overline{a} \mid a \in Act\}$ and $\tau Act = \{\tau_a \mid a \in Act\}$. A port $\overline{a}$ is regarded as a dual port of a, and flow of data on a port τ_a represents the flow on the ports a and $\overline{a}$ simultaneously. The intuition is that the encoding of processes yields behavioural automata whose ports are actions in Act; the encoding of tuples yield behavioural automata whose ports are *dual* actions in $\overline{Act}$; and the composition forces actions and dual actions to synchronise, i.e., to occur simultaneously. We define the global set of ports to be $\mathbb{P} = Act \cup \overline{Act} \cup \tau Act$, and define $\overline{\overline{a}} = a$. The encodings for the fine- (ITS) and coarse-grained (MTS) approaches are defined below.

Notation We write $\mathcal{M}(X)$ to denote the multi-sets over X, which is a shorthand for the functions of type $X \to \mathbb{N}$, mapping each element from X to a natural number. We also write $\oplus$ for both multi-set construction and union, $A(a)$ to denote the number of occurrences of a in a multi-set A, and $a \in A$ to denote that $A(a) > 0$.

ITS to BA

Let $M = P_1 \oplus \cdots \oplus P_n \oplus T$ be a tuple space term (§2.3). In turn, let $T = t_1 \oplus \cdots \oplus t_m$ and $m \geq 0$. We define the encoding of M into a behavioural automaton as follows.

$$[\![M]\!]_{\mathsf{ITS}} = [\![P_1]\!]_{\mathsf{ITS}} \bowtie \cdots \bowtie [\![P_n]\!]_{\mathsf{ITS}} \bowtie [\![T]\!]_{\mathsf{ITS}}$$

Hence, encoding M boils down to encoding Linda processes P_i and the Linda tuple space T. The tuple space and the processes are each encoded separately into a different behavioural automaton. Recall that *match* is a binary relation over tuples such that a tuple t matches a tuple s if t has only $\mathbb{D}$ values, and there is a substitution γ whose domain is the set of free variables of s, and $t = s[\gamma]$. We say t γ-matches s when t matches s and $t = s[\gamma]$, and we write $P[\gamma]$ to denote the process P after replacing all of its the free variables according to the substitution γ.

In both encodings of components and Linda tuple spaces we define labels L as ports, that is, $\mathsf{L} = \mathbb{P} = Act \cup \overline{Act} \cup \tau Act$, and its encoding as atomic steps by the function α defined below.

$$\alpha(a) = \begin{cases} \langle \mathbb{P}, \{a, \tau_{act}\}, \emptyset, \emptyset, \emptyset \rangle & \text{if } a \in Act \cup \overline{Act}, \{act\} = \{a, \overline{a}\} \cap Act \\ \langle \mathbb{P}, \{a\}, \emptyset, \emptyset, \emptyset \rangle & \text{if } a \in \tau Act \end{cases}$$

The composition of two labels $a_1, a_2 \in \mathsf{L}$ is defined as follows.

$$a_1 \otimes a_2 = \begin{cases} \tau_{act} & \text{if } \{a_1, a_2\} \notin \tau Act \ \wedge \ a_1 = \bar{a}_2 \\ \bot & \text{otherwise,} \end{cases}$$

where $\{a\} = \{a_1, a_2\} \cap Act$. The tuple space is used to enforce every action a performed by an actor synchronises with the corresponding action $\bar{a}$ in the tuple space encoded as a behavioural automaton. The definition of $\otimes$ replaces every pair of ports with dataflow a and $\bar{a}$ by a new port with dataflow in τ_a.

We now encode a Linda process P as the behavioural automata

$$[\![P]\!]_{\mathsf{ITS}} = \langle Q_P, \mathsf{L}, \rightarrow_P, \mathcal{O} \rangle$$

with components as defined below.

- The set of states Q_P is given by $Q_P = \mathit{reach}(P)$, where

$$\begin{array}{rcl} \mathit{reach}(\mathbf{out}(t).P) & = & \{\mathbf{out}(t).P\} \cup \mathit{reach}(P) \\ \mathit{reach}(\mathbf{rd}(s).P) & = & \{\mathbf{rd}(t).P\} \cup (\bigcup \{\mathit{reach}(P[\gamma]) \mid s \ \gamma\text{-matches } t\}) \\ \mathit{reach}(\mathbf{in}(s).P) & = & \{\mathbf{in}(t).P\} \cup (\bigcup \{\mathit{reach}(P[\gamma]) \mid s \ \gamma\text{-matches } t\}) \\ \mathit{reach}(P \,[\!]\, P') & = & \{P \,[\!]\, P'\} \cup \mathit{reach}(P) \cup \mathit{reach}(P') \\ \mathit{reach}(\mathbf{end}) & = & \{\mathbf{end}\} \end{array}$$

- The transition relation $\rightarrow_P$ is given by the following conditions.

$$\begin{array}{rcll} \mathbf{out}(t).P' & \xrightarrow{\mathbf{out}(t)} & P' & \text{if} \quad t \in \mathit{Tuple} \\ \mathbf{rd}(s).P' & \xrightarrow{\mathbf{rd}(t)} & P'[\gamma] & \text{if} \quad s \ \gamma\text{-matches } t \\ \mathbf{in}(s).P' & \xrightarrow{\mathbf{in}(t)} & P'[\gamma] & \text{if} \quad s \ \gamma\text{-matches } t \\ P_1 \,[\!]\, P_2 & \xrightarrow{s} & P_1' & \text{if} \quad P_1 \xrightarrow{s} P_1' \\ P_1 \,[\!]\, P_2 & \xrightarrow{s} & P_2' & \text{if} \quad P_2 \xrightarrow{s} P_2' \end{array}$$

- $\mathcal{O}(q) = \tau Act$ for every state q.

We now define the encoding of a Linda tuple space T as:

$$[\![T]\!]_{\mathsf{ITS}} = \langle Q_T, \mathsf{L}, \rightarrow_T, \mathcal{O} \rangle$$

with components as defined below.

- $Q_T = \mathbf{2}^{\mathcal{M}(\mathit{Tuple})}$.
- The transition relation $\rightarrow_T$ is given by the following conditions.

$$\begin{array}{rcll} M & \xrightarrow{\overline{\mathbf{out}(t)}} & M \oplus t & \text{if} \quad t \in \mathit{Tuple} \\ t \oplus M & \xrightarrow{\overline{\mathbf{rd}(s)}} & t \oplus M & \text{if} \quad s \text{ matches } t \\ t \oplus M & \xrightarrow{\overline{\mathbf{in}(s)}} & M & \text{if} \quad s \text{ matches } t \end{array}$$

- $\mathcal{O}(q) = \tau Act$ for every state q, as in the encoding of Linda processes.

Note that the input and output ports of the atomic steps obtained with α, introduced in §3.3.1, are always the empty set, that is, the data value flowing on the ports is not relevant, since the name of the port uniquely identifies the data. Alternative approaches to implement the encoding into behavioural automata that use the data values are also possible, but less transparent. Furthermore, the encoding we propose is *compositional*, since we encode each process and the store with their own behavioural automaton, such that the product of the automata yields the behaviour of the tuple space term. It would be simpler, although less interesting, to encode the tuple space term directly as a behavioural automaton, and ignore the composition of atomic steps.

3.6.2. EXAMPLE. Recall the example presented in §2.3 of a sequence of transitions of a tuple space term in the Linda-Calculus. We present below a simplified version of this example. The labels on the right represent the names of the rules applied at each transition of the Linda-Calculus.

$$\begin{array}{lcl} & & \mathbf{rd}(42,x).P(x) \oplus \mathbf{out}(42,43).P' \\ (out) & \to & \mathbf{rd}(42,x).P(x) \oplus P' \oplus \langle 42,43\rangle \\ (rd) & \to & P(43) \oplus P' \oplus \langle 42,43\rangle \end{array}$$

This examples illustrates the exchange of data between two processes. The corresponding transitions in the encoded behavioural automaton are presented below.

$$\begin{array}{l} [\![\mathbf{rd}(42,x).P(x)]\!]_{\mathsf{ITS}} \bowtie [\![\mathbf{out}(42,43).P']\!]_{\mathsf{ITS}} \bowtie [\![\emptyset]\!]_{\mathsf{ITS}} \\ \xrightarrow{\tau_{\mathbf{out}(42,43)}} [\![\mathbf{rd}(42,x).P(x)]\!]_{\mathsf{ITS}} \bowtie [\![P']\!] \bowtie [\![\langle 42,43\rangle]\!] \\ \xrightarrow{\tau_{\mathbf{rd}(42,43)}} [\![P(43)]\!]_{\mathsf{ITS}} \bowtie [\![P']\!] \bowtie [\![\langle 42,43\rangle]\!] \end{array} \quad \Diamond$$

Observe that we assume an initial empty tuple space, which is encoded as $[\![\emptyset]\!]_{\mathsf{ITS}}$. A more careful analysis shows a one-to-one correspondence between the interleaved transition system and the labelled transition system of the behavioural automaton, which we do not elaborate in this thesis.

MTS to BA

Let $M = P_1 \oplus \cdots \oplus P_n \oplus T$ be a tuple space term, and let $T = t_1 \oplus \cdots \oplus t_m$, as before. We define the encoding of M into a behavioural automaton using the multistep semantics as follows.

$$[\![M]\!]_{\mathsf{MTS}} = [\![P_1]\!]_{\mathsf{MTS}} \bowtie \cdots \bowtie [\![P_n]\!]_{\mathsf{MTS}} \bowtie [\![T]\!]_{\mathsf{MTS}}$$

We encode M by encoding the Linda processes P_i and a Linda tuple space T. As before, the tuple space and the processes are each encoded separately as a behavioural automaton. We define labels as pairs

$$\mathsf{L} = \mathbf{2} \times \mathcal{M}(Act)$$

such that the first element denotes the presence of the tuple space T, while the second denotes a multi-set of actions that are performed simultaneously, which was not possible in the interleaved semantics. The mapping α into atomic steps simply regards the multi-sets of actions as a sets:

$$\alpha(b, as) = \langle \mathbb{P}, as, \emptyset, \emptyset, \emptyset \rangle$$

The composition of labels is defined below, and describes two main cases. In the first case none of the labels is performed in the presence of the tuple space, hence the multi-set of actions are simply joined. In the second case one of the multi-set of actions is performed in the presence of the tuple space T, say as_1. Then we require as_1 to perform only dual actions or τ-actions, and we also require each action from one of the multi-sets to have its dual in the same proportions in the composed multi-set. The composition yields only the corresponding τ actions, as defined below.

$$\langle b_1, as_1 \rangle \otimes \langle b_2, as_2 \rangle = \begin{cases} \langle \mathit{false}, as_1 \oplus as_2 \rangle & \text{if } \neg b_1 \wedge \neg b_2 \\ \langle \mathit{true}, \{ \tau_a \mapsto n \mid a \in Act, as_i(\overline{a}) = n, as_j(a) = n \} \rangle & \\ \qquad \text{if } i, j \in \{1, 2\} \wedge i \neq j \wedge & \\ \qquad\quad b_i \wedge \neg b_j \wedge \forall a \in Act \cdot & \\ \qquad\quad (a \notin as_i \wedge as_i(\overline{a}) = as_j(a)) & \\ \bot & \text{otherwise} \end{cases}$$

Note that we assume that either as_1 or as_2 has the tuple space, and thus can perform dual actions, but never both at the same time. This assumption is a consequence of allowing only one tuple space in our Linda programs. The above definition for the composition of labels enforces actions and their duals to 'synchronise', replacing them by an action in τAct. The multi-set representation of actions allows more than one action to synchronise in a single atomic step.

We now follow the same steps as with the fine-grained semantics for Linda by defining the encoding of a Linda process P as follows:

$$[\![P]\!]_{\mathsf{MTS}} = \langle Q_P, \mathsf{L}, \rightarrow_P, \mathcal{O} \rangle$$

with components defined below.

- The set of states Q_P is given by $Q_P = \mathit{reach}(P)$, defined for the encoding of ITS into behavioural automata.

- The transition relation $\rightarrow_P$ is given by the following conditions.

$$\begin{array}{rcll}
\mathbf{out}(t).P' & \xrightarrow{\langle \mathit{false}, \mathbf{out}(t)\mapsto 1\rangle} & P' & \text{if} \quad t \in \mathit{Tuple} \\
\mathbf{rd}(s).P' & \xrightarrow{\langle \mathit{false}, \mathbf{rd}(t)\mapsto 1\rangle} & P'[\gamma] & \text{if} \quad s\ \gamma\text{-matches } t \\
\mathbf{in}(s).P' & \xrightarrow{\langle \mathit{false}, \mathbf{in}(t)\mapsto 1\rangle} & P'[\gamma] & \text{if} \quad s\ \gamma\text{-matches } t \\
P_1 \,[\!]\, P_2 & \xrightarrow{s} & P_1' & \text{if} \quad P_1 \xrightarrow{s} P_1' \\
P_1 \,[\!]\, P_2 & \xrightarrow{s} & P_2' & \text{if} \quad P_2 \xrightarrow{s} P_2' \\
P_1 \,[\!]\, P_2 & \xrightarrow{\langle \mathit{false}, as_1 \otimes as_2\rangle} & P_2' & \text{if} \quad P_1 \xrightarrow{\langle \mathit{false}, as_1\rangle} P_1' \wedge \\
 & & & \qquad P_2 \xrightarrow{\langle \mathit{false}, as_2\rangle} P_2'
\end{array}$$

  Note that the above conditions are very similar to the conditions for the encoding of $\rightarrow_P$ in the interleaved transition semantics of Linda, with the only difference that the atomic steps are now (singleton) multi-sets of ports instead of a single port.

- $\mathcal{O}(q) = \{\langle b, as\rangle \mid \mathrm{dom}(as) \subseteq \tau\mathit{Act}\}$, which means it is always possible to execute τ actions without considering all processes.

For a Linda tuple space T we define

$$[\![T]\!]_{\mathsf{ITS}} = \langle Q_T, \mathsf{L}, \rightarrow_T, \mathcal{O}\rangle$$

with components as defined below.

- $Q_T = \mathbf{2}^{\mathcal{M}(\mathit{Tuple})}$, as in the encoding of the ITS.

- The transition relation $\rightarrow_T$ is given by the following conditions.

$$\begin{array}{rcll}
M & \xrightarrow{\langle \mathit{true}, \overline{\mathbf{out}(t)}\mapsto 1\rangle} & M \oplus t & \text{if} \quad t \in \mathit{Tuple} \\
t \oplus M & \xrightarrow{\langle \mathit{true}, \overline{\mathbf{rd}(s)}\mapsto 1\rangle} & t \oplus M & \text{if} \quad s \text{ matches } t \\
t \oplus M & \xrightarrow{\langle \mathit{true}, \overline{\mathbf{in}(s)}\mapsto 1\rangle} & M & \text{if} \quad s \text{ matches } t
\end{array}$$

  The above transition relation is the same as the corresponding transition relation for the encoding of the ITS, using singleton multi-sets as labels instead of the corresponding action.

- $\mathcal{O}(q) = \{\langle b, as\rangle \mid \mathrm{dom}(as) \subseteq \tau\mathit{Act}\}$ is the same as for the encoding of processes.

The resulting encoding of the tuple space term M into a behavioural automaton yields the same behaviour as the multistep semantics of the Linda Calculus without the **eval** action, but we choose to not provide any proof to support this claim.

3.7 Related concepts

In this chapter we present a stepwise coordination model based on behavioural automata. This model represents our view of a dataflow-driven coordination system, following the categorisation of Arbab [10]. Each transition in a behavioural automaton represents the *atomic* execution of a number of actions by the coordination system. The behaviour of a system in our stepwise model is described by the *composition* of the behaviour of its sub-systems running concurrently, each with its own behavioural automaton. Furthermore, we allow the *data values* exchanged over the coordination layer to influence the choice of how components communicate with each other as well. We now present a brief discussion about how the three main concepts that underlie our stepwise model—atomicity, composability, and dataflow—are viewed in the computer science community. The goal of this limited survey is to clarify the differences between these three concepts and other similar concepts that are sometimes interpreted as being synonymous. Furthermore, it explains in what sense our stepwise coordination model is (and is not) synchronous and composable, and how (and how not) it uses the dataflow information.

Atomicity. The notions of *atomicity*, *synchrony*, and *transactional execution* have been frequently used in similar settings, with similar, although different, meanings. *Atomicity* is usually associated with a sequence of instructions that are guaranteed to execute atomically, i.e., without being interleaved with interfering instructions. Achieving atomicity typically involves solving a locking problem: how to protect shared resources to guarantee that only a single thread or process has access to it at a given time. Mutexes and semaphores provide common solutions to achieve atomicity [42]. *Synchronisation* provides a restricted sense of atomicity, and it is associated with the notions of causality and time. We interpret *synchronisation* as synchronisation of actions, and two actions synchronise if they occur at the same time. Note that true synchrony is not possible in distributed systems, because there is no way to guarantee that two actions in different computers are performed at the exact same time. Action synchronisation captures the main idea underlying *process synchronisation* [44] and *data synchronisation* [49], whose distinction in the literature simply reflects that the execution of different processes must evolve at the same time, or that multiple copies of the same data must be kept coherent. Finally, atomicity is known in the database community in the context of *transactions*, where actions can be rolled back when necessary.

Our stepwise coordination model is synchronous in the sense of atomicity as explained above. A set of actions (also referred to as a set of *steps*) S executes atomically if no other action can interfere in between the time when the first $a \in S$ starts to execute and when the execution of all of the actions in S is complete.

Other actions can still occur in the same time frame, as long as they do not interfere with the actions in S. The synchronisation of two or more actions yields a new action, obtained by applying a composition operator $\otimes$.

Composability. *Composability* is closely related to *concurrency* and *distribution*. We first clarify the distinction between distributed and concurrent systems. Distribution involves the execution of a coordination system at multiple locations, where locations can be computers, networks, but also CPU's. Concurrency describes how to approach a given problem using independent and communicating threads of execution. A non-concurrent program executes its instructions sequentially. Clearly a concurrent program does not have to be distributed across multiple locations. Furthermore, a program executing in a distributed network does not have to be concurrent either. A non-concurrent program can be executed in a distributed network using mechanisms that involve replication and guarantee consistency of the state of different replicas (e.g., to provide redundancy for fault tolerance, or to exploit special resources available at only specific locations). Thus, concurrency can be seen as a means to have more effective distribution, that is, distributed implementations typically gain from being concurrent.

We now describe what we mean by composability. The systems studied in this thesis have an underlying abstract model representing their coordination. Having an abstract coordination model facilitates the proof of interesting properties, such as correctness or deadlock freedom, and helps us to reason about the behaviour of the coordination system. A composable coordination model is constructed out of building blocks that can be analysed independently. The composition of the building blocks describes how they interact with each other. But are implementations of a composable coordination model always concurrent? No. Although a composable coordination model can describe its behaviour using simpler concurrent composable blocks, its implementation can still require a (partially) non-concurrent execution. Note that the components or services being coordinated can execute concurrently. The coordination model we present in this thesis describes a concurrent description of a concurrent model which, at first sight, seems to require a non-concurrent execution, due to its synchronisation constraints (but, in fact, it does not).

Dataflow networks and dataflow analysis. Important research on dataflow networks can be found in Kahn's seminal work [66] about Kahn Networks (KN) and in Kok's work about the semantics of dataflow for parallel computation [73, 74]. Kahn introduced KN's back in 1974 to model how processes, executing in sequence or in parallel, transform infinite streams of data, using atomic data tokens as the smallest unit of data being transferred. Kok presents two fully abstract semantics for a class of dataflow nets introduced by Kahn, emphasising on their

compositionality aspects. Dataflow models and Kahn networks can be viewed as specialised channel-based models with respect to $\mathcal{R}$eo that incorporate some basic constructs for primitive coordination, as discussed by Arbab [8]. The diluted notion of local time in Reo, which is used to express atomicity and ordering of events, differentiates otherwise 'equivalent behaviour' of networks in other dataflow models that lead to the Brock-Ackerman anomalies [31].

The field of dataflow analysis studies the execution of programs as a series of transformations of the program state by performing code statements. A simple overview of dataflow analysis can be found, for example, in the book of Aho *et al.* on compilers [3]. A general approach is to relate the possible states before and after each statement, referred to as the inputs and outputs, by a *transfer function*. Multiple predecessor statements can be connected to the input, in which case they are combined by a *meet operator*, assuming a meet semilattice over a set of values $\mathbb{D}$. Multiple successors can also be connected to the output, in which case the data is replicated. Furthermore, they assume that the set of all transfer functions has an identity function and is closed under composition. In this setting it is possible to identify properties of dataflow schemas, and prove the implication of these properties on the correctness, precision, and convergence of a dataflow algorithm. This relation between inputs and outputs of statements is closely related to the notion of data constraints in the context of coordination, using a compositional model. However, this thesis focuses on synchronisation constraints and does not exploit this direction.

3.8 Conclusions

We defined the stepwise coordination model to capture the characteristics of the models that can be used by the Dreams framework. The stepwise coordination model abstracts from the *definition* and *composition* of labels, and imposes general properties over atomic steps associated with transportation labels and their composition. Each of the concrete models defined in Chapter 2 is formulated as an instance of a behavioural automaton.

By defining and composing labels only in the concrete models, we focus on the properties provided by behavioural automata which, under certain assumptions, allows aspects other than synchronisation and data to be incorporated in our distributed framework. As an example, we have presented how to incorporate the notion of context dependency (§3.6.2). Other aspects like time [13], quality-of-service guarantees [15], resource bounds [81], or probability [21, 23, 88], can be used in the Dreams framework as well. Note that each label has an associated atomic step, but the opposite does not necessary hold, with the exception of the identity atomic step which we assume to have a special identity label.

The encoded behavioural automata from automata models for $\mathcal{R}$eo are closely

related to its original semantic models, but this is not the case for Linda. We kept a compositional approach in our encoding of Linda as behavioural automata to allow the execution of Linda in our framework over multiple locations, possibly deploying each Linda process and the tuple space in a different location. Alternatively, we could have encoded a tuple space term $M = P_1 \oplus \cdots \oplus P_n \oplus T$ simply by considering the labelled transition system presented in §2.3, and disregarding the composition of the resulting behavioural automata. The specific encoding of each label as an atomic step is not relevant, because in this alternative approach atomic steps never need to be composed. The downside of such an approach is the loss of concurrency, making it less interesting in the context of this thesis.

Chapter 4

Connector colouring & animation

4.1 Introduction

Connector colouring (CC), introduced by Clarke *et al.* [37], further developed in Costa's Ph.D. thesis [41], and extended by Arbab *et al.* using the tile model [14], provides a semantics for the $\mathcal{R}$eo coordination language [8] which improves its preceding semantics in three main aspects.

1. It is the first semantic model of $\mathcal{R}$eo introducing a precise notion of *context dependency*, as explained in §2.2.3 when describing the $\mathcal{R}$eo automata model. We recall this notion below.

2. It is *more intuitive*, in the sense that it exploits $\mathcal{R}$eo's graphical notation, marking each part of a $\mathcal{R}$eo connector with a 'colour' describing its behaviour. The intuition provided by the graphical representation of connector colouring is further explored by the animation framework that we describe later in this chapter.

3. It is better suited for *distribution*. The main support for this claim are the formal properties of the main composition operator, namely associativity, commutativity, and idempotency. Furthermore, in previous (automata) models for $\mathcal{R}$eo, the composition of a pair of connectors c_1 and c_2 yields a new connector c_3 that discards c_1 and c_2 as individual parts of the new connector, and considers only the new global behaviour. Instead, the connector colouring semantics describes the new behaviour of c_1 and c_2 after their composition.

The need for a context dependent semantics for $\mathcal{R}$eo has been motivated and explained in §2.2.3. We recall the basic ideas behind $\mathcal{R}$eo in §4.2, presented in detail in §2.2. Using a context dependent semantics, we distinguish two different kinds of absence of flow: (1) the *possibility* of not having dataflow, and (2) the *need* to have no dataflow. This distinction allows the definition of behaviour where

the absence of flow can be either imposed or required. For example, the context-dependent LossySync is a synchronous channel with a source and a sink end, such data can either be transferred between these two ends or lost after flowing through the source end. However, the channel does not lose data non-deterministically, but it allows data to be lost if the sink end cannot have dataflow. On the other hand, an empty FIFO_1 channel always imposes the absence of dataflow on its sink end. The composition operator defined in the connector colouring semantics requires all absence of flow to have at least one end imposing its absence, extending the expressive power of its preceding semantics.

Motivated by the second and third contribution mentioned above, we developed in joint work with David Costa a compositional animation framework based on connector colouring. The results are explained in detail in Costa's thesis [41], and are more briefly explained in this chapter. The goal of our animation framework is to automatically derive visual animations to help the developer of $\mathcal{R}$eo connectors understand the precise behaviour of his/her own connectors. Our process of generating animations was incorporated in a visual editor for $\mathcal{R}$eo by Christian Krause [16]. These tools and the animations of some $\mathcal{R}$eo connectors can be found online.[1]

Contribution to the thesis Although we presented $\mathcal{R}$eo and its formal semantics in §2.2, we dedicate a chapter for connector colouring and animations because of the relevance of the connector colouring in the remainder of the thesis. The connector colouring focuses on the possible behaviour for a single round of $\mathcal{R}$eo, and preserves the information regarding the behaviour of each of the primitives in the connector. These two factors make connector colouring suitable as a basis for a distributed implementation and motivated the development of the Dreams framework.

We encode the connector colouring model as behavioural automata, stressing the choices we made when defining behavioural automata, and extend it with data-related information and with concurrency predicates, which yield a new concept of *local colourings*. Finally, the animation framework exhibits the need to include the information regarding how data is flowing, which is not modelled by the connector colouring semantics, making the animation framework the first means to model the dataflow of distributed $\mathcal{R}$eo connectors.

Organisation of the chapter We briefly recall $\mathcal{R}$eo and explain the intuition behind the use of colours to describe the behaviour of $\mathcal{R}$eo in §4.2. We then formalise the connector colouring semantics in §4.3, and its encoding into behavioural automata in §4.4. We exemplify the connector colouring semantics with three connectors in §4.5. Our animation framework is described in §4.6, and we wrap up

[1] Available at `reo.project.cwi.nl`.

by describing related workin §4.7 and presenting some conclusions in §4.8.

4.2 Connector colouring overview

The $\mathcal{R}$eo coordination language [8, 9], described in §2.2, is a channel-based language where connectors are compositionally built out of a simple set of primitives connectors. Each primitive connector, also referred to as a primitive, has a set of ends which act as input or output points of data. For consistency with Chapter 3, we say 'port' to each primitive end, and assume a global set of ports $\mathbb{P}$. Channels are special primitives with two ports. Some of the most commonly used $\mathcal{R}$eo channels and primitives are presented in the left columns of Tables 4.1 and 4.2, respectively. The composition of two connectors is performed by joining the shared ports. Ports are joined in a one-to-one manner, and a source port (port that receives data) can only be joined to a sink port (port that sends data). A *mixed node*, depicted by $\bullet$, is the logical place resulting from joining of two ports, and a *boundary node*, depicted by $\circ$, is the logical place with only one port.

The connector colouring (CC) semantics, as presented by Clarke *et al.* [37], is based on the idea of colouring the ports of a connector using a set of three colours—for orientation, $\bullet$ indicates the port. The colour $\longrightarrow\!\!\bullet$ marks ports in the connector where data flow, and the two colours $\text{-}\triangleleft\text{-}\bullet$ and $\text{-}\triangleright\text{-}\bullet$ mark the absence of dataflow. The main idea is that every absence of flow must have a *reason* for excluding the flow of data, for example, because an empty FIFO_1 buffer cannot produce data. The different no-flow colours mark the direction from where the reason originates. The colour $\text{-}\triangleleft\text{-}\bullet$ denotes that the reason for no-flow originates from the context, by exhibiting an arrow coming from the port, and we say that the port *requires a reason* for no-flow. Similarly, $\text{-}\triangleright\text{-}\bullet$ indicates that the reason for no-flow originates from the primitive, using an arrow pointing in the direction of the port, and we say that the port *gives a reason* for no-flow.

Colouring a connector means associating colours to each of its ports in such a way that the colours of two connected ports *match*. The colours of two ports match if both represent flow, or if the reason for no-flow comes from at least one of the ports. That is, the valid combinations are: $\longrightarrow\!\!\bullet\!\!\longrightarrow$, $\text{-}\triangleleft\text{-}\bullet\!\!\leftarrow\triangleleft\text{-}$, $\text{-}\triangleright\text{-}\bullet\!\!\leftarrow\triangleleft\text{-}$ and $\text{-}\triangleright\text{-}\bullet\!\!\leftarrow\triangleright\text{-}$. Invalid matches of colours include $\longrightarrow\!\!\bullet\triangleleft\text{-}$ and $\text{-}\triangleleft\text{-}\bullet\!\!\leftarrow\triangleright\text{-}$; the first pair represents the case when there is dataflow only in one of two connected ports, and the second pair represents a reason being created when joining the ports, which is undesirable in connector colouring. Each primitive has only a specific set of admissible *colourings*, which determine its synchronisation constraints. Each colouring is a mapping from the ports of a primitive to a colour, and the set of the colourings of a connector is called its *colouring table*. Thus, a colouring table represents the set of all possible behaviour of a $\mathcal{R}$eo connector. Tables 4.1 and 4.2 present all possible behaviour of the most commonly used $\mathcal{R}$eo primitives. Composition of two con-

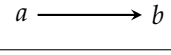
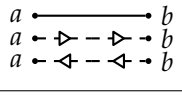
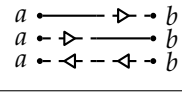
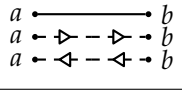
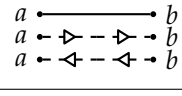
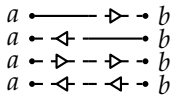
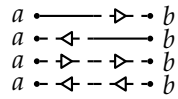
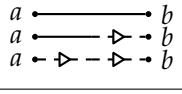
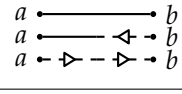
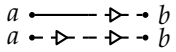
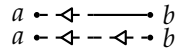
Channel	Colouring table	Channel	Colouring table
Sync 		Priority Drain	
SyncDrain		SyncSpout	
AsyncDrain		AsyncSpout	
LossySync		Context LossySync	
FIFOEmpty$_1$		FIFOFull$_1$	

Table 4.1: Colouring tables for some $\mathcal{R}$eo channels.

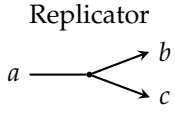
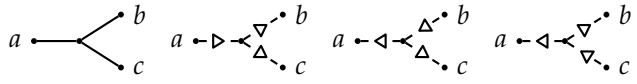
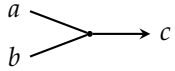
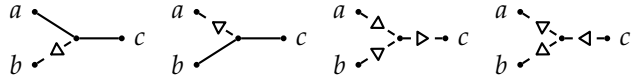
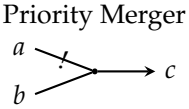
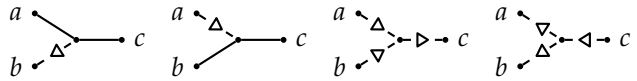
Primitive	Colouring table
Replicator 	
Merger 	
Priority Merger 	

Table 4.2: Colouring tables for some $\mathcal{R}$eo primitives.

nectors is done by creating a new colouring table with all the possible colourings with matching colours at their connected ports. In §4.3 we formalise the connector colouring semantics and the composition of colourings, and define colourings as atomic steps.

4.3 Colourings

We start by formalising the definitions of colouring and colouring tables. Let $\mathcal{C}olour = \{\longrightarrow\bullet, \text{-}\triangleright\text{-}\bullet, \text{-}\triangleleft\text{-}\bullet\}$ be the set of possible colours for each port, and recall that $\mathbb{P}$ is a global set of ports. Furthermore, for any $P \subseteq \mathbb{P}$ let $P^{\updownarrow} = \bigcup_{x \in P} \{x^{\downarrow}, x^{\uparrow}\}$, where $x^{\downarrow}$ denotes that x is a source port, and $x^{\uparrow}$ denotes that x is a sink port.[2] We formalise colourings as follows.

4.3.1. DEFINITION (COLOURING). A colouring over ports $P \subseteq \mathbb{P}^{\updownarrow}$ is a function $c : P \to \mathcal{C}olour$ that maps each port from P to a colour. ◁

We denote by $\mathbb{C}$ the set of all colourings. A colouring over the set P identifies a valid atomic step of a $\mathcal{R}$eo connector with ports P, disregarding any data constraint. For example, the colouring $c_1 = \{a^{\downarrow} \mapsto \longrightarrow\bullet, b^{\uparrow} \mapsto \text{-}\triangleright\text{-}\bullet\}$ describes a scenario where the port a has dataflow and the port b does not have data flow. Furthermore, the port b provides a reason for the absence of dataflow. We drop the superscripts $\uparrow$ and $\downarrow$ on the port names when these can be inferred by inspection of the associated connector. A collection of colourings yields a colouring table, that describes the possible behaviour of a connector.

4.3.2. DEFINITION (COLOURING TABLE). A colouring table over ports $P \subseteq \mathbb{P}^{\updownarrow}$ is a set $T \subseteq \mathbb{C}$ of colourings with domain P. ◁

The colouring c_1 described above represents one of the possible behaviours of the empty FIFO$_1$ channel, with source port a and sink port b. Another possible colouring of this channel is $c_2 = \{a^{\downarrow} \mapsto \text{-}\triangleleft\text{-}\bullet, b^{\uparrow} \mapsto \text{-}\triangleright\text{-}\bullet\}$. For simplicity we also write c_1 as '$a \bullet\!\!-\!\!\triangleright\!\!-\!\!\bullet\, b$' and c_2 as '$a \bullet\!\!-\!\!\triangleright\!\!-\!\!\triangleright\!\!-\!\!\bullet\, b$'. For the empty FIFO$_1$ channel, its colouring table consists of $\{c_1, c_2\}$, describing all the possible steps that this channel can perform. The colouring table of the empty FIFO$_1$ channel can also be found in Table 4.1.

Recall that we colour a $\mathcal{R}$eo connector by selecting colourings of each primitive in the connector and verifying that the colours at shared ports match. We formalise this process by defining the product of colouring tables, presented below.

[2] We use the notation $x^{\uparrow}$ and $x^{\downarrow}$ instead of using disjoint sets of port names, as done by Clarke *et al.* [37], for technical convenience. Our presentation of the connector colouring semantics is closer to automata models for $\mathcal{R}$eo, and our simple extension of data transfer uses the direction of dataflow of the ports to define how data is transferred.

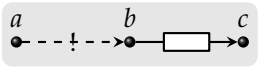
4.3.3. DEFINITION (PRODUCT). The product of two colouring tables T_1 and T_2, denoted by $T_1 \bowtie T_2$, yields the colouring table

$$\{c_1 \cup c_2 \mid c_1 \in T_1, c_2 \in T_2, c_1 \frown c_2\}.$$

We define the compatibility relation $\frown$ below, both for colourings and colours, relating only those that match, where c_1, c_2 are colourings and a, b are colours, and with $\overline{\uparrow} = \downarrow$ and $\overline{\downarrow} = \uparrow$:

$$c_1 \frown c_2 \;\; \textit{iff} \;\; x^\circ \in dom(c_1) \wedge x^{\overline{\circ}} \in dom(c_2) \Rightarrow c_1(x^\circ) \frown c_2(x^{\overline{\circ}})$$
$$a \frown b \;\; \textit{iff} \;\; \langle a, b\rangle \in \{\langle \longrightarrow\bullet, \longrightarrow\bullet\rangle, \langle -\triangleleft-\bullet, -\triangleright-\bullet\rangle, \langle -\triangleright-\bullet, -\triangleleft-\bullet\rangle, \langle -\triangleright-\bullet, -\triangleright-\bullet\rangle\} \qquad \triangleleft$$

4.3.4. EXAMPLE. We present as a simple example the connector resulting from the composition of a context-dependent LossySync with a FIFOEmpty_1 channel, which we call the lossy-FIFO_1 connector:



Recall that this example was also used for the composition of constraint automata and $\mathcal{R}$eo automata, on pages 28 and 34, with the main difference that connector colouring describes only a single step, corresponding in this case to the transitions from the first state of the automata. Composing the colouring tables of both primitives, presented in Table 4.1, we obtain the colouring table of the connector illustrated as follows:

Each image represents a valid colouring, and only these two colourings exist for this connector. The first colouring corresponds to the flow of data through the context-dependent LossySync and into the FIFO_1 buffer, and the second colouring corresponds to the absence of flow in the connector, with a reason for this absence required from port a. In both colourings, the port c of the FIFO_1 buffer ($c^\downarrow$) gives a reason for no dataflow, as the empty buffer can offer no data. $\Diamond$

Example 4.3.4 illustrates that data flowing into the context-dependent LossySync cannot be lost if there is a primitive or component willing to accept that data at its sink port. Furthermore, it also illustrates that an empty FIFO_1 buffer can never produce data, and therefore always gives a reason for no flow to the context.

Our formalisation of connector colouring slightly differs from its original definition [37]. We represent the domain of a colouring as source and sink ports, while the original model does not make this distinction. As a consequence, we avoid

the formulation of the so-called *flip rule*, which states that whenever a colouring table has a colouring mapping a port x to the colour -▷-•, then it also needs to have a similar colouring mapping x to -◁-•. For example, the original model uses two extra colourings to the colouring table of the FIFOEmpty$_1$ channel from Table 4.1: $\{a \bullet\!\!-\!\!\!-\!\!\!\triangleleft\!\!-\!\!\bullet\, b, a \bullet\!\!-\triangleright\!\!-\!\!\!\triangleleft\!\!-\!\!\bullet\, b\}$. The matching function matches only equal colours, and the flip rule states that these extra colourings always exist implicitly, and therefore do not need to be included explicitly in the colouring table.

4.4 Encoding into behavioural automata

We recall the definition of a behavioural automaton, presented in §3.4. We assume global sets of ports $\mathbb{P}$, labels L, atomic steps AS, and *data*, and a global mapping function $\alpha : \mathsf{L} \to \mathsf{AS}$ from labels to atomic steps. A behavioural automaton is a tuple $\langle Q, \mathsf{L}[P], \to, \mathcal{O} \rangle$, where Q is a set of states, $\mathsf{L}[P]$ is a set labels with associated atomic steps over the ports in P, the relation $\to$: $Q \times \mathsf{L}[P] \times Q$ is a transition relation, and $\mathcal{O} : Q \to \mathsf{CP}$ is an observation function, where $\mathsf{CP} = \mathbf{2}^{\mathsf{L}[\mathbb{P}]}$. In this section we start by encoding each colouring as an atomic step, disregarding the data values being transferred, and by defining the composition of colourings based on the product of colouring tables. We then define the observation function and introduce states, extending the connector colouring framework to consider also local behaviour (as a result from introducing concurrency predicates), introducing *local colourings*, and we describe how a connector evolves after performing a given colouring.

4.4.1 Labels as colourings

Clearly the notions of a colouring and an atomic step are closely related. We define labels in behavioural automata to be colourings, that is, $\mathsf{L} = \mathbb{C}$, and encode each colouring $c \in \mathbb{C}$ as an atomic step by defining the function

$$\begin{aligned} \alpha &: \mathbb{C} \to \mathsf{AS} \\ \alpha(c) &= \langle P, X, \emptyset, \emptyset, \emptyset \rangle \,, \end{aligned} \tag{4.1}$$

where

$$P = \{x \mid x^\circ \in \mathrm{dom}(c)\} \quad \text{and} \quad X = \{x \mid c(x^\circ) = \bullet\!\!-\!\!\!-\!\!\bullet\} \,.$$

We define $\mathsf{AS}[P]$ to be the set of all atomic steps with known ports P, and $\mathsf{L}[P]$ as the set of all labels with atomic steps $\mathsf{AS}[P]$. We also write $\mathsf{AS} = \mathsf{AS}[\mathbb{P}]$ and $\mathsf{L} = \mathsf{L}[P]$. Note that the three last arguments of every encoded colouring are always $\emptyset$, that is, the resulting atomic steps do not describe neither the data flowing in the channel nor the dependency between source and sink ports. We will later extend the corresponding atomic steps with data information.

We define below the composition of any two labels (colourings) $c_1, c_2 \in \mathbb{C}$, following closely the Definition 4.3.3 of the product of colouring tables.

$$c_1 \otimes c_2 = \begin{cases} c_1 \cup c_2 & \text{if } c_1 \frown c_2 \\ \bot & \text{otherwise.} \end{cases} \tag{4.2}$$

4.4.2 Local colourings

We now define a *local colouring* of a colouring table $T = T_1 \bowtie T_2$ as a colouring c that belongs to either T_1 or T_2, under certain conditions. These conditions are described by concurrency predicates, introduced in §3.3.2. We associate to each of these colouring tables T_i a concurrency predicate $C_i \subseteq \mathsf{L}$, consisting of a set of colourings from other colouring tables that can be executed independently. Hence we lift the requirement imposed by the original presentation of the connector colouring semantics that, when combining two colouring tables T_1 and T_2 into $T_1 \bowtie T_2$, all the colourings from T_1 must be combined with a colouring from T_2 and vice-versa. We write $\mathsf{CP} = \mathbf{2}^{\mathsf{L}}$ to denote the set of all concurrency predicates, and we write $\mathcal{O}(T)$ to denote the concurrency predicate, introduced in §3.3.2, associated to the colouring table T.

Let P be a set of ports used by a colouring table T, i.e., $P = \{x \mid x^\circ \in \mathrm{dom}(T)\}$. We define the concurrency predicate $\mathcal{O}(T)$ as the set of all colourings that assign the colour -▷-• to every port that is also in P, that is, the colouring table T 'allows' the execution of colourings that give a reason on shared ports with P. We present below the definition of the concurrency predicate $\mathcal{O}(T)$ of a colouring table with known ports P.

$$\mathit{local}(P) = \{c \in \mathbb{C} \mid x^\circ \in \mathrm{dom}(c) \ \wedge \ x \in P \Rightarrow c(x^\circ) = \text{-▷-•}\} \tag{4.3}$$

Thus $\mathcal{O}(T) = \mathit{local}(P)$. We now define local colouring based on this definition of concurrency predicate, and present in Example 4.4.2 a small example that reflects the notion of local steps, similarly to our example in §3.3.2.

4.4.1. DEFINITION (LOCAL COLOURING). A colouring c from a colouring table T is a *local colouring* of T if $T = T_1 \bowtie T_2$, and either $c \in T_1$ and $c \in \mathcal{O}(T_2)$, or $c \in T_2$ and $c \in \mathcal{O}(T_1)$. ◁

4.4.2. EXAMPLE. Consider the composition of the context LossySync channel and the full FIFO_1 channel, presented in Table 4.3. We define $T_L = \{a$ •——— b, a •——◁-▷——• b, a •-▷-◁-▷-• $b\}$ and $T_F = \{b$ •-◁——— c, b •-◁-◁-▷-• $c\}$ as the colouring tables of the context LossySync and the $\mathrm{FIFOFull}_1$ channels, respectively. The full colourings are obtained by the product $T_L \bowtie T_F$, presented in Definition 4.3.3.

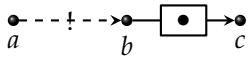
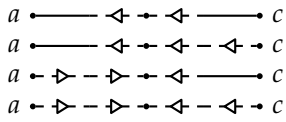
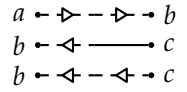
Connector	Full colourings	Local colourings
		

Table 4.3: Colouring table for the lossy-FIFO$_1$ extended with local colourings.

The local colourings of the new connector are all colourings $c_L \in T_L$ such that $c_L \in \mathcal{O}(T_F)$, together with the colourings $c_F \in T_F$ such that $c_F \in \mathcal{O}(T_L)$. The concurrency predicate $\mathcal{O}(T_L)$ is $local(\{a, b\})$, and the concurrency predicate $\mathcal{O}(T_F)$ is $local(\{b, c\})$. The first local colouring in Table 4.3 belongs to T_L, and the remainder local colourings belong to T_F. Note that, when replacing the context LossySync by a Sync channel in this connector, the local colourings remain the same, because the first local colouring in Table 4.3 is also the only local colouring of the Sync channel with respect to its sink port. $\Diamond$

We have now all the ingredients for defining behavioural automata for the primitives presented in this chapter: the definition and composition of labels, and the concurrency predicate of a colouring table.

4.4.3 Colouring tables as states

We encode all the primitives presented in Tables 4.1 and 4.2 as behavioural automata, whose transitions are given by colourings and states are colouring tables. All the primitives mentioned above are stateless, with the exception of the FIFO$_1$ buffer, which has states empty and full. A stateless primitive is a primitive with a single state, where all atomic steps are from and to the unique state. For any stateless primitive p with a colouring table T_p presented in this chapter, we define its behavioural automaton to be as follows:

$$\mathcal{A}_p = \langle \{T_p\}, \mathsf{L}[P], \rightarrow_p, \mathcal{O} \rangle$$

where T_p is the unique state, $\mathsf{L}[P]$ is the set colourings with atomic steps over P, and $\rightarrow_p$ and $\mathcal{O}$ are given below.

$$T_p \xrightarrow{c}_p T_p \quad \textit{iff} \quad c \in T_p \qquad\qquad \begin{array}{l} \mathcal{O} : \{q\} \rightarrow \mathsf{CP} \\ \mathcal{O}(q) = local(P) \end{array}$$

A FIFOEmpty$_1$ behaves as a FIFOFull$_1$ after executing the colouring with flow in its source port, and equivalently the FIFOFull$_1$ channel assumes the behaviour of a FIFOEmpty$_1$ after executing the colouring with flow in its sink port. Let

$P = \{a, b\}$ be the set of known ports, $c_1 = a \bullet\!\!-\!\!\rhd\!\!\rightarrow b$, $c_2 = a \leftarrow\!\!\rhd\!\!-\!\!\rhd\!\!\rightarrow b$, $c_3 = a \leftarrow\!\!\lhd\!\!-\!\!\bullet\, b$, $c_4 = a \leftarrow\!\!\lhd\!\!-\!\!\lhd\!\!\rightarrow b$, $T_e = \{c_1, c_2\}$ be the colouring table of FIFOEmpty$_1$, and $T_f = \{c_3, c_4\}$ be the colouring table of FIFOFull$_1$. We define the behavioural automata of the FIFOEmpty$_1$ channel as follows.

$$\mathcal{A}_F = \langle \{T_e, T_f\}, \mathsf{L}[P], \rightarrow_F, \mathcal{O} \rangle,$$

where $\mathsf{L}[P]$ and $\mathcal{O}$ are defined as above, and $\rightarrow_F$ is defined below.

$$T_e \xrightarrow{c_1}_F T_f \qquad T_f \xrightarrow{c_3}_F T_e$$
$$T_e \xrightarrow{c_2}_F T_e \qquad T_f \xrightarrow{c_4}_F T_f$$

4.4.4 Data transfer

We now redefine labels to include the colouring and the data information as follows:

$$\mathsf{L} = \mathbb{C} \times (\mathbb{P} \rightharpoonup \mathbb{D}).$$

The first element of a label is a colouring describing the synchronisation constraints, and the second element is a mapping $data : \mathbb{P} \rightharpoonup \mathbb{D}$ from ports to the data flowing on it. In our approach, we assume that data is read from the source ports and replicated to the sink ports with dataflow. To avoid deciding for each $\mathcal{R}$eo primitive which source ports should be used when copying data to the sink ports, we impose simple conditions that guarantee that at most one sink port has dataflow when there exists at least one source ports with dataflow. Hence we require that each colouring $c \in \mathbb{C}$ of a primitive connector conforms to at least one of the following scenarios.

- c has no sink ports with dataflow; or
- c has dataflow in a single source port, and the data flowing in all the ports is the same.

These assumptions, stemming from the original description of the connector colouring semantics [37], are made here for all primitives considered in this chapter. Note that we do not require this assumptions for composed connectors. Each colouring can now be associated with several atomic steps, one for each data value being transferred. We start by introducing some auxiliary notation. For any colouring c we define $flow_c = \{x^\circ \mid c(x^\circ) = \bullet\!\!\rightarrow\}$, and P_c and X_c such that $\alpha(c) = \langle P_c, X_c, \emptyset, \emptyset, \emptyset \rangle$. For each primitive with colouring table T, we define the set of possible labels L_T as follows.

$$\mathsf{L}_T = \{\langle c, data_{c,v} \rangle \mid c \in T, v \in \mathbb{D}\} \; data_{c,v} = \{x \mapsto v \mid x^\circ \in flow_c\}$$

We redefine the encoding function $\alpha : \mathsf{L} \to \mathsf{AS}$ to encode colourings and data mappings into atomic steps as follows, where $n \geq 0$.

$$\alpha(\langle c, m\rangle) = \begin{cases} \langle P_c, X_c, \{x_1, \ldots, x_n\}, \emptyset, m\rangle & \text{if } \mathit{flow}_c = \{x_1^{\downarrow}, \ldots, x_n^{\downarrow}\} \\ \langle P_c, X_c, \{x_0\}, \{x_1, \ldots, x_n\}, m\rangle & \text{if } \mathit{flow}_c = \{x_0^{\downarrow}, x_1^{\uparrow}, \ldots, x_n^{\uparrow}\} \end{cases}$$

The composition of labels is presented below, and follows closely the conditions in §3.3.1. Recall that two mappings m_1 and m_2 are compatible, written as $m_1 \frown m_2$, if $m_1(x) = m_2(x)$ for every $x \in \mathrm{dom}(m_1) \cap \mathrm{dom}(m_2)$. The composition of two labels $\langle c_1, m_1\rangle$ and $\langle c_2, m_2\rangle$ is defined as follows.

$$\langle c_1, m_1\rangle \otimes \langle c_2, m_2\rangle = \begin{cases} \langle P_{c_1 \cup c_2}, F_{c_1 \cup c_2}, IP, OP, m\rangle & \text{if } c_1 \frown c_2 \wedge m_1 \frown m_2 \\ \bot & \text{otherwise,} \end{cases}$$

where IP, OP, and m are defined as follows:

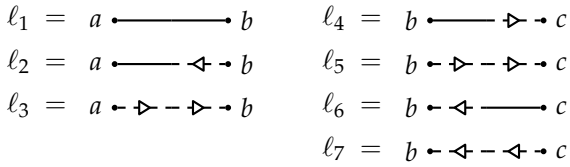
- $IP = (IP_1 \cup IP_2) \backslash (OP_1 \cup OP_2)$;
- $OP = (OP_1 \cup OP_2) \backslash (IP_1 \cup IP_2)$; and
- $m = m_1 \cup m_2$.

4.5 Examples

We illustrate how the connector colouring semantics helps providing intuition about the behaviour of $\mathcal{R}$eo using three main examples: our simple running example of the lossy-FIFO$_1$ connector, the priority exclusive-router connector (a variation of the exclusive router described in §2.2.1), and the alternating coordinator (introduced in §3.2). We abstract from the data being transferred in these examples, being faithful to the original description of the connector colouring model.

4.5.1 Lossy-FIFO$_1$ connector

Recall the Lossy-FIFO$_1$ connector, obtained from composing a LossySync with a FIFO$_1$ channel, used as a running example in this section (see Example 4.3.4 and Example 4.4.2). On top of Figure 4.1 we present the encodings of these two primitives into behavioural automata. The labels ℓ_i, for $i \in 1..7$, are defined below.



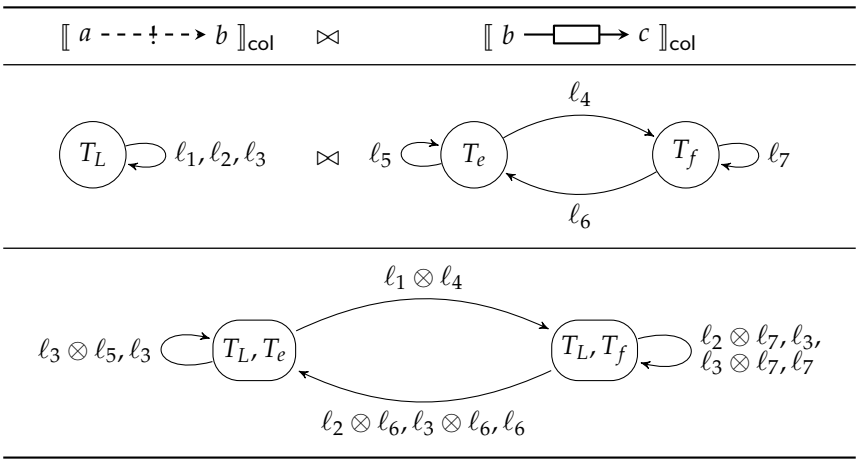
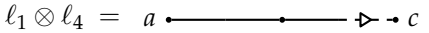
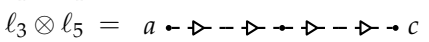
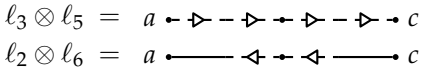
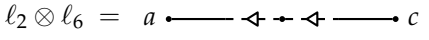
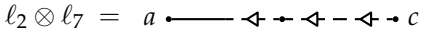
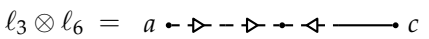


Figure 4.1: Composition of encodings of the context LossySync and the FIFO_1.

The composition of the two behavioural automata yields the behavioural automata depicted at the bottom of Figure 4.1. The product of atomic steps used in the resulting automaton are listed below. Note the existence of *local steps*, which in our example are all atomic steps in the product which do not result from composing other atomic steps. For example, the automata can perform the local step ℓ_6 from $\langle T_L, T_f \rangle$ to $\langle T_L, T_e \rangle$, where the FIFO_1 channel becomes empty without using the colouring table of the LossySync channel. More precisely, ℓ_6 is a local step because $\mathcal{O}(T_L) = \mathit{local}(\{a, b\}) = \{c \in \mathbb{C} \mid x^\circ \in \text{dom}(c) \ \wedge \ x \in \{a, b\} \Rightarrow c(x^\circ) = \text{-▷-•}\}$, and $\ell_6 \in \mathcal{O}(T_L)$.

$\ell_1 \otimes \ell_4 =$ 
$\ell_3 \otimes \ell_5 =$ 
$\ell_2 \otimes \ell_6 =$ 
$\ell_2 \otimes \ell_7 =$ 
$\ell_3 \otimes \ell_6 =$ 
$\ell_3 \otimes \ell_7 =$ 

4.5.2 Priority exclusive-router connector

Our next example results from the composition of stateless primitives, hence it is also stateless. The priority exclusive-router connector is depicted in Figure 4.2. This connector, similarly to the exclusive-router connector presented in §2.2.1, reads data from a source node a and writes data to either the node j or k, but not to both. The only difference with respect to the exclusive-router is the priority merger *f-h-i* instead of a normal merger. This merger enforces data to flow only on j or k, and never on both ports, and the choice between flowing data on j or k

is influenced by the context. We described the behaviour of the priority merger in Table 4.2. If the nodes a, j and k are ready to have dataflow, then data will flow from a to j, and never to k. If j cannot have dataflow, then data can flow from a and k.

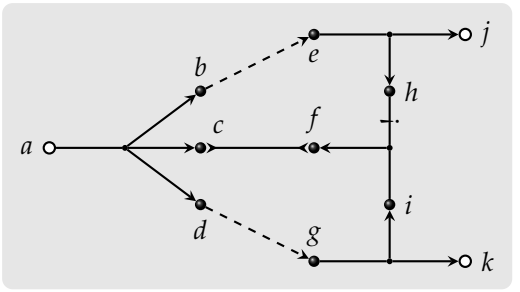


Figure 4.2: Priority exclusive-router connector.

There are three possible scenarios for the behaviour of the priority exclusive-router. Data can flow from a to j or from a to k, or there can be no dataflow in the connector. In Figure 4.3 we present two possible colourings with dataflow on the nodes j and k, respectively. Note an important difference between these two colourings. For the first colouring, the colour of $k^{\uparrow}$ has a triangle pointing outwards, that is, the connector is 'giving' reason for the absence of data flow. For the second colouring, the colour of $j^{\uparrow}$ has a triangle pointing inwards, which means it 'accepts' a reason for the absence of dataflow. Consequently the port j will have higher priority than port k. Note that the two LossySync channels do not need to be context dependent, because the SyncDrain channel will enforce at least one of the LossySync channels to have dataflow, while the merger provides a reason for the absence of flow to the other LossySync.

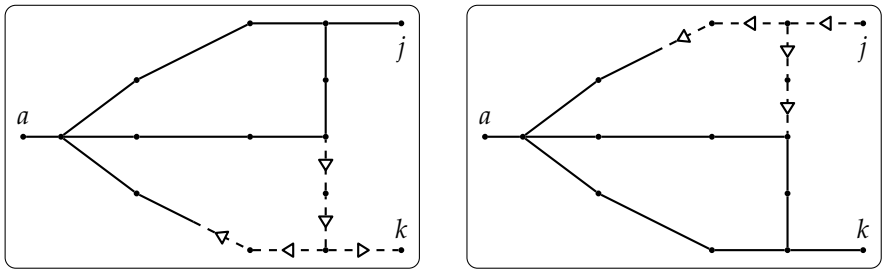


Figure 4.3: Flow colourings for the priority exclusive-router connector.

There are two other colourings with dataflow for the priority exclusive router not presented here. These colourings differ from the left colouring on Figure 4.2 only in the direction of the reasons between the node k and the priority merger,

and describe a scenario that requires $k^{\uparrow}$ to 'accept' a reason. In practice, by analysing the no-flow colours we can justify the absence of flow, and find the origin of the reason for no-flow. In the first colouring the reason originates from the merger, while in the second case the reason originates from node j. Note also that there is no local colouring of this connector with dataflow. The no-flow colouring of the replicator *e-h-j* in the right colouring of Figure 4.3 can be already considered to be a local colouring, since it provides a reason for no-flow on all connected ports. The corresponding behavioural automaton can be easily deduced by constructing a single-state automaton with one reflexive transition for each colouring.

4.5.3 Alternating coordinator

Our next example is the alternating coordinator, used as a running example in Chapter 3, and formalised now using the connector colouring semantics. The alternating coordinator has two possible states, depicted in Figure 4.4. Initially the FIFO_1 buffer is empty, as depicted in the left connector, and it evolves to the state depicted in the right connector, where the FIFO_1 is full. This connector receives data from two ports, a and b, that must arrive simultaneously, and sends the received data through port i, sending first the data from a followed by the data from b. Consequently, it guarantees that data received from a and b is alternated. Other constraints result from the connector: in case of an empty buffer, data can be received on a and b only if the port i can receive data, and no data can be received from either a or b before the previously received data from b is sent via i. The colourings in Figure 4.5 reflect the details of the behaviour just described.

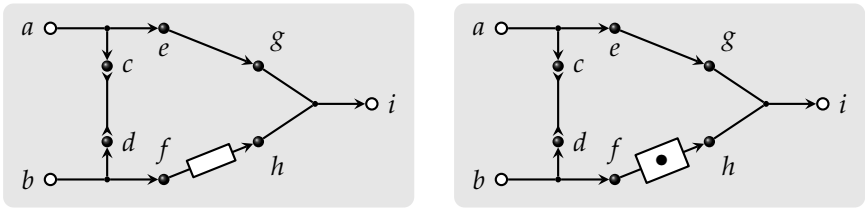


Figure 4.4: Two possible states of the alternating coordinator.

For each of the states presented in Figure 4.4 there is only one possible colouring with dataflow, depicted in Figure 4.5. The colourings with dataflow for the first and second state are depicted on the left and right side of the figure, respectively. There are other similar colourings with dataflow for the second state of the alternating coordinator, that differ only in the direction of the triangles between the SyncDrain and the Sync channels. These variations represent that the Sync channel, the replicator *a-c-e*, and the SyncDrain can receive a reason from either the FIFO_1 channel or from the merger *g-h-i*.

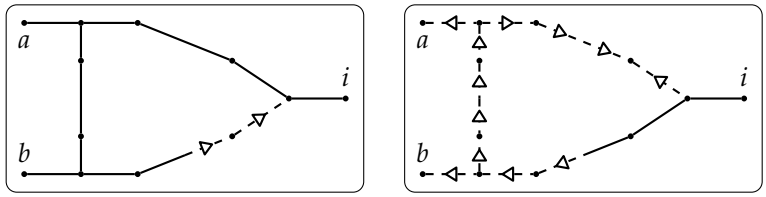


Figure 4.5: Colourings with dataflow of the alternating coordinator.

Other scenarios where the ports a, b, and i receive a reason for the absence of flow are still possible, but the two colourings presented in Figure 4.5 together with a colouring with no-flow everywhere for each state are the only possible colourings when disregarding the direction of dataflow. Let C_1 and C_3 be the set of colourings with no-flow everywhere for the first and second state of the alternating connector, respectively, and let C_2 and C_4 be the sets of colourings with the same dataflow as the left and right colourings from Figure 4.5, respectively. The behavioural automaton of the alternating connector is depicted in Figure 4.6, where label each arrow in the diagram by a set of colourings C to denote multiple transitions, one for each element of C. The state T_e denotes the first state, where the FIFO_1 channel is empty, and the T_f denotes the state where the FIFO_1 channel is full.

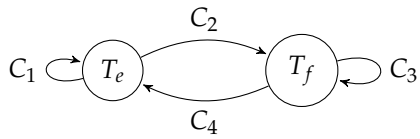


Figure 4.6: Encoding of the alternating coordinator.

4.6 Connector animation

We described the connector colouring model as an intuitive yet precise semantics to $\mathcal{R}$eo, the first successful approach to capture context dependency. We now present the *connector animation* framework, developed in joint work with David Costa, whose main goal is to automatically produce visual animations that simulate the behaviour of a $\mathcal{R}$eo connector, providing a valuable insight when developing connectors. We do not present the full details of this framework, and direct the reader to Chapter 5 of Costa's thesis [41].

The connector animation framework extends the connector colouring semantics with information about the data flowing in the connector, in a similar way to

our approach in §4.4.4 where we extend labels of behavioural automata to include also a data function. These two approaches differ in that our encoding focuses on the data values and the direction of dataflow, while the connector animation focuses on how to represent the movement, replication, creation, and deletion of data tokens in a connector. We describe connector animation by introducing the notions of tokens and locations in §4.6.1, explaining the graphical notation in §4.6.2, presenting an abstract language for animations in §4.6.3 and §4.6.4, and briefly describing how these descriptions are used to produce the final animations in §4.6.5.

4.6.1 Preliminaries

The connector colouring semantics defines colouring tables by assigning colours to source and sink ports that indicate *where* there is dataflow at each round. The connector animation framework describes also how to represent the movement of data through the connector, introducing two new concepts: *data-tokens*, or simply tokens, and *locations*.

Data-token A token represents a unit of information flowing in the connector, and is depicted graphically by ⬟. We denote by $\mathcal{T}oken$ the enumerable set of all data-tokens. Tokens with the same colour are associated to the same data value, and tokens with different colours can have different data values.

Locations A location is a placeholder for a single token. We assume a finite set of locations $\mathcal{L}oc$, and use the names $\ell_1, \ldots, \ell_n$ to range over locations and α, β, γ to range over variables of locations. We say that a location is *occupied* when it holds a token, and is *vacant* otherwise.

The purpose of connector animation is to describe how tokens move within each channel, for a specific colouring, and to combine these descriptions to generate animations of a composed connector. For example, let α, β, and γ be locations of a FIFO_1 buffer corresponding to the source port, sink port, and buffer, respectively. We describe an animation of the empty FIFO_1 channel as follows. When there is dataflow, a token at α moves to γ where it remains for the next round. A full FIFO_1 channel moves a token from γ to β. By providing such descriptions for all primitives used in a $\mathcal{R}$eo connector, we generate automatically visual animations of the composed $\mathcal{R}$eo connectors. These descriptions are formalised by the *animation specification language*, which we will soon present. The generation of the visual animation of a connector c is made in two phases. First, the colouring table of the primitives in c are combined, following Definition 4.3.3 to calculate the product of colouring tables, and a colouring is selected. Second, the animation specifications associated with the selected colouring of each primitive are also combined, yielding a abstract description of the movement of tokens

over a connector. We now introduce some graphical notation used in the resulting animations, together with a simple example.

4.6.2 Graphical notation

We now describe how we choose to represent the animation of a $\mathcal{R}$eo connector, denoting the execution of a colouring (an atomic step). An animation consists of the overlay of three different layers described below, illustrated in Table 4.4.

1. The *connector* representation.
2. The *colouring* being animated, using a thick background line where there is dataflow, and triangles when there is a reason for no-flow.
3. The creation, movement, or deletion of *data-tokens*, represented by the corresponding actions on tokens ⬟ in the animation.

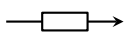
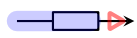
Connector	Colouring	Actions over tokens
		

Table 4.4: Example of the three layers of an animation for the FIFO_1 channel.

We now present a connector obtained from composing a FIFO_1 with a SyncDrain channel, which we call the alternating drain connector, to exemplify an animation generated automatically by the connector animation framework. The images are screenshots from Flash© animations produced in the Eclipse Coordination Tools, that we will explain later in this section.

The alternating drain is depicted on top of Figure 4.7. The animations can only be generated when all boundary nodes ○ are connected to an environment, that in our case consists of two writers, each ready to output a value. Note that writers also have an associated colouring table, which is composed with the colouring tables of the rest of the connector. On the bottom of Figure 4.7 we present six screenshots of an animation illustrating two rounds of the corresponding behavioural automaton. On the left side we see the same colouring decorating the connector, denoting the flow of data from the left writer to the FIFO_1 buffer and providing a reason from the FIFO_1 via the SyncDrain to the right writer for the absence of dataflow. With respect to the token, the following actions are performed: the token is created in the writer, it moves until the FIFO_1, and stays there for the next round. On the right of the figure we see the animation of the second round, where a new token is created by the right writer, and both tokens move to the SyncDrain. The tokens disappear after meeting in the centre of the SyncDrain. Each animation represents a trace of the execution of a connector. In this example, this was the only possible trace with respect to the flow of data.

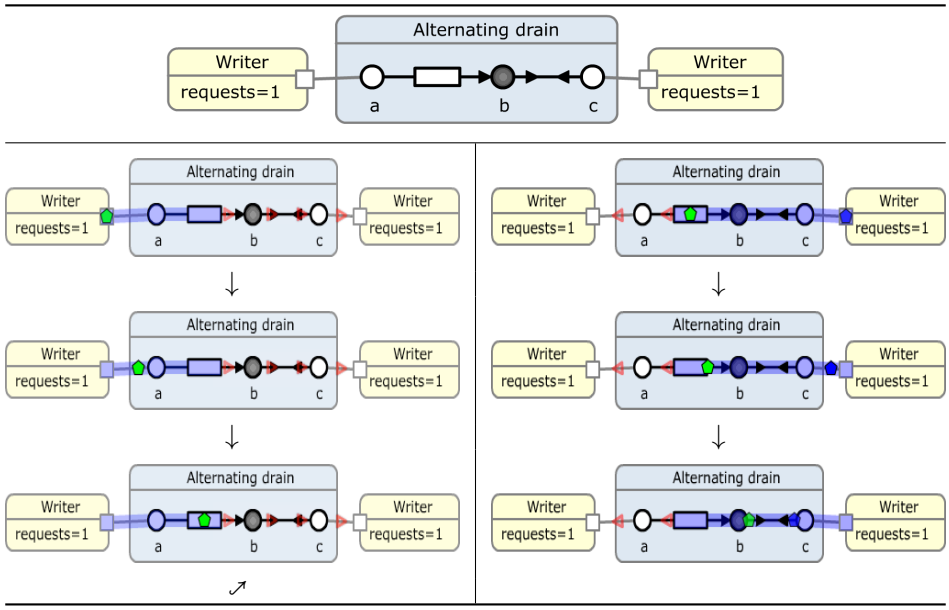


Figure 4.7: Animation of the alternating drain connector.

4.6.3 Animation specifications – Syntax

Connector colouring provides a representation for the presence and absence of dataflow. We now introduce a simple language, the animation specification language (ASL), to represent the four basic dataflow actions for *moving*, *copying*, *creating*, and *deleting* tokens, described below. Recall that we assume a finite set of locations $\mathcal{L}oc$, and use α, β, γ to range over variables of locations.

Move We write $\alpha \triangleright \beta$ to denote the movement of a token from location α to location β. It can only be performed when α is occupied and β is vacant.

Copy We write $\alpha\,!\,\beta$ to denote the duplication of a token in the location α into location β. It can only be performed when α is occupied and β is vacant.

Create We write $\bigstar\,\alpha$ to denote the creation of a new token in location α. It can only be performed when α is vacant.

Delete We write $\boxtimes\,\alpha$ to denote the deletion of a token in location α. It can only be performed when α is occupied.

An *animation specification* is a finite set of basic dataflow actions, where we write 0 to represent an empty animation specification, and we write $as_1 \cup as_2$ to denote the composition of two animation specifications. The animation specification language is formalised below.

$$\text{(MOVE)}\frac{}{\Gamma \vdash(\alpha \triangleright \beta)\rightarrow (\Gamma\setminus\{\alpha\})\cup\{\beta\}} \text{ if } \alpha\in\Gamma, \beta\notin\Gamma$$

$$\text{(SKIP)}\frac{}{\Gamma \vdash(0)\rightarrow \Gamma} \qquad \text{(COPY)}\frac{}{\Gamma \vdash(\alpha \,!\, \beta)\rightarrow \Gamma\cup\{\beta\}} \text{ if } \alpha\in\Gamma, \beta\notin\Gamma$$

$$\text{(CREATE)}\frac{}{\Gamma \vdash(\bigstar \alpha)\rightarrow \Gamma\cup\{\alpha\}} \text{ if } \alpha\notin\Gamma \qquad \text{(DELETE)}\frac{}{\Gamma \vdash(\boxtimes \alpha)\rightarrow \Gamma\setminus\{\alpha\}} \text{ if} \alpha\in\Gamma$$

$$\text{(SEQ1)}\frac{\Gamma \vdash(as_1)\rightarrow \Gamma_1 \quad \Gamma_1 \vdash(as_2)\rightarrow \Gamma_2}{\Gamma \vdash(as_1 \cup as_2)\rightarrow \Gamma_2} \qquad \text{(SEQ2)}\frac{\Gamma \vdash(as_2)\rightarrow \Gamma_2 \quad \Gamma_2 \vdash(as_1)\rightarrow \Gamma_1}{\Gamma \vdash(as_1 \cup as_2)\rightarrow \Gamma_1}$$

$$\text{(PAR)}\frac{\Gamma_1 \vdash(as_1)\rightarrow \Gamma_1' \quad \Gamma_2 \vdash(as_2)\rightarrow \Gamma_2'}{\Gamma_1\cup\Gamma_2 \vdash(as_1 \cup as_2)\rightarrow \Gamma_1'\cup\Gamma_2'} \text{ if } \mathrm{dom}(as_1)\cap\mathrm{dom}(as_2)=\emptyset$$

Table 4.5: Semantics of ASL.

4.6.1. DEFINITION (ANIMATION SPECIFICATION LANGUAGE).
Animation specifications over $\mathcal{L}oc$ are generated by the following grammar.

$$as \;::=\; \alpha \triangleright \beta \mid \alpha \,!\, \beta \mid \bigstar \alpha \mid \boxtimes \alpha \mid 0 \mid as_1 \cup as_2 \qquad \lhd$$

For simplicity we give lower precedence to $\cup$ with respect to the other operators, writing $\bigstar\alpha \;\cup\; \beta \triangleright \gamma$ instead of $(\bigstar\alpha) \;\cup\; (\beta \triangleright \gamma)$. For example, the animation specification $\bigstar \ell_1 \;\cup\; \ell_1 \,!\, \ell_2 \;\cup\; \ell_1 \triangleright \ell_3$ can represent the following behaviour. A token is created in location ℓ_1, duplicated into ℓ_2, and then the original token is moved to ℓ_3. Note that this behaviour results from evaluating the animation specifications from the left to the right. However, we can evaluate it in a different order. For example, if we consider that initially the location ℓ_1 was occupied, this behaviour is no longer possible, because the creation of tokens is only possible in vacant locations. However, it is possible to duplicate the existing token in ℓ_1 to ℓ_2, move the token from ℓ_1 to ℓ_3, and only then create the new token in ℓ_1.

An animation specification can only be evaluated in a given environment, described below. Observe that some animation specifications have no environment under which they be evaluated. For example, $\boxtimes \ell_1 \;\cup\; \ell_1 \triangleright \ell_2$ has no possible behaviour, because both actions remove a token from ℓ_1. Since location ℓ_1 is a placeholder for a single token, it is not possible neither to delete a token in ℓ_1 and then move any token from there, nor the other way around. However, if we compose this animation specification with $\bigstar \ell_1$ then the resulting animation specification already has possible behaviours.

4.6.4 Animation specifications – Semantics

We describe the semantics of animation specifications by formalising the evolution of *frames* by animation specifications. A frame $\Gamma \subseteq \mathcal{L}oc$ is a predicate describing which locations are occupied, i.e., $\alpha \in \Gamma$ iff the location α is occupied. We now define, for each animation specification *as*, an *evaluation relation* over frames. We write $\Gamma \vdash(as)\rightarrow \Gamma'$ to denote that when considering a set Γ of occupied locations, the animation specification evaluates in Γ and produces a new frame Γ' after performing a set of basic dataflow actions. The semantics is given by the set of axioms and inference rules presented in Table 4.5. The side-condition in the PAR-rule refers to the domain of animation specifications, defined below.

4.6.2. DEFINITION (DOMAIN OF ANIMATION SPECIFICATIONS).
We define the domain of animation specifications as follows.

$$\begin{array}{lll} \mathrm{dom}(\alpha \triangleright \beta) = \{\alpha,\beta\} & \mathrm{dom}(\boxtimes\alpha) & = \{\alpha\} \\ \mathrm{dom}(\alpha\,!\,\beta) = \{\alpha,\beta\} & \mathrm{dom}(0) & = \emptyset \\ \mathrm{dom}(\bigstar\alpha) = \{\alpha\} & \mathrm{dom}(as_1 \cup as_2) & = \mathrm{dom}(as_1)\cup\mathrm{dom}(as_2) \end{array} \quad \triangleleft$$

The domain of an animation specification *as* yields the locations used by *as*. The axioms regarding the basic dataflow axioms in Table 4.5 mimic their informal description presented in §4.6.3. The composition of animation specifications deserves some more attention. If the animations specifications refer to different locations, i.e., if their domains are disjoint, then it is safe to perform both of them in simultaneously. Otherwise, the animation specifications evaluate the current frame in sequence. For example, we can show that, for $\Gamma = \ell_1$, the following evaluation is possible.

$$\{\ell_1\} \vdash(\boxtimes\ell_1 \cup \ell_1 \triangleright \ell_2 \cup \bigstar\ell_1)\rightarrow \{\ell_2\} \tag{4.4}$$

To show Equation (4.4) start by observing that $\emptyset \vdash(\ell_1 \triangleright \ell_2 \cup \bigstar l1)\rightarrow \{\ell_2\}$, because (1) $\emptyset \vdash(\bigstar\ell_1)\rightarrow \{\ell_1\}$ by rule (CREATE), (2) $\{\ell_1\} \vdash(\ell_1 \triangleright \ell_2)\rightarrow \{\ell_2\}$ by rule (MOVE), and the evaluations (1) and (2) can be combined with rule (SEQ2) to yield our initial observation. Note that, by rule (DELETE), $\{\ell_1\} \vdash(\boxtimes\ell_1)\rightarrow \emptyset$. Hence, it follows by rule (SEQ1) that Equation (4.4) holds.

4.6.5 Producing visual animations

We extend the information regarding each $\mathcal{R}$eo primitive with animations specifications. We require that each port has an associated location. In our examples, we assume that ports a, b, and c have locations represented by α, β, and ς, respectively, and use the location γ to denote a location represented in the middle of the connector, with no associated port. We also assume that only colourings with dataflow can have non-empty animation specifications, i.e., different from 0. We

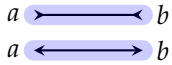
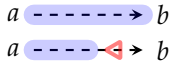
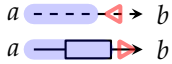
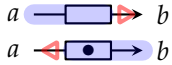
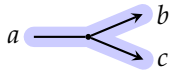
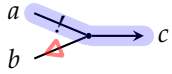
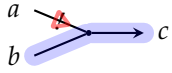
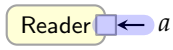
Colourings	Animation specification
	$\alpha \triangleright \beta$
	$\alpha \triangleright \gamma \cup \boxtimes\gamma \cup \beta \triangleright \gamma \cup \boxtimes\gamma$
	$\bigstar\gamma \cup \gamma \triangleright \alpha \cup \bigstar\gamma \cup \gamma \triangleright \beta$
	$\alpha \triangleright \beta$
	$\alpha \triangleright \gamma \cup \boxtimes\gamma$
	$\alpha \triangleright \gamma$
	$\gamma \triangleright \beta$
	$\alpha \,!\, \beta \cup \alpha \triangleright \varsigma$
	$\alpha \triangleright \varsigma$
	$\beta \triangleright \varsigma$
	$\bigstar\gamma \cup \gamma \triangleright \alpha$
	$\alpha \triangleright \gamma \cup \boxtimes\gamma$

Table 4.6: Animations specifications for some $\mathcal{R}$eo channels.

define an *animation table* to be a mapping from colourings (with dataflow) to animation specifications, together with a mapping between ports and the associated locations. In Table 4.6 we present the animation tables for some $\mathcal{R}$eo primitives. In this table we also depict the animation tables of writers and readers, represented as  and  respectively.

Given an animation specification *as* for a connector and an initial frame Γ, we can generate a visual animation by finding a Γ' such that $\Gamma \vdash\!(as)\!\rightarrowtail \Gamma'$, and the tokens are manipulated following the same steps required to prove that $\Gamma \vdash\!(as)\!\rightarrowtail \Gamma'$, i.e., the derivation of this proof. We developed a proof-of-concept implementation for the connector animation framework, called *ReoFlash*. Reoflash provides a set of Haskell libraries to define $\mathcal{R}$eo connecors, colourings, and animation specifications. It generates automatically a *script file* that can be directly fed into the SWFTools,[3] which produces a SWF file ready to be visualised using a Flash Player. Using these libraries we created a small repository of $\mathcal{R}$eo animations, which can be found online.[4]

The current implementation, developed mainly by Christian Krause, has been incorporated in the Eclipse Coordination Tools (ECT) framework [16]. The ECT

[3] `www.swftools.org`
[4] `reo.project.cwi.nl/webreo/`

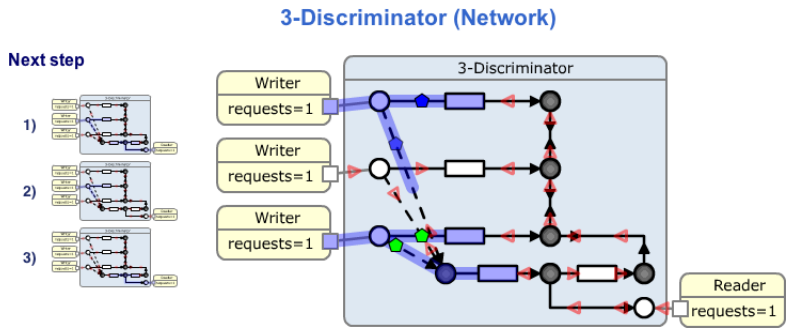


Figure 4.8: Animation of a more complex connector in ECT.

framework consists of a visual $\mathcal{R}$eo editor extended with a set of plug-ins for the Eclipse platform.[5] The animation of $\mathcal{R}$eo connectors is one of such plug-ins. The user can draw $\mathcal{R}$eo connectors in the editor, and Flash© animations are generated in a separate window at runtime. In the animations window a list of possible animations appears, one for every possible behaviour with dataflow, and the animations can be executed for either the one round or for a sequence of rounds. The screenshots in Figure 4.7 were produced using the ECT framework. The ECT becomes especially useful when developing more complex connectors, as suggested by the screenshot in Figure 4.8, where the behaviour is harder to understand. This figure depicts the discriminator connector, whose behaviour is complex but not relevant for this section, and will be used again for the benchmarking of our distributed implementation in Chapter 7.

4.7 Related work

A number of informal and formal models exist for $\mathcal{R}$eo, such as abstract behaviour types [9], automata models [22, 29, 70, 15], models based on a structural operational semantics [89, 67], and models based on logic [34, 39, 40]. Furthermore, the colouring semantics has been modelled using the tiles model [14], which also describes the data being transferred. We will present a more exhaustive comparison between the various $\mathcal{R}$eo models in the next chapter, after introducing a constraint-based model for $\mathcal{R}$eo. The connector colouring model was the first model to capture context dependent behaviour, which earlier models did not, doing so in a simple and precise manner. Clarke *et al.*suggest in the connector colouring paper [37] that Milner's classic SCCS [84] could be an appropriate model for 'implementing' the colouring scheme, by mapping colours to SCCS actions.

The importance of having visual formalisms to model systems accurately has

[5] www.eclipse.org

been long recognised in the scientific community. Visual languages like Petri Nets [91], sequence charts [62] and statecharts [58, 59] were introduced to help building visual system models. For example, for Statecharts, introduced in 1987 by David Harel, the tool Rhapsody [61] that generate running code for Statecharts models, and use simple animations to facilitate the process of model development, specification and analysis. 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* [60, 45]. 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 Tool suite uses the Reactive Animation principle, and the animations are generated and updated on the fly 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.* [99] use it to compare different workflow patterns. To explain the differences between the workflow patterns they have hand-programmed insightful Flash© animations and offer them together with additional information on the web.[6] The visual appearance of our animations is largely influenced by their work. Our approach to obtain the animations differs, though. We generate the animations automatically according to the formal semantics, while in the approach of Van der Aalst it al. animations are created 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. Hung Tran Van *et al.* present animations for Goal-Oriented Requirements [98]. They propose to animate UML state diagrams to visualise and simulate a specific model using visual elements that represent real-life actions of the system to be modelled, such as representing a video with two doors opening associated to the execution of a method called `open-door`. Multiple users can interact with an execution of the model, and property violations are monitored at animation time. Westergaard and Lassen present the BRITNeY Suite Animation tool [102], a tool to create visualisations of formal models, specially 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 produces, on the fly, 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. In contrast, approaches such as the use of goal-oriented requirements force developers to construct both a model for the system and an additional model for its animation.

[6] `http://www.workflowpatterns.com`

4.8 Discussion and conclusions

The connector colouring semantics improves previous $\mathcal{R}$eo semantics models by introducing a more intuitive notation, and by capturing for the first time the notion of context dependency in $\mathcal{R}$eo. The attribution of a visual colour to each channel end, together with a suitable matching of colours, exploits $\mathcal{R}$eo's graphical notation to explain the possible behaviour of each connector. The use of a graphical notation of $\mathcal{R}$eo to provide intuition about the behaviour of $\mathcal{R}$eo connectors is further exploited in the connector animation framework, which simulates the execution of connectors representing data tokens moving over them.

In the process of encoding colouring tables as behavioural automata we extended the colouring semantics in two ways. Firstly, we introduced the notion of local colourings, as a result from our definition of concurrency predicate. Secondly, we can handle state transitions associated to the execution of colourings, following the same approach as the modelling of the colouring semantics using the tiles model [14]. A local colouring does assigns colours to only a subset of known ports, which is not a valid colouring in the original formulation of connector colouring. When a connector a is composed with a connector b, sharing ports $x_1, \ldots, x_n$, then any colouring of a that either assigns the colour -▷-• or does not assign any colour to each port x_i, is also consider to be a valid colouring of $a \bowtie b$. This notion of local colouring emphasises the contribution of the connector colouring model to the development of a distributed implementation of $\mathcal{R}$eo, contributing as well to the colouring semantics by allowing local behaviour. Our Dreams framework, described later in this thesis, exploits this locality.

When analysing the examples presented in §4.5 we can observe several similar colourings that differ only on the direction of the absence of flow. However, not all colourings need to be considered. For example, when calculating the colouring table of a connector we can abstract away from the colours of internal ports (mixed nodes) and focus only on the colours of the boundary nodes. This corresponds to the approach taken by the $\mathcal{R}$eo automata model [29], described in §2.2.3, where internal ports are omitted after the composition of two automata. Furthermore, a colouring where a reason is received can be ignored from the colouring table if a similar colouring exists that sends a reason in the same port. For simplicity, the animations generated by the ECT framework represent only one colouring for each possible dataflow, and do not represent all possible combinations of no-flow colours.

An important aspect covered by Costa's thesis [41], which is not considered here, is causality. In some connectors it is possible to find valid but unrealisable colourings where dataflow or a reason for no-flow results from a loop, as depicted in Table 4.7. These loops are called *causality loops*, and are handled in Costa's thesis by introducing a *causality relation* that, for each colouring, describes the origin and destination of the flow of data or the direction of the no-flow arrow. He introduces

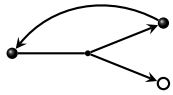
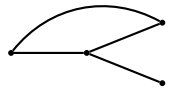
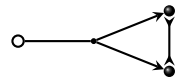
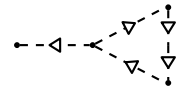
Connector	Causality loop	Connector	Causality loop
			

Table 4.7: Causality loops in two $\mathcal{R}$eo connectors.

the *constructive connector colouring* semantics that admits only colourings that do not have loops on the causality relation, that is, when the transitive closure of the causality relation is anti-reflexive.

We also introduce a simple notion of data transfer, that follows closely the definition of the data functions and its composition in the stepwise coordination model, presented earlier in this thesis. We do not use this extension to the connector colouring in our examples, but we use it later when presenting the implementation of the Dreams framework.

Chapter 5

Constraint-based models for $\mathcal{R}$eo

5.1 Introduction

Wegner describes coordination as constrained interaction [101]. We take this idea literally and represent coordination using constraints to develop an efficient implementation of the $\mathcal{R}$eo coordination model [8]. We will adopt the view that a $\mathcal{R}$eo connector specifies a series of constraint satisfaction problems, and that valid interaction between a connector and its environment (in each state) corresponds to the solutions of such constraints [11]. This idea diverges from the existing descriptions of $\mathcal{R}$eo, which are based on dataflow through channels, but we claim that the viewpoint of constraint satisfaction not only is an appealing alternative way of thinking about $\mathcal{R}$eo connectors, but it also opens the door to more efficient implementation techniques—a claim supported by experimental results in §5.6. We use constraints to describe the semantics of $\mathcal{R}$eo connectors in order to develop more efficient implementation techniques for $\mathcal{R}$eo, utilising existing SAT solving and constraint satisfaction techniques.

The chapter makes the following technical contributions:

- a constraint-based semantic description of $\mathcal{R}$eo connectors; and
- competitive SAT- and CSP-based implementation techniques for $\mathcal{R}$eo connectors.

Organisation of the chapter This chapter is organised as follows. We briefly recall the $\mathcal{R}$eo coordination model in §5.2, where we also present an example of a $\mathcal{R}$eo connector for illustrative purposes further on. §5.3 describes our encoding of $\mathcal{R}$eo-style coordination as a constraint satisfaction problem. §5.4 describes an extension of this encoding to incorporate state, so that connector semantics can be completely internalised as constraints. Correctness and compositionality properties are also presented. §5.5 describes how to encode context dependency, as

formulated in the connector colouring model [37], into constraints. §5.6 presents some benchmarking results comparing an existing engine for $\mathcal{R}$eo based on connector colouring with a prototype engine based on constraint solving techniques, with and without context dependency. §5.7 describes how to guide the underlying constraint solver to achieve fairness and priority. §5.8 presents some implementation issues, in particular, it gives a description of an alternative interaction model that constraint satisfaction enables. §5.9 discusses and compares existing $\mathcal{R}$eo models and implementations with our constraint-based approach, and discusses the position of this approach with respect to the Dreams framework. Finally, §5.10 and §5.11 present related work and our conclusions.

5.2 $\mathcal{R}$eo overview

$\mathcal{R}$eo [8, 9] has been explained in detail in Chapters 2 and 4. We briefly recall the most relevant aspects of $\mathcal{R}$eo for the sake of presentation. $\mathcal{R}$eo is a channel-based coordination model, wherein coordinating *connectors* are constructed compositionally out of more primitive connectors, which we call *primitives*. Primitives communicate through its *ports*, also called ends: primitives consume data through their *source ports*, and produce data through their *sink ports*. The behaviour of each primitive depends upon its current state. The semantics of a connector is described as a collection of possible steps for each state, and we refer to a round as the change of state triggered by one of these steps.

Some of the most commonly used $\mathcal{R}$eo primitives are depicted in Table 5.1. For the purposes of this chapter, we do not distinguish between primitives such as channels used for coordination and the components being coordinated, in that both will offer constraints describing their behavioural possibilities. The main difference between primitives and components is that the connector has more control over and more knowledge about the possible behaviour of the former than the latter, though this distinction is blurred in the present framework.

Connectors are formed by plugging the ports of primitives together in a one-to-one fashion, connecting a sink port to a source port, to form *nodes*. A node is a logical place consisting of a sink port, a source port, or both a sink and a source port. We call nodes with a single port *boundary* nodes, represented by ○, and we call nodes with a sink and a source port mixed nodes, represented by ●. Data flow through a connector from primitive to primitive through nodes, subject to the constraint that nodes cannot buffer data. This means that the two ports in a node are synchronised and have the same dataflow—behaviourally, they are equal. Nodes can be handled transparently by using the same name for the two ports on the node, as the synchronisation and dataflow at the two ports is identical. We recall the example of the exclusive router connector in Example 5.2.1, introduced in Chapter 2, to illustrate $\mathcal{R}$eo's semantics.

5.2.1. EXAMPLE. The connector in Figure 5.1 is an exclusive router built by composing two LossySync channels (*b*-*e* and *d*-*g*), one SyncDrain (*c*-*f*), one Merger (*h*-*i*-*f*), and three Replicators (*a*-*b*-*c*-*d*, *e*-*j*-*h* and *g*-*i*-*k*).

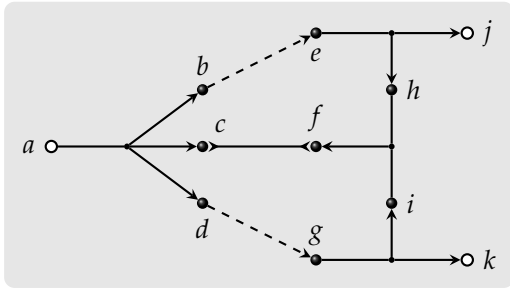


Figure 5.1: Exclusive router connector.

The constraints of these primitives can be combined to give the following two behavioural possibilities (plus the no flow everywhere possibility):

- ports $\{a, b, c, d, e, i, h, f\}$ synchronise and data flow from a to j; and
- ports $\{a, b, c, d, g, k, i, f\}$ synchronise and data flow from a to k.

A non-deterministic choice must be made whenever both behaviours are possible. Data can never flow from a to both j and k, as this is excluded by the behavioural constraints of the Merger *h*-*i*-*f*. ◊

We present in §5.3 a formal definition of the primitives used in this example, and we verify that the composition of the primitives yields the expected behaviour. In this chapter, we address the challenge of implementing $\mathcal{R}$eo by adopting the view of a $\mathcal{R}$eo connector as a set of constraints, based on the way the primitives are connected together, and their current state, governing the possible synchronisation and dataflow at the channel ends connected to external entities.

5.3 Coordination via constraint satisfaction

In this section we formalise the per-round semantics of $\mathcal{R}$eo primitives and their composition as a set of constraints. The possible coordination patterns can then be determined using traditional constraint satisfaction techniques.

The constraint-based approach to $\mathcal{R}$eo is developed in three phases:

synchronisation and dataflow constraints describe synchronisation and, the dataflow possibilities for a single step;

state constraints incorporate next state behaviour into the constraints, enabling the complete description of behaviour in terms of constraints; and

context constraints capture the availability of dataflow for a single step.

The resulting model significantly extends $\mathcal{R}$eo implementations with data-aware and context dependent behaviour, and enables more efficient implementation techniques. We start by focusing on synchronisation constraints (SC) and dataflow constraints (DFC), showing its correctness with respect to the constraint automata model of $\mathcal{R}$eo, before exploring the context constraints and its relation with the connector colouring model.

5.3.1 Mathematical preliminaries

Let $\mathbb{P}$ be a global set of ports of $\mathcal{R}$eo connectors. Let $\widehat{\mathbb{P}}$ denote the set of variables of $\mathbb{P}$ decorated with a little hat. Let $\mathbb{D}$ be the domain of data, and define $\mathbb{D}_\perp \stackrel{\text{def}}{=} \mathbb{D} \cup \{\texttt{NO-FLOW}\}$, where $\texttt{NO-FLOW} \notin \mathbb{D}$ represents 'no dataflow.' Constraints are expressed in quantifier-free, first-order logic over two kinds of variables: *synchronisation variables* $x \in \mathbb{P}$, which are boolean variables, and *dataflow variables* $\hat{x} \in \widehat{\mathbb{P}}$, which are variables over $\mathbb{D}_\perp$. Constraints are formula in the following grammar:

$$
\begin{aligned}
t &::= \hat{x} \mid d && \text{(terms)}\\
a &::= x \mid R(t_1, \ldots, t_n) && \text{(atoms)}\\
\psi &::= a \mid \top \mid \psi \wedge \psi \mid \neg\psi && \text{(formulæ)}
\end{aligned}
$$

where $d \in \mathbb{D}_\perp$ is a data item, $\top$ is *true*, and R is an n-ary predicate over terms. One such predicate is equality, which is denoted using the standard infix notation $t_1 = t_2$. The other logical connectives can be encoded as usual.

A *solution* to a formula ψ defined over ports $\mathcal{P} \subseteq \mathbb{P}$ is a pair of assignments of types $\sigma : \mathcal{P} \to \{\perp, \top\}$ and $\delta : \widehat{\mathcal{P}} \to \mathbb{D}_\perp$, such that σ and δ satisfy ψ, according to the satisfaction relation $\sigma, \delta \vDash \psi$, defined as follows:

$$
\begin{array}{ll}
\sigma, \delta \vDash \top & \sigma, \delta \vDash \psi_1 \wedge \psi_2 \quad \textit{iff} \quad \sigma, \delta \vDash \psi_1 \text{ and } \sigma, \delta \vDash \psi_2\\
\sigma, \delta \vDash x \quad \textit{iff} \quad \sigma(x) = \top & \sigma, \delta \vDash \neg\psi \quad \textit{iff} \quad \sigma, \delta \nvDash \psi\\
\sigma, \delta \vDash R(t_1, \ldots, t_n) \quad \textit{iff} \quad (\delta' \circ (t_1), \ldots, \delta'(t_n)) \in \mathcal{I}(R) &
\end{array}
$$

Each n-ary predicate symbol R has an associated interpretation, denoted by $\mathcal{I}(R)$, such that $\mathcal{I}(R) \subseteq \mathbb{D}_\perp^n$. The data item $\delta'(t)$ is either $\delta(\hat{x})$ or d, depending on t being the variable $\hat{x}$ or the constant d, respectively.

The logic with constraints over boolean variables, plus equality constraints over dataflow variables, arbitrary terms over the flat $\mathbb{D}$ domain, and top-level existential quantifiers is in *NP* [100].

The NO-FLOW value is used in constraints as a special value when no dataflow occurs. A synchronisation variable set to $\bot$ plays exactly the same role. These two facts are linked by the following constraint, which combines synchronisation and dataflow by capturing the relationship between no flow on port $x \in \mathcal{P}$ and the value NO-FLOW:

$$\neg x \leftrightarrow (\widehat{x} = \texttt{NO-FLOW}) \qquad \text{(flow axiom)}$$

This axiom applies to all ports in the connector. Let $\texttt{Flow}(\mathcal{P})$ denote $\bigwedge_{x \in \mathcal{P}}(\neg x \leftrightarrow (\widehat{x} = \texttt{NO-FLOW}))$, where $\mathcal{P} \subseteq \mathbb{P}$. A solution to a set of constraints that satisfies $\texttt{Flow}(\mathcal{P})$ is called a *firing*. Since we are exclusively interested in finding firings (as opposed to other solutions), we assume that the flow axiom holds for all ports involved.

5.3.2 Encoding primitives as constraints

Two kinds of constraints describe connector behaviour: *synchronisation constraints* (SC) and *dataflow constraints* (DFC). The former are constraints over a set $\mathcal{P}$ of boolean variables, describing the presence or absence of dataflow at each port—that is, whether or not those ports synchronise. The latter constraints involve in addition dataflow variables from $\widehat{\mathcal{P}}$ to describe the dataflow at the ports that synchronise.

Table 5.1 provides the semantics of some commonly used channels and other primitives in terms of synchronisation constraints and dataflow constraints. Note that some of our connectors can have an infinite number of states, especially when data is involved—a complete account of stateful connectors is given in §5.4 where state handling is introduced into constraints.

Sync, SyncDrain and SyncSpout channels All three synchronous channels allow dataflow to occur only synchronously at both channel ends. SyncDrains can be viewed as data consumers, and SyncSpouts can be viewed as data generators. A possible variant uses predicates R and Q to constrain the data produced, with a dataflow constraint such as $a \rightarrow (R(\widehat{a}) \wedge Q(\widehat{b}))$.

AsyncDrain and AsyncSpout These asynchronous channels allow flow on at most one of their two ports per round. A refined variant of the AsyncSpout has dataflow constraint $a \rightarrow R(\widehat{a}) \wedge b \rightarrow Q(\widehat{b})$.

Non-deterministic LossySync A LossySync always allows dataflow on port a. It can in addition, non-deterministically, it allows dataflow on port b, in which case the data from a is passed to b. Note that the LossySync here is not context-dependent, i.e., it may lose data when b can accept data. We return to context dependency in §5.5.

Merger A merger permits dataflow synchronously through one of its source ports, exclusively, to its sink port.

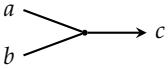
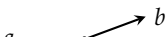
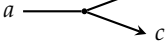
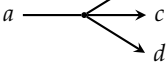
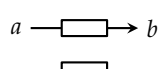
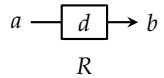
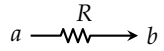
Channel	Representation	SC	DFC
Sync		$a \leftrightarrow b$	$\widehat{a} = \widehat{b}$
SyncDrain		$a \leftrightarrow b$	$\top$
SyncSpout		$a \leftrightarrow b$	$\top$
AsyncDrain		$\neg(a \wedge b)$	$\top$
AsyncSpout		$\neg(a \wedge b)$	$\top$
LossySync		$b \rightarrow a$	$b \rightarrow (\widehat{a} = \widehat{b})$
Merger		$(c \leftrightarrow (a \vee b)) \wedge$ $\neg(a \wedge b)$	$a \rightarrow (\widehat{c} = \widehat{a}) \wedge$ $b \rightarrow (\widehat{c} = \widehat{b})$
Replicator		$(a \leftrightarrow b) \wedge$ $(a \leftrightarrow c)$	$\widehat{b} = \widehat{a} \wedge \widehat{c} = \widehat{a}$
3-Replicator		$(a \leftrightarrow b) \wedge$ $(a \leftrightarrow c) \wedge (a \leftrightarrow d)$	$\widehat{b} = \widehat{a} \wedge$ $\widehat{c} = \widehat{a} \wedge \widehat{d} = \widehat{a}$
FIFOEmpty_1		$\neg b$	$\top$
$\text{FIFOFull}_1(d)$		$\neg a$	$b \rightarrow (\widehat{b} = d)$
Filter(R)		$b \rightarrow a$	$b \rightarrow (R(\widehat{a}) \wedge \widehat{a} = \widehat{b}) \wedge$ $(a \wedge R(\widehat{a})) \rightarrow b$

Table 5.1: Encodings of $\mathcal{R}$eo primitives.

Replicators A replicator and a 3-replicator allows data to flow only synchronously at every channel end. Data is replicated from the source port to every sink port. The constraints for the n-replicator (such as the 3-replicator found in Example 5.2.1) can be easily derived based on the constraints for the replicator, as show above.

FIFOEmpty_1 and $\text{FIFOFull}_1(d)$ FIFO_1 is a stateful channel representing a buffer of size 1. When the buffer is empty it can only receive data on a, but never output data on b. When it is full with data d, it can only output d through b, but cannot receive data on a at the same time.

Filter A filter permits data matching its filter predicate $R(\widehat{x})$ to pass through synchronously, otherwise the data is discarded.

In our approach we use logical formula to describe the behaviour of $\mathcal{R}$eo primitives as both synchronisation and dataflow constraints and require only the flow axiom to hold. Note that this precludes channels which, for example, profess to

offer quality of service guarantees. However, this constraint-based framework allows arbitrary complex concepts to be added to the behaviour of primitives, provided the underlying constraint solver can reason about these.

Notably, the constraints for some channels, such as SyncDrain and SyncSpout, are identical, indicating that the model does not strongly account for the direction of dataflow. Typically, however, some variables will be bound to a value and others will remain unbound, and data can be seen as flowing from the bound variables to the unbound ones. In $\mathcal{R}$eo, the direction of the dataflow is used to govern the well-formedness of connector composition, so that connectors have the expected semantics, but our constraints ignore this. Our constraints will be solved classically, in contrast to the intuitionistic model of Clarke [34], which was designed to avoid causality problems resulting from considering the direction of dataflow. Yet, in our setting the direction of dataflow can be used to optimise the constraint solving, as it is generally more efficient to start with constrained variables than with unconstrained variables, but we do not explore optimisations beyond those provided in the constraint solver underlying our prototype implementation.

Other channels can use non-trivial predicates over more than one argument. For example, it is possible to define a special synchronous drain variant whose predicate $R(\widehat{a}, \widehat{b})$ constrains the data on both of its ports, for instance, by requiring that they are equal or that the content of a field containing the geographic location corresponding to the data on $\widehat{a}$ is nearby the location of $\widehat{b}$. The dataflow constraints of this variation of the synchronous drain can be defined, for example, as $\mathit{SameLocation}(\widehat{a}.\texttt{location}, \widehat{b}.\texttt{location})$, assuming that $-.\texttt{location}$ extracts the location field from the data and the predicate *SameLocation* determines whether two locations are the same or not.

Splitting the constraints into synchronisation and dataflow constraints is very natural, and it closely resembles the constraint automata model [22] (see §5.4.3). It also enables some implementation optimisations. Following Sheini and Sakallah [96], for example, a SAT solver can be applied to the synchronisation constraints, efficiently ruling out many non-solutions. In many cases, a solution to the synchronisation constraints actually guarantees that a solution to the dataflow constraints exists. The only primitive in Table 5.1 for which this is not true is the filter, as it inspects the data in order to determine its synchronisation constraints.

5.3.3 Combining connectors

Two connectors can be plugged together whenever for each port x appearing in both connectors, x is only a sink port in one connector and only a source port in the other.[1] If the constraints for two connectors are ψ_1 and ψ_2, then the constraints for

[1] The information about whether a port is a sink or source needs to be maintained at a level above the constraints. Incorporating such information into the constraints is straightforward.

their composition is simply $\psi_1 \wedge \psi_2$. Existential quantification, such as $\exists x.\exists \widehat{x}.\psi$, can be used to abstract away from variables associated to intermediate channel ends (such as x and $\widehat{x}$).

Top-level constraints are given by the following grammar:

$$\mathcal{C} \;::=\; \psi \mid \mathcal{C} \wedge \mathcal{C} \mid \exists x.\mathcal{C} \mid \exists \widehat{x}.\mathcal{C} \qquad \text{(top-level constraints)}$$

where ψ is as defined in §5.3.1. Top-level constraints are used to introduce the existential quantifier, which hides the names of internal ports after composition is performed. This opens new possibilities for optimisation by ignoring ports that are not relevant for the final solution. This optimisation can be done statically, but our engine implementations do not address this, beyond what is already handled in the SAT and constraint solvers.

The satisfaction relation is extended with the cases:

$$\begin{array}{lll} \sigma, \delta \vDash \exists x.\mathcal{C} & \textit{iff} & \text{there exists a } b \in \{\top, \bot\} \text{ such that } \sigma, \delta \vDash \mathcal{C}[b/x] \\ \sigma, \delta \vDash \exists \widehat{x}.\mathcal{C} & \textit{iff} & \text{there exists a } d \in \mathbb{D}_\bot \text{ such that } \sigma, \delta \vDash \mathcal{C}[d/\widehat{x}]. \end{array}$$

$\mathcal{C}[a/x]$ is the constraint resulting from replacing all free occurrences of x by a in $\mathcal{C}$, in the usual fashion. Similar for $\mathcal{C}[d/\widehat{x}]$.

The following constraints describe the composition of the primitives for the connector presented in Example 5.2.1, abstracting away the internal ports:

$$\begin{array}{ll} \Psi_{SC} & = (a \leftrightarrow b) \wedge (a \leftrightarrow c) \wedge (a \leftrightarrow d) \wedge (e \rightarrow b) \wedge (c \leftrightarrow f) \wedge (g \rightarrow d) \wedge \\ & \quad (e \leftrightarrow j) \wedge (e \leftrightarrow h) \wedge (f \leftrightarrow (h \vee i)) \wedge \neg(h \wedge i) \wedge (g \leftrightarrow i) \wedge (g \leftrightarrow k) \\ \Psi_{DFC} & = (a \rightarrow (\widehat{b} = \widehat{a} \wedge \widehat{c} = \widehat{a})) \wedge (e \rightarrow \widehat{b} = \widehat{e}) \wedge (g \rightarrow \widehat{d} = \widehat{g}) \wedge \widehat{j} = \widehat{e} \wedge \widehat{h} = \widehat{e} \wedge \\ & \quad (h \rightarrow \widehat{f} = \widehat{h}) \wedge (i \rightarrow \widehat{f} = \widehat{i}) \wedge \widehat{i} = \widehat{g} \wedge \widehat{k} = \widehat{g} \\ \Psi & = \exists N.\exists \widehat{N}.(\Psi_{SC} \wedge \Psi_{DFC} \wedge \texttt{Flow}(N \cup \{a, j, k\}) \\ & \quad \text{where } N = \{b, c, d, e, f, g, h, i\} \end{array}$$

A SAT solver can quickly solve the synchronisation constraint Ψ_{SC} (ignoring internal ports):[2]

$$\sigma_1 \;=\; a \wedge j \wedge \neg k \qquad \sigma_2 \;=\; a \wedge \neg j \wedge k \qquad \sigma_3 \;=\; \neg a \wedge \neg j \wedge \neg k$$

We now include the dataflow constraints and use the flow axiom to guarantee consistency between the value of the (boolean) synchronous variables and the dataflow variables. Hence, using these three solutions, Ψ can be simplified using standard techniques as follows:

$$\begin{array}{lll} \Psi \wedge \sigma_1 & \rightsquigarrow & \widehat{j} = \widehat{a} \;\wedge\; \widehat{k} = \texttt{NO-FLOW} \\ \Psi \wedge \sigma_2 & \rightsquigarrow & \widehat{k} = \widehat{a} \;\wedge\; \widehat{j} = \texttt{NO-FLOW} \\ \Psi \wedge \sigma_3 & \rightsquigarrow & \widehat{a} = \texttt{NO-FLOW} \;\wedge\; \widehat{j} = \texttt{NO-FLOW} \;\wedge\; \widehat{k} = \texttt{NO-FLOW} \end{array}$$

These solutions say that data can flow either from port a to j but not to k, or from port a to k but not to j, or that no flow is possible in any of the ports, as expected.

[2] We use the open source SAT solver: `http://www.sat4j.org/`.

5.4 Adding state

Adding constraints to capture stateful primitives is relatively easy. This involves, firstly, adding constraints to capture the pre- and post-states for each interaction described by the constraints, and, secondly, providing details of a $\mathcal{R}$eo engine that updates constraints in each round based on solutions for the current round. In addition, we show the correspondence between our encoding and the semantics of $\mathcal{R}$eo in terms of constraint automata (CA), introduced in §2.2.2.

5.4.1 Encoding state machines

Some primitives, such as the FIFO_1 channel, are stateful, i.e., their state and subsequent behaviour change after data have flown through the channel. This is exemplified in the constraint automata (CA) semantics of $\mathcal{R}$eo [22]. The CA model is described in §2.2.2. We briefly summarise the CA model later in this chapter. For now, consider the CA of a FIFO_1 channel shown in Figure 5.2. Its initial state is `empty`. From this state the automaton can take a transition to state $\texttt{full}(d)$ if there is dataflow on port a, excluding dataflow on port b. The constraint $d = \widehat{a}$ models the storing of the value flowing on port a into the internal state variable d. The transition from $\texttt{full}(d)$ to `empty` is read in a similar way, except that the data is moved from the internal state variable d to port b.

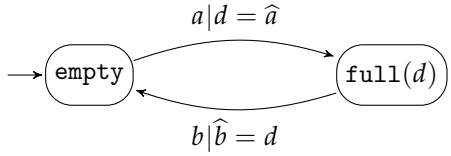


Figure 5.2: Constraint Automata for the FIFO_1 channel

To encode state information, the logic introduced in §5.3.1 is extended so that terms also include n-ary uninterpreted function symbols:

$$t \;::=\; \widehat{x} \mid f(t_1, \ldots, t_n) \qquad \text{(terms)}$$

By including function symbols, a term can have a functional structure rather than just being a variable or a constant. A term t is *ground* iff $t = f(t_1, \ldots, t_n)$ and , for $n > 0$, each t_i for $1 \leq i \leq n$ is ground. Thus a term containing a variable is not ground. The set $\mathbb{D}$ described earlier can now be seen as ground 0-ary uninterpreted function symbols; we now consider $\mathbb{D}$ to be the Herbrand universe over the set of uninterpreted function symbols.

Let S be the set of individual stateful primitives in a connector. Add a new set of term variables $state_p$ and $state'_p$, for each $p \in S$, to denote the state before and

after the present step. State machines of primitives are encoded by encoding their constraint automata [69]. In §5.4.3 we present the correctness and the compositionality of our encoding with respect to constraint automata. For example, the state machine of a FIFO_1 channel is encoded as the formula:

$$
\begin{aligned}
&state_{\text{FIFO1}} = \texttt{empty} \rightarrow \\
&\qquad (\neg b \ \wedge \ a \rightarrow (state'_{\text{FIFO1}} = \texttt{full}(\widehat{a})) \ \wedge \ \neg a \rightarrow (state'_{\text{FIFO1}} = state_{\text{FIFO1}})) \\
&\wedge\, state_{\text{FIFO1}} = \texttt{full}(d) \rightarrow \\
&\qquad (\neg a \ \wedge \ b \rightarrow (\widehat{b} = d \wedge state'_{\text{FIFO1}} = \texttt{empty}) \wedge > \neg b \rightarrow (state'_{\text{FIFO1}} = state_{\text{FIFO1}})) \\
&\wedge\, (\neg a \wedge \neg b) \rightarrow (state'_{\text{FIFO1}} = state_{\text{FIFO1}})
\end{aligned}
$$

The final conjunct captures that no transition occurs when there is no dataflow.

To complete the encoding, we add a formula describing the present state, i.e., we add conjunctively a formula that defines the value of $state_p$ for each stateful primitive p, for the current state. In our example, the fact that the FIFO_1 is in the empty state is recorded by the formula $state_{\text{FIFO1}} = \texttt{empty}$, whereas the fact that it is in the full state, containing data d, is recorded by $state_{\text{FIFO1}} = \texttt{full}(d)$.

In general, the state of primitives will be encoded as a formula of the form $\bigwedge_{p \in S} state_p = t_p$, where t_p is a ground term representing the current state of p. This is called a *pre-state vector*. Similarly, $\bigwedge_{p \in S} state'_p = t_p$, is called the *post-state vector*. The pre-state vector describes the state of the connector before constraint satisfaction; the post-state vector describes the state after constraint satisfaction, that is, it gives the next state. Note that stateless primitives do not contribute to the state vector.

5.4.2 A constraint satisfaction-based engine for $\mathcal{R}$eo

Constraint satisfaction techniques can now form the heart of an implementation of an *engine* performing the coordination described in $\mathcal{R}$eo connectors. The engine holds the current set of constraints, called a *configuration*, and operates in *rounds*, each of which consists of a *solve* step, which produces a solution for the constraints, and an *update* step, which uses the solution to update the constraints to model the transition to a new state. This is depicted in the diagram in Figure 5.3.

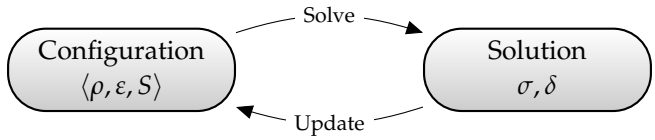


Figure 5.3: Phases of the $\mathcal{R}$eo Engine

The *configuration* of the engine is a triple $\langle \rho, \varepsilon, S \rangle$, where ρ represents *persistent* constraints, ε represents *ephemeral* constraints, and S is the set of stateful primitives in the connector. The persistent constraints are eternally true for a connector, including constraints such as the description of the state machines of the primitives. The ephemeral constraints include the encoding of the pre-state vector. These constraints are updated each round, using a pair of assignments σ, δ that satisfy the persistent and ephemeral constraints. A full round can be represented as follows, where the superscript indicates the round number:

$$\langle \rho, \varepsilon^n, S \rangle \xrightarrow{solve} \langle \sigma^n, \delta^n \rangle \xrightarrow{update} \langle \rho, \varepsilon^{n+1}, S \rangle$$

satisfying:

$$\sigma^n, \delta^n \vDash \rho \wedge \varepsilon^n \qquad \text{(solve)}$$

$$\varepsilon^{n+1} \equiv \bigwedge_{p \in S} state_p = \delta^n(state'_p) \qquad \text{(update)}$$

We use the notation $\langle \rho, \varepsilon, S \rangle \xrightarrow{\sigma,\delta} \langle \rho, \varepsilon', S \rangle$ to denote that $\sigma, \delta \vDash \rho \wedge \varepsilon$ and $\varepsilon' = \bigwedge_{p \in S} state_p = \delta(state'_p)$. The new current state, represented by $state'_p$, is the update of the previous state, represented by $state_p$. Furthermore, we write $c \to c'$ when there is a pair σ, δ such that $c \xrightarrow{\sigma,\delta} c'$, and we write $\to^*$ to denote the reflexive and transitive closure of $\to$. We use *Conf* to denote the set of all configurations. We now provide a more formal account of our encoding of constraint automata into constraints.

5.4.3 Correctness via constraint automata

In this subsection we address the correctness of our approach. We base our argument on the constraint automata model of $\mathcal{R}$eo [22], already described in Chapter 2. We briefly recapitulate the constraint automata model below, and define the language accepted by an automaton. In the process, we fully formalise the encoding described above.

Constraint automata

We recall the formal definition of constraint automata presented earlier in §2.2.2. Define $DC_{\mathcal{P}}$ to be the set of constraints in our language above only over variables in the set $\widehat{\mathcal{P}}$, where $\mathcal{P} \subseteq \mathbb{P}$ and the underlying data domain is $\mathbb{D}$. This excludes constraints over synchronisation variables and over constraints involving `NO-FLOW`. When clear from the context, we write xyz instead of $\{x, y, z\}$ to increase readability, where $x, y, z \in \mathbb{P}$.

A *constraint automaton* over data domain $\mathbb{D}$) is a tuple $\mathcal{A} = \langle Q, \mathcal{P}, \longrightarrow, Q_0 \rangle$, where Q is a set of states, $\mathcal{P} \subseteq \mathbb{P}$ is a finite set of port names, $\longrightarrow$ is a subset of

$Q \times 2^{\mathcal{P}} \times DC_{\widehat{\mathcal{P}}} \times Q$, called the transition relation of $\mathcal{A}$, and $Q_0 \subseteq Q$ is the set of initial states. We write $q \xrightarrow{N|g} p$ instead of $(q, N, g, p) \in \longrightarrow$. For every transition $q \xrightarrow{N|g} p$, we require that g, the guard, is a $DC_{\widehat{N}}$-constraint. For every state $q \in Q$, there is a transition $q \xrightarrow{\emptyset|\top} q$.

The meaning of transition the $q \xrightarrow{N|g} p$ is that in state q data flow at the ports in the set N, while excluding flow at ports in $\mathcal{P} \setminus N$. The data flowing through the ports N satisfies the constraint g, and the resulting state is p. Thus, in constraint automata, synchronisation is described by the set N and dataflow is described by the constraint g. The transition $q \xrightarrow{\emptyset|\top} q$ is present for technical reasons, namely, to simplify the definition of the product, below.

In order to define the language accepted by a constraint automaton,[3] we need to set up some preliminary definitions. Given a set of port names $\mathcal{P}$, define *data assignments* to be given by $\Sigma_{\mathcal{P}} = (\widehat{\mathcal{P}} \rightharpoonup \mathbb{D})$, namely the partial finite maps from the dataflow variables to the data domain. Define $\delta \vDash g$, where $\delta : \widehat{\mathcal{P}} \rightharpoonup \mathbb{D}$ and data constraint $g \in DC_{\mathcal{P}}$, as $\emptyset, \delta \vDash g$ (from §5.3.1). Observe that this will be well formed, as data constraints $DC_{\mathcal{P}}$ do not mention synchronisation variables. We will interpret $\Sigma_{\mathcal{P}}$ as an alphabet. Automata will accept a finite word from the set $\Sigma_{\mathcal{P}}^*$. As automata have no final states specified, we assume that all states are accepting states. We formalise the evolution of a constraint automaton using the notion of a step.

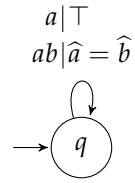
5.4.1. DEFINITION (STEP). A q-step for a constraint automaton $\mathcal{A}$ is given by $q \xrightarrow{\delta} p$, where δ is a data assignment, and there is a transition $q \xrightarrow{N|g} p$ in $\mathcal{A}$ such that $\widehat{N} = dom(\delta)$ and $\delta \vDash g$. ◁

The behaviour of a constraint automaton is expressed in terms of *runs*. A run, as defined by Baier et al. [22], can be described as a sequence of possible steps of the automaton. A q-run of a constraint automaton $\mathcal{A}$ is a finite sequence $q_0 \xrightarrow{\delta_0} q_1 \xrightarrow{\delta_1} \cdots \xrightarrow{\delta_{n-1}} q_n$, where $q_0 = q$ and each $q_i \xrightarrow{\delta_i} q_{i+1}$ is a q_i-step for $\mathcal{A}$. The language accepted by the automaton $\mathcal{A} = \langle Q, \mathcal{P}, \longrightarrow, Q_0 \rangle$ is denoted $\mathcal{L}(\mathcal{A})$, defined as follows, where $\mathcal{L}_q^n$ denotes the words accepted in state q of length n, and $\mathcal{L}_q$ denotes all words accepted in state q:

$$\begin{aligned}
\mathcal{L}_q^0 &= \{\epsilon\} \\
\mathcal{L}_q^{n+1} &= \left\{ \delta.w \mid q \xrightarrow{\delta} q' \text{ is a } q\text{-step for } \mathcal{A}, w \in \mathcal{L}_{q'}^n \right\} \\
\mathcal{L}_q &= \bigcup_{i \geq 0} \mathcal{L}_q^i \\
\mathcal{L}(\mathcal{A}) &= \bigcup_{q \in Q_0} \mathcal{L}_q.
\end{aligned}$$

5.4.2. EXAMPLE. Most primitives in Table 5.1 are stateless, which means each of their corresponding constraint automata has a single state. The LossySync channel is formalised by the automaton $\langle Q, \mathcal{P}, \longrightarrow, Q_0 \rangle$ depicted on the right below, with one state and two transitions, where:

$$\begin{aligned} Q &= \{q\} \\ \mathcal{P} &= \{a, b\} \\ Q_0 &= \ q \\ \longrightarrow &= \{(q, \{a\}, \top, q), (q, \{a, b\}, \widehat{a} = \widehat{b}, q)\}. \end{aligned}$$



◊

5.4.3. EXAMPLE. The FIFO_1 channel, already depicted as a constraint automaton in Figure 5.2, has multiple states, and is formalised by the automaton $\langle Q, \mathcal{P}, \longrightarrow, Q_0 \rangle$ where:

$$\begin{aligned} Q &= \{\texttt{empty}\} \cup \{\texttt{full}(d) \mid d \in \mathbb{D}\} & \mathcal{P} &= \{a, b\} \\ \longrightarrow &= \{(\texttt{empty}, \{a\}, \widehat{a} = d, \texttt{full}(d)) \mid d \in \mathbb{D}\} & Q_0 &= \texttt{empty}. \\ &\cup \{(\texttt{full}(d), \{b\}, \widehat{b} = d, \texttt{empty}) \mid d \in \mathbb{D}\} \end{aligned}$$

◊

The trivial transition has been omitted in the two previous examples.

Encoding constraint automata as constraints

Given a constraint automata $\mathcal{A} = \langle Q, \mathcal{P}, \longrightarrow, Q_0 \rangle$, an obvious correspondence exists between subsets N of $\mathcal{P}$ and functions $\mathcal{P} \to \{\bot, \top\}$. Define $\chi_N : \mathcal{P} \to \{\bot, \top\}$ such that $\chi_N(x) = \top$ if and only if $x \in N$. If δ is defined over $\widehat{N} \subseteq \widehat{\mathcal{P}}$, that is, $\delta : \widehat{N} \to \mathbb{D}$, define δ^+ to extend δ to a map in $\widehat{\mathcal{P}}$ such that for each $\widehat{x} \in \widehat{\mathcal{P}} \setminus \widehat{N}$, $\delta^+(\widehat{x}) = \texttt{NO-FLOW}$.

5.4.4. DEFINITION (ENCODING OF STATES). Recall that we deal with constraints on a per-state basis. The following conditions characterise constraint ψ_q corresponding to state $q \in Q$, where $q \xrightarrow{N_1|g_1} p_1, \cdots, q \xrightarrow{N_n|g_n} p_n$ are the transitions in the automaton with source q:

(*a*) For all $\sigma : \mathcal{P} \to \{\bot, \top\}$, $\delta : \widehat{\mathcal{P}} \cup \{state, state'\} \to \mathbb{D}$ and $p \in Q$, such that $\sigma, \delta \vDash \psi_q$, $\delta(state) = q$ and $\delta(state') = p$, there is a transition $q \xrightarrow{N_i|g_i} p$ such that $\sigma = \chi_{N_i}$ and $\delta \vDash g_i$.

[3]Other notions of the language accepted by a constraint automaton are possible too. The original account [22] considers relations over timed data streams, each of which is a pair of *infinite* streams of positive real numbers and data values. This notion of language is, however, problematic as it fails to account for finite length computations and for infinite computations where there is dataflow at one or more ports only a finite number of times. For example, in such a setting the automaton with no transitions has *no* semantics, instead of the empty set of words.

(*b*) For all transitions $q \xrightarrow{N_i|g_i} p_i$ and for all $\delta : \widehat{N_i} \to \mathbb{D}$ such that $\delta \vDash g_i$, we have that $\chi_{N_i}, \delta^+ \cup \{state \mapsto q, state' \mapsto p_i\} \vDash \psi_q$. ◁

When the conditions (a) and (b) from Definition 5.4.4 hold for a constraint ψ_q, we say that ψ_q *encodes state* q. These two conditions state that for each dataflow described by the constraint, there is a transition in the automaton with the same dataflow, and vice versa. (Note that if there is only one state, all mentioning of variables referring to states can be dropped.)

These constraints are put together to describe the entire state machine as follows:

$$\rho_{\mathcal{A}} \;=\; \bigwedge_{q_i \in Q} ((state = q_i) \to \psi_{q_i}). \tag{5.1}$$

5.4.5. EXAMPLE. The constraints for the FIFO_1 buffer presented in §5.4.1 are correct with respect to the constraint automaton in Example 5.4.3. We only show this for the initial state `empty`, but it can be easily verified also for the other state.

For the `empty` state, the corresponding constraints and their possible solutions are:

$$\psi_{\texttt{empty}} \;=\; \neg b \wedge a \to (state' = \texttt{full}(\widehat{a})) \wedge \neg a \to (state' = state)$$

$$\sigma_1 \;=\; a \wedge \neg b \qquad \sigma_2 \;=\; \neg a \wedge \neg b$$

$$\delta_1 \;=\; state' = \texttt{full}(\widehat{a}) \qquad \delta_2 \;=\; state' = \texttt{empty}$$

In the constraint automaton, the transitions from the `empty` state are:

$$\texttt{empty} \xrightarrow{a|d=\widehat{a}} \texttt{full}(d)$$

$$\texttt{empty} \xrightarrow{\emptyset|\top} \texttt{empty}$$

The two conditions that confirm the correctness of the constraint $\psi_{\texttt{empty}}$ can now be easily verified, after expanding the constraint $state' = \texttt{full}(\widehat{a})$ to the logically equivalent constraint $\exists d.(\widehat{a} = d \wedge state' = \texttt{full}(d))$. ◊

Correctness

Our correctness result shows that, given a constraint automaton of a stateful primitive, every step of the automaton corresponds to a solve-update round in our constraint satisfaction-based engine for $\mathcal{R}$eo, and vice-versa. Recall that $\rho_{\mathcal{A}}$, defined in Equation (5.1), denotes the encoding of the automaton $\mathcal{A}$ as a constraint. In the rest of this chapter we write $\langle \rho_{\mathcal{A}}, state = q \rangle$ to denote a configuration of the constraint solver, where $\rho_{\mathcal{A}}$ is the persistent constraint, $state = q$ is the ephemeral constraint, and the *state* denotes the state variable. Note that we omit the S component containing the set of known stateful primitives, which is used to manage interaction with multiple stateful primitives, as we are dealing with a single automaton only.

5.4.6. THEOREM. *Let $\mathcal{A}$ be a constraint automata and q a state of $\mathcal{A}$. Then the following holds:*

$$q \xrightarrow{\delta} p \text{ is a } q\text{-step of } \mathcal{A} \quad \text{iff} \quad \langle \rho_{\mathcal{A}}, state = q \rangle \xrightarrow{solve} \langle \sigma, \delta' \rangle \xrightarrow{update} \langle \rho_{\mathcal{A}}, state = p \rangle ,$$

where $\widehat{N} = dom(\delta)$, $\sigma = \chi_N$ and $\delta' = \delta^+ \cup \{state \mapsto q, state' \mapsto p\}$.

Proof. Recall Definition 5.4.4 which characterises the encoding of a state q as a constraint ψ_q, and the definition of the solve- and the update-arrow, presented in §5.4.2. The arrow $\langle \rho, \epsilon \rangle \xrightarrow{solve} \langle \sigma, \delta \rangle$ exists if and only if $\sigma, \delta \vDash \rho \wedge \epsilon$, while the arrow $\langle \sigma, \delta \rangle \xrightarrow{update} \langle \rho, \epsilon' \rangle$ exists if and only if $\epsilon' \equiv state = \delta(state')$.

- ($\Leftarrow$) Assume $\langle \rho_{\mathcal{A}}, state = q \rangle \xrightarrow{solve} \langle \sigma, \delta' \rangle \xrightarrow{update} \langle \rho_{\mathcal{A}}, state = p \rangle$. The first solve-arrow indicates that $\sigma, \delta' \vDash \rho_{\mathcal{A}} \wedge state = q$. Note that $\rho_{\mathcal{A}} \wedge state = q \Leftrightarrow \bigwedge_{r \in Q}(state = r \rightarrow \psi_r) \wedge state = q$, which implies ψ_q by modus ponens. Therefore $\sigma, \delta' \vDash \psi_q$. Define $\delta = \delta' \restriction \widehat{N}$, where $\restriction$ denotes the standard restriction of function domains. Observe that $\delta' = \delta^+ \cup \{state \mapsto q, state' \mapsto p\}$. Since $\delta'(state) = q$ and $\delta'(state') = p$, it follows from condition (a) in Definition 5.4.4 that there is a transition $q \xrightarrow{N|g} p$, where $\sigma = \chi_N$ and $\delta' \vDash g$. To show that $q \xrightarrow{\delta} p$ we just need to verify that also $\delta \vDash g$. This follows because g refers only to variables in $\widehat{N}$, and because $dom(\delta) = \widehat{N}$.

- ($\Rightarrow$) Assume $q \xrightarrow{\delta} p$. By the definition of a q-step, there is a transition $q \xrightarrow{N|g} p$ from $\mathcal{A}$ such that $\widehat{N} = dom(\delta)$ and $\delta \vDash g$. Let $\sigma = \chi_N$ and $\delta' = \delta^+ \cup \{state \mapsto q, state' \mapsto p\}$. From condition (b) in Definition 5.4.4, it follows that $\sigma, \delta' \vDash \psi_q$. Observe that, because $\delta'(state) = q$, we conclude that $\sigma, \delta' \vDash state = q$. Hence (1) $\sigma, \delta' \vDash (state = q \rightarrow \psi_q) \wedge state = q$. Furthermore, $\delta'(state) = q$ also implies that for every state $q' \neq q$, the formula $state = q'$ does not hold, thus (2) $\sigma, \delta' \vDash \bigwedge_{r \in Q \setminus \{q\}}(state = r \rightarrow \psi_r)$. From (1) and (2) we conclude that $\sigma, \delta' \vDash \bigwedge_{r \in Q}(state = r \rightarrow \psi_r) \wedge state = q$. Therefore, by the definition of the solve-arrow, $\langle \rho_{\mathcal{A}}, state = q \rangle \xrightarrow{solve} \langle \sigma, \delta' \rangle$. Finally, since $\delta'(state') = p$, we have by the definition of the update-arrow that $\langle \sigma, \delta' \rangle \xrightarrow{update} \langle \rho_{\mathcal{A}}, state = p \rangle$. □

Compositionality

We now argue that the composition of two constraint automata describing two connectors composed appropriately (sink-to-source) corresponds to composition (conjunction) of their corresponding constraints (per state). In both cases, the overlapping of port names corresponds to the places where connectors are joined. As-

sume that we have constraint automata $\mathcal{A}_i$ with domains $\mathcal{P}_i$, for $i \in \{1,2\}$. In constraint automata, the composition of $\mathcal{A}_1$ and $\mathcal{A}_2$ yields a new automaton where the transition relation $\rightarrow$ is given by the condition below.

$$(q_1,q_2) \xrightarrow{N_1 \cup N_2 | g_1 \wedge g_2} (p_1,p_2) \text{ iff}$$
$$q_1 \xrightarrow{N_1|g_1}_1 p_1, q_2 \xrightarrow{N_2|g_2}_2 p_2, \text{ and } N_1 \cap \mathcal{P}_2 = N_2 \cap \mathcal{P}_1$$

Assume that ψ_{q_1} and ψ_{q_2} are the constraints for states q_1 and q_2 of automata $\mathcal{A}_1$ and $\mathcal{A}_2$, respectively, as described above. We claim that $\psi_{q_1} \wedge \psi_{q_2}$ is the constraint modelling state (q_1,q_2) in the composite automaton given by the above rule. Note that we would need to add equations such as $state_{1\times 2} = (state_1, state_2)$ and $state'_{1\times 2} = (state'_1, state'_2)$ to make the format of the equations match. We will not pursue this here.

5.4.7. LEMMA. *Assume condition (a) from Definition 5.4.4 holds for two constraints ψ_{q_1} and ψ_{q_2}, and for automata $\mathcal{A}_1$ and $\mathcal{A}_2$ with domains $\mathcal{P}_1$ and $\mathcal{P}_2$, respectively. Then condition (a) also holds for $\psi_{q_1} \wedge \psi_{q_2}$ and constraint automaton $\mathcal{A}_1 \bowtie \mathcal{A}_2$.*

Proof. Let $\sigma \restriction \mathcal{P}$ denote σ restricted to domain $\mathcal{P}$. Assume that $\sigma, \delta \vDash \psi_{q_1} \wedge \psi_{q_2}$, where $\mathrm{dom}(\sigma) = \mathcal{P}_1 \cup \mathcal{P}_2$, $\mathrm{dom}(\delta) = \widehat{\mathcal{P}}_1 \cup \widehat{\mathcal{P}}_2 \cup \{state_1, state_2, state'_1, state'_2\}$, $\delta(state_i) = q_i$, and $\delta(state'_i) = p_i$, for $i \in \{1,2\}$. It follows that we have $\sigma \restriction \mathcal{P}_1, \delta \vDash \psi_{q_1}$ and $\sigma \restriction \mathcal{P}_2, \delta \vDash \psi_{q_2}$ and, from the properties of ψ_{q_i}, that there exists a transition $q_i \xrightarrow{N_i|g_i} p_i$ such that $\sigma \restriction \mathcal{P}_i = \chi_{N_i}$ and $\delta \vDash g_i$. Clearly, $\delta \vDash g_1 \wedge g_2$, so we are halfway there. Now we want to show $\sigma = \chi_{N_1 \cup N_2}$. This is simple, because $N_1 \cap \mathcal{P}_2 = N_2 \cap \mathcal{P}_1$ guarantees that functions χ_{N_1} and χ_{N_2} agree where their domains intersect. Thus we have $\chi_{N_1 \cup N_2} = \chi_{N_1} \cup \chi_{N_2} = \sigma \restriction \mathcal{P}_1 \cup \sigma \restriction \mathcal{P}_2 = \sigma$. □

5.4.8. LEMMA. *Assume condition (b) from Definition 5.4.4 holds for two constraints ψ_{q_1} and ψ_{q_2} and automata $\mathcal{A}_1$ and $\mathcal{A}_2$ with domains $\mathcal{P}_1$ and $\mathcal{P}_2$, respectively. Then condition (b) also holds for $\psi_{q_1} \wedge \psi_{q_2}$ and constraint automaton $\mathcal{A}_1 \bowtie \mathcal{A}_2$.*

Proof. Given a transition $(q_1,q_2) \xrightarrow{N_1 \cup N_2 | g_1 \wedge g_2} (p_1,p_2)$ in the product automaton, assume that we have δ such that $\mathrm{dom}(\delta) = \widehat{N_1} \cup \widehat{N_2}$ and $\delta \vDash g_1 \wedge g_2$. Firstly, we can conclude both that $\delta \vDash g_i$, for $i \in \{1,2\}$. From condition (b) of Definition 5.4.4 with respect to ψ_{q_i}, we obtain that $\chi_{N_i}, \delta^+ \cup \{state'_i \mapsto p_i\} \vDash \psi_{q_i}$. Now as $N_1 \cap \mathcal{P}_2 = N_2 \cap \mathcal{P}_1$, we obtain $\chi_{N_1} \cup \chi_{N_2} = \chi_{N_1 \cup N_2}$, as in the proof of Lemma 5.4.7. We have immediately that $\chi_{N_1 \cup N_2}, \delta^+ \cup \{state'_1 \mapsto p_1, state'_2 \mapsto p_2\} \vDash \psi_{q_i}$, for $i \in \{1,2\}$, hence $\chi_{N_1 \cup N_2}, \delta^+ \cup \{state'_1 \mapsto p_1, state'_2 \mapsto p_2\} \vDash \psi_{q_1} \wedge \psi_{q_2}$. □

Lemmas 5.4.7 and 5.4.8 show exactly our correctness result of the constraint engine with respect to constraint automata. We make our claim precise in the following theorem.

5.4.9. THEOREM. *If ψ_{q_1} encodes state q_1 from automaton $\mathcal{A}_1$ and ψ_{q_2} encodes state q_2 from automaton $\mathcal{A}_2$, then $\psi_{q_1} \wedge \psi_{q_2}$ encodes state (q_1, q_2) from automaton $\mathcal{A}_1 \bowtie \mathcal{A}_2$.*

Proof. Directly from Lemmas 5.4.7 and 5.4.8. □

5.5 Adding context dependency

We have introduced three different types of constraints to capture the behaviour of $\mathcal{R}$eo connectors, namely synchronisation, data flow, and state constraints. We now explore one further possibility and present context dependency as an additional extension that can be modelled by constraints.

One of the main contributions of the connector colouring (CC) framework [37] is a $\mathcal{R}$eo semantics that expresses *context dependency*, a feature missing from the constraint automata model. A primitive depends on its context if its behaviour changes when there is an increase of data flowing on its ports. That is, when additional dataflow is possible, earlier valid behaviour is not displayed anymore. Two important example primitives that cannot be represented in previous semantic models are:

Context-dependent LossySync This channel loses data written to its source only if the surrounding context is unable to accept the data through its sink; otherwise the data flow through the channel. This corresponds to the original intention of the LossySync channel [8].

Priority merger This is a special variant of a merger that favours one of its sink ports: if dataflow is possible at both sink ports, it prefers a particular port over the other.

Context dependency, as described by the CC semantics, supports a more expressive model than other $\mathcal{R}$eo semantics which lack this notion. Context dependency in $\mathcal{R}$eo is presented in more detail in §2.2.3, where we describe the $\mathcal{R}$eo automata model [29], and in §4.3, where we describe the connector colouring semantics [37]. In the rest of this section we recall the relevant definitions of the connector colouring semantics, which we encode as constraints and use to show the correctness of our approach.

5.5.1 Connector colouring: an overview

We now briefly recall the connector colouring (CC) semantics presented in Chapter 4, that we use as the basis to describe context dependency in our constraint framework. The connector colouring (CC) semantics, as presented by Clarke *et al.* [37], is based on the idea of colouring the ports of a connector using a set $\mathcal{C}olour$ of three colours—for orientation, the • indicates the port. One colour (——•)

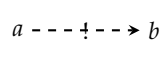
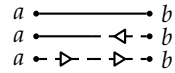
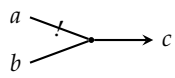
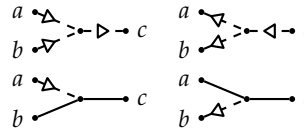
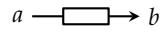
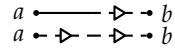
Channel	Representation	Colouring table
Context LossySync		
Priority Merger		
FIFOEmpty$_1$		

Table 5.2: Colouring tables for some primitives.

marks ports in the connector where data flow, and two colours mark the absence of dataflow (-◁-• and -▷-•). -◁-• denotes that the reason for no-flow originates from the context, and we say that the port *requires a reason* for no flow. Similarly, -▷-• indicates that the reason for no-flow originates from the primitive and we say that the port *gives a reason* for no-flow.

For each port $x \in \mathbb{P}$, we write $x^{\downarrow}$ to denote that x is a source port, and $x^{\uparrow}$ to denote that x is a sink port. Let $\mathbb{P}^{\updownarrow} = \{x^{\downarrow} \mid x \in \mathbb{P}\} \cup \{x^{\uparrow} \mid x \in \mathbb{P}\}$, and $X \subseteq \mathbb{P}^{\updownarrow}$. A *colouring* $c : X \to \mathcal{C}olour$ maps each port of a $\mathcal{R}$eo connector to a colour, and a *colouring table* T is a set of colourings for a connector, one for each possible behaviour for the current round. For simplicity, we write x instead of $x^{\downarrow}$ or $x^{\uparrow}$ when the direction of the port name can be inferred by looking at the associated connector, and we write 'a •——▷-• b' to denote the colouring $\{a^{\downarrow} \mapsto$ ——•, $b^{\uparrow} \mapsto$ -▷-•$\}$, when the corresponding direction of a and b can be inferred. We present in Table 5.2 the colouring tables for some primitives.

The composition of two connectors, formally defined below, is done by creating a new colouring table with all the possible colourings that result from matching the colours of their connected ports. The colour of two ports match if both represent flow, or if the reason for no-flow comes from at least one of the ports. That is, the valid combinations are: ——•——, -◁-•-◁-, -▷-•-◁- and -▷-•-▷-.

5.5.1. DEFINITION (PRODUCT). (Also presented in Definition 4.3.3.) The product of two colouring tables T_1 and T_2, denoted by $T_1 \bowtie T_2$, yields the colouring table.

$$\{c_1 \cup c_2 \mid c_1 \in T_1, c_2 \in T_2, c_1 \frown c_2\}$$

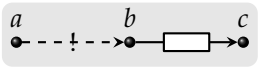
We define below the binary relation $\frown$ for both colourings and colours, relating only those that match, where c_1, c_2 are colourings and a, b are colours, and with

$\overline{\uparrow} = \downarrow$ and $\overline{\downarrow} = \uparrow$:

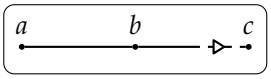
$$c_1 \frown c_2 \;\; \textit{iff} \;\; x^\circ \in dom(c_1) \wedge x^{\overline{\circ}} \in dom(c_2) \Rightarrow c_1(x^\circ) \frown c_2(x^{\overline{\circ}})$$
$$a \frown b \;\; \textit{iff} \;\; \langle a, b\rangle \in \{\langle \longrightarrow, \longrightarrow \rangle, \langle -\triangleleft\!\bullet, -\triangleright\!\bullet \rangle, \langle -\triangleright\!\bullet, -\triangleleft\!\bullet \rangle, \langle -\triangleright\!\bullet, -\triangleright\!\bullet \rangle\} \qquad \triangleleft$$

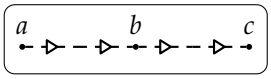
We now use the same example from §4.3 for the composition of two $\mathcal{R}$eo primitives, that we later encode also using our constraint approach.

5.5.2. EXAMPLE. (Also presented in Example 4.3.4.) We compose a context-dependent LossySync with a FIFOEmpty$_1$ channel:



Composing the colouring tables of both primitives, presented in Table 5.2, results in the colouring table of the connector, illustrated as follows:





Each image represents a valid colouring, and only these two colourings exist for this connector. The first colouring corresponds to the flow of data through the context-dependent LossySync and into the FIFO$_1$ buffer, and the second colouring corresponds to the absence of flow in the connector, with a reason for this absence required from port a. In both colourings, port c of the FIFO$_1$ buffer gives a reason for no dataflow. $\Diamond$

5.5.2 Context constraints

To capture context dependency, the constraint-based semantics is extended with an extra set of *context constraints* defined in terms of synchronisation variables ($\mathbb{P}$) and a new set of variables called *context variables*. The flow axiom is also updated to link the two sets of variables.

Context variables represent the direction of the reasons for no-flow, and context constraints reflect the valid combinations for the context variables. Context variables are given by the following set:

$$\{x_{\mathsf{snk}} \mid x \in \mathbb{P}\} \cup \{x_{\mathsf{src}} \mid x \in \mathbb{P}.\}$$

The context variable x_{snk} is used when the port x is a sink port, and x_{src} when the port x is a source port. We also write $\mathcal{P}_{\mathsf{snk}}$ and $\mathcal{P}_{\mathsf{src}}$ to denote the sets $\{x_{\mathsf{snk}} \mid x \in \mathcal{P}\}$ and $\{x_{\mathsf{src}} \mid x \in \mathcal{P}\}$, respectively. Note that for a port x the constraints on the two context variables x_{snk} and x_{src} typically occur in different primitives—the two primitives connected at node x. Thus for each mixed port in a connector, we have

Channel	Context Constraints
	$\neg a \to (\neg b \wedge \neg a_{\mathsf{src}} \wedge b_{\mathsf{snk}}) \ \wedge \ \neg b \to ((a \wedge \neg b_{\mathsf{snk}}) \vee \neg a)$
	$(c \wedge \neg a) \to \neg a_{\mathsf{src}} \wedge$ $(c \wedge \neg b) \to b_{\mathsf{src}} \ \wedge \ \neg c \to ((\neg a_{\mathsf{src}} \wedge \neg b_{\mathsf{src}}) \vee \neg c_{\mathsf{snk}})$
	$(\neg a \to \neg a_{\mathsf{src}}) \wedge b_{\mathsf{snk}}$

Table 5.3: Context constraints for the channels presented in Table 5.2.

constraints defined in terms of variables x, $\widehat{x}$, x_{snk} and x_{src}. The intention is that x_{snk} or x_{src} is true when the port x gives a reason and false if x requires a reason. The values of these variables are unimportant when there is flow on x.

Next, we extend the flow axiom to reflect the matching of reasons:

$$(\neg x \leftrightarrow \widehat{x} = \texttt{NO-FLOW}) \wedge (\neg x \to x_{\mathsf{snk}} \vee x_{\mathsf{src}}) \qquad \text{(updated flow axiom)}$$

Recall that channels are composed using the same name x for a source and a sink port. The constraint $x_{\mathsf{snk}} \vee x_{\mathsf{src}}$ can be interpreted as follows: The reason for no dataflow can either come from the sink port (x_{snk} is true), come from the source port (x_{src} is true), or from both ports at the same time, but the reason can never come from nowhere (both x_{snk} and x_{src} are false). The constraint thus encodes the three valid matching no-flow possibilities given above.

The context constraints for the primitives shown in §5.5.1 are presented in Table 5.3. The other primitives in Table 5.1 must also be extended to reflect the valid combinations of context variables, by encoding their colouring tables (see Clarke *et al*. [37] for examples).

5.5.3. EXAMPLE. Recall Example 5.5.2, where we compose a context-dependent LossySync with FIFOEmpty$_1$ channel. The constraint of this connector is given by Ψ, defined below. We write Ψ_{SC}, Ψ_{DFC} and Ψ_{CC} to denote the synchronous constraints, the dataflow constraints, and the context constraints of our example, respectively.

$$\begin{aligned}
\Psi_{SC} &= b \to a \wedge \neg c \\
\Psi_{DFC} &= b \to (\widehat{a} = \widehat{b}) \wedge \top \\
\Psi_{CC} &= \neg a \to (\neg b \wedge \neg a_{\mathsf{src}} \wedge b_{\mathsf{snk}}) \wedge \\
&\qquad \neg b \to ((a \wedge \neg b_{\mathsf{snk}}) \vee \neg a) \wedge (\neg b \to \neg b_{\mathsf{src}}) \wedge c_{\mathsf{snk}} \\
\Psi &= \exists b, \widehat{b}, a_{\mathsf{src}}, a_{\mathsf{snk}}, b_{\mathsf{src}}, b_{\mathsf{snk}}, c_{\mathsf{src}}, c_{\mathsf{snk}}. \\
&\qquad \Psi_{SC} \wedge \Psi_{DFC} \wedge \Psi_{CC} \wedge \texttt{Flow}(\{a, b, c\}
\end{aligned} \qquad \Diamond$$

The context constraints, represented by Ψ_{CC}, can be understood as follows. The first line says that, if there is no dataflow on a, then there is no dataflow on

b and a reason goes from a to b. The second line says that no dataflow on b implies that b must receive a reason when a has flow. Besides $b, \widehat{b}$, the variables $\{a_{\mathsf{src}}, a_{\mathsf{snk}}, b_{\mathsf{src}}, b_{\mathsf{snk}}, c_{\mathsf{src}}, c_{\mathsf{snk}}\}$ are also bind by the existential quantifier as they are necessary only to exclude non-solutions, when it is not possible to provide reasons for the absence of flow. For example, the cases when the context-dependent LossySync loses data without a reason, or when the FIFO1Empty$_1$ fails to receive a value without receiving a reason. The actual values chosen for such variables do not matter.

A SAT solver can solve the constraint Ψ_{SC} yielding the solutions:

$$\sigma_1 = a \wedge b \wedge \neg c \qquad \sigma_2 = a \wedge \neg b \wedge \neg c \qquad \sigma_3 = \neg a \wedge \neg b \wedge \neg c$$

Using these solutions, we can simplify Ψ using standard techniques, as in §5.3.3, to derive the corresponding dataflow constraints. The novelty in this setting is the context constraints. Using the solutions σ_1, σ_2 and σ_3 to simplify Ψ, we obtain the following constraints on the context variables (which are hidden by the existential quantifier):

$$\begin{array}{lcl} \Psi \wedge \sigma_1 & \rightsquigarrow & c_{\mathsf{snk}} \\ \Psi \wedge \sigma_2 & \rightsquigarrow & \bot \\ \Psi \wedge \sigma_3 & \rightsquigarrow & a_{\mathsf{snk}} \wedge \neg a_{\mathsf{src}} \wedge b_{\mathsf{snk}} \wedge \neg b_{\mathsf{src}} \wedge c_{\mathsf{snk}} \end{array}$$

Only the first and the last constraints are satisfiable. We can conclude that:

- σ_1 is a solution that gives a reason on the port c, without imposing any restrictions on the value c_{src}, as $c_{\mathsf{snk}} \rightarrow c_{\mathsf{snk}} \vee c_{\mathsf{src}}$;
- σ_2 is not a valid solution, i.e., it is not possible for the context-dependent LossySync to lose data since the FIFOEmpty$_1$ does not provide a reason for losing data; and
- σ_3 is a valid solution where no dataflow occurs in the channels, provided that there is a reason given to a, i.e., that a_{snk} is true. As with σ_1, no restrictions are imposed on the value of c_{src}.

5.5.3 Correctness of context constraints

Our approach to context dependency is equivalent to the 3-colouring semantics of Clarke *et al*. [37]. However, there are two main advantages of encoding the 3-colouring semantics as constraints. First, we can use well-known techniques for constraint satisfaction to find solutions for the behaviour efficiently. The benchmark results presented later in this chapter support this claim. Second, it permits the description of other constraints, such as data constraints, which may not be added to the CC framework directly.

σ	$f(\sigma)$	σ	$f(\sigma)$
x	$x^{\downarrow} \mapsto$ —•	x	$x^{\uparrow} \mapsto$ —•
$\neg x \wedge x_{\mathsf{src}}$	$x^{\downarrow} \mapsto$ -▷-•	$\neg x \wedge x_{\mathsf{snk}}$	$x^{\uparrow} \mapsto$ -▷-•
$\neg x \wedge \neg x_{\mathsf{src}}$	$x^{\downarrow} \mapsto$ -◁-•	$\neg x \wedge \neg x_{\mathsf{snk}}$	$x^{\uparrow} \mapsto$ -◁-•

Table 5.4: Definition of f.

In this subsection we formalise the equivalence between these two approaches. We start by introducing some auxiliary definitions, and present our main claim in Theorem 5.5.5. The equivalence of our approach with the 3-colouring follows by construction. The proof of this equivalence results from the following observations.

1. The constraints of each primitive p are defined so that there is a surjection f, defined in Table 5.4, from the solutions of the synchronisation and context constraints onto the entries of the colouring table of p.

2. The mapping f is compositional, namely, when composing two primitives p and q with a shared variable x, composing their colouring tables and applying f to find the possible solutions is equivalent to apply f to each colouring table and then finding the solutions for the conjunction of these constraints and the updated flow axiom.

Let σ be an assignment. For each pair $x, x_s \in dom(\sigma)$, where $s \in \{\mathsf{src}, \mathsf{snk}\}$, $f(\sigma)$ is the colouring table that maps $x^{\downarrow}$ or $x^{\uparrow}$ to the colouring represented in Table 5.4. Furthermore, by the definition of f every colouring is associated to a fixed set of possible solutions, which can be trivially written as a constraint. That is, the inverse function f^{-1} will always produce solutions of possible constraints. It can be easily shown that the colouring tables of the primitives in Table 5.2 are obtained by applying f to the solutions of the synchronisation and context constraints presented in Table 5.3. For example, the synchronisation and context constraints for the FIFOEmpty$_1$ are $\neg b \wedge (\neg a \rightarrow \neg a_{\mathsf{src}}) \wedge b_{\mathsf{snk}}$, and its possible solutions are:

$$\begin{aligned}\sigma_1 &= a \wedge a_{\mathsf{src}} \wedge \neg b \wedge b_{\mathsf{snk}};\\ \sigma_2 &= a \wedge \neg a_{\mathsf{src}} \wedge \neg b \wedge b_{\mathsf{snk}};\ \text{and}\\ \sigma_3 &= \neg a \wedge \neg a_{\mathsf{src}} \wedge \neg b \wedge b_{\mathsf{snk}}.\end{aligned}$$

It follows from the definition of f that $f(\sigma_1) = f(\sigma_2) = a$ •—▷→ b, and $f(\sigma_3) =$ a ←▷– –▷→ b, which is what we expect.

Observe that for every $\mathcal{R}$eo connector the set of variables used in its synchronisation and context constraints must obey certain properties, captured by the notion of *variable set* defined below.

5.5.4. DEFINITION. A *variable set* is a finite set $V \subseteq \mathbb{P} \cup \mathbb{P}_{\mathsf{src}} \cup \mathbb{P}_{\mathsf{snk}}$ such that

$$x \in V \cap \mathbb{P} \quad \textit{iff} \quad x_{\mathsf{src}} \in V \cap \mathbb{P}_{\mathsf{src}} \ \vee \ x_{\mathsf{snk}} \in V \cap \mathbb{P}_{\mathsf{snk}}. \qquad \lhd$$

We write $\mathbb{V}$ to denote the set of all variable sets. We say two sets of variables V_1 and V_2 are *compatible*, written as $V_1 \frown V_2$, if V_1 and V_2 are variable sets, and for every $x \in V_1 \cap V_2 \cap \mathbb{P}$, x is a sink port in V_1 and a source port in V_2 or vice-versa, that is,

$$V_1 \frown V_2 \quad \textit{iff} \quad V_1, V_2 \in \mathbb{V} \ \wedge \ V_1 \cap V_2 \subseteq \mathbb{P} \tag{5.2}$$

Note that if $V_1, V_2 \in \mathbb{V}$ then $V_1 \cup V_2 \in \mathbb{V}$, that is, $\mathbb{V}$ is closed under union. Intuitively, two connectors with constraints over the variable sets V_1 and V_2 can only be composed if the source and sink ports are connected in a one-to-one fashion. Recall that the composition of two $\mathcal{R}$eo connectors, introduced in §5.3.3, requires that every shared port of the composed connectors is a source port in one of the connectors, and a sink port in the other connector.

We now assume that, for every primitive p, the colouring table is given by the surjection f defined in Table 5.4 with respect to the solutions of the synchronisation and context constraints. Let $fv(\cdot)$ be a function that returns the free variables of a constraint. A constraint Ψ is defined over a variable set V whenever $fv(\Psi) \subseteq V$. Let also $[\![\cdot]\!]_V$ be a function that yields the set of all possible solutions of Ψ over V, that is,

$$[\![\Psi]\!]_V \ = \ \{\sigma \mid \sigma \vDash \Psi, dom(\sigma) = V\}.$$

Define F to be the lifting of f to sets, i.e., $F(\Sigma) = \{f(\sigma) \mid \sigma \in \Sigma\}$. Finally, let $\boxtimes$ denote the composition of two synchronisation and context constraints Ψ_1 over the set V_1 and Ψ_2 over the set V_2, where $V_1 \frown V_2$, defined as follows:

$$\Psi_1 \boxtimes \Psi_2 = \Psi_1 \wedge \Psi_2 \wedge \bigwedge_{x \in V_1 \cap V_2} (\neg x \rightarrow x_{\mathsf{snk}} \vee x_{\mathsf{src}}),$$

where the last constraint reflects the update on the flow axiom. The correctness of the composition of our encoding as constraints with respect to the connector colouring semantics is formalised by the following theorem.

5.5.5. THEOREM. *For any pair of constraints Ψ_p over the variable set V_1 and Ψ_q over the variable set V_2 such that $V_1 \frown V_2$, it holds that:*

$$F([\![\Psi_p]\!]_{V_1}) \bowtie F([\![\Psi_q]\!])_{V_2} \quad = \quad F([\![\Psi_p \boxtimes \Psi_p]\!]_{V_1 \cup V_2}).$$

Before proving this theorem we prove four auxiliary lemmas. The first lemma relates shared variables of $\mathcal{R}$eo connectors and the domain of the colourings derived from specific solutions of the same connectors. In the following we use the symbol $\circ$ to range over $\{\uparrow, \downarrow\}$, and define $\overline{\uparrow} = \downarrow$ and $\overline{\downarrow} = \uparrow$.

5.5.6. LEMMA. *Let σ_1 and σ_2 be assignments for variable sets V_1 and V_2, and $V_1 \frown V_2$. Then the following holds.*

$$x^\circ \in dom(f(\sigma_1)) \wedge x^{\overline{\circ}} \in dom(f(\sigma_2)) \quad \textit{iff} \quad x \in V_1 \cap V_2.$$

Proof. Let x be such that $x^\circ \in dom(f(\sigma_1))$ and $x^{\overline{\circ}} \in dom(f(\sigma_2))$. By the definition of f, if $x^\circ \in dom(f(\sigma_1))$ then x must occur also in $dom(\sigma_1)$, and similarly, x must also occur in $dom(\sigma_2)$. Therefore $x \in V_1 \cap V_2$. For the other implication, assume that $x \in V_1 \cap V_2$. Also $x \in \mathbb{P}$ because $V_1 \frown V_2$. By the definitions of f and because $V_1, V_2 \in \mathbb{V}$, we conclude that $x^\uparrow \in dom(f(\sigma_1))$ and $x^\downarrow \in dom(f(\sigma_2))$ or $x^\downarrow \in dom(f(\sigma_1))$ and $x^\uparrow \in dom(f(\sigma_2))$. □

We say two assignments σ_1 and σ_2 are compatible, written as $\sigma_1 \frown \sigma_2$, *iff* $\forall x \in dom(\sigma_1) \cap dom(\sigma_2) \cdot \sigma_1(x) = \sigma_2(x)$. The second lemma provides a sufficient condition for any two assignments σ_1 and σ_2 be *compatible*.

5.5.7. LEMMA. *Let σ_1 and σ_2 be assignments for variable sets V_1 and V_2, where $V_1 \frown V_2$, and $\forall x \in V_1 \cap V_2 \cdot f(\sigma_1)(x^\circ) \frown f(\sigma_2)(x^{\overline{\circ}})$. Then $\sigma_1 \frown \sigma_2$. Moreover, we have that $f(\sigma_1) \cup f(\sigma_2) = f(\sigma_1 \cup \sigma_2)$.*

Proof. We prove $\sigma_1 \frown \sigma_2$ by contraposition. Assuming $\neg(\sigma_1 \frown \sigma_2)$, there exists $x \in V_1 \cap V_2$ such that $\sigma_1(x) \neq \sigma_2(x)$. Note that from Equation (5.2) we conclude that $x \in \mathbb{P}$. Without loss of generality assume $\sigma_1(x) = \top$ and $\sigma_2(x) = \bot$. Then $f(\sigma_1)(x^\circ) = \longrightarrow\!\bullet$ and $f(\sigma_2)(x^{\overline{\circ}}) \neq \longrightarrow\!\bullet$, for some $\circ \in \{\uparrow, \downarrow\}$. Thus $\neg(f(\sigma_1) \frown f(\sigma_2))$. Similarly for $\sigma_1(x) = \bot$, $\sigma_2(x) = \top$. We conclude that $\sigma_1 \frown \sigma_2$. □

The next lemma relates the new constraint from the updated flow axiom to the matching of colourings.

5.5.8. LEMMA. *For the constraint $\neg x \to (x_{\mathsf{snk}} \vee x_{\mathsf{src}})$, where $\{x, x_{\mathsf{src}}, x_{\mathsf{snk}}\} \subseteq V$, it holds that:*

$$c \in F(\llbracket \neg x \to (x_{\mathsf{src}} \vee x_{\mathsf{snk}}) \rrbracket_V) \quad \textit{iff} \quad c(x^\downarrow) \frown c(x^\uparrow). \tag{5.3}$$

Proof. The proof follows by unfolding the definitions of F, and by applying Lemma 5.5.7.

$$\begin{aligned}
& c \in F(\llbracket \neg x \to (x_{\mathsf{src}} \vee x_{\mathsf{snk}}) \rrbracket_V) \\
\equiv\ & \langle \text{By the definition of } F \rangle \\
& c = f(\sigma),\ \sigma \vDash \neg x \to (x_{\mathsf{src}} \vee x_{\mathsf{snk}}),\ \text{and } dom(\sigma) = V \\
\equiv\ & \langle \text{Partition } \sigma \text{ into } \sigma' \text{ and } \sigma'' \text{ such that } dom(\sigma') = \{x, x_{\mathsf{src}}, x_{\mathsf{snk}}\} \rangle \\
& c = f(\sigma' \cup \sigma''),\ \sigma' \vDash \neg x \to (x_{\mathsf{src}} \vee x_{\mathsf{snk}}),\ \text{and } dom(\sigma'') = V \setminus \{x, x_{\mathsf{src}}, x_{\mathsf{snk}}\}
\end{aligned}$$

Observe now that $\sigma' \frown \sigma''$, hence we know that $f(\sigma' \cup \sigma'') = f(\sigma') \cup f(\sigma'')$. Furthermore, the possible solutions for σ' are the following:

$$\begin{array}{llll} x \wedge x_{\mathsf{src}} \wedge x_{\mathsf{snk}} & x \wedge \neg x_{\mathsf{src}} \wedge x_{\mathsf{snk}} & \neg x \wedge x_{\mathsf{src}} \wedge x_{\mathsf{snk}} & \neg x \wedge \neg x_{\mathsf{src}} \wedge x_{\mathsf{snk}} \\ x \wedge x_{\mathsf{src}} \wedge \neg x_{\mathsf{snk}} & x \wedge \neg x_{\mathsf{src}} \wedge \neg x_{\mathsf{snk}} & \neg x \wedge x_{\mathsf{src}} \wedge \neg x_{\mathsf{snk}} & \end{array} .$$

Let Σ be this set. The last step of the proof above is equivalent to

$$c = f(\sigma') \cup f(\sigma''),\ \sigma' \in \Sigma, \text{ and } dom(\sigma'') = V \setminus \{x, x_{\mathsf{src}}, x_{\mathsf{snk}}\}\,.$$

Observe that $f(\sigma')$, for $\sigma' \in \Sigma$, are exactly the set of all possible colourings c' such that $c'(x^{\downarrow}) \frown c'(x^{\uparrow})$, where $dom(c') = \{x^{\downarrow}, x^{\uparrow}\}$. It is now sufficient to observe that $f(\sigma'')$ yields any possible colouring, assigning colours to all the remaining ports apart from x, not mentioned on the right-hand-side of the equivalence in Equation (5.3). □

Our final lemma leading to the proof of Theorem 5.5.5 relates the matching of the colouring yield by two different assignments and a new assignment that satisfies the constraint $\neg x \to x_{\mathsf{src}} \vee x_{\mathsf{snk}}$.

5.5.9. LEMMA. *For σ_1 and σ_2 such that $dom(\sigma_i) = V_i$, for $i \in \{1,2\}$, $\sigma_1 \frown \sigma_2$, and $V_1 \frown V_2$:*

$$\begin{aligned} &f(\sigma_1)(x^{\circ}) \frown f(\sigma_2)(x^{\overline{\circ}}) \text{ iff} \\ &\qquad (\sigma_1 \cup \sigma_2) \vDash \neg x \to x_{\mathsf{src}} \vee x_{\mathsf{snk}} \text{ and } \{x, x_{\mathsf{src}}, x_{\mathsf{snk}}\} \subseteq dom(\sigma_1 \cup \sigma_2). \end{aligned}$$

Proof. For simplicity let $c_1 = f(\sigma_1)$ and $c_2 = f(\sigma_2)$. Let also $\{x, x_{\mathsf{src}}, x_{\mathsf{snk}}\} \subseteq V$. Note that c_1 and c_2 will have disjoint domains, because $V_1 \frown V_2$. Observe that:

$$\begin{aligned} & c_1(x^{\circ}) \frown c_2(x^{\overline{\circ}}) \\ = \ & \langle \text{Because } dom(c_1) \text{ and } dom(c_2) \text{ are disjoint} \rangle \\ & (c_1 \cup c_2)(x^{\circ}) \frown (c_1 \cup c_2)(x^{\overline{\circ}}) \\ = \ & \langle \text{By Lemma 5.5.8} \rangle \\ & c_1 \cup c_2 \in F([\![\neg x \to x_{\mathsf{src}} \vee x_{\mathsf{snk}}]\!]_V) \\ = \ & \langle \text{By the definition of } F \text{ and } [\![\cdot]\!]_V \rangle \\ & c_1 \cup c_2 \in \{f(\sigma) \mid \sigma \vDash \neg x \to x_{\mathsf{src}} \vee x_{\mathsf{snk}}, dom(\sigma) = V\} \\ = \ & \langle \text{Because } \sigma_1 \frown \sigma_2 \text{ and } c_i = f(\sigma_i) \rangle \\ & f(\sigma_1 \cup \sigma_2) \in \{f(\sigma) \mid \sigma \vDash \neg x \to x_{\mathsf{src}} \vee x_{\mathsf{snk}}, dom(\sigma) = V\} \\ = \ & \langle \text{By set inclusion and because } V \subseteq \{x, x_{\mathsf{src}}, x_{\mathsf{snk}}\} \rangle \\ & (\sigma_1 \cup \sigma_2) \vDash \neg x \to x_{\mathsf{src}} \vee x_{\mathsf{snk}} \text{ and } \{x, x_{\mathsf{src}}, x_{\mathsf{snk}}\} \subseteq dom(\sigma_1 \cup \sigma_2) \end{aligned}$$

□

We are now in a position to prove Theorem 5.5.5.

Proof. (Theorem 5.5.5)

$$
\begin{aligned}
& F(\llbracket \Psi_p \rrbracket)_{V_1} \bowtie F(\llbracket \Psi_q \rrbracket)_{V_2} \\
= \; & \langle \text{By the definition of } F \rangle \\
& \{ f(\sigma_1) \mid \sigma_1 \in \llbracket \Psi_p \rrbracket_{V_1} \} \bowtie \{ f(\sigma_2) \mid \sigma_2 \in \llbracket \Psi_q \rrbracket_{V_2} \} \\
= \; & \langle \text{By the definition of } \llbracket \cdot \rrbracket_V \rangle \\
& \{ f(\sigma_1) \mid \sigma_1 \vDash \Psi_p, dom(\sigma_1) = V_1 \} \bowtie \{ f(\sigma_2) \mid \sigma_2 \vDash \Psi_q, dom(\sigma_2) = V_2 \} \\
= \; & \langle \text{By the definition of } \bowtie \rangle \\
& \{ f(\sigma_1) \cup f(\sigma_2) \mid \sigma_1 \vDash \Psi_p, \sigma_2 \vDash \Psi_q, dom(\sigma_1) = V_1, dom(\sigma_2) = V_2, \\
& \quad x^\circ \in dom(f(\sigma_1)) \wedge x^{\overline{\circ}} \in dom(f(\sigma_2)) \Rightarrow f(\sigma_1)(x^\circ) \frown f(\sigma_2)(x^{\overline{\circ}}) \} \\
= \; & \langle \text{By Lemma 5.5.6} \rangle \\
& \{ f(\sigma_1) \cup f(\sigma_2) \mid \sigma_1 \vDash \Psi_p, \sigma_2 \vDash \Psi_q, dom(\sigma_1) = V_1, dom(\sigma_2) = V_2, \\
& \quad x \in V_1 \cap V_2 \Rightarrow f(\sigma_1)(x^\circ) \frown f(\sigma_2)(x^{\overline{\circ}}) \} \\
= \; & \langle \text{By Lemma 5.5.7, from where we also conclude that } \sigma_1 \frown \sigma_2 \rangle \\
& \{ f(\sigma_1 \cup \sigma_2) \mid \sigma_1 \vDash \Psi_p, \sigma_2 \vDash \Psi_q, dom(\sigma_1 \cup \sigma_2) = V_1 \cup V_2, \\
& \quad x \in V_1 \cap V_2 \Rightarrow f(\sigma_1)(x^\circ) \frown f(\sigma_2)(x^{\overline{\circ}}) \} \\
= \; & \langle \text{By Lemma 5.5.9, and because } \sigma_1 \frown \sigma_2 \text{ and } dom(\sigma_1 \cup \sigma_2) = V_1 \cup V_2 \rangle \\
& \{ f(\sigma_1 \cup \sigma_2) \mid \sigma_1 \vDash \Psi_p, \sigma_2 \vDash \Psi_q, dom(\sigma_1 \cup \sigma_2) = V_1 \cup V_2, \\
& \quad x \in V_1 \cap V_2 \Rightarrow (\sigma_1 \cup \sigma_2) \vDash \neg x \to x_{\mathsf{src}} \vee x_{\mathsf{snk}} \} \\
= \; & \langle \text{Because } \sigma_1 \frown \sigma_2 \rangle \\
& \{ f(\sigma_1 \cup \sigma_2) \mid (\sigma_1 \cup \sigma_2) \vDash \Psi_p, (\sigma_1 \cup \sigma_2) \vDash \Psi_q, dom(\sigma_1 \cup \sigma_2) = V_1 \cup V_2, \\
& \quad x \in V_1 \cap V_2 \Rightarrow (\sigma_1 \cup \sigma_2) \vDash \neg x \to x_{\mathsf{src}} \vee x_{\mathsf{snk}} \} \\
= \; & \langle \text{Using } \sigma = \sigma_1 \cup \sigma_2 \text{ and replacing implication by an universal quantifier} \rangle \\
& \{ f(\sigma) \mid \sigma \vDash \Psi_p, \sigma \vDash \Psi_q, dom(\sigma) = V_1 \cup V_2, \forall_{x \in V_1 \cap V_2} \cdot \sigma \vDash \neg x \to x_{\mathsf{src}} \vee x_{\mathsf{snk}} \} \\
= \; & \langle \text{Because } \sigma \vDash \Psi_1 \text{ and } \sigma \vDash \Psi_2 \textit{ iff } \sigma \vDash \Psi_1 \wedge \Psi_2 \rangle \\
& \{ f(\sigma) \mid \sigma \vDash \Psi_p \wedge \Psi_q \wedge \textstyle\bigwedge_{x \in V_1 \cap V_2} \neg x \to x_{\mathsf{src}} \vee x_{\mathsf{snk}}, dom(\sigma) = V_1 \cup V_2 \} \\
= \; & \langle \text{By the definition of } \boxtimes \rangle \\
& \{ f(\sigma) \mid \sigma \vDash \Psi_p \boxtimes \Psi_q, dom(\sigma) = V_1 \cup V_2 \} \\
= \; & \langle \text{By the definition of } F \text{ and } \llbracket \cdot \rrbracket_V \rangle \\
& F(\llbracket \Psi_p \boxtimes \Psi_q \rrbracket_{V_1 \cup V_2}).
\end{aligned}
$$

□

By proving Theorem 5.5.5 we show that our constraint-based approach for describing context dependency is equivalent to the 3-colouring semantics. As a result, searching for valid colourings is reduced to a SAT solving problem, that is, the set of solutions for synchronous and context constraints of a connector coincide with its valid colourings. In the next section we exploit the practical conse-

quences of this equivalence, and compare the execution times of an engine based on the connector colouring semantics with our constraint-based engine.

5.6 Benchmarks

We compare two prototype engines based on constraint satisfaction with an optimised engine for $\mathcal{R}$eo based on the connector colouring semantics [37]. For this, the data constraints are ignored, and solving the constraints only yields *where* data can flow, but not *which* data flow. We evaluate the constraints-based approach using implementations based on both context independent (CI) and context dependent (CD) semantics (§5.5). In the connector colouring semantics, the CI semantics corresponds to using two colours while the CD semantics corresponds to using three colours. We have implemented the context dependent semantics for only one of the constraint engines, as the results we present already provide solid evidence that the constraint-based approach is significantly better than using connector colouring. A few more words regarding the three engines:

CC engine We use an optimised engine based on connector colouring [37] as a reference for our benchmark results. The engine has been incorporated into the Eclipse Coordination Tools.[4] This engine supports both context dependent and independent semantics, as explained in §5.5, and it is, to the best of our knowledge, the fastest existing implementation that computes the behaviour of $\mathcal{R}$eo connectors on-the-fly.

SAT engine This is a constraint engine using the SAT4J Java libraries,[5] which are free and included in the Eclipse standard libraries. The main concern of the project responsible for the SAT4J libraries is the efficiency of the SAT solver. We chose SAT4J because it is a well known library for SAT solving, with the portability advantages provided by the Java platform. We avoid solutions that have no dataflow by adding conjunctively the constraint $\bigvee_{x \in \mathbb{P}} x$ to the constraints of the connector for the CI semantics. We do not add this disjunction in the CD semantics because the context dependency already guarantees that the data flow unless there is an explicit reason that forbids the flow of data.

CHOCO engine We developed a second prototype constraint engine using the CHOCO open-source constraint solver[6] which offers, among other things, state-of-the-art algorithms and techniques,and user-defined constraints, domains and variables. In the future, we expect to achieve finer control over

[4] http://reo.project.cwi.nl/cgi-bin/trac.cgi/reo/wiki/Tools
[5] http://www.sat4j.org/
[6] http://choco.emn.fr/

the strategies for solving the constraints, and to add support for non-boolean variables. The CHOCO-based engine implements only the context independent (CI) semantics. We avoid solutions with no flow by using a strategy that gives precedence to solutions where the synchronisation variables are set to *true*. Using this strategy, the solution with no flow at all in the connector can still be found, but only when it is the only valid solution.

5.6.1 Test cases

We present four test cases constructed out of stateless channels. For each case we replicate part of the connector a number of times and measure the time taken to find a solution for the coordination problem, which includes the time to required build the data structures corresponding to the connector and the encoding of its behaviour as constraints. We also add *active environments* to the connector, i.e., we attach data writers and data readers to every port where we expect data to be written or read, respectively. Defining this environment is important for the CC engine because it reduces the number of possible solutions, and because one of the optimisations of the CC engine is to start computing the colouring table starting from the sources of data. Incorporating an active environment significantly reduces the time taken by the CC engine, allowing for a fairer comparison.

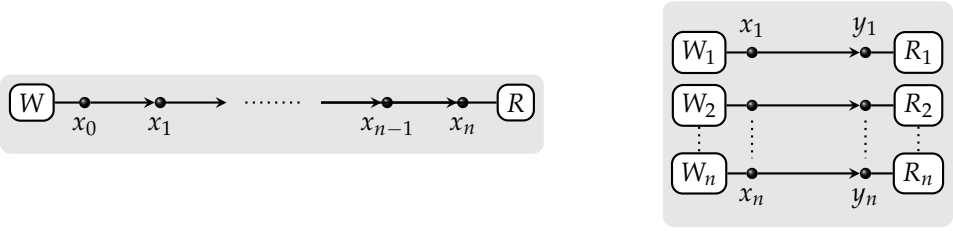


Figure 5.4: Left to right: n synchronous channels composed in sequence (SEQ) and in parallel (PAR).

The first two test cases consist of n synchronous channels in sequence (SEQ) and in parallel (PAR), respectively, as depicted in the left and right of Figure 5.4. Note that a sequence of n synchronous channels is semantically equivalent to a single synchronous channel, but the search for a solution becomes more complex, especially when the topology of the connector is not exploited.

The next test case is a generalisation of the exclusive router (ExR), introduced in §5.5, for n outputs. The generalised exclusive router is depicted on the left side of Figure 5.5. It passes data from the data writer to exactly one of the data readers. Finally, we use a variation of the exclusive router, the *inclusive router* (InR), depicted in the right side of Figure 5.5. The inclusive router replicates data provided by the writer to at least one of the data readers. When compared with

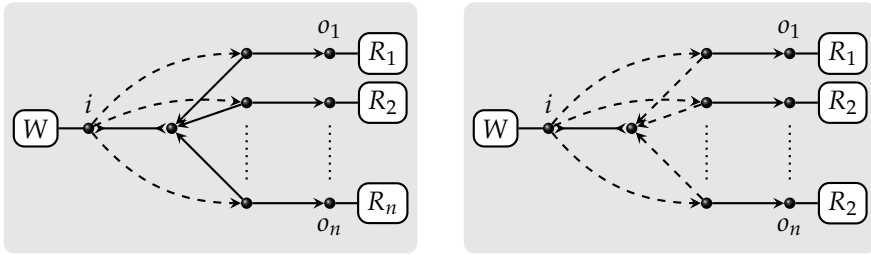


Figure 5.5: Left to right: exclusive (ExR) and inclusive (InR) router connectors generalised for n outputs.

the exclusive router, it uses LossySync channels to connect the data readers to the SyncDrain channel instead of Sync channels. Consequently, more than one reader can receive the data produced by the data writer, although only one of the LossySync channels will not lose data. The number of possible solutions increases exponential when using the CI semantics, which favours the constraint engine (since only one solution is computed). When using the CD semantics with the active environment there is only one solution for the constraints, which consists of the data being replicated to all available readers.

5.6.2 Results

All the benchmarks were executed on a Macbook laptop with a 2 GHz Intel Core 2 Duo processor and 4 GB of RAM, running Mac OS 10.6. For each coordinate in the graphs, we performed 10 different executions and computed the average value. We tested the CC-, SAT-, and CHOCO-based engines using the CI semantics, and the CC- and SAT-based engines using the CD semantics. The benchmark results for the CC engine are presented in Figure 5.6, and the benchmark results for the constraint engines are presented in Figure 5.7, using one graph for each test case. In the graphs we write Seq for the sequence of Sync channels, Par for the set of Sync channels in parallel, ExR for the exclusive router, and InR for the inclusive router. For each engine, semantics (CI and CD), and connector, we selected a range of possible sizes for the connector. For each size, we measured the time for finding a solution 10 times, and represented the average value in the graph using a marking, as explained in the legend of the graphs. Furthermore, a solid line represents the evaluation of the executions using the context independent semantics (CI), and a dashed line represents the evaluation of executions using the context dependent semantics (CD).

The results show that implementations based on constraint-solving techniques scale better and are more efficient than the implementation based on connector

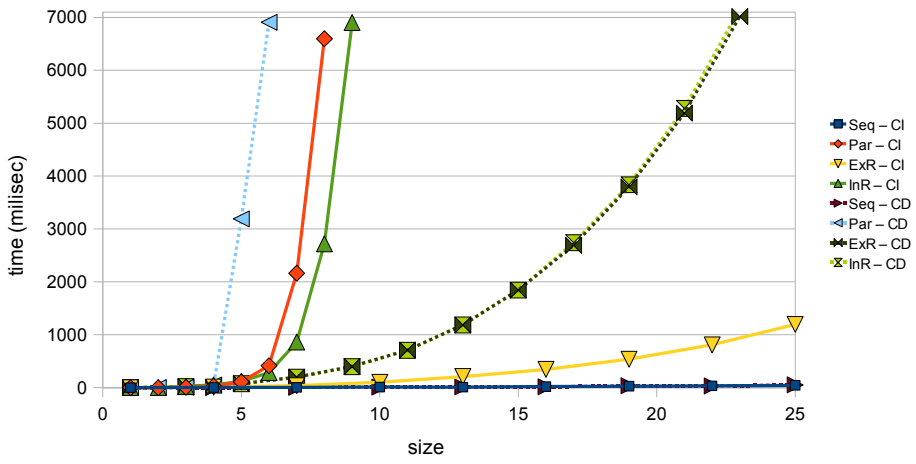


Figure 5.6: Results from the execution of the CC engine for the context dependent semantics (CD) and independent semantics (CI).

colouring.

Firstly, the maximum connector size for the constraint solving approach is much larger than for the CC-based implementation. We measured the maximum size of connectors reached when running out of memory or taking more than 1 minute. While the maximum size of the connectors tested for the CC engine range between 6 (Par-CD) and 100 (Par-CI), for the CHOCO engine these values ranged between 2,000 (InR-CI) and 17,000 (Par-CI), and for the SAT engine these values range between 800 (InR-CD) and 80,000 (Par-CI).

Secondly, the constraint solver-based engines are significantly faster at finding solutions, as can easily be seen from the graphs. We present a comparison based on the *workload*—size of connector—an engine can handle within one second. For the Seq benchmark, the CC-based implementations could only handle connectors of size 89 (CC-CD) to 94 (CC-CI) in 1 second, in contrast to connectors of size 15,500 (CHOCO-CI), 28,000 (SAT-CI) and 27,500 (SAT-CD) in the constraint-based implementations. This means that the increase in workload ranges from 164 to 308. The case for the Par benchmark is more impressive: 5 (CC-CD), 7 (CC-CI) vs. 41300 (CHOCO-CI), 55000 (SAT-CI), and 1000 (SAT-CD). Thus, the increase in workload ranges from 200 to 7857. Notice that as the context dependent semantics require more variables to encode, they take significantly longer to solve in the constraint-based approach compared to context independent semantics. The ExR and InR benchmarks exhibit similar increases in workload, being able to deal in one second with connectors that are between 46 and 1000 times larger.

The main difference between execution of the CC engine and of the constraint

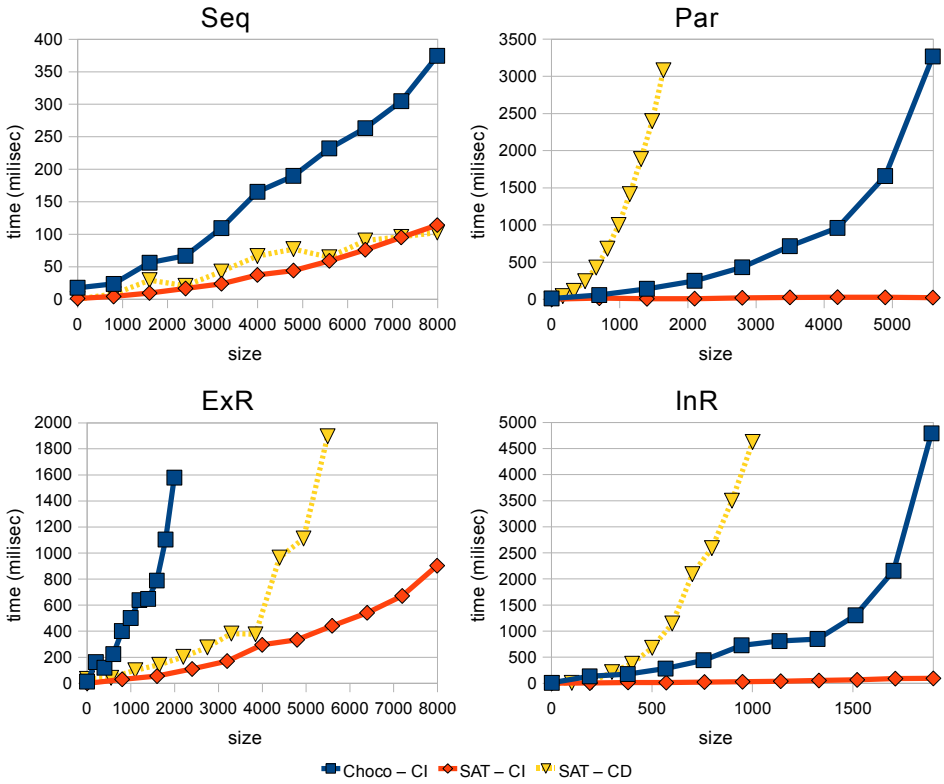


Figure 5.7: Results from executing the CHOCO and SAT engines.

engines is that the former takes into account the topology of the connector, and calculates all possible solutions whenever a new primitive is added to the set of constraints. The constraint engines disregard the topology of the connector, and return only one possible solution. As a consequence, the CC engine favours connectors with a smaller number of solutions, while the constraint engine favours connectors where more solutions can be found. The inclusive router test case in Figure 5.6 illustrates this point very clearly. For the CC engine the use of a context dependent semantics reduces the number of possible solutions, and for that reason the engine is much more efficient than using a context independent semantics. For the SAT engine the number of solutions has the opposite effect, since it is much faster to find a solution under the context independent semantics. This test case also shows that, even though the context dependency requires more complex reasoning, the number of solutions is more relevant for the efficiency of the engine.

Consider now Figure 5.7. The difference between results obtained when using

the CHOCO-based engine and the SAT-based engine are smaller in the connector with a sequence of Sync channels. This is due to the way we avoid no-flow solutions in both cases. By avoiding no-flow solutions, there is only one possible solution to the constraints consisting of data flowing from the writer to the reader. In the CHOCO engine we give precedence to the flow on the ports, so the engine only has to verify that it is in fact a solution. The SAT engine starts by trying the no-flow assignment for each port, with the additional constraint that at least one port has to have flow, which is not optimal for this scenario. And the context has little influence in this case because the variables that deal with the context are not relevant (and thus unconstrained) when the synchronisation variables are set to true.

When performing the benchmarking we also noticed that in certain cases the solution is found unusually rapid. This happened quite frequently, for example, when executing the inclusive router using the context dependent semantics. The existence of these *lucky* runs reflects that the heuristics used by the constraint solver are not as predictable as the compositional method used by the CC engine. The graphs do not exhibit this phenomenon because we only represent the average of the executions.

5.7 Guiding the constraint solver

Constraint satisfaction is performed by splitting the domain of a variable into two or more parts and by propagating constraints [6]. It is well known that the performance of the solver depends on the order in which variables are resolved. We choose to exploit this 'flexibility' for a variety of goals, namely, fairness, priority, avoiding no-flow solutions, and efficiency. This section suggests how the constraint solver can be guided with respect to these goals, but it is not yet supported by tools.

The following criteria, often used in conjunction, can help to guide the constraint solving process:

Variable ordering — Choose which variables to resolve first. For example, try evaluating variables corresponding to ports with data, such as from external components or from a full FIFO_1 buffer, or select variable corresponding to high priority choices. Variables can be ordered locally, within a primitive to achieve a local notion of priority, or across parts of the connector or even globally across the entire connector to achieve a more global notion of priority by making the solver consider particular connectors before others.

Solution ordering — Choose the order in which that values are examined for each variable. For example, try solving constraints with a synchronisation variable set to $\top$ before trying with $\bot$. Solution ordering can be applied

to other data domains, though it is not immediately clear what the consequences of this would be.

We now describe how this can be used. In general, the ordering imposed on variables is partial, leaving room for the constraint solver to makes its own choices.

Fairness — To implement nondeterministic choice fairly, the constraint solver needs to avoid resolving constraints in the same order each time it runs. Otherwise, it is possible that the same solution is always chosen. This can be achieved by randomising the variable ordering each time the constraint solver is invoked and/or changing the order in which the values of split variables are explored. In the presence of other constraints on the variable/-value ordering, randomisation can occur modulo the imposed ordering.

Priority — Priority can be achieved by appropriately ordering the variables and/-or the solutions to achieve the desired effect. The more global or comprehensive the ordering, that is, the more variables the ordering talks about, the more global the notion of priority. Purely local notions concerning as few as one variable are also sensible. For example, preferring flow over no-flow on the output port of a LossySync achieves a local preference for dataflowing through the LossySync.

Avoiding no-flow solutions — To give priority of flow over no-flow, the solver is forced to try $x = \top$ before $x = \bot$ when resolving synchronisation variables $x \in \mathbb{P}$. This needs to be done for all variables corresponding to sources of data.

Efficiency — In general, the most efficient way to solve constraints is by starting with sources of data (such as inputs from components or full FIFO_1 buffers), and moving in the direction of dataflow. Thus, the topology of the connector can also be used to help determine the variable ordering.

5.7.1. EXAMPLE. Consider again the Priority Merger channel from Table 5.3. It performs a merge of ports a and b into port c, giving priority to port a whenever both a and b are possible. The ordering constraints to achieve this consist of visiting variable a before visiting b, and then considering $a = \top$ before $a = \bot$. ◊

The exact degree to which the underlying constraint solver can be manipulated depends upon the implementation of the constraint solver. For instance, CHOCO[7] provides some control, such as setting the order of values for a set of variable, though not to the extent described here. In other settings, the client of the constraint solver may have little influence on its internal algorithms. We also

[7]CHOCO constraint programming system, available at `http://choco.sourceforge.net/`

need to ensure that the various orderings are preserved by optimisations and by the composition of constraints. For example, if a variable is eliminated, what happens to the orderings related to the eliminated variable? More research is required to better understand this issue.

5.8 Implementing interaction

Interaction between components and the engine in our model differs from previous descriptions of $\mathcal{R}$eo, as depicted in Figure 5.8. The usual interaction model for $\mathcal{R}$eo components has two steps: firstly, a component attempts to write or take a data value; secondly, in the current or in some subsequent round the engine replies, with a possible data value. This is how $\mathcal{R}$eo is implemented in Reolite [37] and in the current ECT toolkit [16].

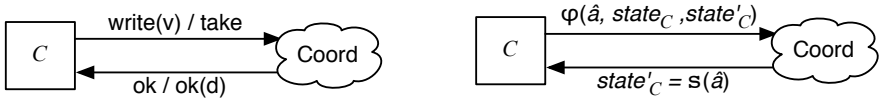


Figure 5.8: (Left) $\mathcal{R}$eo-style interaction. (Right) Interaction in our approach.

In our model, components play a more participatory role, wherein they publish a 'meta-level' description of their possible behaviour in the current round in the form of a constraint. The engine replies with a term that the component interprets as designating its new state and, if required, the dataflow that occurred. This is (a part of) the solution of the constraints, typically just the value assigned to the $state'_C$ variable.

The new (interactive) approach can easily be wrapped to look like the previous (non-interactive) approach as follows, so that components written for an older version of the $\mathcal{R}$eo engine can be used with this version:
write: Component C issues a write of data d to port a:

1. Pass the constraint $a \rightarrow (\widehat{a} = d \wedge state'_C = \texttt{ok}) \ \wedge \ \neg a \rightarrow state'_C = \texttt{no}$ to the constraint solver.

2. If the constraint solver returns $state'_C = \texttt{ok}$, return control to the component.

3. Otherwise, try again with the same constraint in the next solver round.

take: Component C issues take on port a satisfying constraint $R(x)$:

1. Pass the constraint $a \rightarrow (R(\widehat{a}) \wedge state'_C = \texttt{ok}(\widehat{a})) \ \wedge \ \neg a \rightarrow state'_C = \texttt{no}$ to the constraint solver.

2. If constraint solver returns $state'_C = \texttt{ok}(\widehat{a})$, return control with value $\widehat{a}$ to the component.

3. Otherwise, try again with the same constraint in the next solver round.

In general, the interaction protocol between the constraint solver and the component consists of the component/primitive issuing constraints to the solver over a certain set of variables and the solver returning the values of those variables to the component/primitive. Typically, it is sufficient to encode the information returned by the constraint solver in the value stored in the state variable.

5.9 Comparison of $\mathcal{R}$eo models

The surface syntax of $\mathcal{R}$eo is presented in terms of channels and their connecting nodes. In order to offer intuition as to how $\mathcal{R}$eo works, analogies have been drawn between the behaviour of a connector and the flow of electricity in an electrical circuits or water flow through pipes [8, 16]. Such systems have a natural equilibrium-based realisation, which does not extend to dataflow in $\mathcal{R}$eo, and this becomes apparent when implementing $\mathcal{R}$eo. Indeed, even describing $\mathcal{R}$eo purely in terms of dataflow can be misleading, as the direct approach to implementing $\mathcal{R}$eo, by plugging together channels and having them locally pass on data according to the channel's local behavioural constraints, does not work. The channel abstraction offers no help as channels are only capable of locally deciding what to do, whereas the behaviour of a connector typically cannot be locally determined: it is impossible to make choices that are local to channels in order to satisfy the constraints imposed by the entire connector.

This means, for example, that a proposed implementation based on the MoCha middleware [56], which provides all the primitive channels $\mathcal{R}$eo has and nothing else, *cannot* possibly work without additional infrastructure. (This was already identified by Guillen Scholten [56].) Specifically, some form of backtracking or non-local arbitration is required to guarantee the atomicity between the sending of data and the receiving of data, while obeying the constraints imposed by the connector. An attempt to provide a distributed model for $\mathcal{R}$eo based on the speculative passing of data combined with back-tracking has been made [46], but the result was too complex, possibly because it followed too closely the channel metaphor.

To see why, consider the connector in Figure 5.9. Port x could speculatively send data through both the FIFO buffer (x-y), satisfying its local constraints, and through the Sync channel (x-v). Node v would then send the data through the Sync channel (v-w). Node w must also send data into the SyncDrain (w-z), but no data will ever arrive at port z (in this round), so the SyncDrain cannot synchronise. Thus the constraints of all the primitives are not satisfied, based on the wrong

initial choice at x, so the entire dataflow needs to be rolled back. In a distributed setting, this is unlikely to be a feasible approach to implementing $\mathcal{R}$eo.

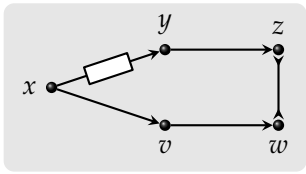


Figure 5.9: Example $\mathcal{R}$eo connector illustrating the need for global choices.

Although $\mathcal{R}$eo is described in terms of channels and their connecting nodes, existing $\mathcal{R}$eo models are instead based on four main concepts: synchronisation, dataflow, state transition, and context dependency. Every model of $\mathcal{R}$eo focusses on one or more of these concepts. Recent efforts to implement $\mathcal{R}$eo are based directly on one of its formal semantic models. This means that the limitations of a semantic model are inherited by implementations based on that model—primitives *cannot* offer arbitrary behavioural possibilities, but are restricted by the semantic model underlying the implementation. However, in general there is considerable freedom in choosing an implementation approach. We therefore present a comparative discussion of some of these models and their implications for the implementation of $\mathcal{R}$eo. We break our discussion into two parts. Firstly, we discuss existing models of $\mathcal{R}$eo. Then we compare existing implementation approaches, including some failed attempts.

5.9.1 $\mathcal{R}$eo models

Models of $\mathcal{R}$eo try to capture one or more of the following features: synchronisation, dataflow, state, context dependency, and, more recently, reconfiguration. Some models aim to be comprehensive, covering as many features as possible, whereas others focus on one or two in order to better understand specific issues.

Synchronisation corresponds to two or more actions occurring atomically; it is the only notion common to all $\mathcal{R}$eo models. We consider mutual exclusion or asynchrony, namely, expressing that two actions cannot occur together, as falling under the umbrella of synchronisation.

Data aware models describe the values of data being communicated, as well as permitting synchronisation that depends upon the value being sent. For example, synchronisation at the sink port of a filter channel, depends upon the value passed through its source port. Models not mentioning data can still be implemented to forward data, but not to transform it nor base synchronisation upon it.

Context dependency is a notion required to model behaviour that changes with the context in which the connector is placed. For example, the original intended

behaviour of a LossySync channel is that it will lose data only if the primitive connected to its sink port does not accept the data [8]. Related to context dependency are the notions of priority, which prefers one transition (in an automata model) over another whenever both are possible, and maximal flow, which prefers data to spread as far as possible into a connector (all other things being equal) [89]. Different approaches have been explored, as it is unclear from the informal descriptions of $\mathcal{R}$eo what exactly context dependency should mean, in particular, regarding its interaction with non-determinism.

Reconfiguration occurs when channels are unplugged from each other and replugged in a new configuration. This can be initiated from within a connector or as an external action. We did not mention reconfiguration before, but recent work proposes a number of approaches [35, 77, 71]. The contemporary Ph.D. thesis of Krausse [76] focuses mainly in the reconfiguration of $\mathcal{R}$eo connectors. For this discussion we restrict to the impact that facilitating reconfiguration has on determining the behaviour of a connector.

Abstract behaviour types The original semantics of $\mathcal{R}$eo is defined in terms of abstract behaviour types [9], which are co-inductively defined relations over timed data streams (a timed stream paired with a value stream). These models account for synchronisation, dataflow, and, implicitly, state changes. They mention neither context dependency nor reconfiguration.

This semantics provides little guidance for implementing $\mathcal{R}$eo, so we do not consider it further.

Automata-based models Various automata-based models for $\mathcal{R}$eo exist. We consider the following four: constraint automata (CA) [22], intensional constraint automata (ICA) [41], port automata (PA) [70], and $\mathcal{R}$eo automata (RA) [29]. These models provide the semantics of each $\mathcal{R}$eo primitive and their composition, by representing the synchronisation possible in a connector and possibly a description of the dataflow in the transitions of the automata. In all of these models synchronisation is represented by a set of ports in each transition, and the dataflow by constraints.

CA is the only automata model of $\mathcal{R}$eo that captures data, as the other models focus on different issues. PA were devised to study decomposition of automata, and hence they capture only synchronisation and state. Neither CA nor PA have context dependency, so they can only express a variant of LossySync which makes a non-deterministic choice between passing the data onwards and losing it. Context dependency was later incorporated in an automata model, firstly in ICA, by reverse engineering ideas from connector colouring, and, subsequently and more compactly, in RA.

Reconfiguration in the presence of an automaton-based model requires both

maintaining a description of the connector and completely re-computing the underlying semantics of a connector whenever reconfiguration occurs [35].

Additional variants of constraint automata proposed as $\mathcal{R}$eo models include notions of time [13], resource bounds [81], quality-of-service guarantees [15], or probability [21, 23, 88].

Connector colouring *Connector colouring* [37] is based on the simple idea that ports in a connector where data flow and where data does not flow can be coloured with different colours (see §5.5). Each primitive has a set of *colourings* describing its possible behaviours. The semantics of a connector is determined by plugging together the colourings of the primitives in such a way that the *colours match*, meaning that dataflow either occurs or does not occur at the node. Connector colouring follows the shape of the connector, and computes all possible behaviours for the given step.

Two variants of connector colouring exist. The first, called 2-colouring (CC2), has one colour representing dataflow and one representing no dataflow. The second, called 3-colouring (CC3), splits the no dataflow colour into two to capture context, as described in §5.5. CC2 captures only synchronisation, whereas CC3 captures both synchronisation and context dependency. Both connector colouring schemes abstract away from the data passed and state transition, but this information can be added in implementations, as long as the value of the data does not affect synchronisation and it is not transformed.

As connector colouring is computed on-the-fly, reconfiguration has no impact on the computation of connector semantics, as the most recent version of the connector is used to compute the colouring table.

SOS Mousavi *et al.* present a structural operational semantics (SOS) formalisation of $\mathcal{R}$eo using Maude [89]. Two versions of the semantics were developed: the original formulation (SOS) and an extension with a notion of maximal flow to capture a notion of context dependency (SOS+FLOW), which is not as flexible as the corresponding notion enforced by 3-colouring. Khosravi *et al.* further explore this idea using Alloy [67]. One advantage of the SOS approach is that it deals with causality issues in connectors (which we deliberately ignore, as most other models of $\mathcal{R}$eo do). Clarke [34] explores causality in depth, presenting $\mathcal{R}$eo semantics in terms of the proof theory of intuitionistic linear logic and zero-safe Petri nets. These approaches do not consider reconfiguration.

An alternative operational semantics is based on the Tile Model [53, 14]. Tiles can be composed in three different ways to generate proof steps: horizontally, vertically, or in parallel. Horizontal composition represents synchronisation, and can be performed only when the effect of one tile matches the trigger for another tile. Vertical composition represents changes of the system in time, requiring that the

final configuration of one tile matches the initial configuration of another tile. Finally, parallel composition represents concurrent execution of tiles. This approach extends connector colouring (both 2- and 3-colouring) to include data, state and a primitive notion of reconfiguration (though not causality). As such, the tile models (TILE2 and TILE3) are two of the most complete semantic descriptions of $\mathcal{R}$eo.

Constraint-based approach Our constraint approach deals with synchronisation, data awareness, state, and context dependency in an orthogonal and uniform way. We defined different constraints for each of these notions, which are then added conjunctively to capture any combination of them (along with certain axioms connecting the various kinds of variables). Beyond the other models, constraints can be used to express that multiple inputs are available on a particular node, whereas other models deal only with a single datum (or do not mention data at all).

Reconfiguration was not considered for the model presented in this chapter, though it would be implemented by rewriting the appropriate constraints.

Model	Data Awareness	State	Context Dependency	Reconfiguration
ABT	✓	✓	✗	None
CA	✓	✓	✗	Re-compute
ICA	✗	✓	✓	Re-compute
PA	✗	✓	✗	Re-compute
RA	✗	✓	✓	Re-compute
CC2	✗	✗	✗	Compatible
CC3	✗	✗	✓	Compatible
SOS	✓	✓	✗	Compatible
SOS+FLOW	✓	✓	✓	Compatible
TILE2	✓	✓	✗	Some
TILE3	✓	✓	✓	Some
Constraints	✓	✓	✓	Compatible

Figure 5.10: Comparison of $\mathcal{R}$eo models. Regarding the reconfiguration, *None* means that reconfiguration is unfeasible; *Re-compute* means that reconfiguration would require re-computing the entire semantics; *Compatible* means that the model is compatible with reconfiguration, because semantics are computed on-the-fly; and *Some* means that the model has some notion of reconfiguration built in.

Table 5.10 presents a comparison of the various approaches along four dimensions of interest (all models express synchronisation). Only three of the models

receive three ticks, namely SOS+FLOW, TILE3, and Constraints. SOS+FLOW was not considered for implementation as the notion of maximal flow is not as flexible as the notion of context dependency enforced by 3-colouring. As TILE3 extends the 3-colouring model to include data awareness and state, and we have shown how to encode synchronisation, data awareness, state, and 3-colouring style context dependency into constraints, we conclude that these are semantically comparable approaches.

5.9.2 $\mathcal{R}$eo engines

The coordination abstractions provided by the $\mathcal{R}$eo model impose some implementation challenges, so most approaches to implementing $\mathcal{R}$eo involve directly implementing some semantic model. Each semantic model induces different characteristics and limitations on implementations based on it. We now compare existing and possible implementations of $\mathcal{R}$eo on the implementation approach, on the numbers of solutions computed, and on whether the behaviour is pre-computed.

implementation approach The approaches we cover include a speculative approach, compilation into automata, connector colouring, a search-based approach, and constraint solving.

number of solutions computed when determining what to do in the next step, two approaches are possible:

1. (all-sol) find all possibilities for a round, and choose one of them non-deterministically (or based on some other scheme); and
2. (one-sol) find only some (typically one) of the possible solutions.

pre-computed behaviour when computing the behaviour of a connector, two approaches are possible:

1. (all-steps) pre-compute all future behaviour *a priori*; and
2. (single-step) compute the behaviour of a single step at a time.

Speculative approach This approach to implementing $\mathcal{R}$eo consists of speculatively trying to send data through channels and rolling back when an inconsistency arises. Had such an approach been successfully implemented, it would have computed one solution at a time (one-sol) for a single step (single-step). This is all speculation, however, as this approach was never successfully implemented due to the inherent impossibility of locally making the globally consistent choices, and the difficulty of managing distributed rollback and subsequent retries.

Automata-based implementations This approach to implementing $\mathcal{R}$eo compiles the behaviour of a connector into an automaton [79], and thus pre-computes all future behaviour at compile-time (all-steps, all-sol). Implementations of all-steps semantics do not scale, since finding all possible behaviour for all possible states of a concurrent system is very expensive and space inefficient. For example, the number of states generally doubles for every FIFO1 buffer in a connector, assuming all states are reachable. Certainly, infinite state spaces are excluded. Furthermore, implementations based on automata models are inherently centralised, and lose all potential parallelism, as a connector is implemented using a single automaton. On the other hand, this means that they can be efficient at run-time, though they need to be re-computed when reconfiguration occurs. They trade off run-time efficiency for flexibility.

Connector colouring-based implementations These implementations encode the behaviour of the next step of a connector as a colouring table and compose the colouring tables using a notion of matching, as described above in §5.5. Implementations based on this approach compute all solutions (all-sol) for a single step (single-step). The disadvantage of computing all steps is the overhead of computing choices that are not used.

Reolite [37] was the first prototype implementation based on connector colouring. A more recent and efficient engine is incorporated into the $\mathcal{R}$eo toolset [16]. The latter version was used as a comparison in our benchmarking. Connector colouring also forms the basis of our upcoming distributed implementation, which we discuss below in §5.9.3.

Search-based implementation These include implementations based on the SOS models and, hypothetically, on the Tile models. The Maude implementation of SOS [89] could find a single solution (one-sol), whereas the implementation of SOS+FLOW [89] computed all possibilities (all-sol), which were then ordered based on a maximal flow test. The result was extremely inefficient, although the encoding of SOS+FLOW in Alloy [67] potentially offers a better one-sol-implementation technique using SAT solving. Unfortunately, no benchmarks were presented in that paper, and as the implementation is very much a prototype, we have not included it here for comparison. We are not aware of an implementation of TILE2 or TILE3 using Maude, as for SOS.

Constraint satisfaction The implementation approach described in this chapter is based on constraint satisfaction techniques to derive efficient executable implementations of $\mathcal{R}$eo. The constraints are solved per round (single-step) and use the heuristics of the constraint solver to stop the search for solutions once a single solution is found, avoiding exploring the full solution space (one-sol).

We summarise the classification of implementation approaches discussed in this section in Table 5.8.

Implementation Approach	Number of Solutions	Pre-computed Behaviour
Speculative approach	one	single
Compilation into automata	all	all
Connector colouring	all	single
Search-based	one	single
Constraint satisfaction	one	single

Table 5.8: Classification of $\mathcal{R}$eo implementations approaches.

5.9.3 Constraints in Dreams

The distributed framework Dreams, the main contribution of this thesis, described in the next chapter, started to be developed prior to our constraint-based approach for coordination. Dreams executes $\mathcal{R}$eo connectors in a distributed environment, and restricts communication so that it can occur only through primitives, and thereby prohibiting both a global agent and direct node-to-node communication. This restriction imposes additional obligations on the implementation of primitives. Specifically, it requires them to play a significant role in the global constraint resolution process, for instance, to pass around colouring tables and to serve as the conduits for all 'coordination communication,' as well as for normal communication.

In the distributed implementation the primitives follow a distributed protocol to achieve consensus regarding how data should flow in each round, and only then data is passed through the primitives. The initial implementation of Dreams relied on connector colouring, but now we incorporated the constraint-solving techniques described in this section to improve efficiency. The Dreams framework and its implementation will be described in Chapters 6 and 7, respectively.

5.10 Related work

Wegner describes coordination as constrained interaction [101]. As such, coordination systems can be modelled by *interaction machines*. Interaction machines react to real-time interactive behaviour, representing the external world by infinite streams of inputs, allowing them to go beyond Turing machines in expressive power. The implementation model of $\mathcal{R}$eo presented in this chapter, which is

extended with an interaction layer, can be regarded as a concrete realisation of Wegner's interaction machine.

However, surprisingly little work takes Wegner's view of coordination as constrained interaction literally, representing coordination as constraints. Montanari and Rossi express coordination as a constraint satisfaction problem, in a similar but more general way [87]. They describe how to solve synchronisation problems using constraint solving techniques. Networks are viewed as graphs, and the tile model is used to distinguish between synchronisation and sequential composition of the coordination pieces. In our approach, we clarify one possible semantics of the coordination language $\mathcal{R}$eo in these terms, giving a clear meaning for each variable, and describing the interaction with the external world within the solve and update stages. Lazovik *et al.* also utilise constraints to solve a coordination problem [78]. They provide a choreography framework for web services, where choreography is formalised as a constraint programming task, and where both the Business Process and the requests are modelled as a set of constraints. This is an example of a concrete application of constraints to coordination, using a centralised and non-compositional approach.

Taking a more practical approach, Minsky and Ungureanu introduce the Law-Governed Interaction (LGI) mechanism [86], implemented by the Moses toolkit. This mechanism targets distributed coordination of heterogenous agents using a policy that enforces extensible laws. Laws are constraints specified in a Prolog-like language, enforced on regulated events of the agents, such as the sending or receiving of messages. The authors give a special emphasis to the deployment and execution of the mechanism, where a trusted server provides certified controllers which enforce the laws, instead of relying on a centralised coordination mechanism. However, laws are local, in the sense that can only refer to the agent being regulated. This allows them to achieve good performance using LGI. In the presence of true global constraints, as in $\mathcal{R}$eo, LGI would require more complex algorithms.

Frølund [52] presents *synchronisers* as a part of an actor coordination framework. He gives semantics for these constructs in terms of constraints, but does not use constraint solving as an implementation technique. The constraints perform the matching of atomic sets of actions along with pattern matching of data, but they do not deal with the communication of data, as our model does. Frølund's synchronisers cannot be plugged together like channels, but they can be composed by overlapping the domains of multiple synchronisers.

The analogy between $\mathcal{R}$eo constraints and constraint solving problems has already been drawn in general publications about $\mathcal{R}$eo [10]. Since then more specific approaches that utilise a constraint-based perspective over $\mathcal{R}$eo have been proposed. Examples of these approaches consider model checking and at the use of mashups. Klüppelholz and Baier describe a symbolic model checking ap-

proach for $\mathcal{R}$eo [69]. Constraint automata are represented by binary decision diagrams, encoded as propositional formulæ. Their encoding is similar to ours, though they use exclusively boolean variables, whilst we deal with a richer data domain. Maraikar *et al.* [79] present a service composition platform based on $\mathcal{R}$eo, adopting a mashup's data-centric approach. They combine several RSS feeds into a user interface using a $\mathcal{R}$eo connector, that is executed using the CHOCO constraint solver (referred in §5.7 and §5.6). The work by Maraikar *et al.* can be seen as an example of an application of the basic ideas that we present in this chapter.

The timed concurrent constraint (tcc) programming framework [94] was introduced by Saraswat *et al.* to integrate the concurrent constraint (cc) programming paradigm [93] with synchronous languages. Time units are rounds, all the constraints are updated in each round, as ours are, whereas inside each round the constraints are computed to quiescence. cc programs are compiled into an automata model, where states are cc programs and transitions represent evolution *within a round* while solving the constraints. In contrast, transitions in the constraint automata model for $\mathcal{R}$eo describe the evolution between rounds. Furthermore, the tcc approach avoids non-determinism as it targets synchronous languages, whilst $\mathcal{R}$eo, as a coordination language, embraces non-determinism.

Andreoli *et al.* [5, 4] also use (linear) logic as the basis for coordination, combining it with the object-oriented paradigm. They utilise proof search to reason about coordination, as opposed to the use of constraint satisfaction techniques to derive efficient implementations. Clarke follows a similar approach and presents a $\mathcal{R}$eo semantics, also based on proof search, using an extension of linear logic with temporal modalities in an intuitionistic setting [34].

A small overview of other coordination models and its relation with $\mathcal{R}$eo, briefly discussed in §3.1, is now in order. The survey of Papadopoulos and Arbab [90] compares several coordination languages, classifying languages based on tuple spaces as data-driven models, as opposed to $\mathcal{R}$eo's channel-driven model. Manifold [28] is another example of a channel-driven model presented in the survey, upon which $\mathcal{R}$eo was built. Linda [54], one of the first coordination languages, provides a simple executable model consisting of a shared tuple space that components use to exchange values. Several other variations, such as Java's popular implementation JavaSpace of Jini [51], and the Klaim language [26], which considers multiple distributed tuple spaces, followed the basic ideas behind Linda. The foundations of Klaim are presented as a process calculus, in particular as a variant of the π-calculus [85] with process distribution and mobility, where communication occurs via shared located repositories instead of channel-based communication primitives. Individual tuple operations in Linda-like languages are atomic, though they do not provide the global synchronisation imposed by $\mathcal{R}$eo.

Coordination languages such as SRML [48] and Orc [68] are oriented towards the coordination of web-services. SRML is a language developed in the context

of the SENSORIA project, where the interaction between two component services can be either synchronous or asynchronous, depending on whether an acknowledgement is required or not. Orc assumes a centralised coordinator that communicate with the component services only via asynchronous messages, instead of describing one to one communication. It also assumes that each service can reply at most once. $\mathcal{R}$eo takes the same exogenous approach as Orc, moving the coordination logic from the components to the coordinator, and introduces new synchronisation capabilities, not captured by the Orc language. An exhaustive formal comparison between $\mathcal{R}$eo and Orc was performed by Proença and Clarke [92].

Coordination models have been applied to coordinate solvers of distributed constraint satisfaction problems (DCSP) [18]. Our coordination model comes full circle and is based on (D)CSP.

5.11 Conclusion and future work

We presented a new semantics and executable model for the $\mathcal{R}$eo coordination model based on constraint satisfaction. This was motivated by one main concern. There is no efficient implementation technique for the existing models of $\mathcal{R}$eo. The channel-view of a $\mathcal{R}$eo connector becomes a mere a metaphor. Instead, a $\mathcal{R}$eo connector is seen as a set of constraints, based on the way the primitives are connected together, and their current state, governing the possible synchronisation and dataflow at the channel ends connected to external entities. The circuit representation of a $\mathcal{R}$eo connector then serves as a convenient graphical representation of how primitive constraints compose to yield the constrained interaction that it expresses.

We contribute to the state-of-the-art of $\mathcal{R}$eo in two ways:

- We identify the four main concepts that characterise the $\mathcal{R}$eo coordination, synchrony, data-awareness, state, and context dependency, and describe these concepts using logical constraints. We show the correctness of our approach with respect to the first three concepts using the constraint automata model as a reference, and we give full proof of correctness for the latter concept based on the connector colouring semantics.

- We reuse existing constraint satisfaction techniques to derive more efficient implementations of $\mathcal{R}$eo. Specifically, we developed two prototype implementations, one which uses a SAT solver and another which uses a constraint solver to search for possible solutions, and compared their performance with an existing $\mathcal{R}$eo engine based on connector colouring. The results strongly support the idea that constraint solving is a viable approach for implementing coordination languages.

In our first publication of this work [39], we explored the decomposition of $\mathcal{R}$eo into constraints, extended the framework presented here to model interaction with an unknown external world, beyond what is currently possible in existing implementations of $\mathcal{R}$eo. In that paper we assume that parts of the constraints are still unknown in the beginning of the constraint solving phase. The corresponding constraints could be requested from external entities when required. Clarke and Proença introduced a *local logic* [38], wherein constraints from only part of a connector need be consulted when searching for valid solutions. This means that partial solutions over only some of the variables are admitted, making this approach more scalable. Furthermore, partiality allows one to reason about 'incomplete' constraints, which can be extended during the constraint satisfaction process, enabling a new model of interaction with the external world.

Ongoing work on $\mathcal{R}$eo tools and on the distributed engine of $\mathcal{R}$eo in Dreams seek to extend the support based on the model proposed here. In particular, the distributed engine currently uses constraint solving techniques for synchronisation and context variables only. This means that data constraints are not incorporated, and the only interaction allowed is writing and reading on channel ends, as explained in §5.8, which can be improved using the constraint-based model with data proposed in this chapter. Constraints provide a uniform and flexible framework in which one may foresee in the future an assimilation of other constraint based notions, such as service-level agreements. Future work will explore these directions, in particular the increased expressiveness offered by constraints and the external interaction modes the model offers. In addition, we plan to exploit the parallelism inherent in constraints.

Chapter 6

The Dreams framework

6.1 Introduction

Dreams[1] (distributed runtime evaluation of atomic multiple steps) is a framework that supports a distributed coordination engine, where each part of the engine runs independently and interacts with the other parts by exchanging behavioural automata, transition labels, and data values.

Figures 6.1 and 6.2 illustrate the main idea behind Dreams. Figure 6.1 depicts a centralised coordination of four different services, connecting a phone-based and an Internet-based service to book hotels. Each hotel provides its own reservation service. All services interact with a single coordination mechanism. So far, all existing implementation approaches for $\mathcal{R}$eo, as we describe in §6.5.4, are centralised. The Dreams framework exploits the compositionality and locality features of behavioural automata to support the execution of the coordination mechanism in a distributed network, based on an asynchronous communication mechanism between all computational nodes of the network. Figure 6.2 illustrates the concept of a distributed coordination engine. Each cloud represents an independent thread of execution of the coordination layer, which cooperates with the clouds it is connected to. The global behaviour results from the composition of the behaviour of all connected clouds.

The Dreams framework was originally developed as a distributed framework for the $\mathcal{R}$eo coordination model, although it can also be used for other coordination models, provided that they can be encoded as behavioural automata. We use the three colouring semantics of $\mathcal{R}$eo (Chapter 4) to illustrate the use of the Dreams framework. Furthermore, we integrate our implementation with the existing tools for the $\mathcal{R}$eo model. For efficiency reasons, our implementation includes a solver that applies constraint satisfaction techniques, as described in Chapter 5.

[1] http://reo.project.cwi.nl/cgi-bin/trac.cgi/reo/wiki/Tools#DistributedReoEngine

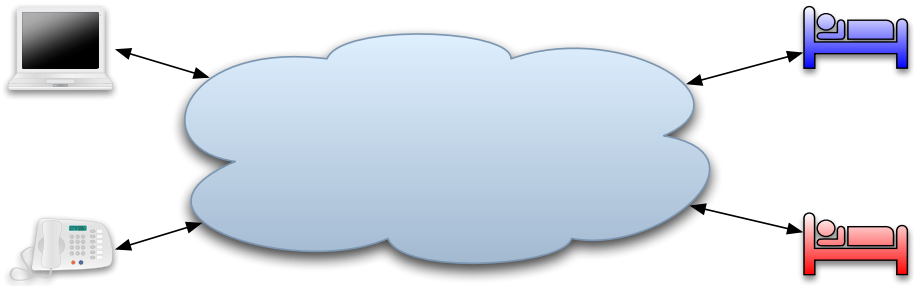


Figure 6.1: Coordination of web-services via a *centralised* engine. A phone- and an Internet-based service to book one of two hotels.

The distribution mechanism introduced by the Dreams framework addresses limitations of centralised approaches to implementing of synchronous languages. The three main concerns tackled by Dreams are not dealt with by all previous implementations of $\mathcal{R}$eo, due to their centralised nature. Note that most other coordination languages and middleware (e.g., JavaSpace [51] and JBoss[2]) also implement only centralised control. Specifically, Dreams supports *decoupling*, it *scales* better than previous approaches, and it supports *reconfiguration* at low cost.

Decoupling Parts of the system should be able to execute independently, subject to the constraints imposed by the coordination specification. For example, two independent communication events that do not depend on each other should be able to proceed in parallel. This provides a high level of concurrency.

Scalability The coordination mechanism should be able to scale up to coordinate a large number of entities, possibly by exploiting multiple CPU cores or by

[2] www.jboss.org

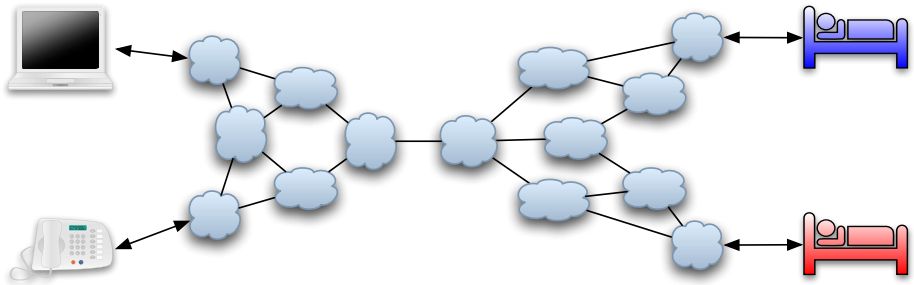


Figure 6.2: Coordination of web services via a *distributed* engine.

distributed execution.

Reconfigurability Reconfiguring an instance of a coordination pattern consists of changing some of its parts. Systems that are expected to be reconfigured frequently should be able to do it in an incremental manner, without requiring the full system to be changed. Furthermore, reconfigurations applied to a small part of the system should be independent of the execution or the behaviour of unrelated parts of the same system.

Dreams achieves decoupled execution in two steps. The core idea behind decoupled execution is the use of an underlying actor model [1, 2]. Primitive coordinators, which we simply call *actors*, exchange asynchronous messages amongst themselves. In the $\mathcal{R}$eo setting, an actor of Dreams manages one or more $\mathcal{R}$eo channels, $\mathcal{R}$eo nodes, or connectors. The communication among actors follows the graph structure of the corresponding $\mathcal{R}$eo connector, which restricts the potential communication partners of each actor to its neighbours only. In the second part of this chapter we explore the locality aspect described in §3.4 and in §4.4.2. By identifying independent regions of the graph of actors, the Dreams framework restricts the number of actors involved in each round of the coordination process, relaxing the restriction that all actors must be involved in every round. We do not address failure in the Dreams framework, and assume that the exchange of messages between actors always succeeds in finite time. However, we envisage failure addressed orthogonally at a later stage, outside this thesis, as suggested by existing techniques for distributed programming [55].

In this following we explain the Dreams framework in two steps. This chapter focuses on how the actors should be connected to guarantee the desired global behavioural automaton, based on the behavioural automata of the individual actors. In Chapter 7 we describe how actors can agree and execute their parts of a global behaviour, and present a distributed algorithm for the involved actors.

Organisation of the chapter In this chapter we present the Dreams framework, described as a system of actors each with two distinct kinds of behaviour. In §6.2 we introduce the notion of an actor, and make the distinction between *reactive behaviour* and *coordination behaviour*. In §6.3 we describe how actors are connected in the Dreams framework, present the basic assumptions regarding their behaviour, introduce the concept of *synchronous regions*, and present an overview of the evolution of the system of actors in time. We explore aspects of locality and the behaviour of actors to define the necessary conditions under which parts of a system can evolve independently of the rest of the system in §6.4. These independent parts are the so-called synchronous regions. We present the ideas behind the Dreams framework in §6.5 using the $\mathcal{R}$eo coordination language to illustrate the approach. We also compare Dreams with other existing ways to implement $\mathcal{R}$eo,

and explain how we improve on the state of the art. We finish with a description of related work outside the $\mathcal{R}$eo world and draw conclusions in §6.6 and §6.7.

6.2 Actors – overview

The primitive entities in the Dreams framework are *actors*. An actor is an active entity that runs concurrently with other actors and communicates with them using a reliable, order-preserving asynchronous message passing mechanism. We describe actors at two different levels of abstraction. At a lower level we use the *actor model* reference to describe how each actor communicates with others and how the message passing mechanism works. We call this the *reactive behaviour*. The actor model is a well studied formal model of concurrent computation, introduced already in 1973 by Hewitt, Bishop and Steiger [63]. We refer the interested reader to Agha's work for an overview of the actor model [1, 2]. At a higher level, we incorporate a specific behavioural automaton into each actor that describes the coordination protocol, and we call this the *coordination behaviour*.

Reactive behaviour The reactive behaviour describes how each actor reacts to incoming messages, following the actor model. In this model *actors* are primitive entities that communicate via asynchronous messages. Actors receive messages, and react by making local decisions, by (1) creating more actors, (2) sending messages, and (3) adapting to respond to future messages.

Coordination behaviour The Dreams framework provides the means to coordinate a set of components or services according to some behavioural automaton (BA), as we define in Chapter 3. We associate with each actor a behavioural automaton, and the coordination behaviour of a system in the Dreams framework is defined by the composition of the behavioural automata of all actors in the system.

In Dreams the execution of a behavioural automaton occurs in rounds, and in each round there is a consensus on which step should be executed next. The reactive behaviour describes how to exchange behavioural automata restricted to the current state, referred to as *frontiers* (at page 49), among a set of connected actors, and also describes how to transfer data between components based on the resulting composition of the frontiers.

A system in the actor model is called a *configuration*, and consists of a collection of concurrently executing actors together with a collection of messages that have been sent but not yet received. Message sending is fair, i.e., the delivery of messages can be delayed only for an arbitrary finite but bounded amount of time. Every actor has a *unique name* and reacts to the messages it receives according to a specific behaviour. The behaviour of an actor is deterministic, and is defined in terms of three basic operations:

1. create new actors;

2. send a finite number of messages to other actors; and
3. aquire new behaviour.

The sending of messages in the actor model is asynchronous. Each actor has an associated *mail queue*, and sending a message from actor A to actor B consists of appending this message to the end of the mail queue of B. After appending the message, A continues its computation and B keeps evaluating the messages in its mail queue, in the order of their arrival.

The Dreams framework distinguishes two orthogonal concerns during the life span of the actors. The main concern in the Dreams framework is to coordinate components or services based on an actor system with a *static topology*. In a static environment, there is no creation of new actors. Furthermore, in the actor model each actor can send messages to any actor in the same system, but we restrict the scope of the potential communication of each actor to its set of neighbours only, which does not change with time. This is a strong assumption that simplifies the reasoning about the reactive behaviour. We formalise this assumption in §7.2.

The second concern is reconfigurability. The evolution of the coordination behaviour in rounds makes reconfiguration versatile, since there is no additional cost for reconfiguring the system between steps. Reconfiguration takes place between rounds, and can therefore be seen as an orthogonal issue to the execution of each round. During a round we assume a static topology.

6.3 The big picture

A system built on top of Dreams consists of a set of actors that can communicate with each other within a static topology, as depicted in the example in Figure 6.3. Each actor has an associated behavioural automaton, characterising its coordination behaviour, and follows a specific protocol to interact with its neighbours to guarantee that the system behaves according to the composite behavioural automata of all connected actors. In this section we give an overview of the distributed protocol, explaining some basic assumptions about the actors, how they are connected to each other, and how the system evolves in time. We leave the full details of the protocol for Chapter 7. Note that an actor is completely specified by its behavioural automaton, hence we often use actors and their behavioural automata interchangeably.

6.3.1 Coordination via a system of actors

We now describe the role of actors in the coordination view of the Dreams framework. The coordination view is represented by a system of actors, as exemplified in Figure 6.3. Each of the clouds depicted in the figure represents a single actor,

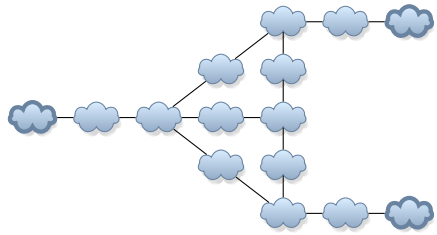


Figure 6.3: Graph structure of a system of actors in Dreams.

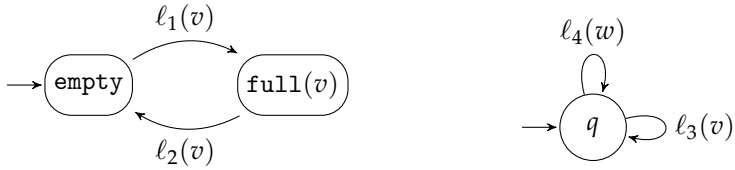
and the edges between clouds reflect the fact that the connected actors know each other as neighbours. Actors can send messages only to their *neighbour actors*. As mentioned at the end of §6.2, we assume that actors cannot send references to actors in messages and cannot create new actors. Thus the topology of the graph is static.

We extend each actor with its coordination information, assuming that each actor has an associated *behavioural automaton* (BA), as described in Chapter 3. The behavioural automaton of an actor is directly related to the topology of the connector as follows. Let $b_1 = \langle Q_1, \mathsf{L}[P_1], \rightarrow_1, \mathcal{O}_1 \rangle$ and $b_2 = \langle Q_2, \mathsf{L}[P_2], \rightarrow_2, \mathcal{O}_2 \rangle$ be two actors. Recall that, according to the definition of behavioural automata, the set $\mathsf{L}[P_i]$ consists of labels of transitions corresponding to actions that b_i can perform, each associated to an atomic step in $\mathsf{AS}[P_i]$ via a function α. The set P_i contains the ports used by these atomic steps. There is an edge between the actors b_1 and b_2 if and only if $P_1 \cup P_2 \neq \emptyset$. Therefore, edges between actors reflect the existence of shared port names between behavioural automata.

Furthermore, we distinguish two different kinds of actors, *proactive actors* and *non-proactive actors*. Intuitively, an actor is proactive if it has a port whose data values do not depend on any data value flowing through another port of the actor, and it is non-proactive otherwise. We formalise a proactive actor as an actor that has at least one *proactive port*, which we define below. But before that, we introduce the notion of value-independence between two ports, which is the key concept behind a proactive port.

6.3.1. DEFINITION (VALUE-INDEPENDENT PORTS).
A port x is value-independent from port y in state q, written $x \nsim_q y$, if for all $P, F, IP, OP \subseteq \mathbb{P}$, and label $\ell_i \in \mathsf{L}[P]$ such that $\alpha(\ell_i) = \langle P, F, IP, OP, data_i \rangle$, the following holds.

$$\begin{aligned} &\text{If } q \xrightarrow{\ell_1} q_1 \text{ and } q \xrightarrow{\ell_2} q_2 \text{ with } data_1(x) = a \text{ and } data_2(y) = b, \\ &\text{then there exists } q \xrightarrow{\ell_3} q_3 \text{ with } data_3(x) = a \text{ and } data_3(y) = b. \end{aligned} \qquad \triangleleft$$



$\alpha(\ell_1(v)) = \{ab, a, a, \emptyset, \{a \mapsto v\}\}$ $\quad$ $\alpha(\ell_3(v)) = \{cd, cd, c, d, \{c, d \mapsto v, v\}\}$
$\alpha(\ell_2(v)) = \{ab, b, \emptyset, b, \{b \mapsto v\}\}$ $\quad$ $\alpha(\ell_4(w)) = \{cd, c, c, \emptyset, \{c \mapsto w\}\}$

Figure 6.4: Behavioural automata of the FIFO_1 (left) and LossySync (right) channels.

Note that ℓ_1, ℓ_2 and ℓ_3 do not need to be distinct. Intuitively, Definition 6.3.1 states that if x can deliver a and y can deliver b, then x and y must be able to deliver a and b in a single step. We now define *proactive ports* as the ports that can send or receive data that originates from or is consumed by the actor, respectively. Intuitively, a port x is a proactive port when its data value does not depend on the value of any of the other ports.

6.3.2. DEFINITION (PROACTIVE PORT). A port $x \in IP \cup OP$ is proactive in state q if there exists a transition $q \xrightarrow{\ell} q'$ such that $\alpha(\ell) = \{P, F, IP, OP, data\}$ and $\forall y \in IP \cup OP \cdot x \nsim_q y$. $\lhd$

Finally, we formalise below a proactive actor as an actor with proactive ports. An actor that is not proactive is called non-proactive actor.

6.3.3. DEFINITION (PROACTIVE ACTOR). We say an actor is *proactive* if its behavioural automata has a proactive port for one of its state. $\lhd$

We refer to a label ℓ with an atomic step $\alpha(\ell) = \langle P, F, IP, OP, data\rangle$ as a *proactive step* if it involves a proactive port $x \in IP \cup OP$. The intuition behind the definition of proactive actors is that these actors will either produce data or consume data *ab initio* : only proactive actors have interest to initiate communication. To exemplify the proactive-related concepts we use the FIFO_1 and the LossySync channels.

6.3.4. EXAMPLE. Recall the FIFO_1 $\mathcal{R}$eo channel, presented in previous chapters using different formalisms (pages 23, 56, 59, 74, and 102). We depict the behavioural automata of a FIFO_1 channel with a source port a and a sink port b on the left side of Figure 6.4. Each arrow labelled with $\ell_i(v)$ represents a collection of labels, one for each data value $v \in \mathbb{D}$. The definition of the labels $\ell_1(v)$ and $\ell_2(v)$ is not important, since we only need to know the corresponding atomic steps. While in state `empty` the port a is proactive, because for any value $v \in \mathbb{D}$ the step $\ell_1(v)$ is proactive. It is enough to observe that $\alpha(\ell_1(v))$ has no output ports, hence $\forall y \in OP \cdot x \nsim_{\texttt{empty}} y$ trivially holds. The port b is not a proactive port in

state `empty` because it is neither an input nor an output port in any of the available transitions. ◊

6.3.5. EXAMPLE. Recall now the LossySync channel. We depict the behavioural automata of a LossySync channel with a sink port c and a source port d on the right side of Figure 6.4. The LossySync channel is not a proactive actor because its two ports are not value-independent. For example, let $v, w \in \mathbb{D}$ such that $v \neq w$. Then $q \xrightarrow{\ell_3(v)} q$ where the value v flows though c, and $q \xrightarrow{\ell_3(w)} q$ where the value w flows through d. However, there is no label ℓ such that $q \xrightarrow{\ell} q'$, v flows through c, and w flows though d. Hence neither c or d are proactive ports. ◊

We depict proactive actors using a thicker line for the border of their clouds, as exemplified in Figure 6.3. Proactive actors are the initiators of the distributed protocol, which we introduce in §6.3.3. Furthermore, proactive actors are the only actors that can decide which atomic step the Dreams configuration performs next.

6.3.2 Synchronous regions

Scalability mentioned in §6.1 is achieved via a true decoupling of the execution, which we address next. The specification of a Dreams configuration as a set of connected actors that can execute concurrently, as depicted in the example from Figure 6.3, already provides a basic decoupling of the execution. We go beyond this basic decoupling by analysing the behavioural automata of the actors and identifying links between actors that require only asynchronous communication. We depict these truly asynchronous connections using dotted lines, as shown in the example in Figure 6.5.

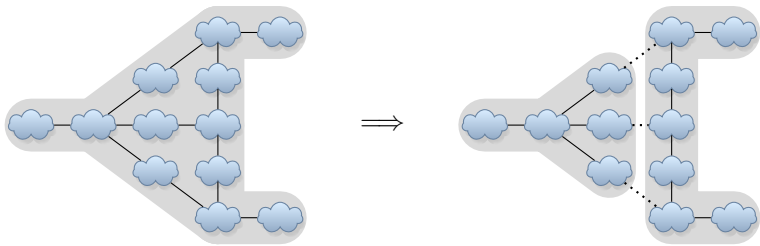


Figure 6.5: Creation of two synchronous regions in a Dreams configuration.

The presence of asynchronous communication yields what we call *synchronous regions*, depicted in Figure 6.5 by a grey background to group actors that belong to the same region. Actors in the same synchronous region must reach consensus among themselves before each round of communication, but actors from different synchronous regions can communicate asynchronously between rounds. Reducing the number of actors involved in the search for a consensus also reduces the

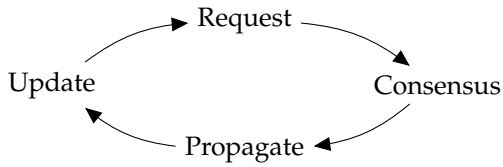
complexity of this search, which is visible in the results of our benchmarks presented in §7.5. This approach is possible only because we reason about both the reactive behaviour (asynchronous communication) and the coordination behaviour (typically synchronous communication) simultaneously.

In §6.4 we address the challenge of how to identify asynchronous communication. Recall that the composition of behavioural automata always requires the shared ports to synchronise, i.e., their actors must communicate in a synchronous manner. Furthermore, two behavioural automata with disjoint sets of ports can always evolve in parallel. Our approach relies on the fact that behavioural automata can have ports that are partially independent. Consider, for example, the behavioural automaton of the FIFO_1 channel, an automaton with two ports that we analyse in detail in §6.5.1. In any of its states, the dataflow on one of the ports is independent of the dataflow on the other port. We formalise how to generate two new behavioural automata by partitioning the ports of a single automaton, and the conditions under which such a partitioning preserves the behaviour of the original automaton, in §6.4.2. In general, such partitioning does not preserve the behaviour of the original automaton, but in truly asynchronous behavioural automata, such as the automaton of a FIFO_1 channel, the behaviour is preserved.

This chapter introduces a special splitting of actors used to identify synchronous regions. The two new resulting actors communicate with each other using asynchronous messages only. More specifically, the two new actors can send information regarding their state updates only to each other, and can produce only state changes that are not associated with the execution of an atomic step. For example, given two split actors A and B, if A changes state after a round it must communicate this change to actor B, which in turn can also change its state, although B may not have performed an atomic step. These state changes can be regarded as reconfigurations, and are not captured by the coordination behaviour.

6.3.3 Evolution in Dreams

This section describes how the execution of actors the Dreams framework evolve in time. Following the same ideas behind the constraint satisfaction-based engine for $\mathcal{R}$eo as discussed in §5.4.2, the system evolves in *rounds*. Each round consists of the following four phases.



Request Every proactive actor with a non-empty set of proactive ports P for the

current state sends a message to each of the actors it is connected to, that is, to every actor with which it shares a port in P, asking for its behavioural automaton. In turn, the neighbours will also ask the behavioural automata of their own neighbours, and so on.

Consensus The actors reply with their behavioural automata until one of the proactive actors collects the behavioural automata of all actors. This proactive actor finds the product of these automata and nondeterministically selects a possible atomic step as such that $q \xrightarrow{as} q'$ where q is the current state in the final behavioural automaton.

Propagate The selected atomic step is propagated to all actors along with the data according to the data function defined in as.

Update Each actor updates the state of its behavioural automaton according to the atomic step as.

In Chapter 7 we describe the algorithm implemented by the reactive behaviour, where each actor has an associated stage and communicates with its neighbour actors using a message passing mechanism. In this section we focus on the assumptions and properties of the Dreams framework, and how it relates to existing work. Note that the consensus phase in each round requires only the transitions from the current state. Therefore, each actor needs only to provide a behavioural automaton restricted to the current state, defined in Definition 3.4.2, as the *frontier*.

Figure 6.6: Simplified diagram relating the possible phases for each actor.

At the system level we have the four phases for each round presented above. At the level of each actor we distinguish different actor phases, to be introduced next. Figure 6.6 presents a simplified diagram with phases for each actor. A complete description of the phases of each actor is presented in §7.3. An actor is initially *idle*, represented by the colour , until it receives a request for its behavioural automaton. After the reception of a request, it forwards the request to its remaining neighbours (all its neighbours except the sender of the request) and becomes *committing*, represented by the colour . Once it has received all replies, the actor collects the required behavioural automata and returns its own behavioural automaton together with the collected replies to the actor that sent the initial request and becomes *committed*, represented by the colour . The consensus phase will be performed by a proactive actor. Once each actor receives

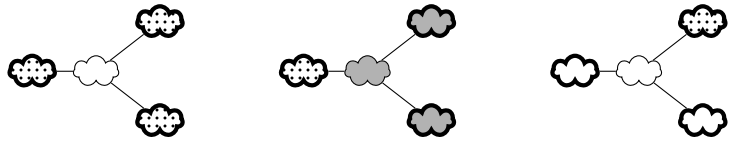


Figure 6.7: Example of the evolution of a round in a system in Dreams.

the resulting atomic step for the current round, it propagates the data, and becomes idle again.

6.3.6. EXAMPLE. In Figure 6.7 we give an example of the evolution of a round of a Dreams configuration, using a simplified version of the graph in Figure 6.3. This example gives some intuition of how the distributed algorithm works. Thick border lines denote proactive actors. We assume that the three proactive actors in Figure 6.7 (a) have a proactive port, and each of them starts the request phase by sending a request message to the actor in the centre. In (b) the three actors in grey are committed, and the left actor is still waiting for a reply to its request for the behavioural automaton of the middle actor. Only the left actor will receive the behavioural automaton of the whole system, while the other actors will only have a partial information about the global behaviour. The details of why only the left actor waits for a reply will be presented in Chapter 7, where we explain the distributed algorithm. Finally, Figure 6.7 (c) represents a possible result of the update phase after the previous round is finished. In this round only the top right actor starts the request phase, where we can safely conclude that only that actor has proactive atomic steps in the new state of its behavioural automaton, i.e., the other proactive actors cannot perform any proactive step to produce or consume data. ◊

Imagine now the following scenario. A proactive actor A can perform a proactive step and sends a request for communication. Imagine that, as a result of the consensus stage, it is not possible to perform the proactive step. None of the actors in the configuration changes the state of its behavioural automaton, and the proactive actor continues its useless efforts again and again to perform its proactive step with no success, since there is no atomic step that causes any change to the current configuration. To avoid this kind of scenarios we introduce two new assumptions that enable a proactive actor to identify when the sending of a request is no longer useful because no dataflow is possible. The first assumption is about the behavioural automata of actors, and the second assumption relates to how the choice of the next step is made in the consensus phase.

6.3.7. ASSUMPTION (SILENT NO-FLOW). *Let b be a behavioural automaton of an actor, and $q_1 \xrightarrow{\ell} q_2$ a transition where $\alpha(\ell) = \langle P, F, IP, OP, data \rangle$. We assume that if $F = \emptyset$, then $q_1 = q_2$, i.e., transitions that do not have dataflow in any of the ports do not change the state of a behavioural automaton.*

6.3.8. ASSUMPTION (AVOID NO-FLOW). *When a proactive actor selects a label ℓ in the consensus phase, where $\alpha(\ell) = \langle P, \emptyset, \emptyset, \emptyset, \emptyset \rangle$, and performs the corresponding transition, then there is no other label ℓ' starting from the current state such that $\alpha(\ell') = \langle P', F', IP', OP', data' \rangle$ and $F' \neq \emptyset$, that is, transitions with dataflow have precedence over transitions without dataflow.*

Using these two assumptions we can guarantee that no dataflow can occur in the current configuration after a step with no dataflow is taken. We make this statement precise in the following proposition.

6.3.9. PROPOSITION. *When all actors comply with the Assumptions 6.3.7 and 6.3.8, whenever a no-flow atomic step is taken (with the empty flow set) all following atomic steps will also be no-flow atomic steps.*

Proof. Let $q \xrightarrow{\ell} q'$ be a transition performed by a behavioural automaton such that $\alpha(\ell) = \langle P, F, IP, OP, data \rangle$ and $F = \emptyset$. By Definition 3.3.1 of an atomic step, also $IP = \emptyset$, $OP = \emptyset$ and $data = \emptyset$. By Assumption 6.3.7 $q = q'$, that is, there is no state change, and by Assumption 6.3.8 there is no transition with dataflow, which could trigger a state change. □

This proposition justifies why a proactive actor does not need to send a request while idle if the previous consensus phase selected a transition with an atomic step $\langle P, \emptyset, IP, OP, \mathbb{D} \rangle$.

We now argue why we consider these assumptions to be reasonable. The general idea is that only dataflow is observable of atomic steps. Since we expect to observe all state changes, transitions with no dataflow should not cause state changes. Assumption 6.3.7 formalises the intuition that a behavioural automaton can change only when there is dataflow. Alternatively, we could have extended the definition of an atomic step with a new boolean to represent whether a state change occurs. We believe that the model is clear and more intuitive without losing expressiveness if we simply forbid state changes labelled with atomic steps with no dataflow. Assumption 6.3.8 justifies the implementation choice of not to perform atomic steps with no dataflow.

Note that in $\mathcal{R}$eo, the context dependency described in Subsections §2.2.3, §4.3, and §5.5.2, conforms to and reinforces Assumption 6.3.8: when data is available in a context, it must flow unless there is a reason for no flow. Furthermore, in our constraint-solving approach for $\mathcal{R}$eo without context dependency, we impose an additional constraint $\bigvee P$, which states that only solutions where one of the ports has flow are relevant.

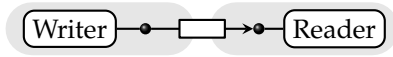


Figure 6.8: Synchronous regions separated by a FIFO_1 channel.

Summary We defined the Dreams framework to be a set of connected actors, where each actor has a behavioural automaton. We distinguished proactive actors from non-proactive actors as the actors that can produce or consume data. We also introduced the notion of a synchronous region, which plays a major role in tackling scalability, and briefly mentioned how we plan to identify such regions. Furthermore, we gave an overview of how each round in Dreams evolves, introducing extra assumptions to detect situations where there is no need to start the request phase.

Next, we explore local executions in §6.4, taking advantage of the behavioural automata of the actors, and the fact that certain regions of the system can stop the protocol until the behavioural automaton of the boundaries of a region changes.

6.4 Decoupled execution

In $\mathcal{R}$eo, constraint solving techniques (Chapter 5) improve the efficiency of the consensus stage compared to previous approaches based on the join of colouring tables. However, the scalability problem remained, as revealed by the benchmarks in §5.6. In these benchmarks, execution time appears to grow exponentially with the size of the connectors involved. Our approach to tackle this problem is to reduce the number of actors involved in the consensus phase. Solving the constraint satisfaction problem of a given connector is in general less efficient than solving twice the constraint satisfaction problem for a connector with half the size.

Recall the empty FIFO_1 channel in $\mathcal{R}$eo. The behaviour of the source and sink ports of this FIFO_1 channel are partially independent. The source port can have dataflow independently of the sink port, and the sink port will always have no dataflow independently of the source port. By considering that the two ports are independent, we can assume that the consensus phases involving the two ports of the FIFO_1 can evolve independently, yielding smaller regions to search for solutions, depicted with a grey background in Figure 6.8. We call these regions *synchronous regions*. We avoid the name 'independent region' because the behaviour of each region is not completely independent: they can evolve independently, but they have to report their state changes to the other regions using asynchronous messages. Below, we justify why actors such as the FIFO_1 can have ports that can be considered independently, based on their behavioural automata. Using this formalisation we show why the FIFO_1 channel can induce different regions, and why another so-called asynchronous $\mathcal{R}$eo primitive—the asynchronous drain

channel—fails to do so.

Up to now we considered all actors in the Dreams framework to be connected and required in each round. However, the stepwise coordination model introduced in Chapter 3 admits the possibility of concurrent executions of a connector, as a consequence of the concurrency predicates described for each state of the behavioural automata. The concurrency predicates describe when atomic steps of a subpart of a global behavioural automaton b can also be considered atomic steps of b, which we called local steps. If two or more disjoint local steps can be performed at some point in time, then they can be performed independently of each other in any order, or even in parallel in the same round.

The Dreams framework evolves by collecting all possible behavioural automata, and only then a proactive actor selects a possible atomic step. The final behavioural automaton is known only after considering all behavioural automata: it is not calculated during the sending of and replying to requests for the behavioural automata. We made this choice to avoid extra communication between actors, although some techniques exist for distributed constraint solving [104] where a possible local solution is communicated and verified, requiring a backtracking mechanism when the verification fails. As a consequence, we cannot find a local step without collecting the behavioural automata of *all* connected actors.

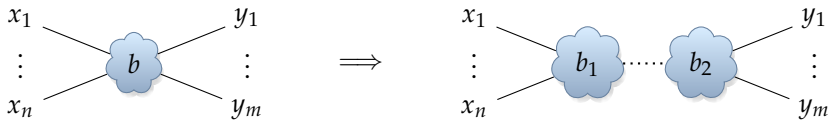


Figure 6.9: Splitting of an actor with asynchronous behaviour.

In our approach, we find actors with *asynchronous behavioural automata* and split them into two new actors, such that the sets of known ports of the behavioural automata of the new actors form a partition of the set of known ports of the behavioural automaton of the original actor. We depict the splitting of actors in Figure 6.9. However, when splitting an actor the coordination behaviour can change. Hence we impose soundness and completeness criteria that guarantee the preservation of the behaviour.

6.4.1 Restricted actors

Each of the new behavioural automata b_1 and b_2 in Figure 6.9 is a copy of the original behaviour b, restricted to its own set of associated ports. The splitting of actors is accomplished through the following restrict and filter functions, which

we define for behavioural automata as follows.

$$\begin{aligned}\alpha(\mathsf{restrict}_X(\ell)) &= \langle P \cap X, F \cap X, IP{\restriction}X, OP{\restriction}X, data{\restriction}X\rangle \\ &\quad \text{if } \alpha(\ell) = \langle P, F, IP, OP, data\rangle \\ \mathsf{filter}_X(\rightarrow) &= \{\langle q, \ell, q'\rangle \mid q \xrightarrow{\ell} q', \alpha(\ell) = \langle P, F, IP, OP, data\rangle, F \cap X \neq \emptyset\}\end{aligned}$$

The operator $\restriction$ denotes the standard restriction of functions. The restriction function $\mathsf{restrict}_X$ over labels needs to be defined for the concrete instance of a label, such as colourings (Chapter 4) or constraints (Chapter 5). The function $\mathsf{restrict}_X$ prunes a label by ignoring all ports not in X, and the function filter_X removes all transitions that do not have dataflow on any of the ports in X. Note that when splitting a behavioural automaton by applying the restrict and the filter functions to a partition of the ports, and considering all combinations of the restricted automaton, we can introduce new undesired behaviour. Later in this chapter we show how splitting an asynchronous drain channel produces such. Therefore, we will introduce some conditions that must hold when splitting actors, which will guarantee that the behaviour of the original behavioural automaton is preserved.

Filtering a set of ports X in the connector colouring model is achieved by removing colourings where some of the ports from X have dataflow. In the constraint-based approach this is achieved by adding conjunctively the constraint $\bigwedge\{\neg x \mid x \in X\}$. Restricting a set of ports X in the connector colouring is done by removing references to the colour of the ports in X. In the constraint approach, this is done by adding an existential quantifier binding the variables in X over the constraints.

In Table 6.1 we exemplify the result from applying the functions $\mathsf{filter}_{\{x\}}$ and $\mathsf{restrict}_{\{x\}}$ in the stepwise coordination model, in the connector colouring model, and in the constraint-based model. For simplicity, we disregard data constraints in the three cases. We choose the non-deterministic LossySync, which is a stateless primitive. The filtering process removes the possibility of having dataflow on y, by extracting ℓ_1 and c_1, and by adding the constraint $\neg y$. The restriction process ignores occurrences of y, by removing it from the remaining atomic steps and colouring tables, and by adding the existential quantifier $\exists y$. We formalise the restriction of an actor as follows.

6.4.1. DEFINITION (RESTRICTED ACTOR). Let $b = \langle Q, \mathsf{L}[P], \rightarrow, \mathcal{O}\rangle$ be a behavioural automaton and X a set of ports. Restricting b to X yields the automaton $\langle Q, \mathsf{L}[P], \rightarrow', \mathcal{O}\rangle$, where

$$\rightarrow' = \left\{\langle q, \mathsf{restric}_X(\ell), q'\rangle \mid \langle q, \ell, q'\rangle \in \mathsf{filter}_X(\rightarrow)\right\}. \qquad \lhd$$

Notation Let $b = \langle Q, \mathsf{L}[P], \rightarrow, \mathcal{O}\rangle$ be a behavioural automaton, X a set of ports, and $\ell \in \mathsf{L}$ a label. We write $b^{(X)}$ to denote the behavioural automaton b restricted

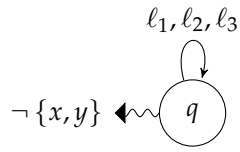
x - - - - → y	$\Longrightarrow$	x - - - $\not\vdash\not\dashv\not\rightarrow\not y$
Beh. automaton	2-Colourings	Sync. constraint
	$q \xrightarrow{c_1} q$ $q \xrightarrow{c_2} q$ $q \xrightarrow{c_3} q$	$y \to x$
$\alpha(\ell_1) = \langle xy, xy, \emptyset, \emptyset, \emptyset \rangle$ $\alpha(\ell_2) = \langle xy, \ y, \emptyset, \emptyset, \emptyset \rangle$ $\alpha(\ell_3) = \langle xy, \emptyset, \emptyset, \emptyset, \emptyset \rangle$	$c_1 = \{x \mapsto \longrightarrow\bullet, y \mapsto \longrightarrow\bullet\}$ $c_2 = \{x \mapsto \longrightarrow\bullet, y \mapsto \text{- - -}\bullet\}$ $c_3 = \{x \mapsto \text{- - -}\bullet, y \mapsto \text{- - -}\bullet\}$	$\sigma_1 = \{x \mapsto \top, y \mapsto \bot\}$ $\sigma_2 = \{x \mapsto \top, y \mapsto \bot\}$ $\sigma_3 = \{x \mapsto \bot, y \mapsto \bot\}$
$\Downarrow$	$\Downarrow$	$\Downarrow$
new atomic steps: $\alpha(\ell_1)$: filtered $\alpha(\ell_2) = \langle x, x, \emptyset, \emptyset, \emptyset \rangle$ $\alpha(\ell_3) = \langle x, \emptyset, \emptyset, \emptyset, \emptyset \rangle$	new colourings: c_1: filtered $c_2 = \{x \mapsto \longrightarrow\bullet\}$ $c_3 = \{x \mapsto \text{- - -}\bullet\}$	new constraint: $\exists y \cdot (\neg y \wedge (y \to x))$ $\sigma_2 = \{x \mapsto \top\}$ $\sigma_3 = \{x \mapsto \bot\}$

Table 6.1: Filtering and restriction for the non-deterministic LossySync.

to X, and ℓ^X to denote the label $\text{restrict}_X(\ell)$. When $X = \{x\}$ we also write ℓ^x and $b^{(x)}$ instead of ℓ^X and $b^{(X)}$.

6.4.2 Splitting actors

We now use restriction of actors to define the splitting of an actor.

6.4.2. DEFINITION (SPLITTING OF AN ACTOR). Let $b = \langle Q, \mathsf{L}[P], \to, \mathcal{O} \rangle$ be a behavioural automaton, and X, Y a partition of P, i.e., $X \cap Y = \emptyset$ and $X \cup Y = P$. We define the splitting of an actor with b as its behavioural automaton over the partition X, Y to be the pair of actors with behavioural automata $b^{(X)}$ and $b^{(Y)}$. $\lhd$

Figure 6.9 represents a split of actors where $b_1 = b^{(X)}$ and $b_2 = b^{(Y)}$, for $X = \{x_1, \ldots, x_n\}$ and $Y = \{y_1, \ldots, y_m\}$. We postpone the presentation of concrete examples to §6.5, where we use $\mathcal{R}$eo and its connector colouring semantics to exemplify the splitting of actors.

By splitting actors we discard some information from the original behavioural automaton. However, we are interested only in specific splitting situations, where the original behaviour is preserved. Let $b = \langle Q, \mathsf{L}[P], \to, \mathcal{O} \rangle$ be an original behavioural automaton, and $b^{(X)} = \langle Q, \mathsf{L}[P_X], \to_X, \mathcal{O}_X \rangle$ and $b^{(Y)} = \langle Q, \mathsf{L}[P_Y], \to_Y, \mathcal{O}_Y \rangle$ be a pair of split actors. Both split actors share the same state space, viz. Q,

and we expect each to mimic the corresponding state changes of the other split actor. This is achieved by each split actor sending an asynchronous message with its atomic step to the other split actor. whenever it changes its state. This is part of the *reactive behaviour*, which we explain in detail in §7.3. Split actors update their status upon receiving such an atomic step from their twin split actors. Semantically, the combined coordination behaviour of the two split actors is subject to soundness and completeness criteria, defined below. These conditions must hold to guarantee the preservation of the coordination behaviour of b.

6.4.3. DEFINITION (SOUNDNESS CRITERIA). Let P be a set of ports and X, Y a partition of P. We say the splitting of $b = \langle Q, \mathsf{L}[P], \rightarrow, \mathcal{O}\rangle$ into $b^{(X)} = \langle Q, \mathsf{L}[P_X], \rightarrow_X, \mathcal{O}_X\rangle$ and $b^{(Y)} = \langle Q, \mathsf{L}[P_Y], \rightarrow_Y, \mathcal{O}_Y\rangle$ is *sound* iff for every pair of transitions $q \xrightarrow{\ell_1}_X q_1$ and $q \xrightarrow{\ell_2}_Y q_2$, there is a pair of labels $r, t \in \mathsf{L}[P]$ where $\ell_1 = r^X$ and $\ell_2 = t^Y$, and a state $q' \in Q$, such that:

$$q \xrightarrow{r} q_1 \tag{6.1}$$

$$q \xrightarrow{t} q_2 \tag{6.2}$$

$$q \xrightarrow{r} q_1 \xrightarrow{t} q' \text{ or } q \xrightarrow{t} q_2 \xrightarrow{r} q', \text{ and } q \xrightarrow{r^X \otimes t^Y} q' \tag{6.3}$$

$$\text{if } q \xrightarrow{r} q_1 \xrightarrow{t} q_1' \text{ and } q \xrightarrow{t} q_2 \xrightarrow{r} q_2', \text{ then } q_1' = q_2'. \tag{6.4}$$

◁

The soundness criteria state that each transition from $b^{(X)}$ or $b^{(Y)}$ corresponds to some transition in b. Note that the state space Q is the same in b, $b^{(X)}$ and $b^{(Y)}$. Equations (6.1) and (6.2) state that each transition in the new behavioural automata corresponds to an analogous state change in b, as can be inferred from the definition of restriction (Definition 6.4.1). Equation (6.3) says that b must be able to perform r and t sequentially in some order, having q_1 or q_2 as an intermediate state, and it must also be able to perform the step $r^X \otimes t^Y$, leading to the same target state q'. Finally, Equation (6.4) says that when b can perform r and t in any order, then they must reach the same ending state, that we know by Equation (6.3) to be the state reached by $r^X \otimes t^Y$. The intuition is that each split actor can evolve independently of the other actor, and can adapt deterministically to state changes performed by its twin split actor, respecting the original behaviour.

6.4.4. DEFINITION (COMPLETENESS CRITERIA). We say that for a partition X, Y the splitting of $b = \langle Q, \mathsf{L}[P], \rightarrow, \mathcal{O}\rangle$ into $b^{(X)} = \langle Q, \mathsf{L}[P_X], \rightarrow_X, \mathcal{O}_X\rangle$ and $b^{(Y)} = \langle Q, \mathsf{L}[P_Y], \rightarrow_Y, \mathcal{O}_Y\rangle$ is *complete* iff for every transition $q \xrightarrow{\ell} q'$ of b one of the follow-

ing conditions holds:

$$q \xrightarrow{\ell^X}_X q' \tag{6.5}$$

$$q \xrightarrow{\ell^Y}_Y q' \tag{6.6}$$

$$\ell = r^X \otimes t^Y, \text{ and } q \xrightarrow{r} q_1 \xrightarrow{t} q' \text{ or } q \xrightarrow{t} q_2 \xrightarrow{r} q' \text{ where } q \xrightarrow{r^X}_X q_1,\ q \xrightarrow{t^Y}_Y q_2. \tag{6.7}$$

◁

The completeness criteria state that every transition of b can be performed by either $b^{(X)}$, $b^{(Y)}$, or by the composition of a step from each of these behavioural automata. Equations (6.5) and (6.6) hold when the transition in q is not filtered in the construction of one of the restricted actors. If a transition by ℓ in b is filtered by both split actors, then Equation (6.7) must hold, that is, ℓ must be obtained by composing the two atomic steps r^X and t^Y, one from each split actor, both starting from the same state as b, and the target state must be reachable by performing these two atomic steps in b in any order.

Consider the behavioural automaton b of the non-deterministic LossySync presented in Table 6.1. The splitting of b over the partition $\{x\}, \{y\}$ yields the behavioural automata $b^{(x)}$ and $b^{(y)}$. In this example we also describe the behavioural automaton $b^{(y)}$. The behavioural automaton $b^{(y)}$ has a single atomic step after the restriction: $\ell_3^y = \langle y, \emptyset, \emptyset, \emptyset, \emptyset \rangle$. This splitting is not complete with respect to b, because the atomic step corresponding to the sending of data through the channel is lost. More specifically, the completeness conditions do not hold. The transition $q \xrightarrow{\ell_1} q$ does not come directly from a transition in $b^{(x)}$ or $b^{(y)}$ (Equations (6.5) and (6.6)). Moreover, Equation (6.7) does not hold either, because there are no transitions r and t in $\rightarrow$ such that r^x and t^y belong to the split actors, and $\ell = r^x \otimes t^y$.

In the next subsection we exemplify a more complex behavioural automaton in the context of $\mathcal{R}$eo that can be split, and another example that cannot be split. We show that the soundness and completeness criteria hold in the first case, and show where they fail for the second case. Furthermore, we present a variation of the second example that satisfies the soundness and completeness criteria.

6.5 Decoupled execution of $\mathcal{R}$eo

This section uses the $\mathcal{R}$eo coordination language to illustrate the decoupling of execution of the stepwise model, described in §6.4, in a more concrete scenario. The existing implementation of the Dreams framework, which we describe in Chap-

ter 7, currently incorporates only two concrete incarnations of behavioural automata: connector colouring and the constraint-based semantic models of $\mathcal{R}$eo.

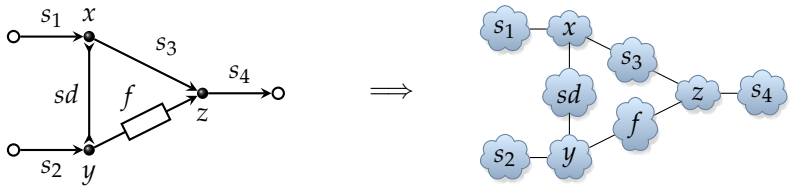


Figure 6.10: Encoding of a $\mathcal{R}$eo connector as a Dreams configuration.

The Dreams framework maps each $\mathcal{R}$eo channel directly to an actor with the concrete incarnation of the behavioural automaton given by its $\mathcal{R}$eo semantics. Merging and replicating of dataflows are performed by $\mathcal{R}$eo nodes with multiple source and sink channel ends, which are also $\mathcal{R}$eo primitives, each with a well-defined behavioural automaton, encoded as a single actor. We depict a possible encoding of a $\mathcal{R}$eo connector into the Dreams framework in Figure 6.10, according to the extreme scenario where every node and channel is encoded as a separate actor. Alternatively, we can also define a single actor with the combined behavioural automaton of the full connector, resulting in a centralised implementation of this connector. In the examples presented in this section we use the connector colouring semantics [37], described in Chapter 4, and give examples of a $\mathcal{R}$eo primitive that can be split, a $\mathcal{R}$eo primitive that cannot be split (although intuition may suggest otherwise), and larger $\mathcal{R}$eo connectors that benefit from the decoupled execution gained by splitting of actors.

Notation We represent labels as colourings with a data function, as defined in §4.4.4. That is, $\mathsf{L} = \mathbb{C} \times (\mathbb{P} \rightharpoonup \mathbb{D})$, where $\mathbb{C}$, $\mathbb{P}$, and $\mathbb{D}$ are the global sets of colourings, ports, and data values. Let *col* be a colouring, that is, a mapping from ports to colours. We write

$$\ell \;=\; col(\widehat{a}_1 = v_1, \ldots, \widehat{a}_n = v_n)$$

to assign to the variable ℓ the label with atomic step $\alpha(\ell) = \langle P, X, IP, OP, data \rangle$, where $n \geq 0$, P is the set of known ports of the primitive with colouring *col*, X is the set of ports with a flow colour, IP and OP form a partition of $\{a_1, \ldots, a_n\}$ containing its source and its sink ports, respectively, and $data(a_i) = v_i$.

6.5.1 Splitting the FIFO$_1$ channel

Recall the behavioural automaton of the FIFO$_1$ channel in Table 6.2. We use the connector colouring semantics, and use parameterised steps and states over a gen-

eral data value v to denote a family of steps and states, respectively. We also use the notation introduced in §3.3.1 for concurrency predicates, where $\neg P$ denotes all atomic steps that do not have flow on any of the ports in the set P.

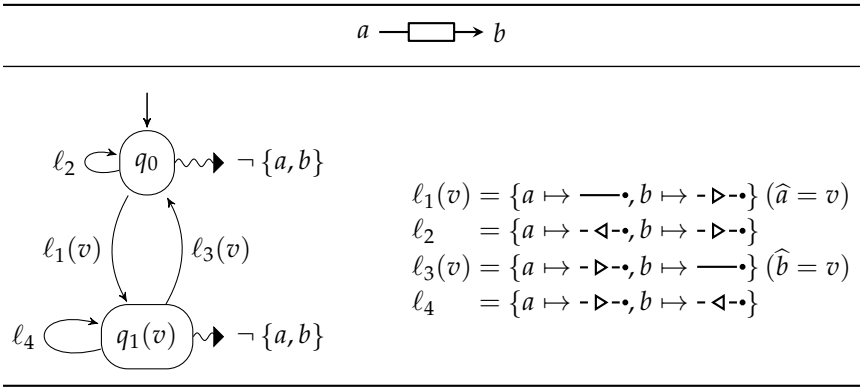


Table 6.2: Behavioural automaton of the FIFO_1 channel.

We split the actor for the FIFO_1 channel into two actors by applying Definition 6.4.2, based on the filtering of its colouring tables and on the restrictions of colourings. Each of the split actors represent one of the two ends of the FIFO_1 channel. The result is presented in Tables 6.3 and 6.4. Recall that filtering a colouring table consists of removing all colourings that have flow on a given set of ports X, and restricting a colouring c consists of restricting the domain of c to X. We depict the removed transitions and colourings in Tables 6.3 and 6.4 in grey colour, by crossing out their text, and using dashed lines in the state diagrams.

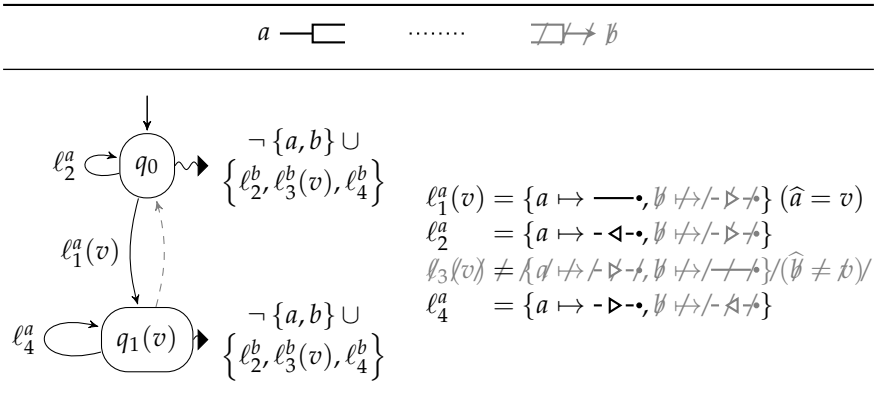


Table 6.3: Behavioural automaton the FIFO_1 after restricting to port a.

We now verify that the soundness and completeness criteria presented in §6.4 hold for the restricted behavioural automata presented in Tables 6.3 and 6.4.

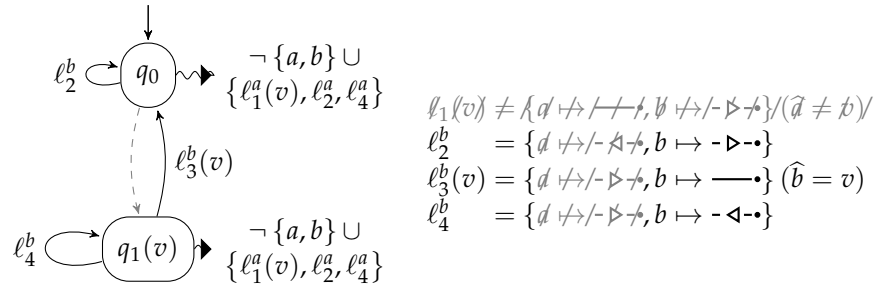


Table 6.4: Behavioural automaton the FIFO_1 after restricting to port b.

Soundness Equations (6.1) and (6.2) can be easily verified by observing that for every ℓ_i^x from the new behavioural automaton the transition ℓ_i exists that performs the same state update in the original behavioural automaton. To show Equation (6.3) observe that the composition of all possible combinations of labels from the same state are defined as follows: $\ell_1^a(v) \otimes \ell_2^b = \ell_1(v)$, $\ell_2^a \otimes \ell_2^b = \ell_2$, $\ell_4^a \otimes \ell_3^b(v) = \ell_3(v)$, and $\ell_4^a \otimes \ell_4^b = \ell_4$. It is now easy to see that for every composition $\ell_i^a \otimes \ell_j^b = \ell_k$ considered before, if $q \xrightarrow{\ell_k} q'$ in the original automaton, then $q \xrightarrow{\ell_i}_a q_1 \xrightarrow{\ell_j}_b q'$ or $q \xrightarrow{\ell_j}_b q_2 \xrightarrow{\ell_i}_a q'$, where $\rightarrow_a$ and $\rightarrow_b$ are the transition relations of each of the new behavioural automata. Finally, Equation (6.4) can also be verified by analysing every possible combination of atomic steps ℓ_i^a and ℓ_j^b starting from the same state, and observing that whenever $q \xrightarrow{\ell_i} q_1 \xrightarrow{\ell_j} q_1'$ and $q \xrightarrow{\ell_j} q_2 \xrightarrow{\ell_i} q_2'$, then $q_1' = q_2'$.

Completeness In this example completeness can be easily shown by observing that for every transition $q \xrightarrow{\ell} q'$ in the original automaton either $q \xrightarrow{\ell^X}_X q'$ or $q \xrightarrow{\ell^Y}_Y q'$ holds in one of the restricted automata. That is, no transition is filtered by both split actors, therefore Equations (6.5) or (6.6) always hold.

6.5.2 The asynchronous drain cannot be split

We follow a similar approach to split the asynchronous drain channel. The split of this channel into two actors with a channel end each is not sound, i.e., does not satisfy the soundness criteria, as we will show soon. The splitting where one of the split actors is restricted to all the ports and the other does not have any ports is trivially sound and complete, although not useful in our framework.

The asynchronous drain is a stateless primitive with two source ends, which

can receive data from either of its ends, but not from both ends at the same time. The purpose of this example is to show that the kind of asynchrony exhibited by the asynchronous drain channel differs from the asynchrony in the FIFO_1 channel. Furthermore we demonstrate a variant of the asynchronous drain channel that makes the same pair of split actors sound and complete.

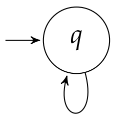
a ≻—‖—≺ b	
 $\ell_1, \ell_2, \ell_3, \ell_4$	$\ell_1 = \{a \mapsto$ ——•$, b \mapsto$ -▷-•$\}$ $\ell_2 = \{a \mapsto$ -◁-•$, b \mapsto$ -▷-•$\}$ $\ell_3 = \{a \mapsto$ -▷-•$, b \mapsto$ ——•$\}$ $\ell_4 = \{a \mapsto$ -▷-•$, b \mapsto$ -◁-•$\}$

Table 6.5: Behavioural automaton of the asynchronous drain channel.

We present in Table 6.5 the behavioural automaton of the asynchronous drain channel using its connector colouring semantics. This channel is stateless, i.e., it has only one possible state. Apart from the data function, the atomic steps are exactly the same as in the FIFO_1 channel, the main difference being that here we have only one state. Following Definition 6.4.2 we obtain two split actors whose behavioural automata have only reflexive transitions, defined in Table 6.6.

a ≻—‖			‖—≺ b		
ℓ_1^a	=	$\{a \mapsto$ ——•$\}$	ℓ_2^b	=	$\{b \mapsto$ -▷-•$\}$
ℓ_2^a	=	$\{a \mapsto$ -◁-•$\}$	ℓ_3^b	=	$\{b \mapsto$ ——•$\}$
ℓ_4^a	=	$\{a \mapsto$ -▷-•$\}$	ℓ_4^b	=	$\{b \mapsto$ -◁-•$\}$

Table 6.6: Behavioural automata of a splitting of the asynchronous drain.

The absence of state changes compared to the FIFO_1 channel breaks one of the soundness criteria, by introducing new behaviour. More specifically, Equation (6.3) does not hold. For example, $q \xrightarrow{\ell_1^a}_a q$ in $b^{(a)}$ and $q \xrightarrow{\ell_3^b}_b q$ in $b^{(b)}$, but it does not hold that $q \xrightarrow{\ell_1 \otimes \ell_3} q$ in the original automaton b, which corresponds to a step with dataflow in both ports. The intuition behind this failure is that the asynchronous drain in fact imposes an exclusion between its ports, but this is lost when we consider each port in isolation. We believe that the name 'asynchronous drain' found in the literature [8, 9, 22] is therefore misleading. In view of the above discussion it would make sense to rename it to 'exclusive drain' or similar. However, for consistency with the literature, we use the established name for this channel.

Consider now a different drain channel, called *non-deterministic drain*. This

b	$b^{(a)}$	$b^{(b)}$
$\ell_1 = \{a \mapsto \longrightarrow\bullet, b \mapsto \longrightarrow\bullet\}$	–	–
$\ell_2 = \{a \mapsto \text{-}\triangleright\text{-}\bullet, b \mapsto \text{-}\triangleright\text{-}\bullet\}$	$\ell_2^a = \{a \mapsto \text{-}\triangleright\text{-}\bullet\}$	$\ell_2^b = \{b \mapsto \text{-}\triangleright\text{-}\bullet\}$
$\ell_3 = \{a \mapsto \longrightarrow\bullet, b \mapsto \text{-}\triangleright\text{-}\bullet\}$	$\ell_3^a = \{a \mapsto \longrightarrow\bullet\}$	–
$\ell_4 = \{a \mapsto \text{-}\triangleright\text{-}\bullet, b \mapsto \longrightarrow\bullet\}$	–	$\ell_4^b = \{b \mapsto \longrightarrow\bullet\}$

Table 6.7: Atomic steps of the non-deterministic drain and its split.

channel differs from the asynchronous drain as follows. The flow of data on both ends is now possible, and it can always provide a reason for no-flow. We define in Table 6.7 the valid colourings for the behavioural automaton b of the non-deterministic drain, and for the behavioural automata $b^{(a)}$ and $b^{(b)}$ of its split actors. The intuition behind this drain is that it can always choose to receive or not receive data through either of its ends, independently of the context and of the behaviour of its other end. The soundness and completeness conditions hold for the splitting of the non-deterministic drain, which can be easily verified as there is only a single state. The soundness and completeness capture the independence of the two ends of the non-deterministic drain, which does not exist in the case of the asynchronous drain. Note that there is no extra communication between the two actors to communicate their state changes, since there is never a state change. Therefore, intuitively the non-deterministic drain must be the same as two completely independent $\mathcal{R}$eo primitives, each with a single end, which is true for the resulting split actors.

6.5.3 Splitting into synchronous regions

We have shown that the FIFO_1 channel and a non-deterministic drain can be split into new actors without changing their original behaviour. Other $\mathcal{R}$eo channels too can each be split into two new actors, each actor engaged with only one of the ends of the channel. Some examples are variations of the FIFO_1 channel, such as the *FIFO*$_2$ channel (a buffered channel with two memory slots instead of one), the *shift lossy FIFO*$_1$ channel (a channel that buffers at most one value, but replaces the content of the buffer when new data arrives), and the *variable* channel (a channel that behaves as the shift lossy FIFO_1 except the buffered value is replicated before sending, so once it becomes full it never gets empty again). Checking soundness and completeness of the splittings of these channels is analogous to our previous examples. We now explore the advantages of splitting actors in $\mathcal{R}$eo connectors, based on the splitting of the FIFO_1 channel.

We use the *sequencer connector* as a motivating example to illustrate the advantages of the splitting of actors. The sequencer connector is depicted in the lower part of the connector in Figure 6.11. It consists of a sequence of FIFO_1 channels

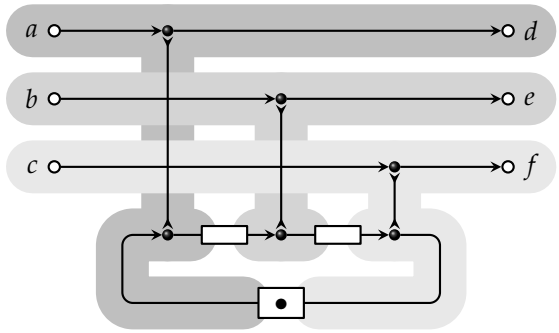


Figure 6.11: Synchronous regions induced in the sequencer connector.

in a loop, where only one of the FIFO's is full. Data is replicated between each pair of FIFO_1 channels, producing data sequentially every time data flows from one FIFO_1 to the next one. The most typical use of this connector is as a synchronisation barrier to alternate the flow of n synchronous channels. Figure 6.11 shows how the sequencer is used to guarantee that a flow from a to d is always followed by a flow from b to e, which is followed by a flow from c to f, and so on. By considering each of the ends of the FIFO_1's as an independent end, we deduce three implicit regions each of which can evolve independently of the other regions, without adding or removing any behaviour. These three regions are depicted using a different background colour for each region. In Figure 6.13 we present the corresponding $\mathcal{R}$eo connector as a system of actors, where each node and channel is implemented by an actor, apart from the FIFO_1 channels, each of which is implemented by a pair of split actors. We depict actors associated with nodes by  to improve readability.

Recall now the synchronising merge connector, presented in Figure 2.2 in §2.2. The synchronising merge connector imposes the following behaviour on the two components A and B. When A, B, or both are invoked by the presence of data on their source ports (on their left), then they cannot be invoked again until each of the previously invoked components produces a value through its sink port (on its right). If the components A and B have independent ends, then two synchronous regions arise from the splitting of the FIFO's, as depicted in Figure 6.12. Therefore, replacing an asynchronous component by a pair of components combined with the synchronising merge connector still yields a new system with independent ports i and o. In Figure 6.14 we present the synchronising merge connector viewed as a system of actors. We assume that the components A and B communicate only asynchronously, depicted by a dotted line between their source and sink ports.

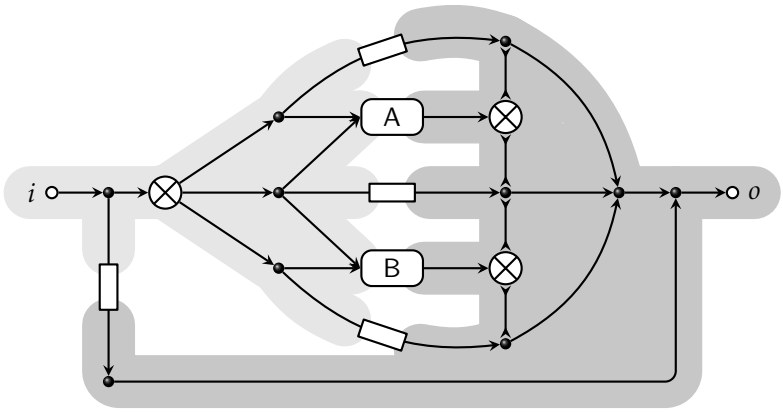


Figure 6.12: Synchronous regions of the synchronising merge connector.

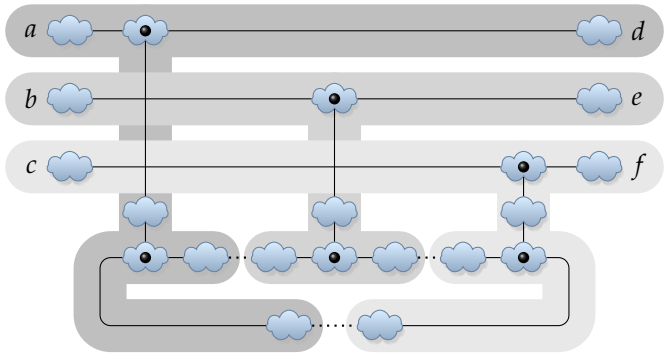


Figure 6.13: System of actors for the sequence connector.

6.5.4 Discussion

Recall the comparison of existing implementation approaches to ℛeo presented in §5.9. We extend this comparison by introducing the distinction between centralised and distributed implementations, and comparing a new approach: the *commit and send* approach, embodied in the Dreams framework. We summarise our findings in Table 6.8.

We compare five different implementation approaches for ℛeo, plus the new commit and send approach. The *speculative* approach consists of trying to send data through the channels and rolling back when an inconsistency arises. The *automata-based* approach [79] pre-computes all future behaviour at compile time. Implementations based on *connector colouring* [37] compute all solutions for the behaviour of each round, described as colouring tables, and deal with data trans-

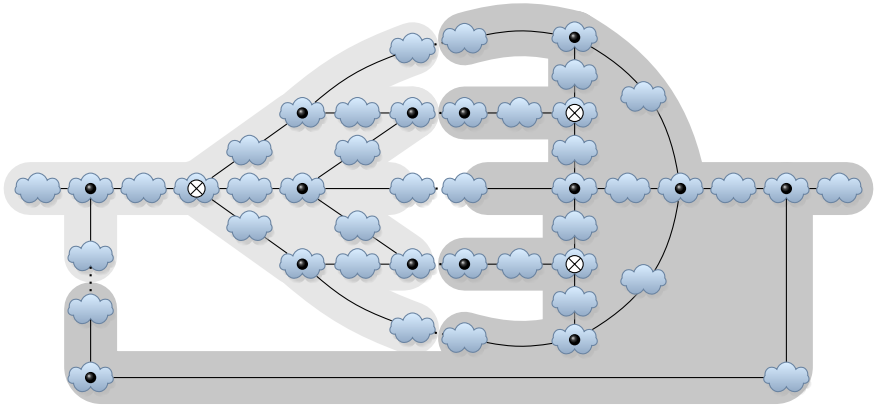


Figure 6.14: System of actors for the synchronising merge connector.

Implementation Approach	Number of Solutions	Pre-computed Behaviour	**Distributed**
Speculative approach	one	single	✓
Compilation into automata	all	all	✗
Connector colouring	all	single	✗
Search-based	one	single	✗
Constraint satisfaction	one	single	✗
Commit and send	one	single	✓

Table 6.8: Classification of $\mathcal{R}$eo implementation approaches.

fer orthogonally. Existing *search-based* implementations are based on SOS models and implemented either in Maude or in Alloy [89, 67], and can also be based on the Tile models [53, 14]. Finally, the *constraint-based* approach [40] described in Chapter 5 utilises SAT solving techniques to search for single solutions in each round.

The speculative approach is the only approach prior to Dreams that aims to achieve a distributed implementation of $\mathcal{R}$eo. However, as mentioned in §5.9.2, this approach was never successfully implemented. Therefore, Dreams is the first successful implementation of a distributed engine for $\mathcal{R}$eo, and is based on a *commit and send* approach. We describe this new approach to compute only one possible solution, and to compute a single behaviour instead of pre-computing all possible behaviour alternatives. The Dreams framework requires only the behavioural automaton of each round to describe the behaviour relative to the current state, and requires the discovery of only a single atomic step in each round.

Note that the perspective of the commit and search approach is different from

the remaining implementation approaches. The number of solutions and the precomputed behaviour are minimum *requirements* of the commit and send approach, while these are *consequences* of the remaining implementation approaches. That is, the commit and send approach does not *implement* a one-solution approach, but *requires* only one solution to be calculated. In practice, the commit phase of the Dreams framework relies on the stepwise coordination model, which is an abstraction of another coordination model such as the connector colouring, constraint automata, or a set of constraints. The fact that the Dreams framework requires only one possible solution makes it suitable to use any of the other centralised approaches to implement $\mathcal{R}$eo as the underlying concrete coordination model of Dreams. As mentioned earlier, we have implemented the connector colouring and the constraint-based models as encodings of the stepwise coordination model.

Hybrid deployment – an ideal scenario. Centralised implementations have an advantage over distributed implementations with respect to their runtime coordination overhead. When it is feasible to precompute all behaviour alternatives a priori, avoiding the consensus phase at each round, this approach makes the execution of a static connector (disregarding reconfiguration) more efficient. For some scenarios, the lower overhead is more relevant than considerations for scalability or easy reconfiguration. This suggests that some complex scenarios may benefit from a *hybrid deployment* using multiple implementations. More specifically, some parts of a connector can be compiled using a centralised approach (e.g., constraint automata), each considered as an individual actor, which can be deployed in a different location. The Dreams framework can then perform the coordination among these connector parts.

Ideally, a connector would be partitioned automatically into its synchronous regions by an automatic splitting mechanism, and each region would be compiled separately to run as a single actor. This results in maximal concurrency among synchronous regions while avoiding costly synchronous communication across actors, which can reside in different physical locations. Note that although we call this an *ideal scenario*, the developer of a system may still desire to force a single synchronous region to be distributed across a network, for example, because of hardware requirements.

6.6 Related work

In this chapter we introduced Dreams, a new framework used to distribute connector-type coordination, and gave a general overview of its implementation. We assume the actor model as the basis for a reliable and asynchronous communication, and presented a small comparison of our distributed approach compared with other approaches to implement $\mathcal{R}$eo in §6.5.4. Now we take a wider perspec-

tive and consider other approaches for distributed coordination. In particular, we start by addressing the most popular formalisms used in the industry, and then look at some languages and tools that involve distributed coordination.

The industry standard for coordination of web services, the business process execution language (BPEL) [30], is a block-structured language that uses a centralised execution model, much like the automata-based implementations of $\mathcal{R}$eo. $\mathcal{R}$eo has also been used for the composition of web services. The mashup environment SABRE [79] built using $\mathcal{R}$eo, provides tools to combine, filter and transform web services and data sources like RSS and ATOM feeds.

Linda [54], one of the first coordination languages, provides a simple executable model consisting of a shared tuple space which components use to exchange values. The survey of Arbab and Papadopoulos [90], and the more recent classification of Arbab [10], classify Linda and its derivatives as *data-oriented* as opposed to $\mathcal{R}$eo's *dataflow-oriented* model. Individual tuple operations in Linda are atomic, but Linda lacks the arbitrary multiparty rendezvous communication that $\mathcal{R}$eo provides through synchrony. There have been many attempts to distribute Linda based on replication [24] or partitioning [27] of the tuple space. These implementations usually use some form of the two-phase commit (2pc) protocol to preserve the atomicity of operations. The current implementation of the Dreams framework is not as mature and optimised as these dedicated approaches for the asynchronous communication described by the shared tuple space mechanism. However, we believe that the Dreams framework can be used to efficiently describe and implement these systems, provided an intelligent use of the splitting of actors is in place to avoid synchronisation of actions when unnecessary.

The two phase commit protocol, as well as some of its variants, is well known in the context of fault tolerance in distributed systems. These models are usually based on a *centralised* coordinator that exchanges asynchronous messages with a set of participants. The protocol checks if all participants agree to perform some transaction, or if there is any that aborts, and communicates this decision back to the participants.

The *distributed two phase commit (d2pc)* protocol [32], introduced by Bruni *et al.* as an extension of the 2pc protocol, can be compared to the distributed agreement protocol in the Dreams framework, which we explain in detail in Chapter 7. Both approaches try to achieve a *global consensus*, which consists of a commit or abort in the case of the d2pc protocol and a description of a step in the case of Dreams. This work served as inspiration for the development of the Dreams framework. The d2pc protocol has been specified in the Join calculus coordination language [50], and differs from the original 2pc protocol in that each participant initially knows only a set of its neighbours, and the participants exchange their sets of known neighbours until they commit and know who the involved participants are, or until all of them abort. Baragati *et al*. developed a prototype application [25] for

two different platforms based on the d2pc protocol, using as a case study a rescue unit composed of a central base and several teams. Experiments in the Dreams framework involving real case studies have not been performed yet.

We now describe a more practical approach, taken by Minsky and Ungureanu, who developed the Law-Governed Interaction (LGI) mechanism [86], implemented by the Moses toolkit. This mechanism coordinates distributed heterogeneous agents, using a policy that enforces extensible laws. Agents execute events that are regulated by some controllers that enforce the laws. Laws are specified in a Prolog-like language, but as opposed to $\mathcal{R}$eo, they reflect local properties only and do not require non-local synchronisation. The authors emphasise the need to replace a centralised controller imposing the laws of the full system by certified controllers, one for each connection. As with Linda and the 2pc protocol, this reflects a need to decentralise coordination, which is the main concern of the Dreams framework.

The Dreams framework attempts to solve (coordination) constraints imposed on concurrent actors, which makes it natural to consider existing techniques for constraint solving in a distributed setting. The constraint-based approach has been successfully applied in the domain of $\mathcal{R}$eo, as described in Chapter 5. Yokoo addresses the distributed constraint satisfaction problem [104] typically by using partial solutions that are exchanged among different participants, and different types of search algorithms such as backtracking and hill climbing. This suggests that the use of a constraint-based approach to implement $\mathcal{R}$eo can open new possibilities for optimising the distributed execution of $\mathcal{R}$eo connectors.

6.7 Conclusions

In this chapter we introduced Dreams, a framework that provides a distributed implementation based on the actor model for coordination models that can be encoded with behavioural automata, and how it is used in the context of $\mathcal{R}$eo. We summarise below our analysis that shows how the Dreams framework achieves the goals we set out in the introduction.

Decoupling The basic building blocks of the Dreams framework are actors, which execute concurrently. Dreams takes advantage of this concurrency by identifying parts of the connector that can be executed independently. We call these parts synchronous regions. We propose a simple approach to exploit these synchronous regions, without computing them statically, yielding a true decoupling of execution.

Scalability As a consequence of decoupling the execution of the instances of stepwise models such as $\mathcal{R}$eo connectors, no global consensus is required. Furthermore, the behaviour is computed only for each step, avoiding the state explo-

sion resulting from computing all possible future behaviour. These two factors form the basis for a scalable implementation. The implementation of Dreams also allows different parts of a connector to execute across physical machine boundaries.

Reconfiguration The single-step semantics provides a very low deployment overhead, reducing the cost of reconfiguration. Furthermore, reconfiguring a connector affects only the synchronous regions that it modifies, while the rest of the connector continues to execute.

Currently we have a functioning implementation of the Dreams framework. We describe the details of the implementation of Dreams in Chapter 7 – Implementing Dreams. The distributed implementation benefits from some of the implementation optimisations offered by centralised schemes, specifically the compilation of a connector into the automata of its synchronous regions.

Chapter 7

Implementing Dreams

7.1 Introduction

The Dreams framework coordinates a set of components or services using a collection of independent executing entities which we call actors. Actors communicate with each other using asynchronous message passing, and their behaviour is described from two different perspectives. The *coordination behaviour* is given by the stepwise coordination model, and describes how data should be propagated between components in each round. The *reactive behaviour* describes how each actor sends and reacts to incoming messages.

In Chapter 6 we described the basic assumptions of the reactive behaviour, and we explained in detail how the coordination behaviour is used. In this chapter we present a more detailed definition of the reactive behaviour:

- by introducing a *syntax* for specifying the reactive behaviour in a variation of SAL (Simple Actor Language);
- by specifying a fixed *set of messages* that actors can send; and
- by defining how each actor *reacts* to each message.

An actor is initially idle, and can evolve in two different ways. It can either participate in a round in the coordination process, or it can enter a suspension phase where the actor can be reconfigured. Either way, it eventually returns to the idle phase. We depict these two scenarios in the simplified diagram in Figure 7.1. The messages exchanged between actors can represent requests for the behavioural automaton for the current round, requests for data, replies with a behavioural automaton or with data, steps for the current round with an optional data value, or suspension and reconfiguration messages.

Our implementation of Dreams is distributed in the sense that actors execute concurrently, and messages are sent asynchronously. Furthermore, the splitting

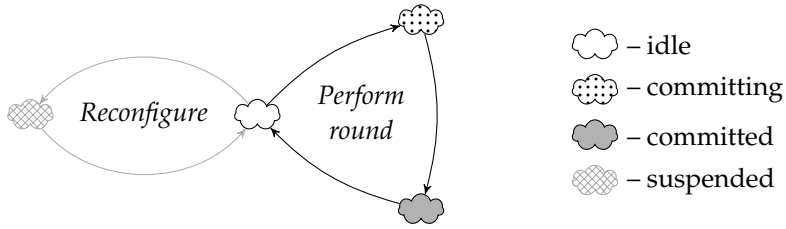


Figure 7.1: Two alternative scenarios for an actor in an idle phase: reconfigure, or perform a round.

of actors described in the previous chapter and our distributed algorithm exploit the idea that concurrent actors can be running in different machines, by allowing independent parts of the coordination layer to execute side by side. However, we do not model failure in our algorithm, which we leave for future work.

Organisation of the chapter This chapter starts by introducing the syntax used to represent actors with an informal description of how an actor evolves in §7.2. We then explain in §7.3 the phases of an actor's lifecycle in Dreams, and for each phase we describe what messages it can receive, and how it reacts to them. To exemplify the protocol induced by the reactive behaviour we show the traces of execution of simple $\mathcal{R}$eo connectors in §7.4, by means of diagrams based on message sequence charts. We compare the performance of Dreams with that of a centralised implementation of $\mathcal{R}$eo in §7.5. In §7.6 we present more details of the current implementation of Dreams, showing how a connector is deployed in a distributed network. We wrap up with conclusions and related work in §7.7.

7.2 Actor definition

We define the *reactive behaviour* as a variation of the Simple Actor Language (SAL) introduced by Agha and Thati [2]. Our variation ignores the creation or sending of actors and uses the pattern-matching process offered by the actor library implemented by Haller *et al.* for the Scala programming language [57]. The actor library of Scala is faithful to the actor model introduced in §6.2. Our Scala implementation of Dreams mimics the reactive behaviour using a syntax different than, yet similar to, the one that we use in this chapter. We give more details of the Scala implementation in §7.6.

An actor a is a triple $\langle A, AB, M\rangle$ of an actor identifier A, a description of the reactive behaviour AB and a queue M of messages sent to a that have not yet been consumed. We call this queue the *mailbox* of a. The only operations on a mailbox M are the sending of a message to M (enqueue) and the consumption of a message

(dequeue) given a list of patterns. The consumption of a message is specified by a list of *patterns* that can be unified with the received messages. When a received message matches a recognised pattern, a specific *command* is triggered. The list of patterns and actions does not need to be exhaustive, i.e., they do not need to capture all possible messages. An actor evolves by reacting to a message, which involves finding the first message in its mailbox that matches one of its current patterns, and executing the command associated with that pattern.

7.2.1. DEFINITION (REACTIVE BEHAVIOUR). The syntax of a reactive behaviour is described by AB in the following grammar.

$$
\begin{array}{rcl}
AB & ::= & BDef_1 \ \ldots \ BDef_n \ \ Com \\
Com & ::= & v := Exp;\ Com \\
 & | & \textbf{if}\ (Exp)\ \textbf{then}\ Com_1\ \textbf{else}\ Com_2 \\
 & | & \textbf{send}\ Exp\ \textbf{to}\ A;\ Com \\
 & | & \textbf{become}\ B(Exp_1, \ldots, Exp_n) \\
BDef & ::= & \textbf{def}\ B(x_1, \ldots, x_n)\ \{ \\
 & & \quad Pat_1 \Rightarrow Com_1; \\
 & & \quad \vdots \\
 & & \quad Pat_m \Rightarrow Com_m; \\
 & & \} \\
Pat & ::= & \texttt{Msg}(Pat_1, \ldots, Pat_n) \\
 & | & x
\end{array}
$$

Com represents a *command* performed by an actor, *BDef* represents a *behaviour definition*, *Pat* represents a *pattern*, *A* represents an *actor*, *Exp* represents an *expression*, *B* is an identifier of a behaviour definition, and x represents a *variable*. ◁

Expressions (*Exp*) Expressions consist of variables, constants, and operations over expressions. We assume the existence of the following data structures: booleans, integers, strings, sets, mappings, actor identifiers, behavioural automata, and transition labels. Booleans have the usual logic operators $\wedge$, $\vee$, and $\neg$. Integers have arithmetic operators $+$, $-$, $\times$, and $/$, and also comparison operators $>$, $<$, and $=$. Strings can also be compared via equality. Sets are defined using a comprehension set syntax $\{element \mid conditions\}$, and have the inclusion operator $\in$. Mappings are a particular case of sets, defined using the same set notation introduced in §3.2, and we write $m \backslash x$ where m is a mapping and $x \in \mathrm{dom}(m)$ to denote the mapping m after removing the entry that maps x to some value. A messages is a tuple consisting of a message identifier `Msg`, that can be regarded as a string, and a sequence of expressions $Exp_1, \ldots, Exp_n$, written as $\texttt{Msg}(Exp_1, \ldots, Exp_n)$. Messages can be compared via the equality operator, but also unified against patterns, as we will describe soon. Actor identifiers also have the equality operator defined.

Finally, behavioural automata and their transition labels have equality operators and specific operators to evolve a behavioural automaton or to select an label from the current state of a behavioural automaton. We will introduce the notation for these operations during the explanation of the distributed algorithm in §7.3. We do not present any type restrictions, although they are present in Scala's implementation.

Commands (*Com*) A command is given by the grammar above, and can take one of four different forms. The command "$v := Exp$; Com" represents the definition of a fresh variable v that can be used only inside the command Com following the attribution. The command "**if** (Exp) **then** Com_1 **else** Com_2" is a traditional if-then-else, where the expression Exp yields either *true* or *false*. The command "**send** Exp **to** A; Com" represents the enqueueing of a message Exp at the end of the mailbox of the actor with identifier A, followed by the evaluation of the command Com. Finally, a command typically finishes with "**become** $B(Exp_1, \ldots, Exp_n)$", where the actor assumes a new reactive behaviour, given by the definition B with actual parameters $Exp_1, \ldots, Exp_n$. Note that, syntactically, a command can be seen as a sequence of instructions terminating on a **become** or an **if-then-else** command (each of whose branches must eventually terminate with a **become**). As a consequence, the syntax ensures that actors never terminate.

Behaviour definitions (*BDef*) A definition of a reactive behaviour consists of an identifier B with a list of variables acting as formal parameters, and a body with a list of possible reactions to messages. The formal parameters are accessible only within the body. Each reaction has a pattern that can be unified with messages from the mailbox. An actor evolves by trying to unify the head of the mailbox with one of the patterns, in their order of appearance. If no pattern is is unified, then the next message in the mailbox is tested against the patterns until all messages are tested or a pattern unifies with a message. In the first case the actor waits for new messages, and in the second case it removes the message from the list and performs the command associated with the pattern, replacing the variables of the pattern with the unified values in the command. When a match is found, the executed command will end up in some "**become** $B(Exp_1, \ldots, Exp_n)$". After this, the actor will try to evaluate the head of the mailbox again, using the reactive behaviour defined by B.

Patterns (*Pat*) A pattern can be either a variable or a message identifier with a sequence of patterns. We say a message matches a pattern if the pattern is a variable, or if the message identifier is the same as the message identifier of the pattern, and the arguments of the message match the arguments of the pattern.

$$\text{(VAR)}\ \frac{}{\langle A\ ,\ BDef^*\ \ v := Exp; Com\ ,\ M\rangle\ ,\Gamma \rightarrow \langle A\ ,\ BDef^*\ \ Com[Exp/v]\ ,\ M\rangle\ ,\ \Gamma}$$

$$\text{(IFTHENELSE-1)}\ \frac{Exp \rightsquigarrow true}{\langle A\ ,\ BDef^*\ \ \textbf{if}\ (Exp)\ \textbf{then}\ Com_1\ \textbf{else}\ Com_2\ ,\ M\rangle\ ,\Gamma \rightarrow \langle A\ ,\ BDef^*\ \ Com_1\ ,\ M\rangle\ ,\ \Gamma}$$

$$\text{(IFTHENELSE-2)}\ \frac{Exp \rightsquigarrow false}{\langle A\ ,\ BDef^*\ \ \textbf{if}\ (Exp)\ \textbf{then}\ Com_1\ \textbf{else}\ Com_2\ ,\ M\rangle\ ,\Gamma \rightarrow \langle A\ ,\ BDef^*\ \ Com_2\ ,\ M\rangle\ ,\ \Gamma}$$

Table 7.1: Basic rules for the evolution of commands in a Dreams configuration, where $Exp \rightsquigarrow v$ represents the evaluation of an expression Exp into a value v.

$$\text{(SND)}\ \frac{}{\langle A\ ,\ BDef^*\ \ \textbf{send}\ Msg\ \textbf{to}\ A'; Com\ ,\ M\rangle\ ,\langle A'\ ,\ AB\ ,\ M'\rangle\ ,\ \Gamma \rightarrow \langle A\ ,\ BDef^*\ \ Com\ ,\ M\rangle\ ,\langle A'\ ,\ AB\ ,\ \mathsf{enqueue}(Msg, M')\rangle\ ,\ \Gamma}$$

$$\text{(RCV)}\ \frac{\textbf{def}\ B(x_1,\ldots,x_n)\ \ \{Pats\} \in BDef^* \qquad \sigma_1 = \{Exp_1/x_1,\ldots,Exp_n/x_n\} \qquad (Msg, Pat \Rightarrow Com, \sigma_2) \xleftarrow{Pats} M}{\langle A\ ,\ BDef^*\ \ \textbf{become}\ B(Exp_1,\ldots,Exp_n)\ ,\ M\rangle\ ,\ \Gamma \rightarrow \langle A\ ,\ BDef^*\ \ Com[\sigma_2,\sigma_1]\ ,\ M\backslash Msg\rangle\ ,\ \Gamma}$$

Table 7.2: Communication rules for the evolution of a Dreams configuration.

The unification of a message with a pattern that matches the message results in a substitution of variables with values in the command associated with the pattern.

Operational semantics We present a simple operational semantics for the reactive behaviour to clarify the narrative explanation. Dreams configuration consists of a collection of actors, where each actor is triple $\langle A, AB, M\rangle$, as explained in the beginning of this section. As a convention, we use Γ to denote a Dreams configuration and we write $BDefs^*$ as a shorthand for a finite sequence of behavioural definitions.

The evolution of a system of actors is described by the rules in Table 7.1 and Table 7.2. The basic rules in Table 7.1 reflect the traditional interpretation of the variable attribution and the if-then-else commands. The communication rules in Table 7.2 explain how the reactive behaviour uses the mailboxes of the actors.

We use the following notation in the communication rules in Table 7.2. The function $\mathsf{enqueue}(x, M)$ represents the mailbox M with the value x at the end of the queue; $M\backslash Msg$ denotes the mailbox M after removing the first occurrence of the message Msg; the variables σ_1 and σ_2 denote substitution of variables; and

$Com[\sigma_1, \sigma_2]$ represents the command *Com* after applying the substitutions σ_1 and then σ_2. Finally, $(Msg, Pat \Rightarrow Com, \sigma_2) \xleftarrow{Pats} M$ denotes a successful search for the first message in M that matches the a pattern from *Pats*. The result of the search is a triple, where *Msg* is the message found in M, $Pat \Rightarrow Com$ is the first pattern found in *Pats* that matches *Msg*, and σ is the substitution obtained by unifying *Msg* with *Pat*.

7.3 Distributed algorithm

We now describe the behaviour of each actor in the Dreams framework. The combined behaviour of all actors yields the evolution of Dreams as described in §6.3.3. Proactive actors initiate the algorithm by requesting the behavioural automata of their neighbours, which in turn send requests to their own neighbours. An actor replies to the request for its behavioural automaton once it has collected all information about its neighbours, which results in a breadth-first traversal of the graph of actors. The graph traversal is not managed by any global entity, and the existence of multiple proactive actors introduces some extra complications.

Recall from §6.2 that the underlying actor model assumes the existence of a unique name for each actor. The Dreams framework assumes also a *total order* among actors based on their unique names.[1] More specifically, it assumes a total order among proactive actors. We specify the order by introducing a unique integer value *rank* associated with each proactive actor, such that the decisions of a proactive actor with a higher *rank* prevails over decisions by an actor with a lower *rank*. We do not address which actors should have higher rank, nor do we exploit a scenario where the order changes during the evolution of an actor. After the initial request of each proactive actor is sent only the proactive actor with highest rank will get a reply with the behavioural automaton of the full system. At this stage the winning proactive actor chooses a step from the behavioural automaton and sends it to its neighbours, who propagate the data across the graph of actors according to the chosen step.

Figure 7.2 depicts the possible phases of an actor. Each phase represents a different reactive behaviour, i.e., it has its own behaviour definition, and reacts differently to received messages. The labels of the diagram depict the received messages, and the messages `Reply`, `AS` and `Data` are shorthands for a set of related messages. We will explain each of these messages later in this chapter. Note that each actor reacts deterministically to each received message, as it will be clear after describing the details of the distributed algorithm, although the simplifications in the diagram from Figure 7.2 suggest otherwise. An actor changes phase when it performs the command **become** $B(Exp_1, \ldots, Exp_n)$, where B is the new behaviour

[1] One can, for instance, imagine the rank of an actor to be the binary representation of the unique string *Name* of the actor, interpreted as an integer.

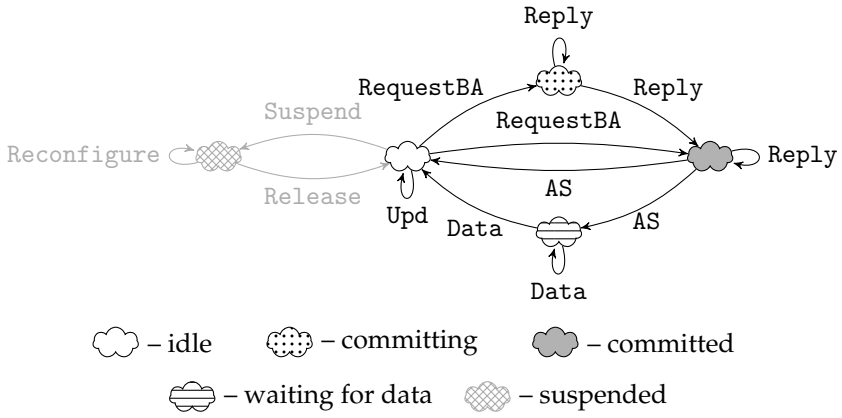


Figure 7.2: Simplified diagram with the transitions between the phases of an actor triggered by received messages.

definition corresponding to the new phase. The intuition is that each actor can be in one of five different phases: idle (), committing (), committed (), waiting for data (), or suspended ().

An idle actor can receive a request for its behavioural automaton and become committing or committed, receive a suspend request and become suspended, or receive an update message from a split actor and update the state of its coordination behaviour. Only split actors can receive these update messages, as explained in §7.3.2. Each request includes the *rank* of the proactive actor responsible for the original request, which we use to resolve race conditions.

A committing actor is an actor that after forwarding the request for a behavioural automaton to all of its neighbours waits for their replies, after which it evolves to the committed phase.

In the committed phase, an actor waits for a transition of the behavioural automata, with an associated step, plus some optional data. If the actor needs more data, it will evolve to the state 'waiting for data' until it collects all needed data.

A suspended actor can receive messages that change its coordination behaviour or even its connections to neighbour actors. We only briefly mention the behaviour of a suspended actor because the focus of this thesis is on the synchronous coordination of components or services, and not on the reconfiguration of connectors. To reason about reconfiguration, we would need to relax the assumption over the reactive behaviour that states that actors cannot be created or sent via messages. We leave the study of techniques to express and apply reconfigurations as future work.

7.3.1 Actor phases

We explain the behaviour definition of each phase individually. In each phase we describe the messages that can be sent and received by an actor in that phase. Note that the Dreams framework utilises a fixed set of messages. We now introduce some conventions that simplify our presentation. For each actor we assume the existence of two variables each with a fixed value, that can be easily modelled by including them as formal parameters of all *BDef*. These variables are *neighbours*, consisting of a set of connected actors, and *ba*, containing the behavioural automaton of the actor. We also assume an extra variable *sender* that refers to the actor that sent the message being processed. Finally, we use the special name INIT to denote a command that initialises the actor. Every actor starts in the *Suspended* phase with a message that triggers the execution of the INIT command. The initialisation of an actor generally consists of the command become *Idle*(), where *Idle*() is defined below. For proactive actors, the initialisation also includes the sending of requests for the behavioural automata that start the distributed algorithm, which we present in §7.3.3.

Idle – 

We define the behaviour of an actor in Dreams in the idle phase as follows:

```
def Idle() {
  Suspend(lock)     ⇒ become Suspended({lock});
  RequestBA(rank) ⇒ children := neighbours \ {sender} ;
    if (children = ∅)
    then send BA(ba) to sender;
         become Committed()
    else sendRequests(rank, children);
         invited = {a ↦ rank | a ∈ children} ;
         become Committing(rank, sender, invited, ba);
  Upd(s)            ⇒ ...
  Admin(action) ⇒ execute(action); become Idle()
  otherwise         ⇒ become Idle();
}
```

To avoid cluttering the code with unnecessary detail we use a sans serif typeface to denote macros that capture sequences of commands that we explain only informally. For such detail, the corresponding code of the implementation is available online.[2]

[2]Available at http://reo.project.cwi.nl/cgi-bin/trac.cgi/reo/browser/reo-engine/cwi.reo.engine/src/cwi/reo/engine/redrum/Primitive.scala.

When an actor executes the reactive behaviour *Idle*() it always consumes the first message in the mailbox, because every message can match one of the patterns. In particular, the last pattern *otherwise* is just a variable that can be unified with any message. This pattern simply discards the message, removing it from the mailbox. For the other patterns, we have four possible cases.

1. The received message is `Suspend`(*lock*), where *lock* is a string that identifies who requested the suspension. In this case the actor changes its reactive behaviour to *Suspended*. The only argument of the new phase is the set of locks that the actor waits on their release before it becomes *Idle* again, in this case {*lock*};

2. The received message is `RequestBA`(*rank*), where *rank* is an integer representing the ranking of the proactive actor that initiated the request. In this case the actor tries to collect the behavioural automaton of all of its connected actors, contained in the variable *children*. When the actor is not connected to any other actor, it immediately replies to the requester with its own behavioural automaton, and becomes *Committed*. Otherwise it forwards the request to all of its children and changes its reactive behaviour to the *Committing* phase, where it waits for the replies from its children. sendRequests(*rank*, *children*) is a macro for a sequence of

   send `RequestBA`(*rank*) `to` *a*

   commands, for all $a \in children$. The arguments of the new *Committing* phase are, respectively: the rank of the requester; the actor to whom to reply after collecting all behaviour; a set of invited neighbours to return their behavioural automaton, each associated with the rank used in the sending of the request; and the behavioural automaton obtained from the replies of the children (initially the behavioural automata *ba* of the actor). Note the redundancy of adding the value *rank* both as an argument of *Committing* as well as associating it with each invited actor. We will clarify this when we describe the *Committing* phase, but the intuition is to allow the actor to update its rank parameter without sending new requests for a behavioural automaton with the new rank, or send new requests with a new rank to just some of the invited actors.

3. The `Upd`(s) message can be received only by a split actor, and represents a state update of the behavioural automaton by the step s of the receiver's split actor. We leave this command undefined for now, and will return to it in §7.3.2.

4. The `Admin`(*action*) messages represent administration operations, and are present for the sake of completeness with respect to the existing implementation. Possible actions include monitoring operations of the actor, such as

a request for the current phase, or even a request to force the termination of the actor. We do not explain these actions, and we use the faded colour to show that these actions are present but not interesting for the purpose of the distributed algorithm.

Recall the diagram in Figure 7.2. The label `Reply` denotes a group of two possible messages, `BA` and `Busy`. The message `BA` is used in the definition of *Idle*(), and contains the behavioural automaton requested by the *sender*. The message `Busy` is used when the actor already received a previous request for its behavioural automaton, which we will see when describing the *Committing* phase.

Committing – 

An idle actor with more than one neighbour that receives a request for its behaviour changes its reactive behaviour to *Committing*. We define the behaviour of an actor in Dreams in a committing phase as follows:

```
def Committing(rank, root, invited, tempBA) {
  RequestBA(newrank)     ⇒ processRequest(newrank);
  StrongerReq(newrank)   ⇒ processRequest(newrank);
  BA(b)                  ⇒ updCommitting(tempBA ⋈ b);
  Busy()                 ⇒ updCommitting(tempBA);
  Admin(action)          ⇒ execute(action);
                           become Committing(rank, root,
                                  invited, tempBA);
}
```

The *Committing* phase is parameterised by four arguments. The variable *rank* represents the rank of the proactive actor that initiated the request for a behavioural automaton; *root* is the neighbour to whom the actor should reply a behavioural automaton, *invited* is a mapping from neighbours to ranks, consisting of the neighbours to whom a request for their behavioural automata was sent but from who no reply has yet been received, and the rank used in that request; and *tempBA* is a temporary behavioural automaton that combines all behavioural automaton already received from the neighbours. Note that, in contrast to the definition of the *Idle* phase, the list of patterns is not complete, because not all incoming messages can be unified with one of the patterns. This means that incoming messages that cannot be unified will remain in the mailbox of the actor until the reactive behaviour changes into a phase where they can be reacted to. Besides the administrative messages, we have two kinds of messages to react to: requests for behavioural automata or replies to these requests. We proceed by explaining each of these cases.

Requests The Dreams framework recognises two different messages for sending requests, `RequestBA` and `StrongerReq`, which are not distinguished in the definition of *Committing*. We will explain this distinction when describing the reactive behaviour of the *Committed* phase. The processRequest macro executes the following commands in sequence:

1. If $newrank = rank$ then the actor starts by performing the command

   $$\mathsf{updCommitting}(tempBA).$$

   Recall that $invited(a) = i$ represents that a request to the actor a was sent with rank i, but a reply has not yet been processed. Note also that $sender \in \mathrm{dom}(invited)$, which we explain only informally. The mapping *invited* contains all the neighbours except the *root*, who sent the request responsible for the actor becoming *Committing*, and the neighbours that already replied to the actor, who cannot have sent another request in the current round. If $invited(sender) < newrank$, then the actor also performs the following command before the updCommitting command:

   send `Busy` **to** *sender*.

2. If $invited(sender) \leq newrank < rank$ then the actor sends a stronger request with the command

   send `StrongerReq`$(rank)$ **to** *sender*.

3. If $newrank < invited(sender) \leq rank$ then the message is consumed and discarded, because the actor has already sent a request with a rank greater than *newrank* to *sender*.

4. If $newrank > rank$ then the value of *root* of the *Committing* phase changes to the *sender*. The *root* is the neighbour to who the actor has to reply a behavioural automata. A naive approach is for the actor to send a stronger request for their behavioural automata to all its neighbours. Instead of sending new invitations we simply update the value of *rank*, and postpone the sending of stronger invitations until a reply is received from the invited actors, that may or may not trigger the send of the stronger request. This behaviour is achieved by executing the macro changeRoot, defined below.

   changeRoot =
   send `RequestBA`$(newrank)$ **to** *root*;
   $invited' := (invited \setminus \{sender\}) \cup \{root \mapsto newrank\}$;
   become $Committing(newrank, sender, invited', tempBA)$.

Replies The possible replies for requests of behavioural automata consist of BA(b) and Busy messages. The first message is received when a neighbour replies with a behavioural automaton b, and the second when a neighbour decides it does not need to return a behavioural automaton, since it will do so to another root. The actor evolves by removing the *sender* from the *invited* list, executing the macro updCommitting(b'), where b' is the combined behavioural automaton of the invited neighbours. In order to accomodate proative actors, a slightly more complex version of this macro (page 191) is necessary. Ignoring this, for now we define updCommitting as follows.

updCommitting(b') =
  $invited' := invited \setminus \{sender\}$;
  if ($invited' \neq \emptyset$)
  then become *Committing*($rank, root, invited', b'$)
  else send BA(b') **to** *sender*;
    become *Committed*()

Note that some of the received RequestBA messages are also interpreted as replies, when the rank of the request is the same as the rank of the committing actor, as shown before.

Committed – 

A *Committed* actor has already sent its own behavioural automaton in a reply, together with the behavioural automaton of its other neighbours, and is waiting for a step for the current round. The step can arrive from any of its neighbours, and not only from the root of the *Committing* phase.

def *Committed*() {
  StrongerReq($rank$) ⇒ **send** Busy **to** *sender*;
    become *Committed*();
  AS(s) ⇒ processAS(s);
  AS+Req(s) ⇒ processAS+req(s);
  AS+Data(s, d) ⇒ processAS+data(s, d);
  Data(d) ⇒ **become** *Committed*();
  Admin(*action*) ⇒ execute(*action*);
    become *Committed*();
}

We distinguish between the RequestBA and StrongerReq messages in the definition of the *Committed* phase. Whenever a StrongerReq message is received, the actor replies with a Busy message, and remains *Committed*. A *Committed* actor cannot change its previously replied behavioural automaton. The roots of the current

actor and the *sender* will also eventually intersect, because the `StrongerReq` message is only sent when invitations from more than one neighbour have the same rank, that is, the *sender* was previously in a *Committing* phase with rank value *rank*.

The incoming `RequestBA` messages do not match any of the patterns in this phase, and are therefore kept in the mailbox of the actor until they can be processed (in the *Idle* phase).

Apart from the `StrongerReq` messages and administrative messages, a *Committed* actor reacts only to messages with a step s, denoted in Figure 7.2 simply as `AS`. We admit three different forms for the messages represented in this diagram as `AS`: $\texttt{AS}(s)$, $\texttt{AS+Req}(s)$ and $\texttt{AS+Data}(s, d)$. The message $\texttt{AS}(s)$ is received when there is no data communication between the *sender* and the *Committed* actor. The message $\texttt{AS+Req}(s)$ is received when the *sender* provides the step for the current round, but it stays in the phase '*waiting for data*', to whom the *Committed* actor should provide a data value. Finally, the message $\texttt{AS+Data}(s, d)$ is received when there is a data value d sent from the *sender* to the *Committed* actor.

Steps as groups of labels As discussed in §3.5, the step s consists of a group γ_s of labels of transitions from the global behavioural automata b, whose atomic steps differ only in the data-values flowing through the ports. We recall below the properties of γ_s. This approach allows a label to be chosen without deciding a priori which data values will be transferred. Let $b = \langle Q, \mathsf{L}[P], \rightarrow, \mathcal{O} \rangle$. The group γ_s is indexed by an assignment of data of the input ports, that is, the type of γ_s is

$$\gamma_s : (IP \to \mathbb{D}) \to \mathsf{L}[P].$$

The conditions over γ_s are specified as follows.

$$\forall m : IP \to \mathbb{D} \;\cdot\; \alpha(\gamma_s(m)) = \langle P, F, IP, OP, data_m \rangle \,, \\ \text{for some fixed } F, IP, OP, \text{ and where } data_m \frown m$$

Recall that $m_1 \frown m_2$ if the values of m_1 and m_2 coincide for the common domain of both maps, and α is a global function that maps each label $\ell \in \mathsf{L}$ of transitions of behavioural automata into an atomic step.

Let $b = b_1 \bowtie \cdots \bowtie b_n$, where each b_i is the behavioural automata of each of the actors involved in the current round, and let $b_i = \langle Q_i, \mathsf{L}[P_i], \rightarrow_i, \mathcal{O}_i \rangle$. As mentioned in Chapter 3, the group γ_s can be projected into each behavioural automaton b_i, denoted by $\gamma_s[b_i]$, whose labels have atomic steps restricted to the set of ports P_i. In the rest of this chapter we write $data_s[m](x) = d$ if there exists a data value d such that, for all data functions $data_s$ of the atomic steps associated to labels in $\gamma_s[b_i]$, $data_s(x) = d$. Furthermore, we refer to such groups of labels simply as 'steps'.

Processing of a step We now describe the processAS+data macro, and explain how the macros processAS and processAS+req differ from it. The processAS+data macro analyses the projection $\gamma_s[b]$ and deduces the neighbours from which a data value should be requested, the neighbours to which a data value should be sent, and the neighbours which require only the step s. The details of this macro follow below. The macro describes the sending of the appropriate messages to the respective neighbours.

1. Define $\gamma_s[b]$ to be the projection of s for the behavioural automata b, and P, IP, and OP to be the sets of known, input, and output ports, respectively, from the atomic steps of the labels in $range(\gamma_s[b])$. Also define $rcvd := \{sender \mapsto d\}$, which is a mapping of the data values received so far.

2. Send the step s to all the neighbours whose ports are neither in IP nor in OP. Note that each neighbour is represented in the actor by the port that is shared with the neighbour. For all $x \in P \setminus (IP \cup OP)$ perform the command:

$$\textbf{send } \texttt{AS}(s) \textbf{ to } x.$$

3. Send the step s with a request for data to all ports in $ip \in IP \setminus \{sender\}$, by performing the following command:

$$\textbf{send } \texttt{AS+Req}(s) \textbf{ to } ip.$$

4. Send the step s to all output ports whose data value is already known. Recall that $data_s[m](x) = d$ if there exists a data value $d \in \mathbb{D}$ such that $data_s(x) = d$, for all data functions $data_s \in range(\gamma_s[b])$. Then, for all ports $op \in OP$ such that there exists a data value $d \in \mathbb{D}$ where $data_s[rcvd](op) = d$, we perform the command:

$$\textbf{send } \texttt{AS+Data}(s, d) \textbf{ to } op.$$

5. Define $R = IP \setminus \mathrm{dom}(rcvd)$ to be the set of the remaining input ports whose data values are unknown. If $R = \emptyset$, then $IP \subseteq \mathrm{dom}(rcvd)$ and $\gamma_s[rcvd] = \{s_b\}$, because of the properties of $\gamma_s[b]$ described above, where s_b is the step that b will perform. Consequently, the actor updates the state of its behavioural automaton and becomes idle. If $R \neq \emptyset$, then the actor waits for the replies from actors in IP. This is achieved by the command:

$$\begin{array}{l}\textbf{if } (R = \emptyset)\\ \textbf{then } \texttt{updBA}(\gamma[rcvd])\\ \textbf{else become } \mathit{WaitingData}(R, \emptyset, rcvd, s)\end{array}$$

   The macro $\texttt{updBA}(\gamma[rcvd])$ updates the behavioural automaton b of the actor, evolving b by the step s_b. After updating the behaviour, updBA initialises the

actor by performing the command INIT, described in the beginning of this section, where the actor returns to the *Idle* phase. The new *WaitingData* phase is parameterised on the neighbours that still need to send a data value, the mapping *rcvd* of received data values, and the step s for the current round. We will explain the arguments in more detail below.

The macros processAS and processAS+req are analogues to processAS+data. In the first case the mapping *rcvd* is empty, and in the second case we use $\{sender\}$ instead of the empty set as the second argument of the behaviour definition *WaitingData*.

Waiting for data – 

An actor moves to the phase *WaitingData* when it receives the step s to be performed in the current round, but it still needs data from some of its neighbours to perform the required communication. The reactive behaviour is defined as follows.

def *WaitingData*$(sendAS, sendData, rcvd, s)$ {
  Data(d) $\Rightarrow$ updData(d);
  AS+Data(d) $\Rightarrow$ updData(d);
  AS$(_)$ $\Rightarrow$ **become** *WaitingData*$(sendAS, sendData, rcvd, s)$;
  AS+Req(s) $\Rightarrow$ $sendAS' := sendAS' \setminus \{sender\}$;
  $sendData' := sendData' \cup \{sender\}$;
  become *WaitingData*$(sendAS', sendData', rcvd, s)$;
  Admin$(action)$ $\Rightarrow$ execute$(action)$;
  become *WaitingData*$(sendAS, sendData, rcvd, s)$;
}

WaitingData is parameterised by four arguments. The variable *sendAS* is a set with the neighbours that still need to send a data value, *sendData* is a set of neighbours that already know the step s but still need a data value from this actor, *rcvd* is a mapping from the neighbours that already sent a data value to their corresponding data values, and s is the step for the current round. The macro updData(d) is unfolded to the following command. Note that the variables *sendAS*, *sendData*, and *rcvd* are in the scope of this macro, and used by tryToSend below.

updData(d) =
  $rcvd' := rcvd \cup \{sender \mapsto d\}$;
  tryToSend$(d, rcvd')$;
  if $(sendAS' = \emptyset \wedge SendData' = \emptyset)$
  then updBA$(s[b][rcvd'])$
  else become *WaitingData*$(sendAS', sendData', rcvd', s)$

The macro updData(d) starts by storing the information about the data values already received in the variable $rcvd'$, and the macro tryToSend($d, rcvd'$) checks if the data for any of the actors in *sendAS* or *sendData* is available after receiving d. The macro tryToSend sends the message `AS+Data(s,d')` to every actor $a \in sendAS$ such that there is a value $d' \in \mathbb{D}$ where $data_s[rcvd'](a) = d'$ with respect to the projection $\gamma_s[b]$, and similarly it sends the message Data(d') for every actor in *sendData* with an associated data value d'. The variables $sendAS'$ and $sendData'$ are defined to be the updated variables of *sendAS* and *sendData* after removing the actors that were replied to. The last if-then-else checks if there are still actors that are waiting for data. If not, the actor evolves it's behavioural automaton and initialises again by performing the macro updBA, described for the *Committed* phase. Otherwise, the parameters of *WaitingData* are updated. Note that $\gamma_s[b][rcvd']$ will always be a singleton set because $sendAS' = \emptyset$ and $SendData' = \emptyset$, which imply that all the input ports in *IP* are completely defined by $rcvd'$.

When the message `AS` is received the actor consumes and discards message with the step, because the step for the current round is already known. Finally, when a message `AS+Req` is received the actor updates the values of *sendAS* and *sendData*, since the *sender* does not need the step anymore, but only its data value.

Suspended – 

Every actor is created in the *Suspended* phase, with a message `Release()` in the mailbox to release this suspension. Therefore the definition of *Suspended*(*locks*) presented below plays two roles: (1) initialise the reactive behaviour, and (2) allow the reconfiguration of the actor without interfering with the coordination behaviour. The initialisation is achieved by executing INIT, introduced in the beginning of §7.3.1. We incorporate these two roles into the same reactive behaviour because every time an actor stops being suspended it has to be considered to be a new actor, since it could have been reconfigured.

An idle actor becomes suspended by receiving the Suspend(*lock*) message. Its reactive behaviour is described by *Suspended*(*lock*), defined below. The variable *lock* identifies who requested the suspension, to permit the suspension of an actor from independent entities. We do not explore the reconfiguration aspect in this thesis, and simply use a general `Reconfigure` message parameterised on an instruction *inst*. Reconfiguration plays an orthogonal role to the communication of data, and can be dealt with independently.

```
def Suspended(locks) {
  Suspend(lock)      ⇒ become Suspended(locks ∪ {lock});
  Reconfigure(inst)  ⇛ reconfigure(inst);
  Release(lock)      ⇒ locks′ := locks\lock;
                       if (locks′ = ∅)
                       then INIT
                       else become Suspended(locks′)
  Admin(action)      ⇒ execute(action);
                       become Suspended(locks′);
}
```

The code is self-explanatory. The phase parameters are always updated to store which locks were sent, and which locks were released. reconfigure is an operation that cannot be captured by our reactive behaviour definition. A reconfiguration typically requires creation of actors or sending and receiving of actors, in order to change how an actor is connected to its neighbours.

7.3.2 Split actors –

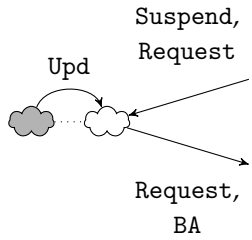


Figure 7.3: Messages used by split actors in the *Idle* phase.

Recall from §6.4 that split actors communicate the state changes of their behavioural automata between themselves at the level of the reactive behaviour. We make this concrete by defining the reactive behaviour of split actors. The diagram in Figure 7.3 illustrates the possible communication from and to a split actor in the *Idle* phase. A split actor in the *Committed* phase sends an Upd(s) message to its connected split actor when it updates the state of its behavioural automaton. This message is received only during the *Idle* phase of the target split actor. Each split actor can also exchange other types of messages with its connected actors, as described before. We define the behaviour of split actors by redefining the macro updBA(s), and explaining the command associated to the Upd message in the *Idle* phase.

1. Let a_2 be the connected split actor of the current split actor. The macro updBA(s) described in §7.3.1 for the *Committed* case evolves the state of the behavioural automaton by performing the step s, and becomes *Idle*. We redefine this macro to perform the same state update, followed by the command:

```
updBA(s) =
    ...
    if (changedBA)
    then send Upd(s) to a2; INIT
    else INIT
```

   assuming that the macro changedBA yields true if the step s produced a state change.

2. An actor in the *Idle* phase can receive an Upd(s) message from the twin split actor. We left the details of the handling of this message undefined during the presentation of the *Idle* phase in §7.3.1. The definition of *Idle* becomes complete by associating the following command to the pattern Upd(s).

```
def Idle() {
  ...
  Upd(s)  ⇒ processUpd(s); become Idle();
  ...
}
```

   The macro processUpd(s) performs the state update appropriate for a step s performed by the connected split actor a_2. Note that this state update is deterministic because of the soundness and completeness criteria defined in §6.4.2.

7.3.3 Proactive actors – ☁

We now define *how the algorithm starts*. The distributed algorithm starts when a proactive actor has a proactive step, and consequently it sends a Request(*rank*) message, where *rank* is its own rank. This was briefly described in §6.3.1, and is now made precise by defining its corresponding reactive behaviour. We assume each proactive actor has a *rank* variable defined, and a macro hasActiveStep that returns true if and only if there is a proactive step from the current state of the actor's behavioural automaton.

The initialisation of a proactive actor differs from the initialisation of non-proactive actors. Therefore, we redefine the INIT command as follows.

$$
\begin{array}{l}
\mathsf{INIT} = \\
\quad \textbf{if } (\mathsf{hasActiveStep}) \\
\quad \textbf{then } \mathsf{sendRequests}(\mathit{rank}, \mathit{neighbours}); \\
\qquad\quad \mathit{invited} := \{a \mapsto \mathit{rank} \mid a \in \mathit{neighbours}\}\,; \\
\qquad\quad \textbf{become } \mathit{Committing}(\mathit{rank}, *, \mathit{invited}, \mathbf{0}_{\mathsf{BA}}) \\
\quad \textbf{else } \textbf{become } \mathit{Idle}()
\end{array}
$$

Note the absence of the root in the committing phase, denoted by the asterisk $*$, and the use of the identity element $\mathbf{0}_{\mathsf{BA}}$ for the composition of behavioural automata. We redefine the macro $\mathsf{updCommitting}(ba)$ (page 184) to accommodate the absence of a root. The only difference with the original definition of $\mathsf{updCommitting}(b')$ is the underlined part.

$$
\begin{array}{l}
\mathsf{updCommitting}(b', \underline{\mathit{rank}}) = \\
\quad \mathit{invited}' := \mathit{invited} \setminus \{\mathit{sender}\}\,; \\
\quad \textbf{if } (\mathit{invited}' \neq \emptyset) \\
\quad \textbf{then become } \mathit{Committing}(\underline{\mathit{rank}}, \mathit{root}, \mathit{invited}', b') \\
\quad \textbf{else } \underline{\mathsf{processAS}(\mathsf{selectAS}(\mathit{ba}))}
\end{array}
$$

The last line expresses that when a proactive actor receives all replies for the requests of behavioural automata, it choses the step to be performed in the current round, and sends it to its neighbours. The macro processAS is described in the explanation of the *Committed* phase (page 186), and manages which actors still need to receive data and/or the step of the current round. We assume that the selection of a step from a behavioural automaton b is performed by the macro $\mathsf{selectAS}(b)$.

We parameterise the updCommitting macro on the rank that the actor will have. We introduce this parameter because we need to use it when redefining the macro changeRoot, defined in the explanation of the *Committing* phase (page 183). Since there is no root, changing the root is equivalent to updating the *Committing* phase, as shown below.

$$
\begin{array}{l}
\mathsf{changeRoot} = \\
\quad \mathsf{updCommitting}(\mathit{ba}, \mathit{newrank})
\end{array}
$$

The redefined changeRoot macro performs the command updCommitting instead of forwarding the value of *newrank* to the previous root, which does not exist. In the original definition of changeRoot the actor remained in a *Committing* phase, but because of the absence of a root there might be no more invited neighbours left. This case is considered by the macro updCommitting. After the execution of the command updCommitting the actor will either become *Committing* with the *newrank*, become *WaitingData*, or initialise again via the command INIT.

7.4 Distributed $\mathcal{R}$eo

We exemplify the distributed algorithm using the coordination behaviour specified by the $\mathcal{R}$eo coordination language. We show two different connectors, each attached to exactly one writer and one reader. The first connector has only synchronous and stateless channels, and the topology of the connector forms a loop. The second connector consists of a FIFO_1 channel, which is split into two different actors.

7.4.1 Example: a reliable LossySync

We start by presenting a simple example where a writer is connected to a reader by a LossySync channel. We also attach a SyncDrain channel to the two ends of the LossySync, so that the coordination behaviour of the connector coincides with the coordination behaviour of the Sync channel. We call this connector a *reliable LossySync*, which is depicted in Figure 7.4. The goal of this example is to show how messages are communicated across a small Dreams configuration that has loops.

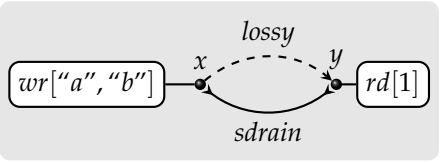


Figure 7.4: Reliable LossySync connector attached to a writer *wr* ready to send two elements, and a reader *rd* ready to consume one element.

We define the writer as a proactive actor with rank 1 and with two data values ready to be sent: a string "*a*" and a string "*b*". The reader is a proactive actor with rank 2 ready to consume at most one data value. The rank value is used when requests from the two actors 'collide', which is resolved by giving preference to the higher rank value. Upon creation, all actors are in the *Suspended* phase with their corresponding release messages in their mailboxes. In the diagram in Figure 7.5 we depict a possible execution of the reliable LossySync connector. Note that the behaviour of the connector is deterministic: initially the data value "*a*" has to be sent from the writer to the reader, and after that no other transition can be taken. However, the speed of sending and receiving messages is not deterministic, therefore the evolution of the global system is not deterministic.

In the diagram we use the following conventions. The vertical dotted lines represent the life line of each actor, i.e., the passage of time for each actor. The rectangles under the actor names and overlapping the dotted lines reflect the existence of messages in the mailboxes of the actors that have not yet been consumed.

Arrows between life lines of actors represent the sending of a messages to the mailboxes of other actors, but not their consumption, i.e., not the matching of the message to a pattern. The clouds at the ends of the life lines represent the phases of the reactive behaviours at the end of the trace. In this case, all actors are committing with the exception of the reader who is idle. Furthermore, only the actor y has messages in its mailbox at the end of the trace, depicted by drawing the rectangle until the cloud on the bottom.

Some comments about the trace of the execution of the reliable LossySync in Figure 7.5 are now in order. Initially all actors are suspended with a `Release` message in each of their mailboxes, and they are initialised via the INIT command once the message is received. The trace depicted in the diagram starts after the `Release` message is consumed. In the beginning, only the writer and the reader send a request each, as a result of being released from their suspension, become *Committing*, and all of the other actors become *Idle*. Consider now the top rectangle on the life line of actor x. Observe that three requests for a behavioural automaton arrive in the mailbox of actor x before it sends a message. The messages are processed in the order they were sent, and as a consequence x starts to propagate the first (weaker) request to all of its neighbours. Because of the delay in sending the request, we assume x sends a request for a behavioural automaton to *lossy* even after *lossy* sent a (stronger) request, and the same happens between x and *sdrain*. Once x receives the request from *lossy* it sends a new request to the writer, but it does not send any request to *sdrain* since it is still waiting for the reply to its initial (weaker) request. Also when the `Req`(2) message from *lossy* is received x replies by sending a `Busy` message to *lossy*. When *wr* receives `Req`(2) it becomes committed and replies by sending its behavioural automaton to x. Reacting to this message, x replies with its behavioural automaton together with *wr*'s behavioural automaton to *lossy*, and *lossy* replies with the received behavioural automaton and its own behavioural automaton to y. Meanwhile, *sdrain* also replies with its behavioural automaton to y, and y sends the composition of the received behavioural automata and its own behavioural automaton to *rd*. Finally, *rd* receives the reply to its initial request with the behavioural automaton of the rest of the connector. At this point the reader, *rd*, composes the received behavioural automaton with its own behavioural automaton, selects a step s_1 from this automaton, sends the step together with a request for data, and changes its phase to *WaitingData*. Note that once the writer receives the step s_1 it sends the data value, followed by a request for a behavioural automaton for the next round. Therefore, the writer is already committing, while the *sdrain* is still committed and waiting for the step of the previous round. This is not problematic because the step will always be sent to *sdrain* before the request for the new behavioural automaton.

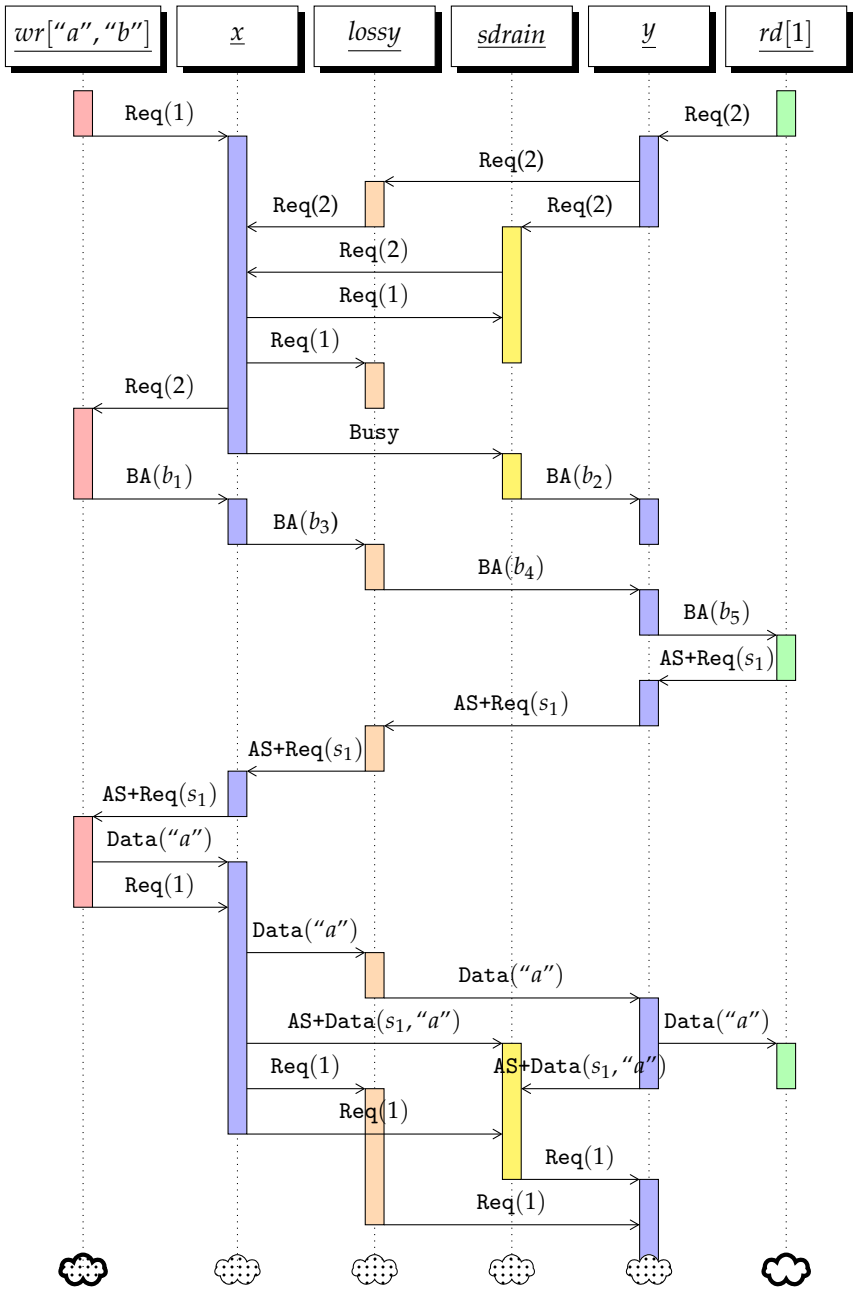


Figure 7.5: Trace of the execution of a reliable LossySync connector in Dreams, attached to a writer of rank 1 and a reader of rank 2.

7.4.2 Example: a split FIFO$_1$

In our second example we connect a writer *wr*, a FIFO$_1$ *fifo*, and a reader *rd*, as depicted in Figure 7.6. When encoding the connector in Dreams, we split the FIFO$_1$ channel into two actors. The split was described in §6.5.1, and we name the actor attached to *x* *fifo-in*, and the actor attached to *y* *fifo-out*. The split increases the concurrency and avoids the requests for behavioural automata to propagate between these two actors, which restricts the number of behavioural automata that are composed in each round.

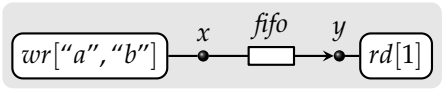


Figure 7.6: Empty FIFO$_1$ channel *fifo* attached to a writer *wr* ready to send two elements, and a reader *rd* ready to consume one element.

We depict a possible trace of the execution of the FIFO$_1$ connector in Figure 7.7. We omit the nodes *x* and *y* because these actors mainly forward messages and do not play a relevant role in the trace, leaving the actors *wr*, *fifo-in*, *fifo-out*, and *rd*. All these actors are proactive actors, because their behavioural automata contain proactive steps in their transition relations. Recall that proactive steps have dataflow on ports that do not depend on any other port. Initially all actors have proactive ports for the current state of their behavioural automata with the exception of *fifo-out*, since an empty FIFO$_1$ cannot output data. Until the sending of the $\texttt{Upd}(s_1)$ message two independent lines of communication can be observed. The actors *wr* and *fifo-in* exchange messages until they succeed in sending a data value "*a*" from *wr* to *fifo-in*. At the same time, the reader sends a request but there is no data available, so the actors *fifo-out* and *rd* become idle. After the sending of the message $\texttt{Upd}(s_1)$ the situation is inverted. The *fifo-in* replies to a request for its behavioural automaton, but the state of its behavioural automaton is full and it cannot receive data. The *fifo-out* sends a request and manages to send the data value "*a*" to the reader. The actors for *fifo-out* and *rd* become idle, because neither one is ready to send or receive data, and therefore no proactive steps are available. An $\texttt{Upd}(s_4)$ is then sent to the *fifo-in* actor, which ends up receiving the data value "*b*". In the end of the trace, the *fifo-out* actor still has an active step (the buffer is full), but the step s_5 does not have any dataflow, therefore *fifo-out* refrains from sending a new request, because there was no state change.

7.4.3 Dreams vs. d2pc

We explored the most relevant work related to the Dreams framework in §6.6, including the distributed two phase commit (d2pc) protocol, introduced and stud-

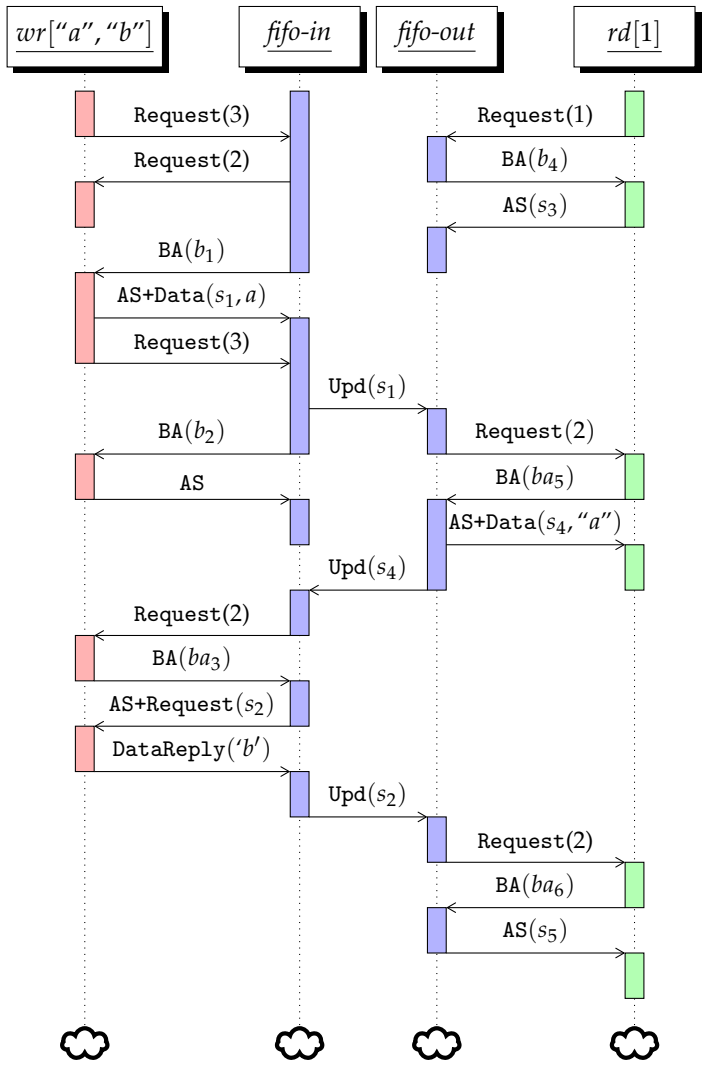


Figure 7.7: Trace of the execution of a system with a producer, a buffer, and a consumer.

ied by Bruni *et al.* [32]. The d2pc protocol was an important source of inspiration for the committing stage of our distributed algorithm, and therefore we pay some more attention to it.

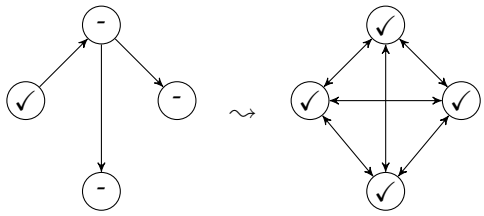
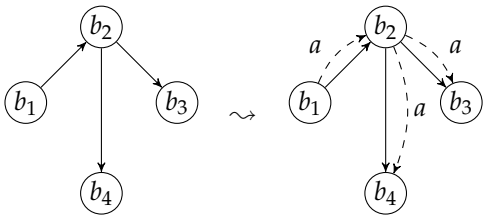
Before	After	Comments
		Only *topology* matters. Initially each actor knows some neighbours. In the end each actor knows every other actor, and whether or not everyone agrees on some request (denoted by ✓).
		Mainly the *behaviour* matters. At each round the actors agree on how data should flow, and perform the data transfer. In this case the data token *a* is sent between actors.

Table 7.3: Main difference between the application of the d2pc protocol (top) and the application of our distributed algorithm (bottom).

The main difference between the approaches taken by the Dreams framework and the d2pc protocol is their primary purposes. We use the diagram in Table 7.3 to illustrate this difference. The d2pc protocol starts with a set of actors with a limited set of connections, and after executing the protocol all actors become connected and know if anyone aborts the consensus. The Dreams framework does not change the topology of the connector (i.e., the connected actors) after each round. The focus is shifted from the topology of the connector to *how* and *where* data flow within the connector. Instead of agreeing on a yes or no answer, the Dreams framework agrees on a behaviour to send data though the connector. The evaluation of synchronous and data constraints required to reach a consensus in Dreams require a more complex mechanism than Bruni *et al.* 's approach.

Assuming only one of the actors initiates the d2pc protocol, the number of messages exchanged is exactly 2 for each possible pair of actors. In the Dreams framework, if only one proactive actor has an active port, the number of messages until a final behavioural automaton is known is exactly 2 for each pair of connected actors. In general, this is much less than the number of messages exchanged in the d2pc protocol, although the d2pc protocol can terminate faster, assuming that the cost of sending simultaneous messages is the same as sending a single message. Furthermore, the d2pc protocol has been fully formalised in the Join Calculus [50], whereas we leave the full formalisation of the distributed

algorithm as future work.

7.4.4 Discussion

In the reliable LossySync example we saw that it is possible for actors in different states and rounds to coexist in the same snapshot of the connector. We claim that this is not problematic because the order of messages sent will always guarantee a correct behaviour. This scenario reveals that the distributed algorithm introduces a level of concurrency not permitted before by previous implementations of $\mathcal{R}$eo, which makes it a non-trivial problem to guarantee that the system always evolves according to the global behaviour of all actors. We support this claim based on the runs of the Dreams framework only, and not by formal verification, which we leave as future work.

Together with Jaghoori, we have modelled in Rebeca [64, 65] a simplified case of a sequence of two and of three split FIFO_1 buffers. The behavioural automata of the current round were reduced to simple booleans stating whether the FIFO_1 buffers were empty or full, and we used a data domain with a single element. Furthermore, we assumed an infinite producer of data on the source port of the connector, and a reader that can always accept data on its sink port. As a result, we managed to fix a few mistakes in the then-current version of the algorithm, and we verified that the following properties hold for the simplified Rebeca implementation:

1. the actors never deadlock, i.e., can always send and receive messages;
2. each actor never has more than five messages in its mailbox; and
3. each FIFO_1 buffer changes its state infinitely often.

The first and second properties are specified as part of the model to be verified. The third property is specified as a separate LTL (linear temporal logic) formula. The generated model used to verify the first two properties has 254 states for two FIFO_1 channels, and it has 1820 states for three FIFO_1 channels. The generated model used to verify the third property, using three FIFO_1 channels, already has 5013 states. For this small example the size of the generated models are manageable. However, the fast growth of the number of states clearly shows that only small examples can be model checked. The analysis of traces and the formal verification of a simple connector lead us to believe that the algorithm is correct, although to claim correctness we expect to extend the formal verification to the full algorithm described in this chapter in the future.

We now stress the practical relevance of the actor splitting, and the resulting synchronous regions. The conditions for splitting an actor reflect the independence of each of the split actors from the other. It also pinpoints the parts of the

connector that always communicate in an asynchronous manner. This is also supported by our example of the split FIFO_1, and a trace of its execution in Figure 7.7. Between the two split actors there is no agreement before communication, and the transition labels are sent after being performed. Therefore, we say splitting yields *local autonomy*. The result of a splitting is a pair of split actors, each of which belongs to an independent synchronous region. The actors in a synchronous region do not have to interact with the actors in any other synchronous region to decide how to evolve. Specifically, a split actor does not need to consult with its connected split actor before deciding how to evolve. It only need to report its committed choices to its twin split actor.

In our distributed implementation we do not capture the notion of failure, which is a dominant concern in distributed systems. Instead, we rely on the actor model and assume that all communication is reliable. Note that existing distributed programming techniques typically treat reliable communication as an orthogonal problem, as suggested, for example, by Guerraoui and Rodrigues [55]. However, there are still open questions about how the system should react when some actors become disconnected during the run of the distributed algorithm. Some of such failures may also require changes in the coordination behaviour. We leave such questions out of the scope of this thesis.

7.5 Benchmarks

We evaluate the performance of our implementation of the Dreams framework by considering the time to create a $\mathcal{R}$eo connector and to perform the communication to pass a sequence of data values from some writers to some readers. For comparison we use the constraint automata-based implementation of $\mathcal{R}$eo, described in Chapter 5, which we call the CA engine. However, our main concern in our comparison in this chapter is different than the benchmarks performed in §5.6. For the analysis of the constraint-based approach for $\mathcal{R}$eo we compared the performance of composing the possible behavioural automaton for a single round. Now we compare the performance of a connector that evolves in time, acknowledging the time to deploy a connector and the time to send data through the connector. We present examples of stateful connectors, with one exception for comparison. More details about the two engines used in our benchmark follow below.

CA engine The CA engine is a code generator and interpreter of $\mathcal{R}$eo connectors included in the suite of Eclipse[3] plug-ins for $\mathcal{R}$eo [16].[4] It uses the context independent semantics of $\mathcal{R}$eo, following the ideas of port automata [70]

[3] http://www.eclipse.org

[4] Available at http://reo.project.cwi.nl/cgi-bin/trac.cgi/reo/browser/ea-codegen

extended with memory and data transfer functions, similar to how we extended the connector colouring semantics with data information in §4.4.4.

Dreams engine The Dreams engine implements the Dreams framework as described in this chapter, using the behavioural automata of $\mathcal{R}$eo primitives and nodes. We use the constraint-based approach with context dependency, as described in Chapter 5. In this benchmark we consider the extreme (and thus inefficient) case where every node, channel, and component is deployed as an independent actor that performs the distributed algorithm explained in §7.3.

By associating each node, channel, and component to an independent actor we emphasise the differences between these the CA and the Dreams engines. More optimal scenarios for the Dreams engine would deploy one actor for a group of connected $\mathcal{R}$eo primitives, using the combined behavioural automaton, as explained in §6.5.4.

7.5.1 Test cases

We present four test cases of $\mathcal{R}$eo connectors deployed and executed in the CA and Dreams engines. For each test case we replicate a part of the connector n times, following the same approach as in the previous benchmarks performed in §5.6.

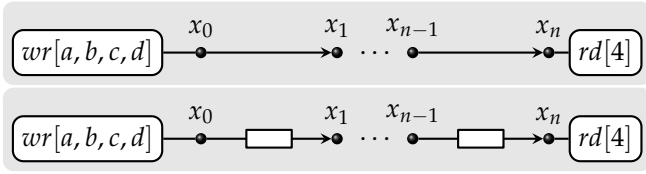


Figure 7.8: Sequences of n Sync channels (above) and FIFO_1 channels (below).

The first two test cases consist of a sequence of Sync channels (Syncs) and a sequence of FIFO_1 channels (Fifos), as depicted in Figure 7.8. These test cases are more academic, and allow us to explore the best and the worse case scenarios of each of the two engines. In both cases we defined the writer component to have four data values ready to send, and the reader component to be able to receive at most four values. The sequence of Sync channels provides an ideal scenario for the CA engine, since after hiding its intermediate ports (implemented by the CA engine) the automaton of the global system is the same as the automaton of a single Sync channel. Thus, it is better to precompute the behaviour of this connector. On the other hand, the sequence of FIFO_1 channels is optimal for the Dreams

engine, since it consists of only small synchronous regions that can run independently. For the CA engine the number of possible states doubles with every new FIFO_1 channel, making the construction of the automaton for the whole connector infeasible for large n.

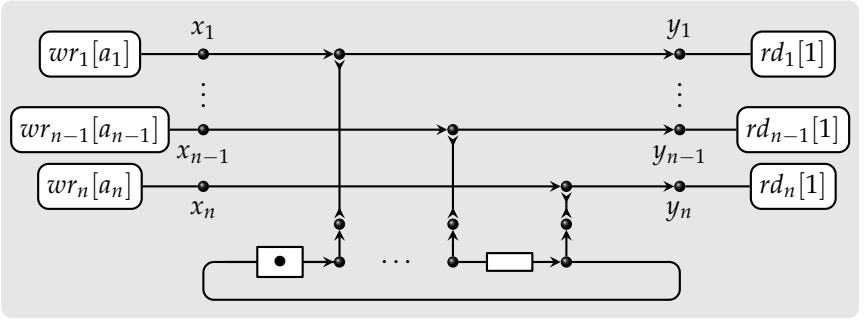


Figure 7.9: Sequencer restricting the flow of n synchronous channels.

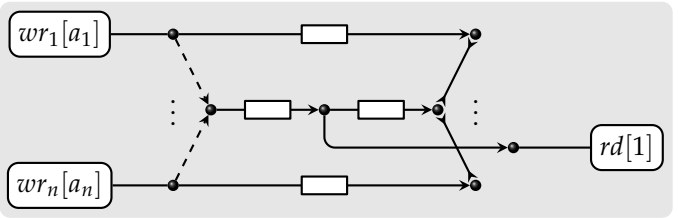


Figure 7.10: Discriminator connector attached to n writers.

The last two test cases represent more useful patterns of coordination. The sequencer connector (Seq), depicted in Figure 7.9, has been presented before in §6.5.3 to illustrate the existence of synchronous regions. In our test case we assume that each writer component can produce only one data value, and each reader component can receive one value. The last test case is the *discriminator* connector (Disc), depicted in Figure 7.10, which receives data from n sources and sends the first data value received to a single port. The discriminator is one of the workflow patterns proposed by van der Aalst [99]. In our example we have connected the discriminator connector to a reader component and n writer components, each ready to exchange one data value.

7.5.2 Results

All benchmarks were executed on a PC with an Intel® Core™2 Quad CPU Q9550 processor at 2.83GHz and with 7.8GB of RAM, running Fedora release 10. For

each value of n, we performed 10 different executions and used the averages of the measured values. Furthermore, for each test case we evaluate two different aspects, the time to *build* a connector and the time to *exchange* data.

Build time The creation of the connector is performed once, after which the connector can be executed multiple times. In the CA engine creating a connector corresponds to calculating the product of the automata representation of each channel and node, deploying a centralised engine and the components, and connecting the engine to the components. In the Dreams engine, the creation of the connector consists of the deployment of each actor, and the creation of the connections between the actors. Note that the CA engine performs the creation sequentially, therefore the evaluation of the build time is performed simply by measuring the time before and after the creation process. On the other hand, the Dreams engine performs the connections between actors in parallel. Consequently, the Dreams engine uses a shared variable v to know when all actors are connected, using the traditional Java approach to lock v every time v is updated by one of the actors—an unnecessary and unfair handicap necessitated only by our need for measurement.

Exchange time After a connector is deployed and connected, the exchange time consists of the time required to exchange a sequence of messages between the components until no more data can be exchanged. The *exchange time* is calculated in both engines by measuring the time when the first message is sent, and the time when the last message is processed.

Actors in sequence

The build and exchange times for the sequence of n synchronous channels and n FIFO_1 channels are presented in Figure 7.11. On top we present the measurements for the sequence of synchronous channels, and on the bottom graphics we present the measurements for the sequence of FIFO_1 channels. The left graphics represent the results of the CA engine, while the right graphics represent the results of the Dreams engine. Each bar of the graphics is split into two parts, the bottom part represents the build time, while the top part represents the send time.

Start by observing the top graphics of Figure 7.11 with the measurements of the CA engine. The sequence of Sync channels have the most impressive results, with a linear growth of the build time, taking less than 3 seconds to compose 3000 channels, and with constant send time. The constant time is easily explained by the fact that the composition of the automata of two Sync channels is again the automaton of a Sync channel. The main problem is shown in the compilation times of the FIFO_1 channels. The automaton for a sequence with more than 5 FIFO_1 channels already takes more than 36 seconds to generate. An automaton resulting from the composition of 6 FIFO_1's has already $2^6 = 64$ states and an

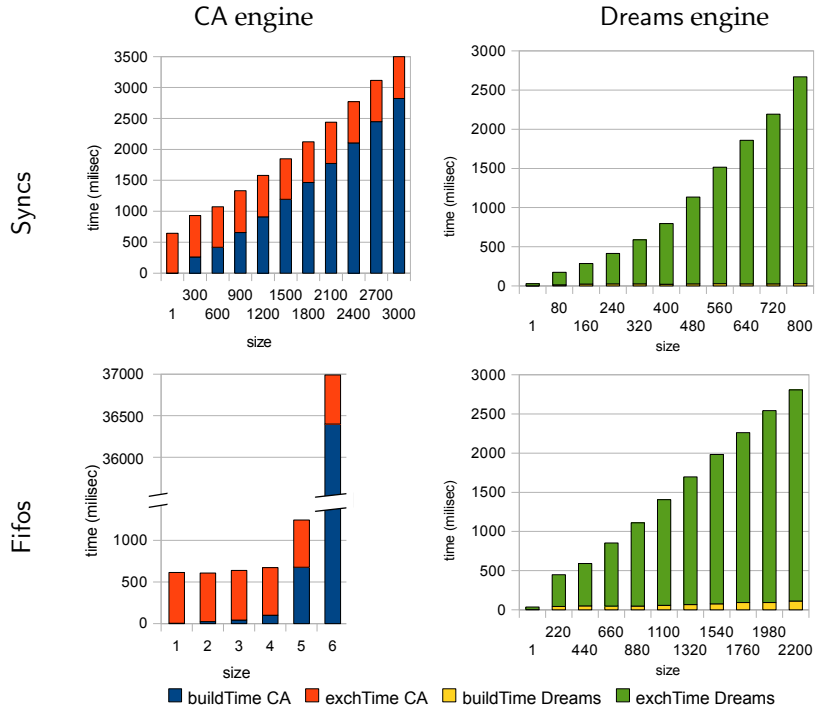


Figure 7.11: Results of the evaluation of the build and send times for the Syncs (top) and Fifos (bottom) connectors, using the CA (left) and the Dreams (right) engines. Each bar in the graphics is divided into two parts. The top part represents the exchange time, and the bottom part represents the build time.

even larger number of transitions, increasing the time to compose the automata exponentially.

The Dreams engine has an extremely small build time when compared with its execution time, which grows linearly for the sequence of $FIFO_1$ channels, due to the splitting of actors. Furthermore, the time taken to deploy the $FIFO_1$'s and send four data values is less than three seconds for 2200 channels, overcoming successfully the small number of $FIFO_1$ channels supported by the CA engine. The sequence of Sync channels exhibits an exponential growth in Dreams, due to the constraint solving process performed once for each sending of data, although the growth is still relatively slow when compared to the exponential blowup of the sequence of $FIFO_1$ channels with the CA engine.

Sequencer and Discriminator

The build and exchange times for the sequencer and the discriminator connectors are presented in Figure 7.12. On top we present the measurements for the sequencer connector (Seq), which favours the Dreams engine because of the big number of synchronous regions. The bottom graphics present the measurements for the discriminator connector (Disc), which has a fixed number of synchronous regions, making the time to reach a consensus in Dreams grow exponentially with the size of the connector. As before, the left graphics represent the results of the CA engine, and the right graphics represent the results of the Dreams engine.

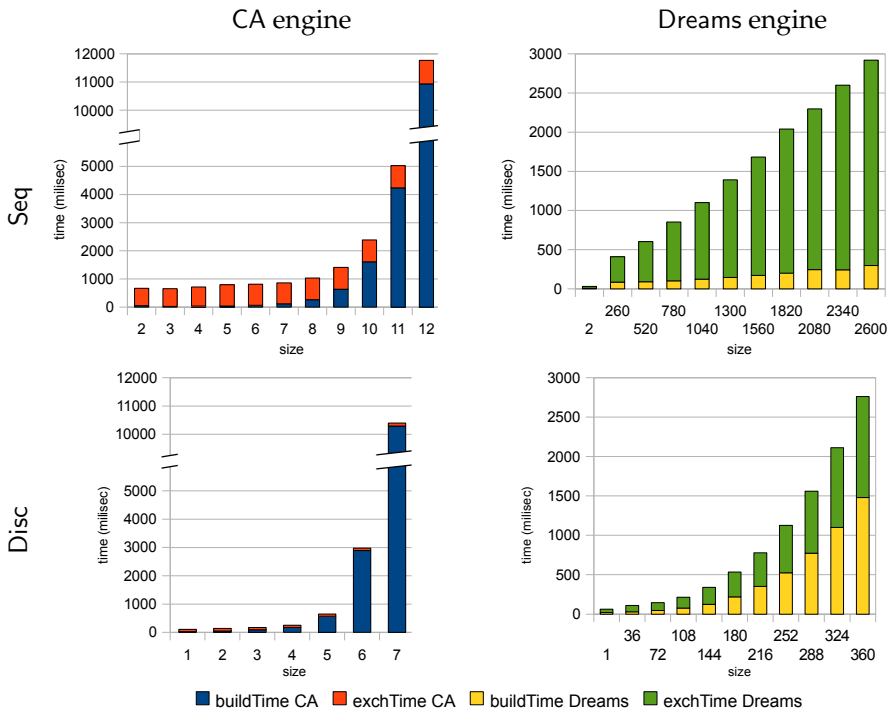


Figure 7.12: Results from the evaluation of the build and exchange time for the Seq (top) and Disc (bottom) connectors, using the CA (left) and the Dreams (right) engines. Each bar of the graphics is divided in two parts. The top part represents the exchange time, and the lower part represents the build time.

Start by observing the evaluation of the CA engine. A curious fact is that the sequencer connector can reach almost double the size of the sequence of FIFO_1 channels before a big blow-up (6 vs. 12). Note that the number of possible states of the automaton of the sequencer connector is n, because only one of the FIFO_1 channels in the loop can have a data token, while in the sequence of FIFO_1 chan-

nels any combination of empty and full $FIFO_1$'s is possible. However, intuition says that the sequencer connector of size n would still need to calculate the behaviour of a sequence of $n-1$ $FIFO_1$ channels before the loop can be closed, reducing the number of possible states. So, why is it that the CA engine can handle more $FIFO_1$'s in the sequencer connector than in the case of a series of $FIFO_1$'s? This is explained by the order in which the CA engine composes its automata. Instead of composing the automata in sequence $(a_1 \bowtie a_2) \ldots \bowtie a_n$, it builds a balanced tree to with the automata to be composed and calculates $(a_1 \ldots \bowtie \ldots a_{n/2}) \bowtie (a_{n/2+1} \ldots \bowtie \ldots a_n)$. The goal of this heuristic is to compose automata with similar sizes, assuming that in general the composition of two automata yields a larger automaton than any of the initial automata. Hence when calculating the sequencer with twelve $FIFO_1$'s, the CA engine calculates two automata with six $FIFO_1$'s each, which should take around 2×3700, and only than calculates its composition, which will yield a much smaller automaton. The build time of the discriminator also explodes for relatively small connectors, as expected.

The Dreams execution of Disc reveals an unexpected observation. The build time appears to grows exponentially, as opposed to the previous examples where it grows linearly. This is explained by a choice in the current implementation of the nodes in the Dreams engine. When a new port is added to a node, the Dreams engine builds the new constraint for the coordination behaviour of the node from scratch, disregarding its previous constraint. Therefore, to build a node with n ports the engine finds the constraints for 1 port, then for 2 ports, until n ports. A simple optimisation is to postpone the calculation of the coordination behaviour until the suspension is released.

Discussion

These four examples show that the Dreams engine is specially relevant in two main situations: (1) when the connector needs to be re-compiled (or reconfigured) frequently, and (2) when the pre-computation is infeasible because of the excessive time required to compile it. These benchmarks also support the discussion in §6.5.4, where we claim that an ideal scenario needs to have a good balance of pre-computation (as implemented in the CA engine) and on-the-fly computation (as implemented in the Dreams engine).

The key insight obtained from these benchmarks is how automata-based implementations fail to execute more complex connectors, where the number of possible states explodes exponentially. It highlights the tradeoff between pre-computation and on-the-fly computation, suggesting different scenarios require different approaches. Note, however, that the true power of the automata models is in the verification of systems, which is not addressed by the Dreams framework. In fact, the Dreams framework exploits the fact that discovering only one possible

step at each round is enough for the execution of a connector, and the fact that not all actors require synchronous communication. Furthermore, Dreams does not require any further knowledge beyond the next step, which may not even be available or known for some primitives.

7.6 Scala implementation

The Dreams framework is implemented in Scala,[5] a language that integrates features of object-oriented and functional languages, and that is fully interoperable with Java. More specifically, Dreams uses the actor library of Scala [57]. Scala programs are compiled into Java binary classes.

The most recent version of the source code of our implementation is available in our $\mathcal{R}$eo repository.[6] The source code is grouped into packages. The most relevant packages are presented below. Later in this section we describe other packages, also available in our repository.

- `cwi.reo.engine.common` – defines behavioural automata and atomic steps;
- `cwi.reo.engine.redrum`[7] – defines $\mathcal{R}$eo primitives;
- `cwi.reo.engine.colouring` – defines the CC semantics as behavioural automata;
- `cwi.reo.engine.SAT` – defines the constraint-based semantics as behavioural automata.

The definitions of $\mathcal{R}$eo primitives follow directly the definitions of the reactive behaviour presented in §7.3. We define proactive actors as an extension of normal primitives, which we call *initiators* in the source code. The behaviour of each $\mathcal{R}$eo primitive, such as a $\mathcal{R}$eo channel or a $\mathcal{R}$eo node, is defined inside the `colouring` and the `SAT` packages. We implemented the context sensitive semantics for both the CC (using three colours) and for the constraint-based approach (using context variables, as described in §5.5.2).

We also performed an optimisation with respect to the data function. The behavioural automata of the primitives encoded using the CC and constraint-based semantics do not depend on the data to agree on the final behavioural automaton. Furthermore, by knowing which step to perform without the data function, each primitive can deduce how data should be transferred. This step corresponds to the notion of group of labels with the same sets of ports with dataflow, as described in §3.5. Therefore, we store the data functions within each primitive, and we do not include it in the behavioural automaton.

[5] `http://www.scala-lang.org`

[6] `http://reo.project.cwi.nl/cgi-bin/trac.cgi/reo/browser/reo-engine`

[7] The name 'redrum' stands for $\mathcal{R}$eo *distributed machine*.

Note on heterogenety The Scala implementation of Dreams includes primitives whose coordination behaviour is given by colouring tables (as described in Chapter 4) or by constraints (as described in Chapter 5). However, the execution depends only on the definitions of behavioural automata and atomic steps, available in the package `cwi.reo.engine.common`. Therefore, the same implementation can be used for different coordination models, including the Linda [54] and Orc [68] coordination models, or even other semantic models for $\mathcal{R}$eo. Note that all connected actors must use the same coordination model, since the composition operator is defined at the level of the concrete model, and must be closed. In principle, it is also possible to intermix different coordination models (e.g., $\mathcal{R}$eo, Orc, Linda, etc.), the main challenge being to define a common underlying model for them that can be encoded as behavioural automata.

7.6.1 Deployment

The Dreams framework is capable of executing relatively complex $\mathcal{R}$eo connectors. This section shows how the current implementation of Dreams can be used to define and execute a connector. For the benchmarks performed in §7.5 we simply defined a Scala program that creates a fixed connector, which after its creation, sends a `Release` message to all actors to start the execution of the connector. The source code of the benchmarks is available in the package `cwi.reo.engine.sandbox`.

To make the Dreams framework more usable, we recognise the need for the deployment phase, introducing two new classes: an Engine and an Engine Manager, available in the package `cwi.reo.engine.redrum.distributed`. In this section we give only an overview of the architecture of the implementation of Dreams, and do not go into details of deployment.

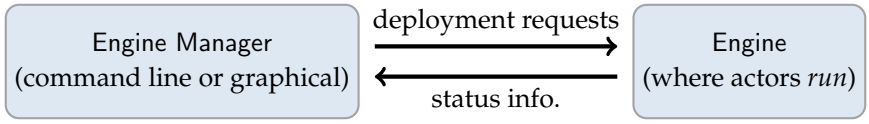


Figure 7.13: Relation between the Dreams engine and the engine manager.

We use an extension of Scala actors called `RemoteActors`, which differ from normal Scala actors because they can be accessed from outside of their executing Java Virtual Machines via TCP/IP. The engine and the engine manager are instances of remote actors that communicate with each other via the TCP/IP socket mechanism, as depicted in Figure 7.13. An instance of Engine receives messages with *deployment requests* to create, delete, and update actors from an Engine Manager, and sends information about its current phase and relevant changes to all observing Engine Managers. The Engine Manager provides methods to communicate with instances of Engine. More than one instance of an Engine Manager can be

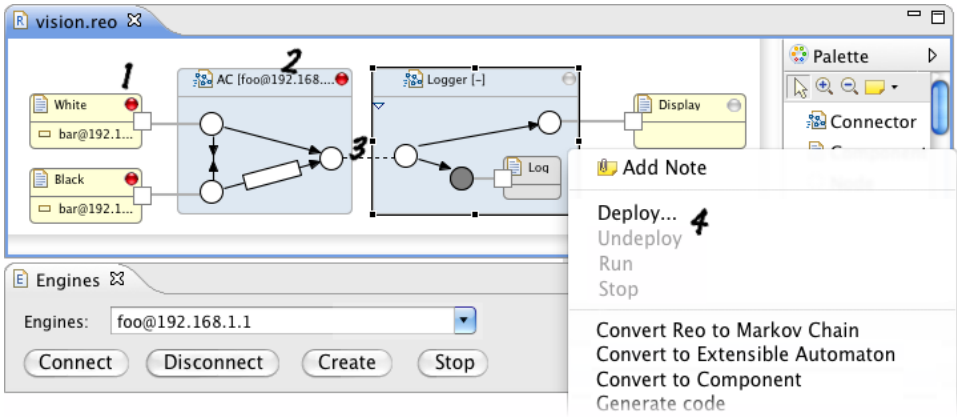


Figure 7.14: Screenshot of the proposed interface for the user.

connected to one engine, and each instance of Engine Manager can be connected to multiple Engines.

We developed two packages in Java that use the Engine and Engine Manager classes. The first package is `cwi.reo.engine.standalone`, and contains Java applications that receive command line arguments with the parameters required to create and manage engines. It is currently the most complete way to access all the functionality provided by the engines. The second package is `cwi.reo.engine.ui` which is a graphical plug-in for Eclipse integrated in the Eclipse Coordination Tools (ECT) [16] framework. The ECT framework incorporates a set of $\mathcal{R}$eo-related tools, including a $\mathcal{R}$eo editor, an animation generator, model checkers, QoS modelling and analysis tools, and the code generator used in our benchmarks in §7.5.

7.6.2 Proposed graphical plug-in

We have developed a graphical plug-in for Eclipse, which we describe in §7.6.4, that acts as an interface for the users. It sits between the editor of $\mathcal{R}$eo connectors and the distributed engines where $\mathcal{R}$eo connectors can be deployed. The existing implementation is not yet as user-friendly as planned. We first present here the graphical interface that we envision in a near future, before presenting the existing user interface for deployment.

In Figure 7.14 we present a screenshot of the two views of Eclipse that we propose, used for the deployment process. On top we see the $\mathcal{R}$eo editor enhanced with the following features, whose numbering corresponds to the numbers in the figure.

1. Representation of the status of the deployed connector or component: green

circle (●) denotes that the actors are running, red circle (●) denotes that the actors are suspended, and grey circle (●) denotes the connector or component is not deployed.

2. Representation of where the connector or component is deployed.
3. Unification of nodes from different engines, which we explain in §7.6.3.
4. New actions in the context menu of the connectors (i) to deploy the connector by creating the required actors in a given engine, (ii) to remove (undeploy) all the actors associated with the connector, (iii) to release the associated actors from suspension and start the algorithm, and (iv) to stop the algorithm and suspend the associated actors, e.g., to allow reconfiguration.

On the bottom part of Figure 7.14 we show the Eclipse view that manages the set of engines connected to the graphical interface. Running engines can be added by pressing the "Connect" button, and removed or disconnected by pressing the "Remove" or "Disconnect" button, respectively. Furthermore, new local engines can also be created, using the "Create" button.

A developer of connectors can draw new connectors, and later perform the action "Deploy..." to create the required actors in a given engine. Once deployed, the actors associated with the connector will be suspended, and any change to the connector will automatically result in a reconfiguration. Only after a connector C and its connected connectors or components are deployed can C be executed, using the action "Run" in the context menu. When connecting to a new engine using the "Connect" button, all existing connectors and components will automatically be imported into the graphical editor, and deleting the connectors in the editor will result in the deletion of the corresponding actors from the running engine.

Note that it is not possible to perform actions on individual actors, such as the actor of a $\mathcal{R}$eo node. Only the full set of actors associated with a connector can be manipulated by these operations. This will give the freedom for each of the engines to optimise the number of actors it creates to achieve the behaviour of a deployed connector. Recall the discussion in §6.5.4 where we refer to a hybrid deployment. Associating actions to connectors instead of $\mathcal{R}$eo primitives allows the implementation of a hybrid deployment by the engines, without the user being aware of it.

7.6.3 Distributed engines

We use `RemoteActors` to define Engine and Engine Manager, to allow multiple engines and managers to execute in different JVMs (Java Virtual Machines), which can be running in different computers on a network. We now extend this concept to implement the coordination of a connector that is deployed in multiple cooperating engines.

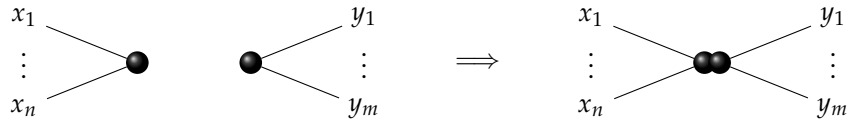


Figure 7.15: Unification of remote nodes.

A naive approach is to assume that all actors are remote actors, accessible from the outside of their JVMs. We avoid the overhead of making all actors accessible from the outside, and restrict the remote actors to only a new $\mathcal{R}$eo primitive that we call *remote node*. Consequently, all remote communication is embodied in a single type of actor. Note that we do not forbid the creation of other remote actors, although in most scenarios this is not required. An interesting direction would be to define remote split actors for efficiency reasons, since this requires fewer messages to be exchanged. However, as we will see shortly, remote split actors are not as general as remote nodes.

A *remote node* is an actor with the same behavioural automaton as that of a traditional $\mathcal{R}$eo node. However, a remote node is also accessible from other JVMs, and can be *unified* with other remote nodes, as depicted in Figure 7.15. When two remote nodes are unified they exhibit the same behavioural automaton as that of a single $\mathcal{R}$eo node with the union of all their ports. More than two remote nodes can be unified in a pairwise fashion. A pair of unified remote nodes share the information regarding which ports they have, and follow the same distributed algorithm described in this chapter. We have adapted the algorithm presented earlier in §7.3 to accommodate the communication between unified nodes because in this case communication can occur in both directions depending on the step of the current round, but we omit these details here.

We extend the idea of connected engines via remote nodes to allow general components to be connected to already running $\mathcal{R}$eo connectors. We define a new class `ComponentPort` that wraps a variation of a data writer or reader connected together with a remote node. The component ports are made available to any Java developer, providing methods to send data to or receive data from the component port, as well as to unify other remote nodes. The source code of the remote nodes and the source code of the component ports are available in the package `cwi.reo.engine.redrum.distributed`. Furthermore, small examples of the use of the component ports can be found in the package `cwi.reo.engine.standalone`.

7.6.4 Existing graphical plug-in

The existing graphical deployment plug-in for Eclipse is depicted in Figure 7.16. It uses the $\mathcal{R}$eo editor included in the ECT framework to extract information from

the connectors being developed. The user, the developer of a $\mathcal{R}$eo application, can then deploy and run actors associated with primitives in the editor using a graphical interface. It is possible to select a running engine by introducing its IP address and port number, or even to create a local engine. A wizard will guide this process by clicking on the "Add" button.

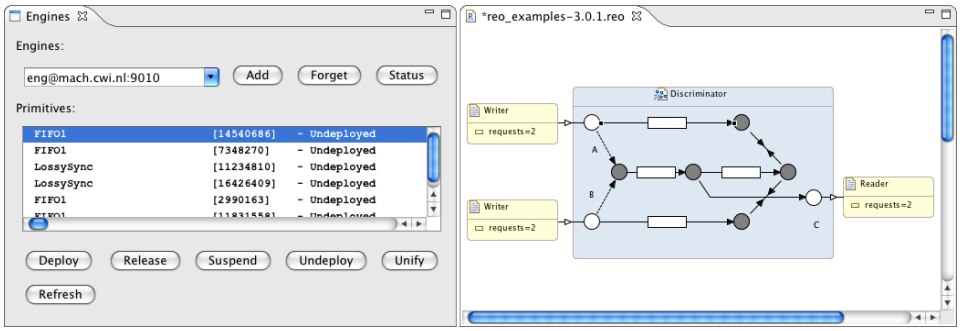


Figure 7.16: Deployment plug-in for the Dreams framework for $\mathcal{R}$eo (left) and ECT's editor (right).

To use the plug-in the user has to select a connector in the editor (by clicking on the blue rectangle), and a list of primitives become available on the deployment view (on the left of Figure 7.16). The next step is to select an engine on the top combo box. If necessary, new connections to existing engines can be added by pressing the "Add" button. By selecting the list of primitives that we want to deploy, and pressing the "Deploy" button, a request to the selected engine is sent to create and deploy the engines. A node is automatically created when two connected primitives are deployed.

Upon deployment, primitives are in the *Suspended* phase, where they can be reconfigured. Deploying a new channel attached to a previously deployed node triggers a reconfiguration of this node, changing its behavioural automaton to reflect its new behaviour after adding new ports. The button "Unify" allows the unification of nodes executing in different engines, as explained in §7.6.3. To execute a connector, the user has to release the suspension of all involved primitives, by selecting them and pressing the "Release" button.

Several optimisations and user-friendly additions can still be incorporated, but these issues have not been a priority concern compared with the correct behaviour of the system. We provide an interface between a visual editor and a running engine as a proof-of-concept binding, which we expect to improve in the future.

7.6.5 A guided example

The goal of this subsection is to illustrate how the tools can be used to deploy and execute a $\mathcal{R}$eo connector running in multiple engines, and to execute components in the JVMs in the same network as the engines which they are attached to. Note that the deployment aspect of the Dreams implementation is not yet very user friendly.

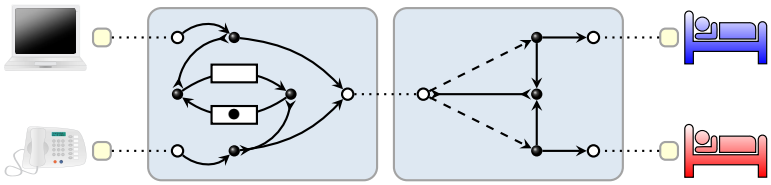


Figure 7.17: Example of four services connected by two engines.

We exemplify the application of the Dreams framework using the command line and the graphical plug-ins included in the ECT framework, for the connector depicted in the diagram in Figure 7.17. The connector on the left consists of a sequencer alternating the flow from the two source ports, and a merger that offers the values from the two source ports through the same sink port. The connector on the right is an exclusive router that sends data to one of its sink ports on the right, non-deterministically. The notation used in the diagram is the following. The four images on the corners represent Java implementations of services. An *online* service and a *phone* service on the left try to send reservation requests to one of the hotel services on the right. As an example, we define four toy Java applications, one for each service. Each of the small squares next to each service represents a `ComponentPort` (see §7.6.3) used by the Java application, wrapping a distributed node. The dotted lines represent the unification of two distributed nodes (see §7.6.2). Finally, the two big rectangles surrounding $\mathcal{R}$eo connectors represent two engines executing in two different JVMs running on the same network. The possible steps to set up this system are described below.

1. *Start the engines.* We create the left engine on a machine `foo` and the right engine on a machine `bar` by opening a terminal window on each machine, changing directory to where the package `cwi.reo.standalone` is compiled, and executing each of the following commands on its respective machine:

   ```
   bash:foo# java engine.Run fooEngine 192.168.1.1 9011
   bash:bar# java engine.Run barEngine 192.168.1.2 9012
   ```

   The first command creates an engine entitled `fooEngine` on the machine where this command is executed (`foo`), using the TCP port 9011, where

`192.168.1.1` is the IP address used to access this machine. Similarly, the second command creates an engine called `barEngine` on the machine where this command is executed (`bar`), using the TCP port 9012, with IP address `192.168.1.2`.

2. *Deploy the connectors.* The connectors will be created using the graphical deployment plug-in, depicted in Figure 7.16. For that we open Eclipse on a machine in the same network as `foo` and `bar` with the editor and deployment views. In the deployment view we click the "Add" button in the engines area, and insert the information regarding the `fooEngine`, and repeat the same for the `barEngine`. We create two connectors in the editor, one for each part (the two central boxes) of the connector in Figure 7.17. We click on the left connector, and the list of primitives in the deployment view is updated. We select the `fooEngine`, then all primitives in the list, and press the `Deploy` button. We repeat this process for the `barEngine` with the right connector.

3. *Connect the engines.* We unify the sink node of the left connector with the source node of the right connector using the Eclipse plug-in. We select the two nodes that we want to unify in the $\mathcal{R}$eo editor, and press the "Unify" button in the deployment view. We make the boundary nodes that will be attached to components accessible via TCP/IP using the manager in the command-line tools. We read the identifier *prim* of the channel attached to each boundary node from the list of primitives in the graphical deployment plug-in, and use the manager in the package `cwi.reo.engine.standalone` as follows.

```
bash:foo# java engine.Manager fooEngine 192.168.1.1 9011 \
          Awake prim 9013
```

   The command-line instruction above assumes the correct classpath is defined and *prim* is the identifier of a boundary node deployed in `fooEngine`.

4. *Create the components.* The components are developed in plain Java, importing the relevant Java binaries resulting from the compilation of the Scala implementation of Dreams. In Listing 7.1 we present a Java program `OnlineBooking` that can be used to implement the *online* service in Figure 7.17. We abstract from certain technicalities such as error handling. The program uses an extension of `ComponentPort` called `SinkPort`, imported in line 1 and instantiated as port in line 17. The method `addDatum` used in line 32 adds the string stored in variable *line* (obtained earlier on line 25 from the standard input) to the queue of available messages to be sent by the remote node in the component port. How the component port is connected is not of concern inside the `OnlineBooking` program: it is the responsibility of the coordination

engine. The other components can be created in a similar way, using the extension `SourcePort` of `ComponentPort` instead of `SinkPort` for the hotel services. The source code of these extensions of component ports is available in the package `cwi.reo.engine.redrum.distributed`, and examples of simple Java programs that send and receive text to component ports are available in the package `cwi.reo.engine.standalone. components`.

5. *Running the system.* The execution of the `OnlineBooking` Java program connects to the distributed node identified in its command line arguments upon creation of the `SinkPort`. The port is initially suspended with a default lock, and is released when its `release` method is invoked in line 17. After its release it becomes idle because no data is available for it to send. We repeat the same process for the other components. The `OnlineBooking` program can try to send a booking request already, but its message will not be consumed by the connector because the other actors are still suspended. We release the connector using the deployment view in the Eclipse plug-in, by selecting all channels and pressing the `Release` button. The components can now request data to be sent or received, changing the states of their behavioural automata when they become idle, and engage in the distributed algorithm described in this chapter.

7.7 Conclusions

This chapter takes a more practical view of the Dreams framework, and presents how the distributed algorithm for committing to a behavioural automaton and sending data operates in detail. It also shows how a developer of $\mathcal{R}$eo connectors can use the current implementation of Dreams in the context of ECT.

The interplay between synchrony and asynchrony, reactive behaviour and coordination behaviour, open new possibilities for scalability and reconfiguration of synchronous languages such as $\mathcal{R}$eo. We showed based on benchmarks that precompiling the full behaviour is not always possible or advantageous. The separation of a single connector into independently executing sub-connectors also allows for a more flexible framework for development and concurrent reconfiguration of larger coordination specifications.

For future work we aim to compare the execution of connectors on multiple computers. At the moment the process of deploying a connector over a network is not yet fully automatised, leaving new research questions. These questions include: how to optimise the number of actors and behavioural automata for a given connector; how to automatically chose in which machine should each actor be deployed, and how to dynamically adapt a running connector based properties of the network and demand for communication.

```
1  import cwi.reo.engine.redrum.distributed.SinkPort;
2   (...)
3
4  public final class OnlineBooking {
5    /**
6     * @param args <thisip> <rank> <otherremotenode>
7     *                  <otherip> <otherport>
8     */
9    public static void main(String[] args) {
10      (...)
11
12     // Create a component port and connect it to a distributed node using the
13     // array args (rank, remote node's ID, IP and TCP port, and local IP).
14     SinkPort port = new SinkPort( (...),
15       args[1], args[2], args[3], args[4], args[0]);
16
17     port.release ();
18
19     while(true) {
20       // Prompt user for a message and read answer from standard input.
21       BufferedReader br = (...)
22       String line = null;
23       System.out.print("Enter reservation request: ");
24       try {
25         line = br.readLine();
26       }
27       catch (Exception e) {
28          (...)
29       }
30        (...)
31       System.out.println("Sending reservation request.");
32       port.addDatum(line);
33        (...)
34     }
35   }
36 }
```

Listing 7.1: Example code for the online booking component, where some details are hidden by using the symbol "(...)".

Chapter 8

Conclusions

This thesis is an exploration in the field of coordination, with special focus on models with synchronisation constructs. Most of the work was developed to support the coordination language $\mathcal{R}$eo, although the results extend beyond and are applicable outside the scope of $\mathcal{R}$eo.

The main question addressed in this thesis is how to manage the complexity of synchronous models. Hitherto existing implementations of $\mathcal{R}$eo represent a connector by a monolithic state-based system that controls all communication. However, this approach does not scale up to larger connectors, due to the exponential growth of connectors obtained when composing the building blocks of an elaborate coordination layer.

Our key contribution is the development of new implementation techniques for $\mathcal{R}$eo-like languages, which exploits concurrency and shifts part of the calculation of the coordination behaviour into runtime. We successfully deal with larger connectors than previous $\mathcal{R}$eo engines, and we can execute such synchronous connectors in a distributed network.

Main contributions

In Chapters 2, 3, and 4 of this thesis, we gave a formal presentation of existing coordination models and introduced the stepwise coordination model. We explored how to use the connector colouring semantics in a distributed implementation. The stepwise model focuses on the same key concerns as the connector colouring semantics: the compositional development of synchronous coordination systems. We encoded connector colouring in the stepwise model, extending it with state changes, with data constraints, and with the notion of local colourings. We also presented an animation framework to exemplify the challenges of using the connector colouring semantics as the basis of an implementation that sends data across a $\mathcal{R}$eo connector. The animation framework provided the starting point for developing the Dreams distributed engine, introduced in this thesis.

In Chapters 5, 6, and 7, we improved existing techniques to execute synchronous coordination models such as $\mathcal{R}$eo in three ways:

- by increasing performance using constraint satisfaction techniques;
- by improving scalability by identifying synchronous regions; and
- by supporting decoupled execution and lightweight reconfiguration.

We briefly review each of these contributions below.

Constraint satisfaction This work alleviates some of the performance and scalability limitations of existing implementation of $\mathcal{R}$eo by shifting part of the computation of the composition of behaviour from compile time to runtime. As explained in Chapter 5, we used existing constraint satisfaction techniques to calculate the coordination behaviour of each atomic step, reducing the complexity of the composition operation.

Synchronous regions In Chapter 6 we introduced Dreams, a framework that provides a distributed implementation based on the stepwise coordination model, using the actor model to describe the primitive communication between concurrent elements. Equipped with the notion of locality from the stepwise model and the asynchronous properties of the actor model, we statically find parts of the coordination system that can perform local atomic steps. We refer to these regions as *synchronous regions*. Synchronous regions communicate with each other using only asynchronous requests for state updates, without changing the behaviour of the overall system. Hence, adding new synchronous regions to a connector does not affect the behaviour of each of the synchronous regions, allowing scalable implementations.

Decoupled execution The Dreams framework is structured as an Actor Model [1]. Its implementation, described in Chapter 7, uses the actor library of Scala. Actors are primitives that communicate with each other using a reliable and order-preserving message passing mechanism. Our main contribution is to assign to each actor a description of the coordination behaviour, given by the stepwise model, and to send data through the connector according the composed behaviour of all actors. We developed a distributed algorithm where actors associated with the same synchronous region reach a consensus regarding the atomic step to be performed before sending the data through the connector. The computation of this atomic step is performed by one of the actors of the synchronous region, and not by a centralised entity.

As an orthogonal contribution, this thesis addressed $\mathcal{R}$eo's semantics, giving an exhaustive comparison of existing models and implementations, with special emphasis on to three particular models. We explained what is salient in $\mathcal{R}$eo and

what aspects of coordination are left uncovered. We argue that, common to all existing models, are *synchronous* constraints over ports in a *compositional* framework. We use these notions in our implementation, and include dataflow information and locality aspects.

Future work

The stepwise coordination model described in Chapter 3 presents only the required aspects of coordination for our distributed framework. As future work, we leave a more complete formal treatment of this model, that is, we leave open the question of how to reuse formal proofs over the stepwise coordination model that can be directly applied in more concrete models. Furthermore, we provide an informal description of the correctness of the instantiation of concrete coordination models into the stepwise model, leaving as future work a complete formalisation of these proofs.

The animation framework, presented in Chapter 4, creates visual animations of $\mathcal{R}$eo representing differently different data elements flowing in the network, and including state information. A possible way to improve the current animation engine is to include local steps, and to show the behaviour of only a subset of a connector. Furthermore, the animations can be extended with new information regarding, for example, properties of the underlying network, amount of expected traffic, or security level of the data channels.

In Chapter 5 we use constraint satisfaction techniques to calculate the coordination behaviour of each step. Regarding this a number of questions remain unaddressed in this thesis.

- What other relevant notions, besides synchronisation, data, and context dependency, are useful to model existing coordination systems?
- How can a constraint solver be guided more efficiently to find coordination solutions?
- How can the existing constraints be extended, e.g., with special symbols that are initially unknown, but are replaced by their interpretations only when needed?
- How to perform constraint solving using these special symbols?
- How to find a solution σ for a subset of a conjunctive collection of constraints, such that σ is also a solution for the global collection? This would allow, for example, to represent constraints that grow in a per-need basis, without size restrictions.

We gave several suggestions how to guide the constraint solver in §5.7, without going into detail. The last two questions are partially addressed in existing work regarding constraint-based coordination [38, 40]. Furthermore, we presented in this thesis an alternative approach to search for solutions for only a subset of constraints based on synchronous regions, described below. We statically identify a collection of constraints whose solutions will always be solutions of a more global set of constraints. However, how to allow these sets to be discovered dynamically during constraint satisfaction is ongoing work.

The Dreams framework is introduced in Chapter 6, where we describe how and why different regions, dubbed synchronous regions, can evolve in parallel without requiring a global consensus at each step. We identify two possible directions of future work regarding the use of synchronous regions.

- Partition a connector into its synchronous regions by an *automatic* splitting mechanism. Currently we analyse manually each building block of the coordination model and try to find ports that communicate asynchronously, without any tool support. Ideally, this process would be automated, or alternatively, the discovery of synchronous regions can happen at runtime, as suggested in the discussion above regarding constraint satisfaction.

- Find a balance between pre-compilation and runtime computation of the coordination behaviour, as discussed in §6.5.4. We utilise the extreme case where the behaviour of every building block is composed only at runtime, to better evaluate the pros and cons of our approach compared to static approaches. In practice, it makes sense to precompile the behaviour of some parts of the connector, in particular when these do not change over time and always require synchronous communication.

In Chapter 7, the last one of this thesis, we describe the communication between actors in more detail, and provide more technical details regarding the implementation of the Dreams framework. This implementation can still be improved in several ways. We list some possible directions.

- Integrate Dreams with existing web-service protocols, using, for example, SOAP messages.[1]

- Automate of the deployment process as suggested in §7.6.2.

- Benchmark the execution of connectors in a distributed network. That is, consider the time to send messages between different machines and not only between threads.

[1] `http://www.w3.org/TR/#tr_SOAP`

- Optimise the number of actors for a given connector, by composing simpler actors into more complex actors at compile time.
- Find automatically on which machine each actor should be deployed.
- Dynamically adapt a running connector based on properties of the network and demand for communication.
- Model failure, that is, describe how the system should behave if, at any time, the connection between two actors is broken.

In conclusion, the Dreams framework answers questions raised concerning the applicability of synchronous coordination models: are they too complex to be practically useful? The techniques proposed in this thesis provide increased performance of synchronous coordination systems by diminishing the computational complexity of their execution. We believe our contributions of exploiting constraint-solving and introducing independent asynchrony indeed make synchronous coordination practically useful.

Bibliography

[1] Gul Agha. *Actors: a model of concurrent computation in distributed systems*. MIT Press, Cambridge, MA, USA, 1986. 1, 6.1, 6.2, 8

[2] Gul Agha and Prasanna Thati. An algebraic theory of actors and its application to a simple object-based language. In Olaf Owe, Stein Krogdahl, and Tom Lyche, editors, *Essays in Memory of Ole-Johan Dahl*, volume 2635 of *Lecture Notes in Computer Science*, pages 26–57. Springer, 2004. 6.1, 6.2, 7.2

[3] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986. 3.7

[4] Jean-Marc Andreoli, Steve Freeman, and Remo Pareschi. The coordination language facility: coordination of distributed objects. *Theory and Practice of Object Systems*, 2(2):77–94, 1996. 5.10

[5] Jean-Marc Andreoli and Remo Pareschi. Linear objects: logical processes with built-in inheritance. *New Generation Computing*, pages 495–510, 1990. 5.10

[6] Krzysztof Apt. *Principles of Constraint Programming*. Cambridge University Press, 2003. 5.7

[7] Farhad Arbab. Coordination of mobile components. *Electronic Notes in Theoretical Computer Science*, 54, 2001. 1

[8] Farhad Arbab. Reo: a channel-based coordination model for component composition. *Mathematical Structures in Computer Science*, 14(3):329–366, 2004. 1, 2.1, 2.2, 2.2.3, 3, 3.7, 4.1, 4.2, 5.1, 5.2, 5.5, 5.9, 5.9.1, 6.5.2

[9] Farhad Arbab. Abstract behavior types: a foundation model for components and their composition. *Science of Computer Programming*, 55:3–52, 2005. 2.1, 2.2, 4.2, 4.7, 5.2, 5.9.1, 6.5.2

[10] Farhad Arbab. *Composition of Interacting Computations*, chapter 12, pages 277–321. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006. 2.1, 3.7, 5.10, 6.6

[11] Farhad Arbab. Elements of interaction. In Marc Aiguier, Francis Bretaudeau, and Daniel Krob, editors, *Complex Systems Design and Management*, pages 1–28. Springer Berlin Heidelberg, October 2010. 5.1

[12] Farhad Arbab, Lacramioara Astefanoaei, Frank S. de Boer, Mehdi Dastani, John-Jules Ch. Meyer, and Nick A. M. Tinnemeier. Reo connectors as coordination artifacts in 2apl systems. In The Duy Bui, Tuong Vinh Ho, and Quang-Thuy Ha, editors, *PRIMA*, volume 5357 of *Lecture Notes in Computer Science*, pages 42–53. Springer, 2008. 1

[13] 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. 1, 3.8, 5.9.1

[14] Farhad Arbab, Roberto Bruni, Dave Clarke, Ivan Lanese, and Ugo Montanari. Tiles for Reo. In Andrea Corradini and Ugo Montanari, editors, *Recent Trends in Algebraic Development Techniques*, volume 5486 of *Lecture Notes in Computer Science*, pages 37–55. Springer Berlin/Heidelberg, 2009. 3.1, 4.1, 4.7, 4.8, 5.9.1, 6.5.4

[15] Farhad Arbab, Tom Chothia, Sun Meng, and Young-Joo Moon. Component connectors with QoS guarantees. In Amy L. Murphy and Jan Vitek, editors, *COORDINATION*, volume 4467 of *Lecture Notes in Computer Science*, pages 286–304. Springer, 2007. 1, 3.8, 4.7, 5.9.1

[16] Farhad Arbab, Christian Koehler, Ziyan Maraikar, Young-Joo Moon, and José Proença. Modeling, testing and executing Reo connectors with the Eclipse Coordination Tools. In *Proceedings of FACS*, SCP, 2008. Tools available at `http://reo.project.cwi.nl/cgi-bin/trac.cgi/reo/wiki/Tools`. 1, 1, 2.2, 3.6.1, 4.1, 4.6.5, 5.8, 5.9, 5.9.2, 7.5, 7.6.1

[17] Farhad Arbab, Natallia Kokash, and Sun Meng. Towards using Reo for compliance-aware business process modeling. In Tiziana Margaria and Bernhard Steffen, editors, *ISoLA*, volume 17 of *Communications in Computer and Information Science*, pages 108–123. Springer, 2008. 1

[18] Farhad Arbab and Eric Monfroy. Coordination of heterogeneous distributed cooperative constraint solving. *SIGAPP Applied Computing Review*, 6(2):4–17, 1998. 5.10

[19] 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 *Lecture Notes in Computer Science*, pages 34–55. Springer, 2002. 2.2.3

[20] Farhad Arbab, Meng Sun, and Christel Baier. Synthesis of Reo circuits from scenario-based specifications. *Electronic Notes in Theoretical Computer Science*, 229(2):21–41, 2009. 1

[21] Christel Baier. Probabilistic models for Reo connector circuits. *Journal of Universal Computer Science*, 11(10):1718–1748, 2005. 1, 3.8, 5.9.1

[22] Christel Baier, Marjan Sirjani, Farhad Arbab, and Jan J. M. M. Rutten. Modeling component connectors in Reo by constraint automata. *Science of Computer Programming*, 61(2):75–113, 2006. 2, 2.2.1, 2.2.2, 2.2.3, 2.2.6, 3.4.1, 3.6.1, 4.7, 5.3.2, 5.4.1, 5.4.3, 5.4.3, 3, 5.9.1, 6.5.2

[23] Christel Baier and Verena Wolf. Stochastic reasoning about channel-based component connectors. In Paolo Ciancarini and Herbert Wiklicky, editors, *COORDINATION*, volume 4038 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 2006. 1, 3.8, 5.9.1

[24] David E. Bakken and Richard D. Schlichting. Supporting fault-tolerant parallel programming in Linda. *IEEE Transactions on Parallel and Distributed Systems*, 6, 1995. 6.6

[25] Alberto Baragatti, Roberto Bruni, Hernán Melgratti, Ugo Montanari, and Giorgio Spagnolo. Prototype platforms for distributed agreements. *Electronic Notes in Theoretical Computer Science*, 180(2):21–40, 2007. 6.6

[26] Lorenzo Bettini, Viviana Bono, Rocco De Nicola, Gianluigi Ferrari, Daniele Gorla, Michele Loreti, Eugenio Moggi, Rosario Pugliese, Emilio Tuosto, and Betti Venneri. The Klaim project: Theory and practice. In *Global Computing: Programming Environments, Languages, Security and Analysis of Systems*, volume 2874 of *Lecture Notes in Computer Science*, pages 88–150. Springer-Verlag, 2003. 2.3, 5.10

[27] Robert D. Bjornson. *Linda on distributed memory multiprocessors*. PhD thesis, Yale University, New Haven, CT, USA, 1993. 6.6

[28] Marcello M. Bonsangue, Farhad Arbab, Jaco de Bakker, Jan J. M. M. Rutten, Adriano Scutellà, and Gianluigi Zavattaro. A transition system semantics for the control-driven coordination language Manifold. *Theoretical Computer Science*, 240(1):3–47, 2000. 2.1, 5.10

[29] Marcello M. Bonsangue, Dave Clarke, and Alexandra Silva. Automata for context-dependent connectors. In John Field and Vasco Thudichum Vasconcelos, editors, *COORDINATION*, volume 5521 of *Lecture Notes in Computer*

Science, pages 184–203. Springer, 2009. 2.2.3, 2.2.8, 2.2.3, 2.2.3, 3.4.1, 4.7, 4.8, 5.5, 5.9.1

[30] BPEL4WS. *Business Process Execution Language for Web Services*, May 2003. 6.6

[31] Jarvis Dean Brock and William Ackerman. Scenarios: A model of non-determinate computation. In J. Díaz and I. Ramos, editors, *Formalization of Programming Concepts*, volume 107 of *Lecture Notes in Computer Science*, pages 252–259. Springer Berlin / Heidelberg, 1981. 10.1007/3-540-10699-5-102. 3.7

[32] Roberto Bruni, Cosimo Laneve, and Ugo Montanari. Orchestrating transactions in Join calculus. In Lubos Brim, Petr Jancar, Mojmír Kretínský, and Antonín Kucera, editors, *CONCUR*, volume 2421 of *Lecture Notes in Computer Science*, pages 321–337. Springer, 2002. 6.6, 7.4.3

[33] Paolo Ciancarini, Keld K. Jensen, and Daniel Yankelevich. On the operational semantics of a coordination language. In Paolo Ciancarini, Oscar Nierstrasz, and Akinori Yonezawa, editors, *Object-Based Models and Languages for Concurrent Systems*, volume 924 of *Lecture Notes in Computer Science*, pages 77–106. Springer-Verlag, 1995. 2.3, 2.3

[34] Dave Clarke. Coordination: Reo, nets, and logic. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever, editors, *FMCO*, volume 5382 of *Lecture Notes in Computer Science*, pages 226–256. Springer, 2007. 4.7, 5.3.2, 5.9.1, 5.10

[35] Dave Clarke. A basic logic for reasoning about connector reconfiguration. *Fundamenta Informaticae*, 82(4):361–390, 2008. 1, 5.9.1, 5.9.1

[36] Dave Clarke, David Costa, and Farhad Arbab. Modelling coordination in biological systems. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods*, volume 4313 of *Lecture Notes in Computer Science*, pages 9–25. Springer Berlin, 2006. 1

[37] Dave Clarke, David Costa, and Farhad Arbab. Connector colouring I: Synchronisation and context dependency. *Science of Computer Programming*, 66(3):205–225, May 2007. 1, 2, 2.2.1, 2.2.3, 3.4.1, 4.1, 4.2, 2, 4.3, 4.4.4, 4.7, 5.1, 5.5, 5.5.1, 5.5.2, 5.5.3, 5.6, 5.8, 5.9.1, 5.9.2, 6.5, 6.5.4

[38] Dave Clarke and José Proença. Coordination via interaction constraints i: Local logic. *CoRR*, abs/0911.5445, 2009. 5.11, 8

[39] Dave Clarke, José Proença, Alexander Lazovik, and Farhad Arbab. Deconstructing Reo. In *Proceedings of the International Workshop on the Foundations of*

Coordination Languages and Software Architecture (FOCLASA). Elsevier, 2008. 4.7, 5.11

[40] Dave Clarke, José Proença, Alexander Lazovik, and Farhad Arbab. Channel-based coordination via constraint satisfaction. *Science of Computer Programming*, In Press, Accepted Manuscript, 2010. 4.7, 6.5.4, 8

[41] David Costa. *Formal Methods for Component Connectors*. PhD thesis, Vrije Universiteit Amsterdam, 2010. 1, 2.2.3, 4.1, 4.1, 4.6, 4.8, 5.9.1

[42] Pierre J. Courtois, F. Heymans, and David L. Parnas. Concurrent control with "readers" and "writers". *Commun. ACM*, 14(10):667–668, 1971. 3.7

[43] Régis Cridlig and Eric Goubault. Semantics and analysis of linda-based languages. In Patrick Cousot, Moreno Falaschi, Gilberto Filé, and Antoine Rauzy, editors, *WSA*, volume 724 of *Lecture Notes in Computer Science*, pages 72–86. Springer, 1993. 2.3, 2.3

[44] Allen B. Downey. *The Little Book of Semaphores*. Green Tea Press, second edition, 2007. 3.7

[45] Sol Efroni, David Harel, and Irun R. Cohen. Reactive Animation: Realistic modelling of complex dynamics systems. *Computer*, 38(1):38–47, 2005. 4.7

[46] Kees Everaars, David Costa, Nikolay Diakov, and Farhad Arbab. A distributed computational model for Reo. Technical Report SEN-E0601, CWI, Amsterdam, The Netherlands, February 2006. 5.9

[47] Ming-Dong Feng, Weng-Fai Wong, and Chung-Kwong Yuen. Balinda lisp: Design and implementation. *Computer Languages*, 22(4):205–214, 1996. 2.3

[48] José Luiz Fiadeiro, Antónia Lopes, and Laura Bocchi. A formal approach to service component architecture. In Mario Bravetti, Manuel Núñez, and Gianluigi Zavattaro, editors, *WS-FM*, volume 4184 of *Lecture Notes in Computer Science*, pages 193–213. Springer, 2006. 5.10

[49] Nate Foster, Michael B. Greenwald, Christian Kirkegaard, Benjamin C. Pierce, and Alan Schmitt. Exploiting schemas in data synchronization. In Gavin M. Bierman and Christoph Koch, editors, *DBPL*, volume 3774 of *Lecture Notes in Computer Science*, pages 42–57. Springer, 2005. 3.7

[50] Cédric Fournet and Georges Gonthier. The join calculus: A language for distributed mobile programming. In Gilles Barthe, Peter Dybjer, Luis Pinto, and João Saraiva, editors, *APPSEM*, volume 2395 of *Lecture Notes in Computer Science*, pages 268–332. Springer, 2000. 6.6, 7.4.3

[51] Eric Freeman, Ken Arnold, and Susanne Hupfer. *JavaSpaces Principles, Patterns, and Practice*. Addison-Wesley Longman Ltd., Essex, UK, UK, 1999. 2.3, 5.10, 6.1

[52] Svend Frølund. *Coordinating Distributed Objects*. The MIT Press, Cambridge, Massachusetts, USA, 1996. 5.10

[53] Fabio Gadducci and Ugo Montanari. *The tile model*, pages 133–166. MIT Press, Cambridge, MA, USA, 2000. 3.1, 5.9.1, 6.5.4

[54] David Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985. 2.1, 2.3, 2.3, 3.6.3, 5.10, 6.6, 7.6

[55] Rachid Guerraoui and Luís Rodrigues. *Introduction to Reliable Distributed Programming*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006. 6.1, 7.4.4

[56] Juan Visente Guillen Scholten. *Mobile channels for exogenous coordination of distributed systems : semantics, implementation and composition*. PhD thesis, LIACS, Faculty of Mathematics and Natural Sciences, Leiden University, January 2007. 5.9

[57] Philipp Haller and Martin Odersky. Scala actors: Unifying thread-based and event-based programming. *Theoretical Computer Science*, 410(2-3):202–220, 2009. 1, 7.2, 7.6

[58] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987. 4.7

[59] David Harel. On visual formalisms. *Communications of the ACM*, 31(5):514–530, 1988. 4.7

[60] David Harel, Sol Efroni, and Irun R. Cohen. Reactive Animation. In *FMCO 2002*, volume 2852 of *Lecture Notes In Computer Science*, pages 13–153. Springer Verlag, 2003. 4.7

[61] David Harel and Hillel Kugler. The Rhapsody Semantics of Statecharts (or, on the executable core of the UML). *Lecture notes in computer science*, 2004. 4.7

[62] David Harel and P. S. Thiagarajan. *Message sequence charts*, pages 77–105. Kluwer Academic Publishers, Norwell, MA, USA, 2003. 4.7

[63] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor formalism for artificial intelligence. In *IJCAI*, pages 235–245, 1973. 6.2

[64] Mohammad Mahdi Jaghoori, Ali Movaghar, and Marjan Sirjani. Modere: the model-checking engine of Rebeca. In Hisham Haddad, editor, *SAC*, pages 1810–1815. ACM, 2006. 7.4.4

[65] Mohammad Mahdi Jaghoori, Marjan Sirjani, Mohammad Reza Mousavi, Ehsan Khamespanah, and Ali Movaghar. Symmetry and partial order reduction techniques in model checking rebeca. *Acta Informatica*, 47(1):33–66, 2010. 7.4.4

[66] Gilles Kahn. The semantics of simple language for parallel programming. In *IFIP Congress*, pages 471–475, 1974. 3.7

[67] Ramtin Khosravi, Marjan Sirjani, Nesa Asoudeh, Shaghayegh Sahebi, and Hamed Iravanchi. Modeling and analysis of Reo connectors using Alloy. In Doug Lea and Gianluigi Zavattaro, editors, *COORDINATION*, volume 5052 of *Lecture Notes in Computer Science*, pages 169–183. Springer, 2008. 4.7, 5.9.1, 5.9.2, 6.5.4

[68] David Kitchin, William R. Cook, and Jayadev Misra. A language for task orchestration and its semantic properties. In Christel Baier and Holger Hermanns, editors, *CONCUR*, volume 4137 of *Lecture Notes in Computer Science*, pages 477–491. Springer, 2006. 5.10, 7.6

[69] Sascha Klüppelholz and Christel Baier. Symbolic model checking for channel-based component connectors. *Electronic Notes in Theoretical Computer Science*, 175(2):19–37, 2007. 5.4.1, 5.10

[70] Christian Koehler and Dave Clarke. Decomposing port automata. In *SAC '09: Proceedings of the 2009 ACM symposium on Applied Computing*, pages 1369–1373, New York, NY, USA, 2009. ACM. 4.7, 5.9.1, 7.5

[71] Christian Koehler, David Costa, José Proenca, and Farhad Arbab. Reconfiguration of Reo connectors triggered by dataflow. In *GT-VMT'08: Proc. 7th International Workshop on Graph Transformation and Visual Modeling Techniques*, volume 10 of *Electronic Communications of the EASST*, 2008. 1, 5.9.1

[72] Christian Koehler, Alexander Lazovik, and Farhad Arbab. Connector rewriting with high-level replacement systems. *Electronic Notes in Theoretical Computer Science*, 194(4):77–92, 2008. 1

[73] Joost N. Kok. Denotational semantics of nets with nondeterminism. In Bernard Robinet and Reinhard Wilhelm, editors, *ESOP*, volume 213 of *Lecture Notes in Computer Science*, pages 237–249. Springer, 1986. 3.7

[74] Joost N. Kok. A fully abstract semantics for data flow nets. In J. W. de Bakker, A. J. Nijman, and Philip C. Treleaven, editors, *PARLE (2)*, volume 259 of *Lecture Notes in Computer Science*, pages 351–368. Springer, 1987. 3.7

[75] Natallia Kokash, Christian Krause, and Erik P. de Vink. Data-aware design and verification of service compositions with Reo and mCRL2. In *SAC'10: Proc. of the 2010 ACM Symposium on Applied Computing*, pages 2406–2413, New York, NY, USA, 2010. ACM. 1

[76] Christian Krause. *Reconfigurable component connectors*. PhD thesis, LIACS, Faculty of Mathematics and Natural Sciences, Leiden University, 2011. 1, 5.9.1

[77] Christian Krause, Ziyan Maraikar, Alexander Lazovik, and Farhad Arbab. Modeling dynamic reconfigurations in Reo using high-level replacement systems. *Science of Computer Programming*, 76(1):23–36, 2011. 1, 5.9.1

[78] Alexander Lazovik, Marco Aiello, and Rosella Gennari. Choreographies: Using constraints to satisfy service requests. In *AICT-ICIW '06: Proceedings of the Advanced International Conference on Telecommunications and International Conference on Internet and Web Applications and Services*, page 150, Washington, DC, USA, 2006. IEEE Computer Society. 5.10

[79] Ziyan Maraikar, Alexander Lazovik, and Farhad Arbab. Building mashups for the enterprise with SABRE. In Athman Bouguettaya, Ingolf Krüger, and Tiziana Margaria, editors, *ICSOC*, volume 5364 of *Lecture Notes in Computer Science*, pages 70–83, 2008. 1, 5.9.2, 5.10, 6.5.4, 6.6

[80] Antoni Mazurkiewicz. *Concurrent program schemes and their interpretations*. Aarhus Universitet. Department of Computer Science; DAIMI-PB 78. Aarhus, 1977. 2

[81] Sun Meng and Farhad Arbab. On resource-sensitive timed component connectors. In Marcello M. Bonsangue and Einar Broch Johnsen, editors, *FMOODS*, volume 4468 of *Lecture Notes in Computer Science*, pages 301–316. Springer, 2007. 1, 3.8, 5.9.1

[82] Sun Meng and Farhad Arbab. Web services choreography and orchestration in Reo and constraint automata. In *SAC '07: Proceedings of the 2007 ACM symposium on Applied computing*, pages 346–353, New York, NY, USA, 2007. ACM. 1

[83] Robin Milner. *A Calculus of Communicating Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1982. 1

[84] Robin Milner. Calculi for synchrony and asynchrony. *Theoretical Computer Science*, 25:267–310, 1983. 4.7

[85] Robin Milner. *Communicating and Mobile Systems: the Pi-Calculus*. Cambridge University Press, June 1999. 1, 5.10

[86] Naftaly H. Minsky and Victoria Ungureanu. Law-governed interaction: a coordination and control mechanism for heterogeneous distributed systems. *ACM Transactions on Software Engineering and Methodology*, 9(3):273–305, 2000. 5.10, 6.6

[87] Ugo Montanari and Francesca Rossi. Modeling process coordination via tiles, graphs, and constraints. In *3rd Biennial World Conference on Integrated Design and Process Technology*, volume 4, pages 1–8, 1998. 5.10

[88] Young-Joo Moon, Alexandra Silva, Christian Krause, and Farhad Arbab. A compositional semantics for stochastic Reo connectors. In Mohammad Reza Mousavi and Gwen Salaün, editors, *FOCLASA*, volume 30 of *EPTCS*, pages 93–107, 2010. 1, 3.8, 5.9.1

[89] MohammadReza Mousavi, Marjan Sirjani, and Farhad Arbab. Formal semantics and analysis of component connectors in Reo. *Electronic Notes in Theoretical Computer Science*, 154(1):83–99, 2006. 4.7, 5.9.1, 5.9.1, 5.9.2, 6.5.4

[90] George A. Papadopoulos and Farhad Arbab. Coordination models and languages. In *M. Zelkowitz (Ed.), The Engineering of Large Systems*, volume 46 of *Advances in Computers*, pages 329–400. Academic Press, 1998. 1, 2.1, 2.3, 5.10, 6.6

[91] Carl Adam Petri and Wolfgang Reisig. Petri net. *Scholarpedia*, 3(4):6477, 2008. 1, 4.7

[92] José Proença and Dave Clarke. Coordination Models Orc And Reo Compared. In *Proceedings of the International Workshop on the Foundations of Coordination Languages and Software Architecture (FOCLASA)*. Elsevier, 2007. 5.10

[93] Vijay A. Saraswat. *Concurrent constraint programming*. MIT Press, Cambridge, MA, USA, 1993. 5.10

[94] Vijay A. Saraswat, Radhakrishnan Jagadeesan, and Vineet Gupta. Timed default concurrent constraint programming. *Journal of Symbolic Computation*, 22(5–6):475–520, November–December 1996. Extended abstract appeared in the *Proceedings of the 22nd ACM Symposium on Principles of Programming Languages*, San Francisco, January 1995. 5.10

[95] Edward Joseph Segall. *Tuple space operations: multiple-key search, on-line matching and wait-free synchronization*. PhD thesis, Department of Electrical and Computer Engineering, Rutgers University, New Brunswick, NJ, USA, 1993. 2.3

[96] Hossein M. Sheini and Karem A. Sakallah. From propositional satisfiability to satisfiability modulo theories. In Armin Biere and Carla P. Gomes, editors, *SAT*, volume 4121 of *Lecture Notes in Computer Science*, pages 1–9. Springer, 2006. 5.3.2

[97] Geoff Sutcliffe. Prolog-d-linda v2: A new embedding of linda in SICStus prolog. In *Proc. Workshop on Blackboard-based Logic Programming*, pages 105–117, 1993. 2.3

[98] Hung Tran Van, Axel van Lamsweerde, Philippe Massonet, and Christophe Ponsard. Goal-oriented requirements animation. In *RE*, pages 218–228. IEEE Computer Society, 2004. 4.7

[99] Wil M. P. van der Aalst, Arthur H. M. ter Hofstede, Bartek Kiepuszewski, and Alistair P. Barros. Workflow patterns. *Distributed Parallel Databases*, 14(1):5–51, 2003. 2.2.1, 4.7, 7.5.1

[100] K. N. Venkataraman. Decidability of the purely existential fragment of the theory of term algebras. *J. ACM*, 34(2):492–510, 1987. 5.3.1

[101] Peter Wegner. Coordination as constrained interaction (extended abstract). In *Coordination Languages and Models*, volume 1061 of *Lecture Notes in Computer Science*, pages 28–33, 1996. 5.1, 5.10

[102] Michael Westergaard and Kristian Bisgaard Lassen. The BRITNeY Suite Animation Tool. In Susanna Donatelli and P. S. Thiagarajan, editors, *ICATPN*, volume 4024 of *Lecture Notes in Computer Science*, pages 431–440. Springer, 2006. 4.7

[103] Min Xie. Specification Of E-Business Process Model For PayPal Online Payment Process Using Reo. Master's thesis, Leiden University, the Netherlands, 2005. 1

[104] Makoto Yokoo. *Distributed constraint satisfaction: foundations of cooperation in multi-agent systems*. Springer-Verlag, London, UK, 2001. 6.4, 6.6

Index

Summary

Coordination is a relatively recent field, considerably inspired by concurrency theory. Coordination languages and models are based on the philosophy that an application or a system should be divided into the parts that perform computations, typically components or services, and the parts that coordinate the results and resources required to perform the computations. The coordination aspect focuses on the latter, describing how the components or services are connected. We study a specific class of coordination models, namely synchronous, exogenous, and composable models, and we exploit implementation techniques for such models in distributed environments. Our work concentrates on the $\mathcal{R}$eo coordination model as the main representative of this class of coordination models.

Current engines that execute $\mathcal{R}$eo allow the coordination layer to run only in a single thread of execution, although the components can execute in parallel or on a distributed platform. Furthermore, due to the synchrony aspect these engines only support small systems, and do not scale. To address these limitations, our approach to implement $\mathcal{R}$eo-like models makes a tradeoff between pre-compiling the possible behaviour and calculating it at runtime.

Our work contributes to the field of coordination, in particular to $\mathcal{R}$eo, by improving existing approaches to execute synchronisation models in three major ways. First, this work supports *decoupled execution* and *lightweight reconfiguration*. We introduce a distributed protocol that allows actors to reach consensus about data exchange, and performs the actual communication of data. We developed a prototype Dreams engine to test this protocol, using an actor library for the Scala language. Reconfiguration of a small part of the system is independent of the execution or behaviour of unrelated parts of the same system. Second, Dreams *outperforms* previous $\mathcal{R}$eo engines by using *constraint satisfaction* techniques. In each round of the execution of the Dreams framework, descriptions of the behaviour of all building blocks are combined and a coordination pattern for the current round is chosen. This choice is made using constraint satisfaction techniques, requiring less time than previous approaches that collect all patterns before selecting one.

Third, our work improves *scalability* by identifying *synchronous regions*. We statically discover regions of the coordination layer that can execute independently, thus achieving a truly decoupled execution of connectors. Consequently, the constraint problem representing the behaviour at each round is smaller and more easily solved.

Samenvatting

Coördinatie is een relatief jong onderzoeksgebied dat aanzienlijk geïnspireerd is door *concurrency theory*. Coördinatietalen en -modellen zijn gebaseerd op de filosofie dat een toepassing of een systeem moet worden opgesplitst, enerzijds in de onderdelen die berekeningen uitvoeren, meestal componenten of diensten genoemd, en anderzijds onderdelen die de berekeningen en de middelen die hiervoor nodig coördineren. Het coördinatiegedeelte richt zich op het laatste en beschrijft hoe de componenten of diensten verbonden zijn. We bestuderen een specifieke klasse van coördinatiemodellen, namelijk synchrone, exogene en samengestelde modellen, en we verkennen de implementatietechnieken voor dergelijke modellen in gedistribueerde omgevingen. Ons werk concentreert zich op het $\mathcal{R}$eo-coördinatiemodel als de belangrijkste vertegenwoordiger van deze klasse van de coördinatiemodellen. De huidige *engines* die $\mathcal{R}$eo uitvoeren laten de coördinatielaag slechts toe om in n enkele thread te draaien hoewel componenten parallel of op een gedistribueerd platform uitgevoerd kunnen worden. Bovendien ondersteunen deze *engines*, vanwege synchronisatie, alleen kleine systemen en zijn ze slecht schaalbaar. Om deze beperkingen te verzachten maakt onze aanpak om $\mathcal{R}$eo en vergelijkbare modellen te implementeren een afweging tussen het voorcompileren van het mogelijke gedrag enerzijds en het te berekenen tijdens de uitvoeringsfase anderzijds.

Ons werk draagt bij aan het onderzoek naar coördinatie, en in het bijzonder betreffende $\mathcal{R}$eo, door bestaande aanpakken om synchronisatiemodellen te executeren te verbeteren op drie belangrijke vlakken. Ten eerste ondersteunt het werk ontkoppelde uitvoering en lichtgewicht herconfiguratie. We introduceren een gedistribueerd protocol waarmee actoren consensus kunnen bereiken over de uitwisseling van gegevens, en dat de eigenlijke communicatie van de gegevens uitvoert. We ontwikkelden een prototype, de Dreams *engine*, om dit protocol te testen met behulp van een bibliotheek voor de Scala-taal. Herconfiguratie van een klein deel van het systeem is onafhankelijk van de uitvoering of het gedrag van niet-verbonden delen van hetzelfde systeem. Ten tweede, is Dreams een verbe-

tering ten opzichte van eerdere $\mathcal{R}$eo *engines* door het gebruik van *constraint satisfaction*-technieken. In elke uitvoeringsronde van het Dreams-raamwerk worden beschrijvingen van het gedrag van alle bouwstenen gecombineerd en wordt een coördinatiepatroon voor de huidige ronde gekozen. Deze keuze wordt gemaakt met behulp van *constraint solving* dat tijdsefficienter is in vergelijking met andere aanpakken, waarbij alle patronen verzameld worden alvorens een te selecteren. Ten derde verbetert ons werk de schaalbaarheid door het identificeren van synchrone regio's. Er wordt statisch naar regio's gezocht die zelfstandig uitgevoerd kunnen worden en een ontkoppelde uitvoering van connectoren mogelijk is. Daardoor is het *constraint satisfaction*-probleem dat het gedrag bij elke ronde representeert kleiner en eenvoudig op te lossen.

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 Multi-disciplinary 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

M.M. Jaghoori. *Time at Your Service: Schedulability Analysis of Real-Time and Distributed Services*. Faculty of Mathematics and Natural Sciences, UL. 2010-11

R. Bakhshi. *Gossiping Models: Formal Analysis of Epidemic Protocols*. Faculty of Sciences, Department of Computer Science, VUA. 2011-01

B.J. Arnoldus. *An Illumination of the Template Enigma: Software Code Generation with Templates*. Faculty of Mathematics and Computer Science, TU/e. 2011-02

E. Zambon. *Towards Optimal IT Availability Planning: Methods and Tools*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-03

L. Astefanoaei. *An Executable Theory of Multi-Agent Systems Refinement*. Faculty of Mathematics and Natural Sciences, UL. 2011-04

J. Proença. *Synchronous Coordination of Distributed Components*. Faculty of Mathematics and Natural Sciences, UL. 2011-05