Universiteit
Leiden
The Netherlands

Reconfigurable component connectors

Krause, C.

Citation
Krause, C. (2011, June 21). *Reconfigurable component connectors*. *IPA Dissertation Series*. Retrieved from https://hdl.handle.net/1887/17718

Version:	Corrected Publisher's Version
License:	Licence agreement concerning inclusion of doctoral thesis in the Institutional Repository of the University of Leiden
Downloaded from:	https://hdl.handle.net/1887/17718

Note: To cite this publication please use the final published version (if applicable).

Reconfigurable Component Connectors

Christian Krause

Promotiecommissie

Promotor:	Prof. Dr. F. Arbab	Universiteit Leiden
Copromotor:	Dr. E.P. de Vink	TU Eindhoven
Other members:	Prof. Dr. W.M.P. van der Aalst	TU Eindhoven
	Prof. Dr. F. de Boer	Universiteit Leiden
	Dr. M. Bonsangue	Universiteit Leiden
	Prof. Dr. J.N. Kok	Universiteit Leiden
	Prof. Dr. G. Taentzer	Phillips Universität Marburg

The work in this thesis has been carried out at the Centrum Wiskunde & Informatica (CWI), under the auspices of the research school IPA (Institute for Programming research and Algorithmics). The author was funded by the NWO GLANCE project *Workflow Management for Large Parallel and Distributed Applications (WoMaLaPaDiA)*.

Copyright © 2011 Christian Krause
ISBN: 978–90–6464–475–7
IPA Dissertation Series 2011–08
Printed by Ponsen & Looijen

Reconfigurable Component Connectors

PROEFSCHRIFT

ter verkrijging van

de graad van Doctor aan de Universiteit Leiden,

op gezag van Rector Magnificus prof. mr. P.F. van der Heijden,

volgens besluit van het College voor Promoties

te verdedigen op dinsdag 21 juni 2011

klokke 15:00 uur

door

Christian Krause (geb. Köhler)

geboren te Chemnitz (Karl-Marx-Stadt), Duitsland

in 1981

The difficulty lies, not in the new ideas, but in escaping the old ones, which ramify, for those brought up as most of us have been, into every corner of our minds.

John Maynard Keynes, 1987 (quoted in [33])

CONTENTS

Contents **v**

1 Introduction **1**
1.1 Coordination models . . . 1
1.2 Component connectors in Reo . . . 2
1.3 Dynamic reconfiguration . . . 2
1.4 Formal methods and their use . . . 3
1.5 Goals of this thesis . . . 4
1.6 Thesis overview and contributions . . . 5

2 Channel-based coordination with Reo **7**
2.1 The Reo coordination language . . . 7
2.1.1 Channels . . . 7
2.1.2 Nodes . . . 8
2.1.3 Components . . . 9
2.1.4 Connectors and networks . . . 9
2.2 The Eclipse Coordination Tools . . . 12
2.2.1 Tools overview . . . 13
2.2.2 The Reo meta-model . . . 15
2.2.3 Custom primitives in ECT . . . 19
2.3 Conclusions . . . 23

3 Automata-based semantics for Reo **25**
3.1 Constraint and port automata . . . 25
3.1.1 Definition . . . 26
3.1.2 Join and hiding operations . . . 26
3.1.3 Bisimulation . . . 27
3.1.4 Port automata . . . 28
3.1.5 From Reo to automata models . . . 30
3.2 Decomposition of port automata . . . 31
3.2.1 Stateless port automata . . . 31
3.2.2 General decomposition scheme . . . 33
3.2.3 Related work . . . 36
3.3 The Extensible Automata framework in ECT . . . 38
3.3.1 The EA meta-model . . . 38
3.3.2 Extension providers . . . 39
3.3.3 Product providers . . . 40

3.3.4 From Reo to automata models . . . 42
3.3.5 CA runtime and code generation . . . 43
3.4 Conclusions . . . 44
3.5 Discussion . . . 45

4 Verification by model checking 47
4.1 Overview . . . 47
4.2 The mCRL2 specification language . . . 49
4.2.1 Actions . . . 49
4.2.2 Processes . . . 49
4.2.3 Data types . . . 50
4.2.4 Tools . . . 50
4.3 Encoding Reo in mCRL2 . . . 50
4.3.1 Join and hiding operations . . . 52
4.3.2 General port automata encoding . . . 53
4.3.3 Encoding of the coloring semantics . . . 55
4.3.4 Encoding context-dependency in port automata . . . 58
4.3.5 Encoding of Timed Reo . . . 59
4.4 Verification tools in ECT . . . 60
4.4.1 The mCRL2 conversion tool . . . 60
4.4.2 The Vereofy model checker . . . 64
4.4.3 Bounded model checking for timed constraint automata . . . 65
4.4.4 Stochastic analysis using PRISM . . . 66
4.4.5 Stochastic analysis based on discrete event simulation . . . 66
4.5 Related work . . . 67
4.6 Conclusions . . . 67

5 Reconfiguration by graph transformation 69
5.1 Motivation . . . 69
5.2 Reconfiguration by graph transformation . . . 70
5.2.1 Reo networks as typed hypergraphs . . . 71
5.2.2 Double pushout rewriting of Reo networks . . . 72
5.2.3 Critical pair analysis in AGG . . . 75
5.3 Modeling dynamic reconfiguration . . . 76
5.3.1 State space analysis in Henshin . . . 77
5.3.2 Transparent dynamic reconfiguration . . . 82
5.4 Support for reconfiguration in ECT . . . 84
5.5 Dynamic reconfiguration in ReoLive . . . 85
5.6 Reconfigurable coordination of YAWL workflows . . . 86
5.7 Related work . . . 87
5.8 Conclusions . . . 88

6 Distributed networks and reconfiguration 89
6.1 Motivation . . . 89

6.2 Distributed graphs and Reo networks 90
6.2.1 Distributed graphs 90
6.2.2 Extended typing for distributed Reo networks 92
6.3 Reconfiguring distributed networks 93
6.3.1 Local reconfigurations 93
6.3.2 Synchronizing local reconfigurations 94
6.3.3 Coordinating local reconfigurations 97
6.4 Flattening of distributed graphs 98
6.5 Related work 99
6.6 Conclusions 99

7 Distributed port automata **101**
7.1 Overview 101
7.2 The category of port automata 102
7.3 The category of distributed port automata 106
7.3.1 Encoding of Reo networks 107
7.3.2 Encoding of Petri nets 108
7.3.3 Composing distributed port automata 108
7.3.4 Semantics of distributed port automata 110
7.4 Towards dynamic reconfiguration 111
7.5 Related work 112
7.6 Conclusions and future work 113

8 Conclusions and further directions **115**

Bibliography **117**

A Proofs **127**
A.1 Proof for Theorem 4.4 127
A.2 Proof for Theorem 6.9 129

B Listings **131**
B.1 Vereofy library for context-dependent primitives 131

C Summary **133**

D Samenvatting (dutch) **135**

Chapter 1

Introduction

The large complexity of today's software systems is caused not only by their sheer size in terms of lines of code. In the age of service-oriented and cloud computing, applications use and integrate functionality from third-party providers, which can be distributed over the Internet, implemented in different languages and running on different types of hardware. Therefore, interoperability and robustness have become key requirements for building software. This raises the questions of not only how we compose different pieces of software, and how we can reason about the properties of the resulting systems, but also, how we can change or *reconfigure* applications at runtime. The modeling and formal analysis of reconfigurable software is the topic of this thesis.

1.1 Coordination models

The component-based and particularly the service-oriented design patterns are becoming increasingly relevant in modern software engineering. In both paradigms, applications are composed out of a set of smaller functional building blocks, i.e., components and services, respectively. The step from components to services involves –at the technical level– a proper handling of distribution and heterogeneity. Another important difference between the two approaches is that services are loosely coupled and the fact that they can be discovered dynamically, i.e., at runtime. However, the core ideas of bundling and black-boxing of functional software artifacts is inherent in both component-based and service-oriented systems. Thus, building software by composing a set of services or components is –at the conceptual level– very similar.

A major challenge in component-based as well as service-oriented software is the proper coordination of the active entities that comprise a system. Components are essentially self-contained functional building blocks. Services, furthermore, can be owned by third parties and are usually accessible only via published interfaces. Therefore, it is a natural choice to use an additional, external coordination mechanism to control the interaction between components and services from outside. Exogenous

coordination models and languages provide the means to specify this external coordination mechanism explicitly. This is particularly important in order to avoid ad hoc composition approaches. While the functional aspects of an application are implemented using components, their coordination can be realized using so-called component *connectors*, which control the way the components communicate with each other. Component connectors provide the *glue code* that is required for describing the interaction in component-based and service-oriented software systems.

The implementation of component connectors involves a number of technical details, e.g. the use of inter-platform communication protocols and interchange formats. However, considering component connectors explicitly at the modeling level has the advantage that the structural and behavioral properties of a system can be described independently from its implementation. According to the paradigm of model-driven engineering, this enables the use of code generation for deriving implementations. Moreover, it also allows to use formal methods for verification. Thus, coordination models provide an appropriate abstraction for studying the interaction in component-based and service-oriented systems.

1.2 Component connectors in Reo

In this thesis, we study component connectors in the channel-based coordination language Reo [1]. Component connectors in Reo are built compositionally out of primitive channels that can be plugged together using nodes. These channel-based connectors control the dataflow between the components and, thereby, enforce communication protocols among them. This enforcement involves a number of different dataflow aspects, such as synchronous vs. asynchronous communication, buffering, filtering and data manipulation, causality, context-dependent behavior, and mobility. Due to its feature-rich models, Reo offers powerful means for describing the coordination that turns a set of components or services into a coherent working application.

Since the notion of channels in Reo supports mobility, problems in the area of dynamic reconfigurations can be studied concretely in the context of Reo. Moreover, Reo has formal semantics which enables the use of formal methods for verification.

1.3 Dynamic reconfiguration

To *re-configure* an application, i.e., to change its configuration, is a common task in all branches of software engineering and is not very challenging in itself. However, nowadays more and more domains emerge where it is crucial to be able to perform a reconfiguration dynamically, i.e., at runtime. This is particularly the case in areas where the components and services cannot simply be shutdown or restarted.

An example are the complex embedded systems used in the automotive industry. The software operating the hundreds of electronic control units in modern cars is required to run in different modes in which specific configurations and scheduling policies must be adhered to. Switching between different modes entails that all

components are synchronously updated and that they form –together– a consistent system state. This constitutes one of the key challenges in the area of dynamic reconfiguration. Another, less critical, application involves service-oriented systems that use third-party services which are not under the control of the owner of the application, and thus cannot be restarted or reset in general. A reconfiguration of a service-based application can become necessary if one of the used services suddenly becomes unavailable or its quality of service becomes unacceptable. In such a scenario, a reconfiguration can be as simple as switching to an alternative service implementation, or as challenging as to modify the complete application architecture.

While in some domains switching between a finite set of configurations is sufficient, in others a rule-based approach for reconfiguration is more suitable. For instance, the embedded software in modern cars, as mentioned above, usually operates in a fixed set of predefined modes. However, if the number of components in a software system is not known a priori (such as in peer-to-peer networks), switching between a finite set of configurations is not feasible. In such domains, reconfigurations that modify a system property, e.g. the topology of a network, based on interpreted reconfiguration rules are better suited. In general, rule-based reconfiguration provides a more powerful approach since it supports an unbounded number of different configurations.

As mentioned above, component connectors facilitate the description of the interaction between the services or components that comprise an application. As one of the central claims of this thesis, we believe that the study of systematic approaches for reconfiguration of component connectors is of major relevance, because in modern software, the assumptions of *static* configurations, protocols and even whole system layouts are no longer tenable.

1.4 Formal methods and their use

When a computer scientist is asked about the relevance of formal methods, the answer mainly depends on his or her personal opinion and background. While the need for modeling in general is more or less accepted in many industrial branches of computer science, the benefits of *formal* modeling and verification are to a large extent still valued only in the academic environment. For instance, in the industry, software specifications are commonly given in natural, rather than formal languages, and implementations are validated using testing, rather than model checking or theorem proving. Although the correctness of an application is considered as the most important quality measure of software in general, the choice against formal methods is often being justified by higher production costs or by the argument that they are not feasible in industrial applications.

The acceptance of formal modeling techniques depends greatly on their expressive power. For instance, in the area of embedded systems, issues involving real-time are of major importance. In service-oriented systems, quantitative aspects are relevant for quality of service assurances. However, the success of formal methods also depends to

a large extent on their usability and scalability. To apply formal methods in practice, it is crucial to have efficient algorithms and proper tool support for verification.

In the context of this thesis, we apply formal methods to problems that exist in the area of component coordination, with particular emphasis on verification of static and dynamically reconfigurable component connectors. For this purpose, we exploit methods from automata theory, process algebra, model checking, as well as graph transformation and category theory. Each of these approaches provides a means to solve a specific problem and is therefore properly motivated. As argued above, we believe that it is important for the proposed methods to be applicable in practice and, therefore, we focus on approaches with proper tool support. Although our examples are not of industrial strength, the modeling and verification tools discussed in this thesis have the potential of being applied in large scale settings.

1.5 Goals of this thesis

The central topic of this thesis is the formal modeling and analysis of *Reconfigurable Component Connectors* in Reo. Our goals can be summarized as follows:

Verification of static connectors

Based on existing formal semantical models for Reo, we show that formal verification of statically defined component connectors can –in practice– be done using model checking techniques.

Reconfigurable connectors

While the existing semantic models for Reo provide a solid basis for analyzing static connectors, we claim that they are inadequate for describing dynamically reconfigurable ones. As one of our central goals, we argue in this thesis that the theory of algebraic graph transformation provides a powerful framework for formal modeling of reconfigurable connectors. We justify the use of a formal approach to reconfiguration by showing how to analyze dynamic reconfiguration, again using model checking.

Distributed connectors

Centralized approaches to component coordination have a limited range of application. Therefore, we provide a framework for modeling distributed component connectors and their reconfiguration based on the theory of distributed graph transformation.

Integration of structure and semantics

We argue in our approach for reconfigurable connectors, that modeling the execution and reconfiguration in the same formalism, i.e., as graph transformation rules,

enables the use of model checking for formal verification of dynamic reconfigurations This approach, however, cannot always be applied since it lacks compositionality. Moreover, issues involving state transfer and state consistency in dynamic reconfiguration cannot be handled properly. Therefore, we consider also an alternative approach in which we integrate the graph transformation based techniques for reconfiguration with an existing semantic automata model of Reo.

Tool support

Tool support is crucial for all formal and informal modeling approaches. The verification techniques presented in this thesis are all implemented based on existing model checking tools. To provide a uniform development framework, an integrated environment for modeling and analysis of Reo connectors based on the Eclipse platform [35] has been implemented and is discussed in this thesis.

1.6 Thesis overview and contributions

We now give an overview of the contents of this thesis and sketch our contributions.

Chapter 2. In this chapter, we give an overview of the channel-based coordination language Reo and, in its second part, introduce the Eclipse Coordination Tools (ECT): a graphical development environment for Reo that has been implemented in the context of this thesis.

Chapter 3. In this chapter, we recall the constraint automata semantics for Reo and introduce port automata as an abstraction of constraint automata. We present a decomposition scheme for port automata which can be used for synthesis of Reo connectors. We then discuss a framework for automata-based models in ECT.

The decomposition scheme in this chapter is based on [90].

Chapter 4. In this chapter, we introduce an approach for model checking of Reo connectors using the behavioral specification language mCRL2 [49], based on the automata semantics presented in Chapter 3. Moreover, we give an overview of the verification tools available in ECT.

The mCRL2-based verification approach in this chapter is based on [65, 62, 63, 64]. The author of this thesis further contributed to two approaches for stochastic analysis in Reo, respectively based on so-called *stochastic Reo automata* [75, 74] and discrete event simulation [106]. Furthermore, the author contributed to a paper that shows that context-dependency in Reo can be encoded in basic two-color models (cf. [55]). We give an overview of this and related work.

Chapter 5. In this chapter, we present an approach for modeling and verification of dynamically reconfigurable connectors using the theory of algebraic graph

transformation. Thus, we drop the restriction to static connectors which was necessary for the verification approach in Chapter 4. To this end, we use the AGG [87] and Henshin [6] tools, and furthermore exploit validation using mCRL2 [49], CADP [43], PRISM [67] and OCL [76]. We consider behavioral, structural, as well as quantitative analysis of dynamic reconfigurations.

The material on modeling and verification of dynamic reconfigurations in this chapter is based on [98]. The Henshin transformation language and toolset used in this chapter were introduced in [6]. Additional work by the author of this thesis related to reconfiguration in Reo can be found in [91, 92, 93].

Chapter 6. In this chapter, we extend the modeling approach of Chapter 5 to distributed environments. Specifically, we show how to realize reconfiguration of distributed Reo connectors based on the theory of distributed graph transformation. Moreover, we show in this chapter that the flattening operation of distributed graphs is compositional.

The distributed reconfiguration approach in this chapter is based on [89]. The compositionality of the flattening functor was shown (among other things) in [96].

Chapter 7. In this chapter, we introduce *distributed port automata*, which integrate the automata based semantics in Chapter 3 and the graph transformation based approach for reconfiguration in Chapter 5. We show compositionality of the semantics of distributed port automata using a theorem for the flattening operation of distributed graphs presented in Chapter 6. The approach in this chapter has applications in the area of dynamic reconfiguration. Specifically, it allows to reason about the problems of state transfer and state consistency.

The results in this chapter are a generalization of the tailored approach in [95] and were first presented in [96].

Chapter 8 contains concluding remarks and future directions.

Chapter 2

Channel-based coordination with Reo

In this chapter, we recall the most important features and concepts of the coordination language Reo and give an introduction to the Eclipse Coordination Tools (ECT), which is an integrated development environment for Reo that has been implemented in the context of this thesis.

2.1 The Reo coordination language

The coordination paradigm [44] proposes to divide software systems into two orthogonal aspects:

(i) the computation performed by a set of autonomous components or services, and

(ii) their coordination using some kind of 'glue code'.

Reo [1] is a channel-based coordination language that provides means to construct such glue code. Coordination in Reo is performed using circuit-like connectors which are built out of primitive channels and nodes. These connectors coordinate components or services from outside and without their knowledge, which is also referred to as *exogenous* coordination. Reo connectors define and implement the allowed interactions between the active entities in a network by means of communication protocols. This includes aspects of concurrency, buffering, ordering, data flow and manipulation.

In the following, we recall the basic notions of Reo and give some introductory examples.

2.1.1 Channels

Channels in Reo are entities that have exactly two ends, which can be either *source* or *sink* ends. Source ends accept data into, and sink ends dispense data out of their channels. Reo allows directed channels as well as so-called *drain* and *spout* channels, which have respectively two source and two sink ends. Channels impose constraints on the data flow at their ends. In particular, they can synchronize or mutually exclude

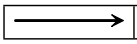
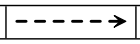
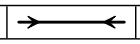
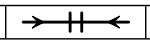
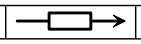
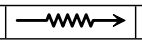
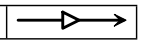
Sync	*LossySync*	*SyncDrain*	*AsyncDrain*	*FIFO1*	*Filter*	*Transform*
						

Table 2.1: graphical notations of some basic Reo channels

data flow, provide buffers or apply data transformations. Although channels can be defined by users in Reo, a set of basic channels suffices to implement rather complex coordination patterns.

A set of commonly used channels is summarized in Table 2.1. The *Sync* channel consumes data items at its source end and dispenses them at its sink end. The I/O operations are performed synchronously and without any buffering. Consequently, the channel blocks if the party at the sink end is not ready to receive any data. The *LossySync* channel behaves in the same way, except that it does not block the party at its source end. Instead, the data item is consumed and destroyed by the channel if the receiver is not ready to accept it. The *SyncDrain* channel is also a synchronous channel, but it differs in the fact that it has two source ends through which it consumes and destroys data items synchronously. Complementarily, the *AsyncDrain* consumes a data item from only one of its source ends and can therefore be used to realize a mutual exclusion. None of the channels considered so far buffer data items. Buffering can be implemented using the *FIFO1* channel, which is a directed, asynchronous channel with a buffer of size one. The *Filter* channel uses a data constraint, e.g. a regular expression, to decide whether a data item should be passed to the sink end or destroyed by the channel. Finally, the *Transform* channel applies a function to all data items and can therefore be used for data conversion.

2.1.2 Nodes

To construct connectors, channels can be joined together using nodes. A node can be of one out of three types: source, sink or mixed, depending on whether all coinciding channel ends are source ends, sink ends or a combination of both. Source and sink nodes together form the boundary of a connector, allowing interaction with its environment. A source node acts as a synchronous replicator, i.e., it atomically copies incoming data items to all of its outgoing source ends. On the other hand, a sink node acts as a non-deterministic merger, i.e., it randomly chooses a data item from one of the sink ends for delivery to its connected component. Mixed nodes combine both behaviors by atomically consuming a data item from one sink end and replicating it to all source ends. This can be seen as a 1:*n* synchronization, as opposed to 1:1 synchronizations (Milner style), or synchronizations of all coinciding channel ends (Hoare style). It is also important to stress that nodes do not perform any buffering. As a consequence, synchrony propagates through connectors, e.g. an arbitrarily long sequence of *Sync* channels has the same qualitative behavior as a single *Sync* channel.

2.1.3 Components

We use the term *component* for an active entity with a fixed interface that consists of a number of source and sink ends. Therefore, components can be seen as a generalization of channels where the requirement of having exactly two ends is dropped. The difference between channels and components is somewhat similar to the step from edges to hyperedges in graph theory. We refer to channels and components commonly as *primitives*.

Regarding the behavior of components, we often interpret them as black boxes, i.e., we do not make any assumptions about their behavior. However, for analysis it is often beneficial to take into account the behavior of components as well, e.g. to detect potential deadlocks or to validate temporal properties. For this purpose, we allow a component to be annotated with a specification that reflects its semantics. In general, we do not constrain the specification format. In practice, we mostly choose formal specification formats that can be used either for analysis or execution. The formats supported by our tools are discussed in detail in Section 2.2.3.

As the most basic examples of components we often consider simple *Writers* and *Readers*, which have respectively a single sink and a single source end through which they write and read data items. We allow these components to delay the data flow, e.g. *Writers* can provide no data items and *Readers* may refuse to accept data items. We do not make any fairness assumption in this context.

2.1.4 Connectors and networks

As elucidated above, channels and nodes can be used to construct connectors, which have a well-defined interface, i.e., their set of source and sink nodes. A connector can be considered as an open system, in that it offers interaction points to its environment which is not further specified. However, for analysis it is often beneficial to also have the knowledge of the components that are coordinated by the connector. We refer to a system of components and connectors without any dangling interfaces as a *network*, which contrary to connectors are closed systems where all constituents are known a priori. In the following, we discuss three example networks.

REMARK 2.1 (boundary nodes)**.** For clarity, we depict boundary nodes as open circles and mixed nodes as filled circles in all diagrams. ▽

EXAMPLE 2.2 (exclusive router)**.** Figure 2.1 shows a network containing three simple *Writer* and *Reader* components, which are coordinated by a connector called *exclusive router* (cf. [10]). This connector routes data items synchronously from the *Writer* to exactly one of the two *Readers*. If both of them are ready to accept data, the choice of where the data item goes is made non-deterministically. This is due to the fact that node *D* merges its inputs without priority, i.e., exactly one of the *Sync* channels is activated, the data item on the active side is replicated to the corresponding *Reader* and the data item on the other side is destroyed by its *LossySync*. Note that this connector never loses data items. △

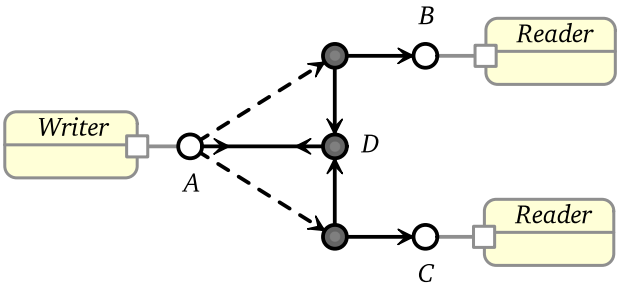


Figure 2.1: example network: exclusive router

EXAMPLE 2.3 (ordering)**.** Figure 2.2 shows a network of two *Writers*, one *Reader* and a connector referred to as *ordering* or *alternator* (cf. [1]). This connector enforces an ordered output of the data items provided by the two *Writers*. The *SyncDrain* is used to synchronize the inputs. The *FIFO1* stores the data item from *B* and makes it available in the next execution step. Since the *FIFO1* cannot store more than one data item, a stored data item has to be released first before new data items can be read. This way, an alternating output is guaranteed. △

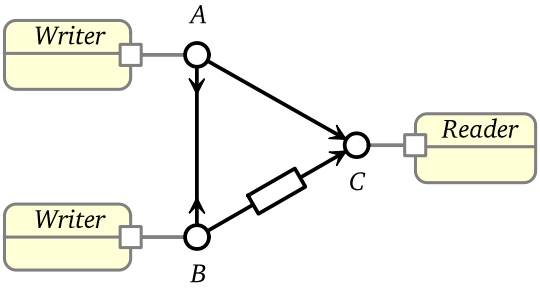
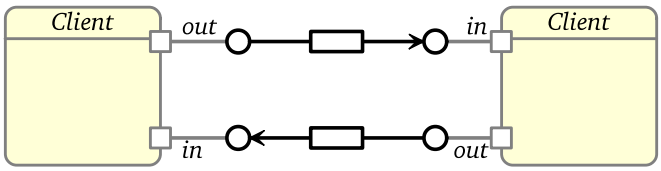
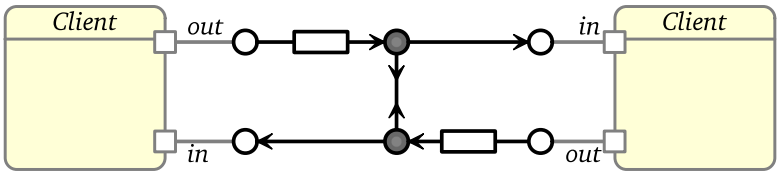


Figure 2.2: example network: ordering

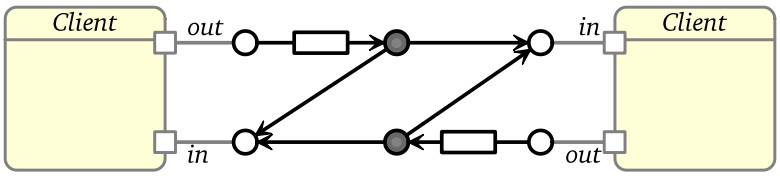
EXAMPLE 2.4 (instant messenger)**.** Figure 2.3 depicts three variations of a simple instant messenger application. Two *Client* components exchange messages via a connector. In variant 2.3a messages are simply exchanged using two buffered channels, in this case two *FIFO1* channels. In variant 2.3b an additional *SyncDrain* is used to ensure that message retrievals are synchronized, i.e., one client may receive a message only if it also sends one to the other client. Finally, variant 2.3c shows a case where the clients get –as an acknowledgment– a copy of their own message when the other client has successfully received it. △



(a) simple message passing using *FIFO1* buffers



(b) synchronized message retrieval



(c) messenger with automatic acknowledgments

Figure 2.3: example network: instant messenger

The above examples show how various coordination patterns can be achieved in Reo with a small set of primitive channels. Note also that data dependencies and manipulations can also be directly modeled in Reo using *Filter* and *Transform* channels. Furthermore, there also exist channels for modeling context and time-dependent behavior in Reo. An example of a context-dependent channel is the *LossySync* which loses data items only when the party at its sink end is not ready to accept data (see [26, 64]). Timed behavior on the other hand can be modeled using the *Timer* channel (see [2, 63]).

An important aspect of Reo's approach to component coordination is that the coordinated entities are unaware of their environment and that the coordination protocols are enforced by the connector from outside of the components. This enables a clear separation of concerns between (i) the computation performed by the components and (ii) their exogenous coordination realized by the connectors.

In the following section we introduce the Eclipse Coordination Tools (ECT), which is a comprehensive and extensible implementation of Reo in the Eclipse development environment.

2.2 The Eclipse Coordination Tools

The Eclipse Coordination Tools (ECT) is a set of integrated plug-ins for the Eclipse platform, which offer a graphical development environment for the specification, analysis and execution of component-based software systems using the coordination language Reo. Among other applications, the ECT have been used in the EU projects Credo [30, 48] and COMPAS [27]. The features of ECT can be summarized as follows:

- graphical definition of connectors and networks with hierarchical structuring,
- definition of user-defined component and channel types,
- automatic generation of connector and network semantics,
- reusability of components and connectors using libraries,
- definition and execution of reconfigurations,
- code generation and deployment onto a distributed execution engine,
- qualitative analysis using model checking and animation,
- performance evaluation using stochastic model checking and simulation.

Figure 2.4 depicts the Eclipse environment with a typical ECT set-up. In the following, we give an overview of the toolset as a whole and discuss its architecture, the underlying models and some of its design principles.

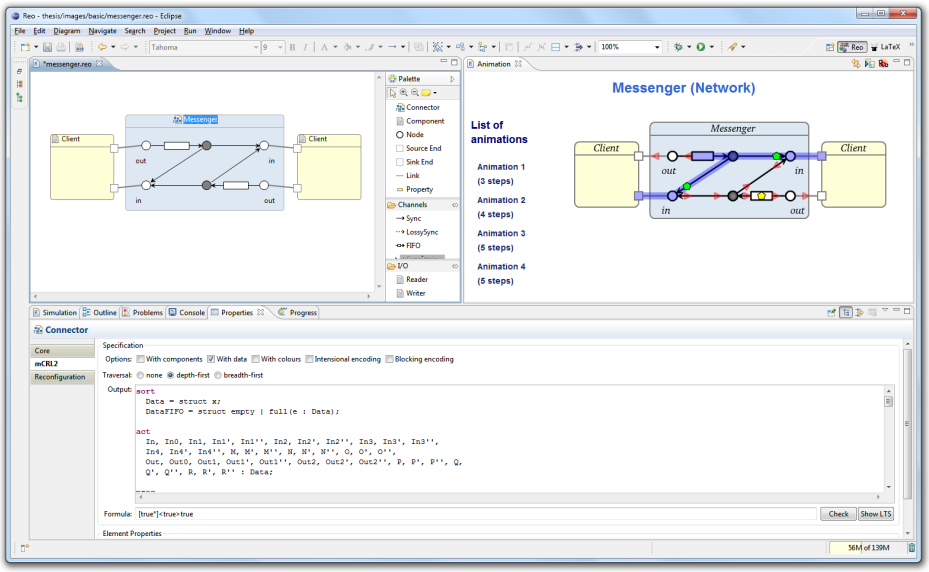


Figure 2.4: editing and analyzing Reo connectors in ECT

2.2.1 Tools overview

Since ECT contains a large variety of different tools implemented by a number of programmers at the CWI and other research institutes, we give here an overview of the toolset as a whole and describe the functionalities of the different tools. Some of the tools will be discussed in more detail later. For the rest we give references here for further reading.

Graphical Reo editor

The top-left part of Figure 2.4 shows the graphical Reo editor which supports the basic channel types from Table 2.1. The editor has been implemented using the Graphical Modeling Framework (GMF) and is based on the Reo meta-model which is defined using the Eclipse Modeling Framework (EMF). We discuss the Reo meta-model in detail in Section 2.2.2. The Reo meta-model and the graphical editor were written the author of this thesis.

Animation tool

The tool on the right-hand side of Figure 2.4 is the animation tool which can be used to generate Adobe® Flash® animations of Reo networks on-the-fly. The animation tool is integrated into the graphical Reo editor and provides a quick and intuitive way of simulating connectors and networks. The tool is based on the animation semantics introduced by Costa et al. [29], which can be seen as an extension of the Reo's coloring semantics [26] (see also Section 4.3.3) with data flow actions. These animations essentially visualize the token game in Reo connectors. Figure 2.5 shows the animation tool in more detail. The coloring and animation semantics, as well as the animation tool itself have been implemented by the author of this thesis.

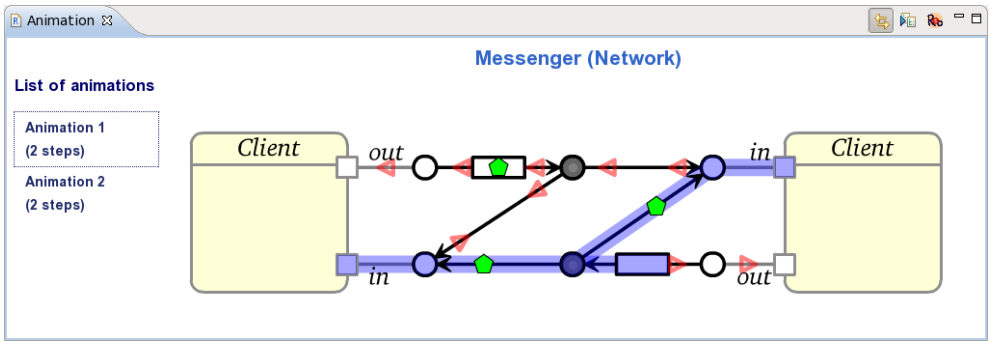


Figure 2.5: the messenger network animated using the ECT animation tool

mCRL2 conversion tool

The bottom part in Figure 2.4 is the mCRL2 [49] conversion tool used for model checking in ECT. While the animation tool gives an intuition about the behavior of a network, the mCRL2 converter can be used to do formal verification based on model checking and state space visualization. The tool is based on an encoding of Reo in the mCRL2 specification language, which we discuss in detail in Chapter 4. Note that the mCRL2 converter can handle data-, context- and time-dependent behavior, which is to the best of our knowledge not provided by any other comparable tool. Note also that the mCRL2 specifications are automatically derived from Reo models specified in the graphical Reo editor in ECT. The mCRL2 converter was written by the author of this thesis.

The Extensible Automata (EA) framework

A large number of semantical models exist for Reo and most of them are automata models. ECT therefore contains the *Extensible Automata* (EA) framework which provides a unified framework for deriving automata-based models from Reo. The framework contains a graphical automata editor and can also be used outside of the context of Reo. We discuss the EA framework in detail in Section 3.3. The following persons have made contribution to the EA framework: Stephanie Kemper, Ziyan Maraikar, Young-Joo Moon and the author of this thesis.

Stochastic modeling and analysis

For performance evaluation, there are two tools available in ECT. On the one hand there is a tool that generates automata models with stochastic information from graphical Reo models. These so-called *quantitative intensional automata* (QIA) can be used to generate Continuous Time Markov Chains (CTMCs) which can then be fed into the PRISM model checker [67] for further analysis. This approach has been developed and implemented by Young-Joo Moon within the EA framework discussed above. An alternative approach for stochastic analysis is based on a simulator for Reo. Similar to the animation tool, the simulator is based on Reo's coloring semantics [26]. This approach has been implemented by Oscar Kanters in his Master's thesis [56]. We give a more detailed overview of both tools in Section 4.4.

Execution engines

Currently there exist two implementations for executing Reo connectors. On the one hand, a code generation framework and runtime based on constrained automata (see Section 3.1) can be used to derive centralized implementations of Reo. This approach was implemented by Ziyan Maraikar and is further discussed in Section 3.3.5. Related to this implementation, the ReoLive web service discussed in Section 5.5 provides a dynamically reconfigurable implementation of Reo using standard web service tech-

nologies, on top of a centralized constraint automata based coordinator implementation. This engine for Reo was written by Ziyan Maraikar and the author of this thesis.

An alternative approach is provided by the distributed engine written by José Proença, for which we refer to [83] for more details.

Conversion tools

Reo can serve as a formal basis of other high-level modeling languages, too. ECT therefore contains conversion tools from various modeling languages to Reo. Most importantly, conversion from BPEL, BPMN, and UML2 sequence diagrams is supported. For more information we refer to [24]. The conversion tools have been developed by Behnaz Changizi in the context of the EU COMPAS project [27].

Vereofy

The Vereofy tool [105] provides a powerful means for model checking Reo connectors. Vereofy is developed by the group of Christel Baier at the Technical University of Dresden. Vereofy is a standalone tool and not part of ECT. However, it includes also an integration with ECT and can therefore be used directly within ECT. We discuss some of the unique features of Vereofy in Section 4.4.2.

In the next part of this section we introduce the Reo meta-model. It not only forms the core of ECT, but also provides a formal view on the underlying structural models.

2.2.2 The Reo meta-model

Following the model-driven approach to software engineering, we have defined and implemented Reo based on a meta-model. We used the Eclipse Modeling Framework (EMF) [40], which is the standard modeling framework in Eclipse, offering powerful code generation and runtime based on purely structural meta-models. The core of the Reo meta-model is depicted in Figure 2.6. The shown classes and interfaces are part of the package *cwi.reo*. For brevity, multiplicities of associations are omitted if they are 0..1. In the following, we explain the Reo meta-model in detail.

Core package

The abstract class *Composite* represents a collection of connectors. There are two concrete implementations of this class: Module and Network. Both of these classes have references to Component and Connector. However, these references are of containment type for modules, but not for networks. Modules serve indeed as containers for components and connectors and usually correspond one-to-one with Reo files. Networks on the other hand are runtime objects which describe an interconnected graph of connectors and components. Its method update() is used to compute the transitive closure of the current contents of a Network. Note that a module can include multiple, logically disconnected networks and vice versa, a network can span over multiple

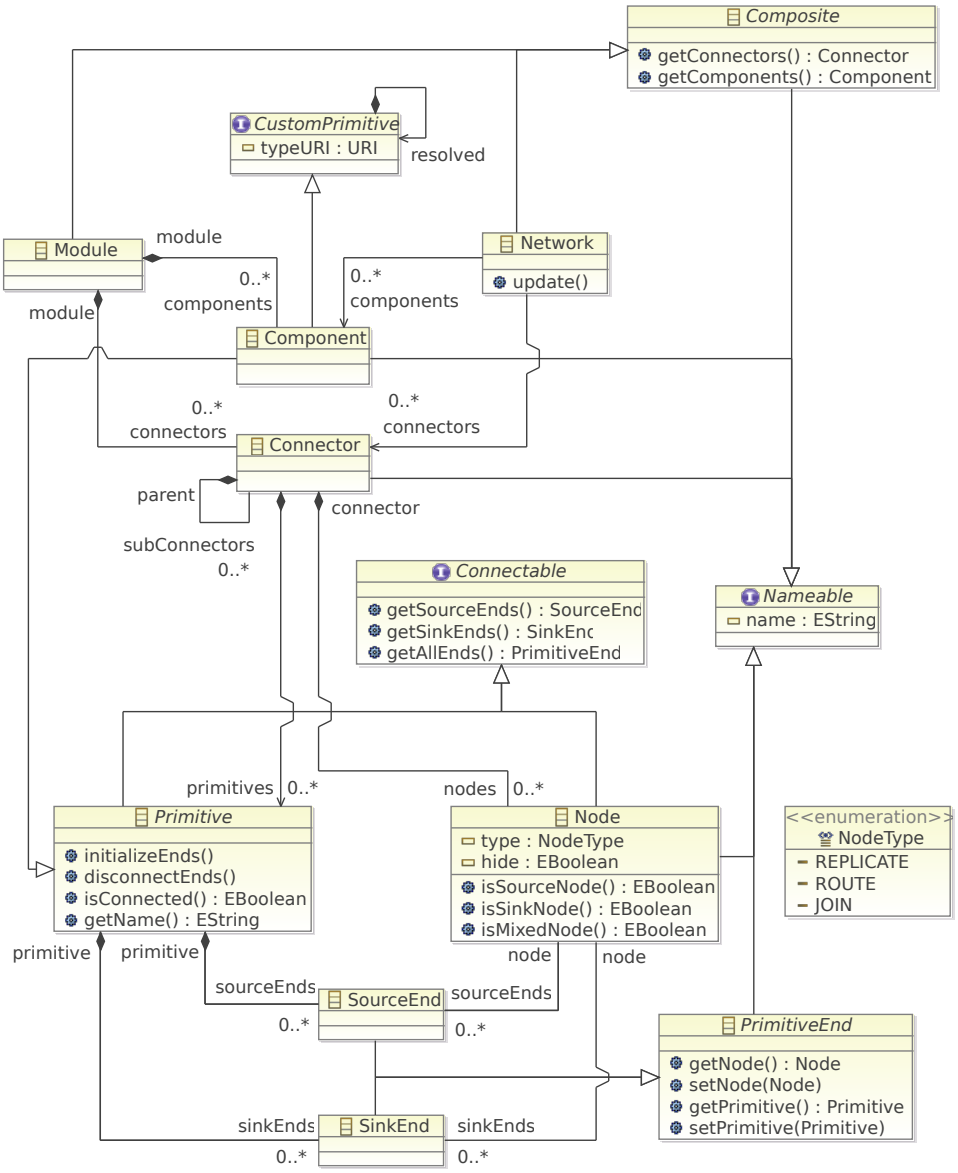


Figure 2.6: Reo meta-model, package *cwi.reo*

modules and hence multiple files. Thus, a module serves as a mere container object, whereas a network describes a closed, transient system that can be used for execution and analysis.

The class Connector represents a collection of nodes and primitives (channels or internal components). The class Component is a concrete implementation of the abstract class *Primitive*. Connectors are essentially containers for nodes and primitives and can be nested using the containment reference subConnectors. On the bottom of Figure 2.6 the abstract class *PrimitiveEnd* together with its concretizations SourceEnd and SinkEnd are shown. Primitive ends serve as connection points between primitives and nodes. A primitive end contains a reference to at most one node and at most one primitive. As indicated by its name, a primitive end always belongs to a primitive, never to a node. This is witnessed by the fact that the associations from *Primitive* to SourceEnd and SinkEnd are containments. Thus, the ends of a primitive can be seen as its public interface. Most types of primitives have a fixed interface, which can be initialized using the method initializeEnds(). The method isConnected() checks whether all ends of this primitive are connected to a node. The method disconnectEnds() can be used to disconnect the primitive from all its adjacent nodes.

For nodes it can be checked whether they are source, sink or mixed nodes. We consider nodes as source nodes if they are not connected to a sink end, or if the sink end belongs to an external component, and analogously for sink nodes. Moreover, nodes have a type attribute, which can have the values REPLICATE, ROUTE or JOIN. The first one of these represents Reo's default merger-replicator semantics of nodes. The second type models an exclusive router, i.e., incoming data items are merged and non-deterministically routed to one of the outgoing ends. The third type of node joins the inputs of all incoming ends into a tuple and forwards it to one of the outgoing ends. From a mere synchronization point of view, join nodes are the dual of the default node semantics of Reo. Nodes and primitives implement the common interface *Connectable*, which can be used to access the source and sink ends of an entity, regardless of whether it is a node or a primitive. This interface also contains a method for accessing all primitive ends together. Primitive ends, nodes, components and modules all implement the interface *Nameable* which can be used to assign names to these entities. Primitives also have a name attribute, but it is derived and can be accessed only using the method getName().

The interface *CustomPrimitive* indicates that a primitive is user-defined. This can include its interface (source and sink ends), as well as its semantics. Custom primitives are dynamic entities and can be either defined *inline* or by reference. We discuss custom primitives in detail in Section 2.2.3. The interface *PropertyHolder* is not shown for clarity in the diagram in Figure 2.6. This interface is implemented by all important classes, including Module, Connector, *Primitive*, Node and *PrimitiveEnd*. It can be used to annotate entities with simple key-value pairs, which for instance can be used to associate semantics to a component or to supply information needed for code generation.

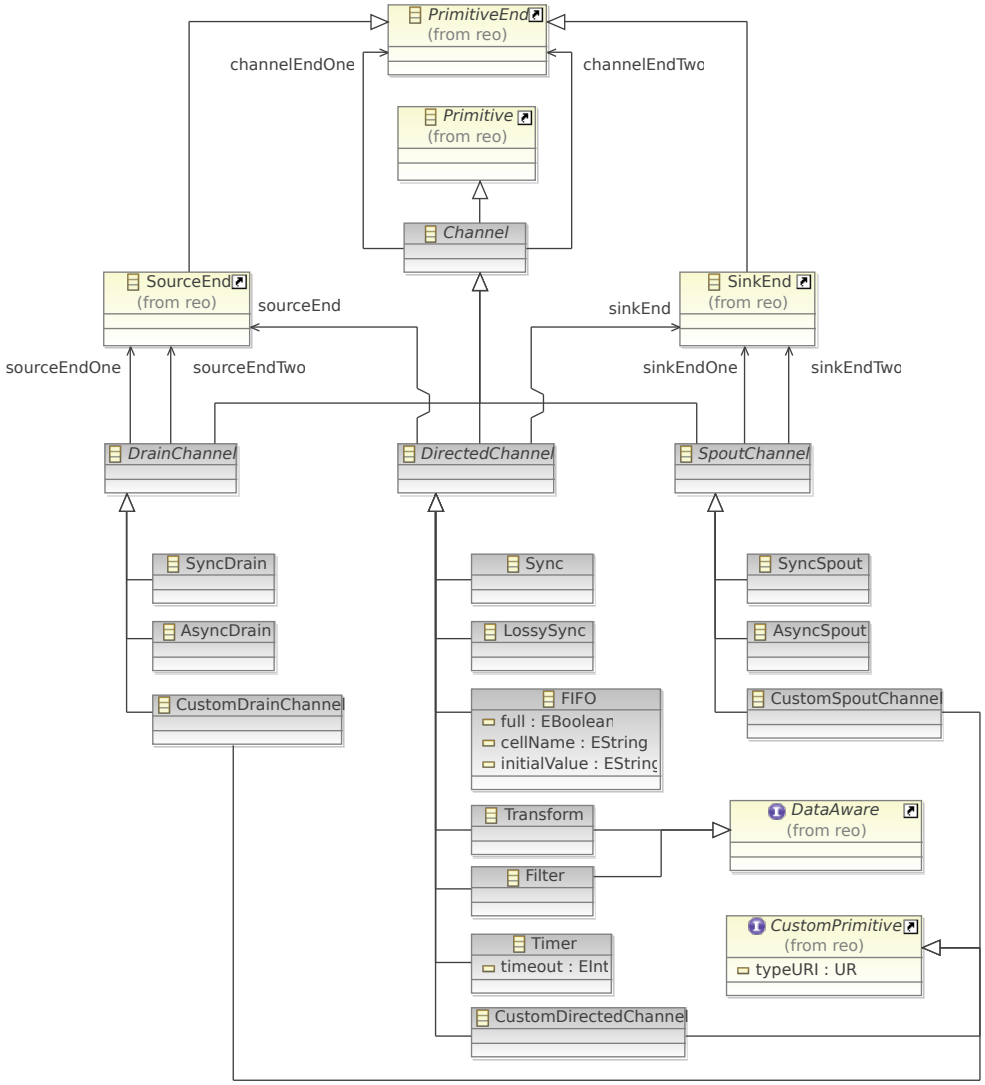


Figure 2.7: Reo meta-model, package *cwi.reo.channels*

Channels package

Channels are defined in the package *cwi.reo.channels*, which is depicted in Figure 2.7. The abstract class *Channel* extends *Primitive* and defines two references to *PrimitiveEnd*, representing the two channel ends of the channel. Note that these references specialize the sourceEnds and sinkEnds containment references in *Primitive*. The channel class is further specialized into the abstract classes *DirectedChannel*, *DrainChannel* and *SpoutChannel*, which define references to SourceEnd and SinkEnd which further specialize the aforementioned references to *PrimitiveEnd*.

The bottom of Figure 2.7 contains the concrete implementations of the three basic channel types, including all channels from Table 2.1. It also defines three concrete classes for custom channels, which all implement *CustomPrimitive*. Just like components, custom channels can be used to define new types of primitives.

2.2.3 Custom primitives in ECT

The pre-defined channel types provide a rich framework for defining complex connectors and communication protocols, including synchrony and asynchrony, buffering, sorting, data manipulation, and data and time-dependent behavior. Moreover, the Reo meta-model allows horizontal and hierarchical structuring of connectors. However, for a true modularization already defined functionalities should be reusable in a different context, without copying the underlying connectors. Moreover, for full flexibility, users should be allowed to define their own components and channel types from scratch or using existing artifacts. Both aspects are supported by ECT using the concept of custom primitives.

Custom primitives are ordinary primitives that additionally implement the interface *CustomPrimitive* and that can be annotated with semantical information. Concrete classes for custom primitives are Component, CustomDirectedChannel, CustomDrainChannel and CustomSpoutChannel. For the latter three, the interface is fixed, whereas a component may have an arbitrary number of source and sink ends.

Semantical annotations for custom primitives

The semantics for normal primitives is fixed already at compile time. For custom primitives, semantics can be defined at runtime by annotating the component or custom channel with behavioral specifications. The format of the specification is not fixed. Currently supported formats are:

- coloring tables and animation specifications,
- constraint automata (Section 3.1),
- mCRL2 specifications (Section 4.2), and
- RSL and CARML code using the Vereofy plug-in [7, 8] (Section 4.4.2).

For a given custom primitive, multiple specifications in different formats can be made. Because of the different types of semantics and expressiveness, consistency between

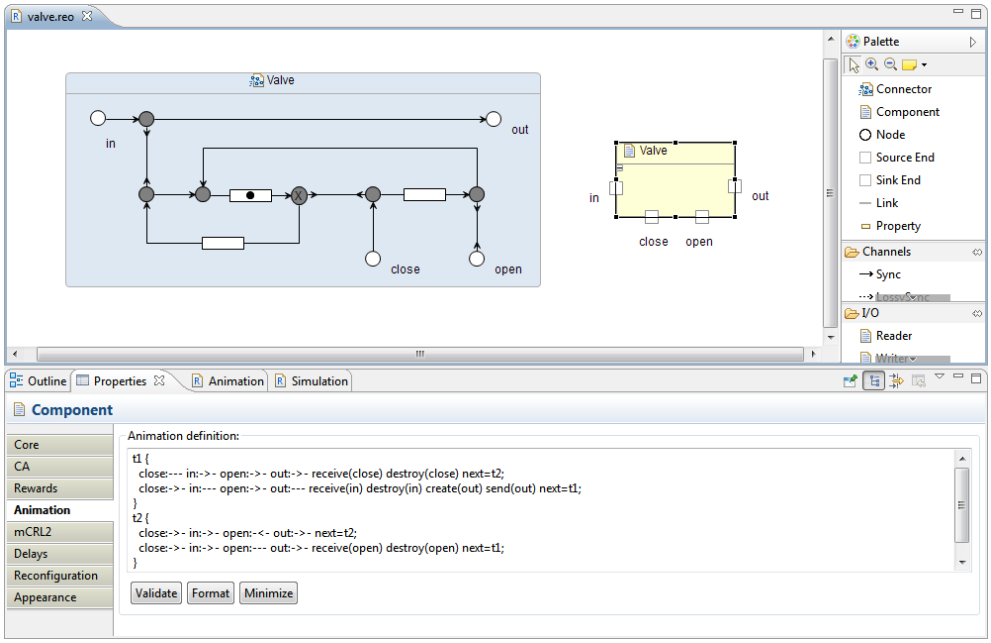


Figure 2.8: deriving components from connectors in ECT

the specifications has to be ensured by the user. However, conversion from constraint automata into the coloring model with two colors has been implemented and can be used to generate animations on-the-fly, without specifying an animation semantics.

Deriving custom primitives from connectors

A typical task is to generate a component or a custom channel from a given Reo connector. The idea is to derive a custom primitive with the same interface and the same behavior as the connector. In ECT, this functionality has been implemented in an extensible framework and can be invoked directly in the graphical Reo editor. Figure 2.8 depicts an example where a connector called *Valve* in the top-left part of the editor has been converted to a component with the same interface in the top-right part. Moreover, the derived component contains semantical annotations that describe its behavior, e.g. in terms of an animation specification as shown in the bottom of Figure 2.8. To generate these behavioral descriptions automatically, for every semantical specification format there exists a so-called *textual semantics provider*, which is registered in the runtime of ECT. The conversion from a connector to a custom primitive is then performed in three steps:

(i) instantiate a custom primitive with the same interface as the given connector,

(ii) invoke all semantics providers to generate textual representations of the connectors semantics, and

(iii) annotate the new custom primitive with the generated specifications.

In this way, new custom primitives can be obtained from existing ones in a modular way. Note that in step (ii), providers may have to incorporate behavioral specifications of other custom primitives. The above scheme has been fully implemented for coloring tables and animation specifications, as well as constraint automata.

Converting a connector into a primitive using the above scheme always produces a self-contained entity which has no relationship to the original connector anymore. The behavioral specifications are bound to that particular primitive since they are directly attached as annotations. Custom primitives derived or directly specified in this way are therefore called *inline* primitives. Note that inline primitives can also be defined from scratch.

Custom primitives with type references

A complementary approach is to define the semantics of a primitive by reference. In ECT, custom primitives have an attribute called *type URI*, which can be used for this very purpose. Using this property, primitives can be defined externally and reused without copying them. A custom primitive with a set type URI essentially serves as a stub that can be resolved to an inline primitive, which in turn can be used for analysis and execution. The resolution of custom primitives is implemented using a lazy loading scheme: the primitive is resolved only when necessary. This strategy saves resources and can in fact be found everywhere in the Eclipse platform, e.g. in the proxy concept of EMF, where a proxy object is resolved when its contents are accessed for the first time.

The content type of the artifact located at the type URI is not fixed. It can be one of the above mentioned specification formats, but also a Reo artifact itself. In the latter case, the URI must point to a Reo file and include a unique identifier in that file. This identifier can be the name of a primitive or a connector. If it is a primitive, the stub is simply resolved to that primitive. If it is a connector, the connector is first converted into a primitive, as described above, and then this derived primitive is used. Note that this is done on demand, as opposed to the inline approach.

Example 2.5 (Components with type references)**.** Figure 2.9 shows an example of components with type references in the animation tool. The upper part contains a connector called *Counter*. This connector has two source nodes *increase* and *reset*, and one sink node *count*. It moreover contains nodes with exclusive router semantics (cf. Example 2.2). The semantics of this connector can be described as follows: on every second token at *increase*, a token is produced at *count*, unless there was a token at *reset* before. This essentially models a 2-counter with reset.

The bottom part of Figure 2.9 shows a network that has been animated using ECT's animation tool. This network contains two components with type references, i.e., they both contain a property '*type=#Counter*'. The value of this property is a relative URI pointing to an artifact with name *Counter* in the same Reo file. Thus, it refers to the counter connector in the upper part. To compute an animation as shown in this

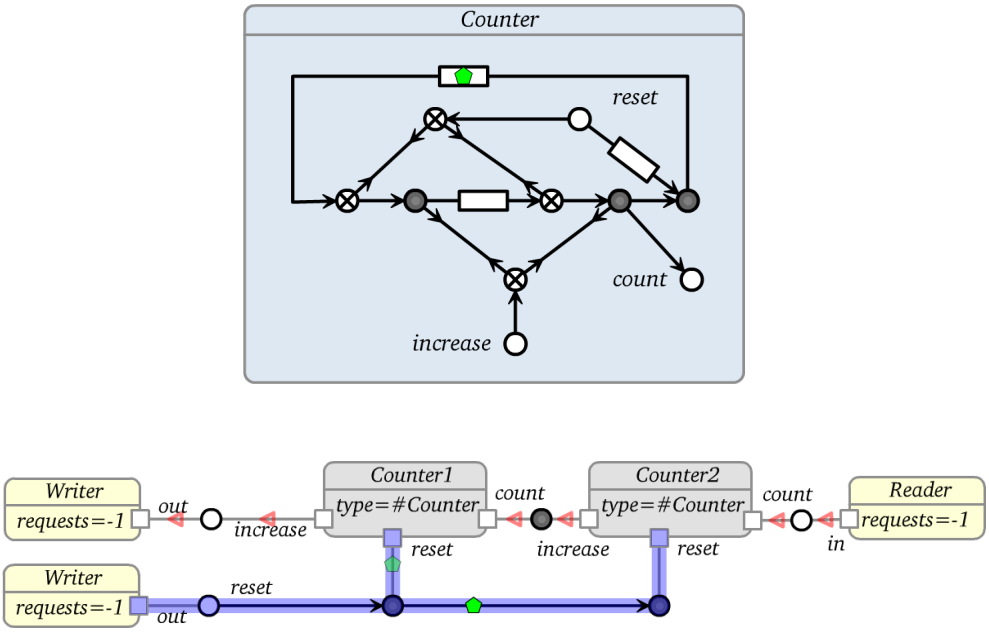


Figure 2.9: components with type references

example, the animation tool resolves the two components *Counter1* and *Counter2* to the connector *Counter* and generates an animation based on the connector semantics. Note that since this is a definition by reference, changes in *Counter* are immediatly reflected in the network semantics. In this particular example we have built a 4-counter with reset using two 2-counters with reset. △

Note that not only components but also custom channels can be used in such a way, i.e., the semantics of a custom channel can be defined using a connector. The concept of textual semantics providers in ECT, moreover, allows to use arbitrary formats for the specification of custom primitives. Currently implemented semantics providers support the definition of custom primitives using Reo artifacts (components or connectors), constraint automata, and CARML and RSL code using the Vereofy plug-in [105]. Note also that since the type reference is given in terms of a URI, virtually arbitrary locations can be used to store definitions. For instance, it is possible to directly access Reo artifacts located on websites using *http*-URIs. Thus, type references can be used to associate behavior to black-boxed components in a modular and reusable way. This is in particular useful for defining and structuring large networks.

2.3 Conclusions

Reo is an expressive coordination language offering the possibility of defining custom channels and components with potentially data-, context- and time-dependent behavior. Extensive tool support for Reo is provided in the Eclipse Coordination Tools (ECT). In ECT, connectors can be used directly or as black-boxes for the definition of components and channels, providing a powerful abstraction mechanism. Furthermore, connectors can be structured hierarchically and reused in other contexts. Custom primitives, i.e., user-defined channels and components may be defined inline or by reference. Multiple formats for the specification of primitives are supported using a modular and extensible specification framework in ECT. Moreover, the animation tool in ECT provides means for visualizing network and connector behavior in an intuitive way. Note also that many other extensions and tools exist for formal analysis and execution of Reo models, which we will present in more detail later.

Chapter 3

Automata-based semantics for Reo

In this chapter, we discuss two automata-based semantics for Reo which will serve as the basis for the verification approach in Chapter 4. We first recall constraint automata as one of the standard semantics for Reo. We then introduce port automata as an abstraction of constraint automata and present a decomposition scheme for it. Finally, we discuss an extensible implementation for automata-based semantics in ECT, including conversion from Reo to various automata types, and an automata-based, centralized execution engine for Reo.

3.1 Constraint and port automata

The use of an automata-based semantics for Reo is motivated by the fact that connectors in general are stateful entities and their behavior can be expressed in terms of data flow events on their ports[1]. Somewhat similar to Petri nets, Reo allows true concurrency, i.e., multiple ports can fire together synchronously, as opposed to merely interleaved semantics. However, since Reo is a data flow oriented model, an explicit handling of data dependencies and transformations is also required.

Constraint automata [10] are a widely used semantical model for Reo, which capture the most important aspects of channels and connectors, i.e., synchrony vs. asynchrony, state, and data constraints. The constraint automata model is, moreover, compositional, in the sense that it comes equipped with well-behaved product and hiding operators.

Regarding their use in tools, constraint automata form the basis for a centralized execution engine, as we discuss in Section 3.3.5. Furthermore, there exist three analysis tools that internally use or encode constraint automata:

- the mCRL2 converter (Section 4.4.1),
- the Vereofy model checker (Section 4.4.2), and
- an analysis tool based on symbolic execution [82].

[1] We use the term *port* as a synonym for primitive ends in semantical settings.

3.1.1 Definition

Before giving the formal definition of constraint automata, we first need to recall the language for data constraints to be used. Given a finite set of port names $N = \{A_1, \ldots, A_n\}$ and a finite data domain $Data = \{d_1, \ldots d_m\}$. The language of data constraints $DC(N, Data)$, or just DC for short, is defined using the following grammar:

$$g ::= \bot \mid \top \mid data(A) = d \mid g_1 \vee g_2 \mid g_1 \wedge g_2$$

where $data$: $N \to Data$ is a partial data-assignment function. Following the notation of [10], we often abbreviate $data(A)$ as d_A and write expressions like $d_A = d_B$ for

$$\bigvee_{d \in Data} (data(A) = d \wedge data(B) = d).$$

We now formally define constraint automata, as introduced in [10].

Definition 3.1 (constraint automaton)**.** Given a finite data domain $Data$. A *constraint automaton* $CA = (Q, N, T, q_0)$ consists of a set of states Q, a set of port names N, a transition relation $T \subseteq Q \times 2^N \times DC(N, Data) \times Q$, and an initial state $q_0 \in Q$. ◊

We usually write transitions as $p \xrightarrow{S,g} q$ with $p, q \in Q$ source and target states, $S \subseteq N$ the set of synchronously firing ports, and g the enabled data constraint or *guard*. We require that guards refer to firing ports only, i.e., $g \in DC(S, Data)$.

The constraint automata for the basic channels are summarized in Table 3.1. Note that we also include two primitives each with three ports: the *Merger* and the *Replicator*. They can be used to compositionally construct the behavior of nodes (cf. [26]). Note also that the constraint automaton for the *FIFO1* is with respect to the data domain $Data = \{0, 1\}$ and that it can be adapted for any other finite data domain.

3.1.2 Join and hiding operations

In the following, we recall the join operator for constraint automata, which is the most important ingredient for the compositionality of the constraint automata model.

Definition 3.2 (join of constraint automata)**.** Given two constraint automata $CA_1 = (Q_1, N_1, T_1, q_0^1)$ and $CA_2 = (Q_2, N_2, T_2, q_0^2)$. The constraint automaton $CA_1 \bowtie CA_2$ is defined as:

$$CA_1 \bowtie CA_2 = \left(Q_1 \times Q_2, N_1 \cup N_2, T, \langle q_0^1, q_0^2 \rangle\right)$$

where the transition relation T is defined by the following rules:

$$\frac{q_1 \xrightarrow{S_1,g_1}_1 p_1 \quad S_1 \cap N_2 = \emptyset}{\langle q_1, q_2 \rangle \xrightarrow{S_1,g_1} \langle p_1, q_2 \rangle} \qquad \frac{q_2 \xrightarrow{S_2,g_2}_1 p_2 \quad S_2 \cap N_1 = \emptyset}{\langle q_1, q_2 \rangle \xrightarrow{S_2,g_2} \langle q_1, p_2 \rangle} \tag{3.1}$$

$$\frac{q_1 \xrightarrow{S_1,g_1}_1 p_1 \quad q_2 \xrightarrow{S_2,g_2}_1 p_2 \quad S_1 \cap N_2 = S_2 \cap N_1}{\langle q_1, q_2 \rangle \xrightarrow{S_1 \cup S_2, g_1 \wedge g_2} \langle p_1, p_2 \rangle} \tag{3.2}$$

◊

While rules (3.1) describe an interleaving of actions, rule (3.2) models a synchronous and hence truly concurrent firing of ports. Note that, for applying any of the rules, every port shared by the two automata must be either enabled or disabled in both transitions, as captured by the condition $S_1 \cap N_2 = S_2 \cap N_1$.

Another important operation for constraint automata is hiding, which intuitively removes all occurrences of a set of given port names from the automaton.

DEFINITION 3.3 (hiding for constraint automata)**.** Let $CA = (Q, N, T, q_0)$ be a constraint automaton and $M \subseteq N$. The constraint automaton $CA\backslash M$ is defined by $CA\backslash M = (Q, N\backslash M, T_1, q_0)$ where T_1 is given by:

$$q \xrightarrow{S,g} p \quad \Leftrightarrow \quad q \xrightarrow{S\backslash M, g\backslash M}_1 p \qquad \Diamond$$

Note that $g\backslash M$ is the guard where every term in g that involves a port in M is replaced by *true*. We mention also that our notion of hiding is simpler than the one used in [10] and that it may lead to τ-steps, modeling internal behavior. For examples of applications of the join and hiding operators we refer to [10].

3.1.3 Bisimulation

To reason about constructions we further need a notion of behavioral equivalence for constraint automata. Again, we follow [10] and use strong bisimulation [71] as behavioral equivalence. In essence, two states in the same or in two different automata or labeled transition systems are bisimilar if they cannot be distinguished by an external observer.

DEFINITION 3.4 (constraint automata bisimulation)**.** Let $CA_1 = (Q_1, N, T_1, q_0^1)$ and $CA_2 = (Q_2, N, T_2, q_0^2)$ be two constraint automata. A relation $R \subseteq Q_1 \times Q_2$ is called a *bisimulation* if for all $\langle q_1, q_2 \rangle \in R$:

(i) if $q_1 \xrightarrow{S,g_1}_1 p_1$ then there exists $p_2 \in Q_2$ and $q_2 \xrightarrow{S,g_2}_2 p_2$ with $g_1 \equiv g_2$ and $\langle p_1, p_2 \rangle \in R$,

(ii) if $q_2 \xrightarrow{S,g_2}_2 p_2$ then there exists $p_1 \in Q_1$ and $q_1 \xrightarrow{S,g_1}_1 p_1$ with $g_1 \equiv g_2$ and $\langle p_1, p_2 \rangle \in R$.

Two states $q_1 \in Q_1$, $q_2 \in Q_2$ are called *bisimilar*, written as $q_1 \sim q_2$, if there exists a bisimulation R with $\langle q_1, q_2 \rangle \in R$. The two constraint automata CA_1 and CA_2 are bisimilar, written as $CA_1 \sim CA_2$, if $q_0^1 \sim q_0^2$. $\Diamond$

Bisimilarity of constraint automata qualifies as a notion for behavioral equivalence since it is a congruence for join and hiding, as summarized in the following lemma.

LEMMA 3.5. *Bisimilarity is a congruence for the join and hiding operators:*

(i) $CA_1 \sim CA_2$ *and* $CA_1' \sim CA_2' \Rightarrow \left(CA_1 \bowtie CA_1'\right) \sim \left(CA_2 \bowtie CA_2'\right)$.

(ii) $CA_1 \sim CA_2 \Rightarrow \left(CA_1\backslash M\right) \sim \left(CA_2\backslash M\right)$.

PROOF. For showing (i) we observe that the relation

$$\left\langle \left\langle q_1, q_1' \right\rangle, \left\langle q_2, q_2' \right\rangle \right\rangle \in R \Leftrightarrow q_1 \sim q_2 \text{ and } q_1' \sim q_2'$$

defines a bisimulation on the joined automata. For (ii) it suffices to show that a bisimulation remains a bisimulation after a removal of ports. □

3.1.4 Port automata

Constraint automata are an expressive model for capturing data flow dependencies. However, to allow an easier way of reasoning we introduce an abstraction where data constraints are omitted. To avoid confusion, we explicitly call these automata *port automata*.

DEFINITION 3.6 (port automaton)**.** A constraint automaton $CA = (Q, N, T, q_0)$ is also called a *port automaton* if for every transition $q \xrightarrow{S,g} p$ it holds that $g \equiv \top$. ◊

Join, hiding and bisimulation are inherited from constraint automata and it is easy to show that port automata form a closed subclass of constraint automata, as stated in the following, trivially provable, lemma.

LEMMA 3.7. *Port automata are closed under join, hiding and bisimilarity.* □

Because of the suppressed guards, we use a slightly different set of primitives for the port automata model. To distinguish them from the constraint automata in Figure 3.1 we use a different font for the name of primitives in the port automata model. Figure 3.2 depicts a set of basic port automata.

Note that in the port automata setting, the channels *Sync*, *SyncDrain* and *Transform* are modeled using the same automaton, i.e., $\mathcal{S}$ync. Moreover, the *Filter* behaves like a *LossySync*. We also introduce a new primitive, called $\mathcal{XOR}$, which models both the *Merger* and the exclusive router presented in Example 2.2. Finally, the *FIFO1* channel is reduced to the port automaton $\mathcal{FIFO}1$ which has just two states, respectively modeling that the *FIFO1* is empty or full. Formally, this port automaton can be derived from the corresponding constraint automaton in Table 3.1 by first setting all constraints to *true* and then minimizing it modulo bisimulation. Moreover, we introduce a new primitive with four ports, called $\mathcal{F}$lip$\mathcal{F}$lop. This primitive can be interpreted as a $\mathcal{FIFO}1$ that can be tested for its current state using the additional ports C and D. As we will show in the following section, the $\mathcal{F}$lip$\mathcal{F}$lop primitive arises as an intermediate result of a decomposition scheme for port automata.

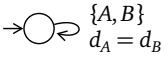
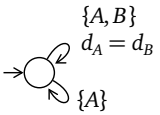
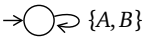
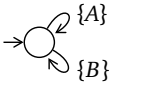
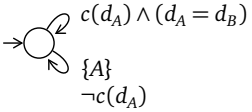
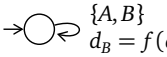
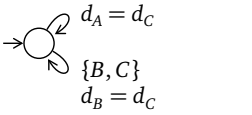
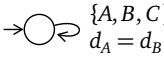
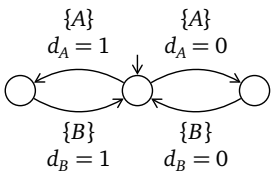
$Sync(A,B) =$ 	$LossySync(A,B) =$ 
$SyncDrain(A,B) =$ 	$AsyncDrain(A,B) =$ 
$Filter(A,B) =$ 	$Transform(A,B) =$ 
$Merger(A,B,C) =$ 	$Replicator(A,B,C) =$ 
$FIFO1(A,B) =$ 	

Table 3.1: constraint automata for common primitives

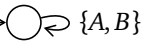
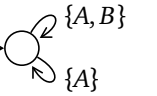
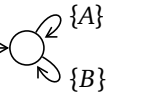
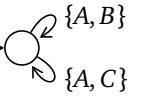
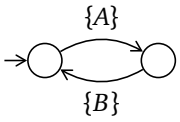
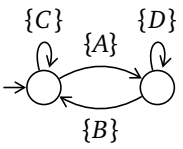
$\mathcal{S}\text{ync}(A,B) =$ 	$\mathcal{L}\text{ossy}\mathcal{S}\text{ync}(A,B) =$ 
$\mathcal{A}\text{sync}(A,B) =$ 	$\mathcal{XOR}(A,B,C) =$ 
$\mathcal{FIFO}1(A,B) =$ 	$\mathcal{F}\text{lip}\mathcal{F}\text{lop}(A,B,C,D) =$ 

Table 3.2: port automata for common primitives

3.1.5 From Reo to automata models

The constraint automata or port automata semantics of a Reo connector or network is derived in two steps: (i) use the join operator to compose the automata of all primitives, and (ii) remove internal ports using the hiding operator. Note that the result is uniquely determined because the join operator is associative and commutative.

Example 3.8 (port automaton for instant messenger)**.** We consider the instant messenger network depicted in Figure 3.1. For simplicity, we compute the port automaton for the connector only, and omit the clients. Moreover, we do not model the *Replicator*s at the nodes X and Y explicitly, since the same effect can be achieved by using a single port name for all coinciding channel ends. The *Merger*s at the nodes B and C must be modeled explicitly though. We denote the incoming channel ends of these *Merger*s by B_1, B_2 and C_1, C_2, respectively, and their outgoing ends by B and C. We can now compute the port automaton for the complete connector by joining all primitive port automata and hiding the internal port names. Formally, we construct the port automaton as shown in Figure 3.2. The states in the resulting port automaton are a combination of the states of the two involved $\mathcal{FIFO}1$ buffers, e.g. states *ee* and *ff* are the states in which both $\mathcal{FIFO}1$ are respectively empty and full. △

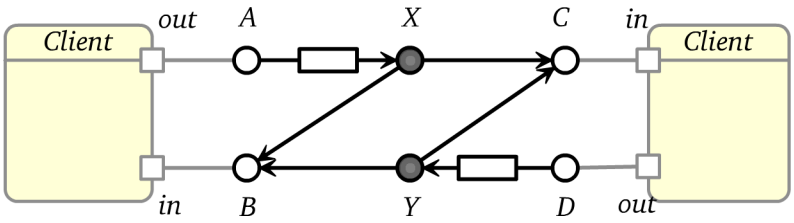
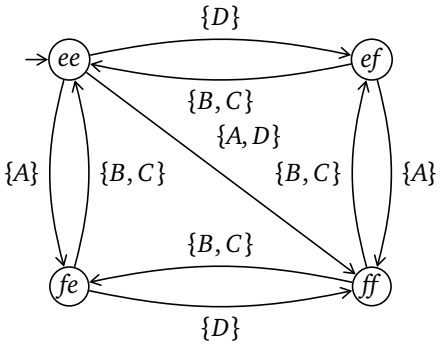


Figure 3.1: instant messenger network (see also Example 2.4)

$$\mathcal{M}essenger(A,B,C,D) =$$
$$\big(\mathcal{FIFO}1(A,X) \bowtie \mathcal{FIFO}1(D,Y)$$
$$\bowtie \mathcal{S}ync(X,C_1) \bowtie \mathcal{S}ync(X,B_1)$$
$$\bowtie \mathcal{S}ync(Y,B_2) \bowtie \mathcal{S}ync(Y,C_2)$$
$$\bowtie \mathcal{XOR}(B,B_1,B_2) \bowtie \mathcal{XOR}(C,C_1,C_2)\big)$$
$$\setminus \{X,Y,B_1,B_2,C_1,C_2\}$$

(a) construction



(b) result

Figure 3.2: deriving the port automaton for the instant messenger connector

3.2 Decomposition of port automata

For a compositional language such as Reo a natural question that arises is whether there exists a finite set of primitives that, when composed in the right way, are expressive enough to model any possible behavior. If yes, one may also ask for an efficient synthesis algorithm that ideally yields a compact and human-readable system specification. In this section, we show how an arbitrary port automaton can be decomposed into a compact representation based on a finite set of of primitives and the join and hiding operators. This decomposition is a tailor-made approach in which the port automata are examined directly. This has the advantage that the language generated by an automaton does not have to be taken into account.

In the context of Reo, our result means that a finite number of channel types is sufficient to implement every behavior expressible in port automata. Moreover, we show that the decomposition yields a very compact representation of the system, which can be easily extended to a connector synthesis algorithm. We propose a decomposition scheme in two steps. First, we show how stateless port automata –port automata with just one state– can be decomposed into $\mathcal{XOR}$s. In the second step, we consider the full class of port automata and show how they can be efficiently decomposed using just two primitives, i.e., $\mathcal{XOR}$s and $\mathcal{FIFO}1$s.

3.2.1 Stateless port automata

In this part, we show how stateless port automata, i.e., port automata with just one state, can be decomposed into a collection of $\mathcal{XOR}$s. As an example, consider the following decompositions of some basic stateless port automata.

Example 3.9 (stateless port automata decompositions)**.**

$$\begin{aligned}
\mathcal{L}\text{ossy}\mathcal{S}\text{ync}(A,B) &= \mathcal{XOR}(A,B,X) \setminus \{X\} \\
\mathcal{A}\text{sync}(A,B) &= \mathcal{XOR}(X,A,B) \setminus \{X\} \\
\mathcal{S}\text{ync}(A,B) &= \big(\mathcal{XOR}(A,X,Y) \bowtie \mathcal{XOR}(B,X,Y)\big) \setminus \{X,Y\}
\end{aligned}$$ △

From these examples, it appears that the $\mathcal{XOR}$ is a possible candidate primitive for a general decomposition scheme for stateless port automata. $\mathcal{XOR}$s can be composed in multiple ways. To illustrate this, we use a connector-like notation in which $\mathcal{XOR}$s are represented as $-\!\!\prec$, and hidden and normal ports are depicted by $\circ$ and $\bullet$, respectively. We distinguish two specific ways of composition, as shown in Figure 3.3.

Figure 3.3a shows how to build an n-ary $\mathcal{XOR}$ out of binary ones. This type of composition essentially splits a transition with firing ports $S \cup \{X\}$ into two transitions with respective firing sets $S \cup \{B_2\}$ and $S \cup \{B_3\}$. This construction yields a mutual exclusion of ports B_2 and B_3. In the following, we will use the notation $\mathcal{XOR}(A,\mathcal{B})$ to describe an n-ary $\mathcal{XOR}$ with $\mathcal{B} = \{B_1,\ldots,B_n\}$, which can be built in this way.

The second possibility of composing $\mathcal{XOR}$s is depicted in Figure 3.3b. This construction uses the internal and mutually excluded ports to synchronize the external ports. In this particular example, ports A and B can be activated only together, either

Figure 3.3: two ways of composing $\mathcal{XOR}$s using join and hiding

via port X or Y. The resulting port automaton coincides with $\mathbb{S}\mathrm{ync}(A,B)$, as shown in Example 3.9. In the following, we show how these two simple constructions can be used to build any stateless port automaton.

Lemma 3.10 (decomposition of stateless port automata)**.** *Let $PA = (\{q\}, N, T, q)$ be a finite, stateless port automaton with $N = \{A_1, \ldots, A_n\}$ and $T = \{\langle q, S_1, q\rangle, \ldots, \langle q, S_m, q\rangle\}$. Let D and $\mathcal{X} = \{X_1, \ldots, X_m\}$ be fresh port names and define for every $A \in N$ the set*

$$\mathcal{X}_A = \left\{ X_i \in \mathcal{X} \mid \exists\, (q \xrightarrow{S_i} q) \colon A \in S_i \right\}$$

Then the following equivalence holds:

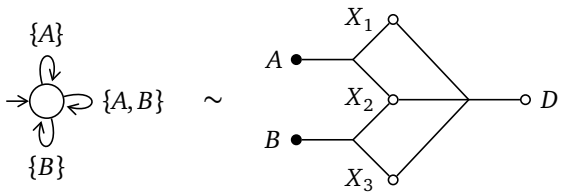
$$PA \sim \left(\mathcal{XOR}(D, \mathcal{X}) \bowtie \mathcal{XOR}(A_1, \mathcal{X}_{A_1}) \bowtie \ldots \bowtie \mathcal{XOR}(A_n, \mathcal{X}_{A_n}) \right) \setminus (\{D\} \cup \mathcal{X})$$

Proof. The ports of the constructed automaton are the same as in PA, since D and all ports in $\mathcal{X}$ are hidden. The constructed port automaton is also stateless. Moreover, every new port $X_i \in \mathcal{X}$ corresponds to the PA-transition $q \xrightarrow{S_i} q$ in the following way:

$$A \in S_i \iff X_i \in \mathcal{X}_A,$$

i.e., port X_i is enabled exactly when all $A \in S_i$ are enabled. Hence, X_i synchronizes the ports in S_i. Moreover, $X_i \in \mathcal{X}$ can be enabled only if D is enabled and whenever D is enabled exactly one $X_i \in \mathcal{X}$ is enabled. Hence, no concurrent activation of the port sets S_i is possible. □

Example 3.11 (stateless port automaton decomposition)**.** Consider the following decomposition of a stateless port automaton into $\mathcal{XOR}$s (again in connector notation):



There is a one-to-one correspondence between the created ports $\mathcal{X} = \{X_1, X_2, X_3\}$ and the transitions in the original automaton, i.e., X_1 represents the transition $\{A\}$, X_2 represents $\{A, B\}$, and finally X_3 stands for the transition $\{B\}$. The $\mathcal{XOR}$ from D ensures that no concurrent activation of these ports is possible. Formally, the automaton is constructed as:

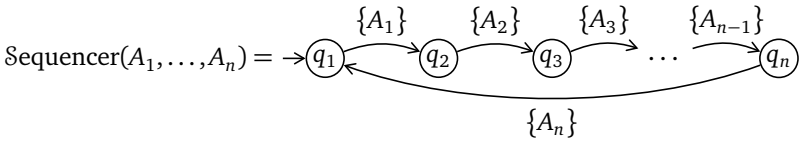
$$\left(\mathcal{XOR}(A, \{X_1, X_2\}) \bowtie \mathcal{XOR}(B, \{X_2, X_3\}) \bowtie \mathcal{XOR}(D, \{X_1, X_2, X_3\})\right) \setminus \{X_1, X_2, X_3, D\} \quad \triangle$$

We have seen that, using join and hiding, every stateless port automaton can be represented using (binary) $\mathcal{XOR}$s. In the following section we will extend this decomposition to the full class of port automata.

3.2.2 General decomposition scheme

We consider now the full class of port automata and start again with an example.

Example 3.12 (sequencer)**.** The *Sequencer* (cf. [1]) is a connector that activates a collection of ports $A_1, \ldots, A_n$ sequentially, one after another. The corresponding port automaton is given by:



If we would try to build a connector with this behavior, we could simply compose n *FIFO1* channels in a loop and make the last one *full*, i.e., it already carries a token in the beginning. Intuitively, the token moves from one *FIFO1* channel to another in one step and starts from the beginning after n steps. In the port automata model, the difference between empty and full $\mathcal{FIFO}1$s is realized by switching the ports, so that $\mathcal{FIFO}1(B, A)$ models a full *FIFO1* from A to B. Hence, we can construct the n-ary *Sequencer* in the port automata model as follows:

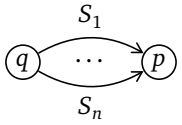
$$\begin{aligned}\mathcal{S}\text{equencer}(A_1, \ldots, A_n) = \mathcal{FIFO}1(A_1, A_2) &\bowtie \mathcal{FIFO}1(A_2, A_3) \bowtie \ldots \\ &\bowtie \mathcal{FIFO}1(A_{n-1}, A_n) \bowtie \mathcal{FIFO}1(A_1, A_n)\end{aligned}$$

where the last *FIFO1* is full. Note that this construction also introduces some unreachable states which are not shown in the automaton above. $\triangle$

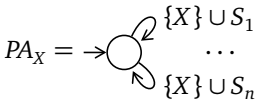
The general decomposition of an arbitrary port automaton involves the following three steps, which do not change the behavior of the automaton, up to bisimilarity. Step (i) removes all parallel transitions, and labels every transition uniquely. Step (ii) breaks the resulting automaton into a product of port automata each containing not more than two states and four transitions. There is only a finite number of such automata; we show that the largest of these, which we will call $\mathcal{F}$lip$\mathcal{F}$lop, suffices. Finally, step (iii) reduces this automaton to $\mathcal{XOR}$s and $\mathcal{FIFO}1$s, completing the decomposition scheme.

Step (i) – Removing parallel transitions

Given an arbitrary port automaton PA, construct a new port automaton PA' which has the same number of states. For all parallel transitions between each pair of states:



in PA introduce a single transition $q \xrightarrow{\{X\}} p$ in PA' where X is a fresh port. The port set of PA' consists exactly of these ports: $N' = \{X_1, \ldots, X_m\}$. For $X \in N'$ corresponding to transitions with labels $S_1, \ldots, S_n$, construct the following stateless port automaton:



Joining these stateless port automata with PA' and hiding the newly introduced ports yields the original automaton:

$$PA \sim \left(PA' \bowtie PA_{X_1} \bowtie \ldots \bowtie PA_{X_m}\right) \setminus N'$$

up to a change of state names. The stateless automata PA_{X_i} can be further decomposed into $\mathfrak{XOR}$s, as described above. The automaton PA' has the property that every transition in it is labeled with a singleton port name set and every port name occurs in exactly one transition (there is a one-to-one correspondence between transitions and port names). We call an automaton with this property *singleton* port automaton.

Step (ii) – Splitting singleton port automata

Let $PA = (Q, N, T, q_0)$ be a singleton port automaton with more than 2 states. The following steps decompose it into two singleton port automata, one with 2 states, the other with $\lceil |Q|/2 \rceil$ states.

If the number of states in the automaton is odd, introduce an unreachable dummy state to make the total number of states even. Let n be the new total number of states. Now organize the states into a $2 \times (n/2)$ grid with dimensions $\{a, b\}$ and $\{1, 2, \ldots, n/2\}$, where position $(a, 1)$ is reserved for the initial state, and the remainder of the states of PA are placed arbitrarily in the grid. Now construct two port automata:

$$PA_1 = \left(\{a, b\}, N, T_1, \{a\}\right),$$
$$PA_2 = \left(\{1, \ldots, n/2\}, N, T_2, \{1\}\right)$$

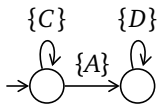
using the following rule:

$$\frac{\langle i, j\rangle \xrightarrow{\{X\}} \langle i', j'\rangle \quad i, i' \in \{a, b\} \quad j, j' \in \{1, \ldots, n\}}{i \xrightarrow{\{X\}}_1 i' \qquad j \xrightarrow{\{X\}}_2 j'}$$

It is clear from the construction that $PA \sim (PA_1 \bowtie PA_2)$ and that they are both again singleton port automata.

By repeating this step one can decompose a singleton port automaton into a product of singleton port automata with not more than two states. By applying step (i) to these automata, the number of transitions can be reduced to at most four, since all parallel transitions are removed. The *largest* of these two-state automata, i.e., the one with four transitions coincides with the $\mathcal{F}$lip$\mathcal{F}$lop primitive (cf. Table 3.2). All other possible port automata of this shape can be constructed by removing transitions from this one. Transitions can be removed by first creating a new transition using an $\mathcal{XOR}$ and then synchronizing it with the transition to be removed, as shown in the following example.

EXAMPLE 3.13 (two-state port automaton decomposition)**.**



$$\sim \big(\mathcal{F}\text{lip}\mathcal{F}\text{lop}(A,B,C,D) \bowtie \mathcal{XOR}(A,A_1,A_2) \bowtie \mathcal{S}\text{ync}(A_2,B)\big) \setminus \{B,A_1,A_2\} \triangle$$

Hence, an arbitrary port automaton can be decomposed into $\mathcal{F}$lip$\mathcal{F}$lops and $\mathcal{XOR}$s (the $\mathcal{S}$ync can be replaced by two $\mathcal{XOR}$s, as shown in Example 3.9). Note also that the number of required $\mathcal{F}$lip$\mathcal{F}$lops is $\lceil \log n \rceil$, where n is the number of states of the original automaton to be decomposed. However, as the $\mathcal{F}$lip$\mathcal{F}$lop does not correspond to a channel in Reo, we decompose it now further into ordinary $\mathcal{FIFO}$1s.

Step (iii) – Removing self loops

The final decomposition step replaces all occurences of $\mathcal{F}$lip$\mathcal{F}$lops by ordinary $\mathcal{FIFO}$1s. In essence, the difference between the two is that $\mathcal{FIFO}$1s have no self loop transitions. Figure 3.4 shows how to *unroll* such self loops, resulting in an automaton with four states. A more general version of this automaton, where each transition is uniquely labeled, and its further decomposition are depicted in Figure 3.5. The original can be regained by composing it with $\mathcal{XOR}(A,A_1,A_2)$, $\mathcal{XOR}(B,B_1,B_2)$, $\mathcal{XOR}(C,C_1,C_2)$ and $\mathcal{XOR}(D,D_1,D_2)$, and hiding $\{A_1,A_2,B_1,B_2,C_1,C_2,D_1,D_2\}$. Each of the four remaining primitives in Figure 3.5 can be constructed using a $\mathcal{FIFO}$1 and two binary $\mathcal{XOR}$s. For example, the top left hand automaton is equivalent to:

$$\big(\mathcal{FIFO}1(A,B) \bowtie \mathcal{XOR}(A,A_1,C_1) \bowtie \mathcal{XOR}(B,B_1,C_2)\big) \setminus \{A,B\}$$

Note that this step is an instance of a more general construction, wherein any port automaton without self loops can be directly constructed from $\mathcal{FIFO}$1s and $\mathcal{XOR}$s which more or less encode the automaton structure using the *FIFO1* buffers.[2] However, the number of $\mathcal{FIFO}$1s required in this construction is linear in the number of states and hence the resulting connector becomes unnecessarily large in practice.

[2]Based on an idea of Christel Baier.

In summary, we have shown that arbitrary port automata can be decomposed into $\mathcal{XOR}$s and $\mathcal{FIFO}1$s. The number of required $\mathcal{FIFO}1$s is $4\lceil \log n \rceil$, since we need four $\mathcal{FIFO}1$s to encode one $\mathcal{F}$lip$\mathcal{F}$lop. For example, an n-$\mathcal{S}$equencer requires n $\mathcal{FIFO}1$s in the original construction, or $\lceil \log n \rceil$ $\mathcal{F}$lip$\mathcal{F}$lops or $4\lceil \log n \rceil$ $\mathcal{FIFO}1$s using our techniques. For an 8-$\mathcal{S}$equencer, the required stateful primitives are 8 (original), 3 (using $\mathcal{F}$lip$\mathcal{F}$lops), and 12 (after encoding $\mathcal{F}$lip$\mathcal{F}$lops), whereas for a 16-$\mathcal{S}$equencer, the numbers are 16, 4, and 16, after which even our encoding using $\mathcal{FIFO}1$s becomes superior to the original.

3.2.3 Related work

Decomposition results for (deterministic) finite automata have a very long history, starting with the Krohn-Rhodes Theorem [66] from 1965. The collection of primitive automata underlying the Krohn-Rhodes Theorem include a $\mathcal{F}$lip$\mathcal{F}$lop, similar to but not the same as ours. By treating the labels of our automata as letters of the alphabet, these results apply to our automata, but do not give the results presented here. The key difference is that our labels have structure and our product and hiding operations manipulate that structure.

Port automata are an abstraction of constraint automata [10]. Arbab et al. [3] show how to synthesize Reo connectors from constraint automata, based on a small set of Reo primitives (hence, a small set of constraint automata). The approach, however, takes a side step through scheduled-data expressions (effectively, regular expressions for constraint automata), and hence the resulting connectors are large and unwieldy, and the resulting automata are *not* bisimilar to the original automata. As their approach deals with data, a larger number of primitives (9) are required. Our approach directly encodes automata in terms of other automata, which can then be represented as Reo connectors, requires a smaller number of primitives (2), and its encoding produces a bisimilar result.

An algebra for stateless connectors [22] was suggested by Bruni et al. The authors give a completeness result by showing that a finite number of primitives is sufficient to model all stateless connectors of Reo in this framework, although their axiomatization is not finite. The two main differences between this work and our model is that we do not consider the direction of data flow and that the approach of Bruni is restricted to stateless connectors only.

Expressions in Milner's SCCS [70] have actions corresponding to a multiset of atomic actions (elements of an Abelian group). Synchronization is built into the model automatically via the group's product operation. Hiding of names corresponds to matching actions with their co-action (inverse in the group). Synchronization in our model is different, based on the correspondence of names in the two automata being joined. We are unaware of any decomposition theorems for SCCS.

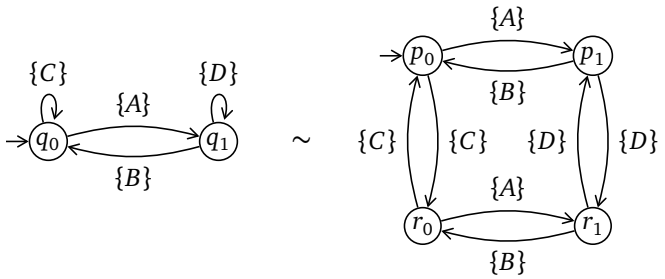


Figure 3.4: unrolling the self loops in a $\mathcal{F}$lip$\mathcal{F}$lop

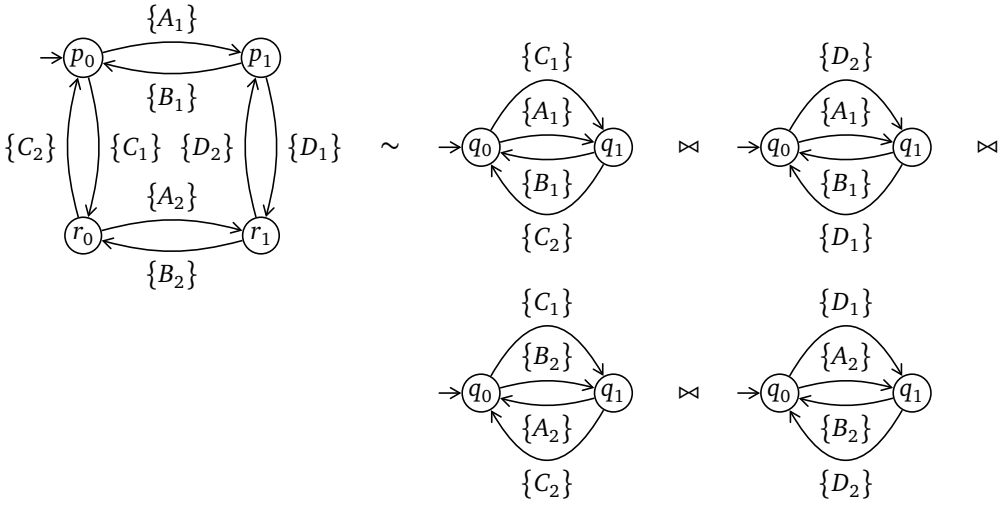


Figure 3.5: decomposing the unrolled automaton

3.3 The Extensible Automata framework in ECT

To support automata-based semantics for Reo in ECT, we have implemented the Extensible Automata (EA) framework. As its name suggests, extensibility was the guiding requirement for the design of this framework. This is due to the fact that besides constraint and port automata, there exist a number of variations of constraint automata and other automata models for Reo. The following list of automata types are currently supported by the EA tools:

- Port automata (PA) [90]
- Constraint automata (CA) [10]
- Constraint automata with memory (CAM) (see [82])
- Timed constraint automata (TCA) [2]
- Quantitative constraint automata (QCA) [5]
- Intensional automata (IA) [29]
- Quantitative intensional automata (QIA) [4]
- Reo automata (RA) [18]

Since many of these automata models are variations of each other, we chose to implement our automata tools in an extensible way. The EA framework consists of a meta-model for automata, together with an extensible implementation of product and hiding operations and a graphical automata editor. In the following we describe the architecture of this extensible framework in detail. The core of the EA framework was written by the author of this thesis.

3.3.1 The EA meta-model

The Extensible Automata framework in ECT is based on the EA meta-model, which is divided into two packages: *cwi.ea.automata* and *cwi.ea.extensions*, respectively shown in Figures 3.6 and 3.7.

The core automata model depicted in Figure 3.6 contains classes for automata, states and transitions, as well as modules, which serve as container objects for automata. The classes for states, transitions and automata all extend the abstract base class *ExtensibleElement* from the package *cwi.ea.extensions*. This package contains the two interfaces *IExtensible* and *IExtension* which form the core of the framework. An instance of *IExtensible* owns a number of extensions, which are typed using an ID. The method findExtension() in *IExtensible* can be used to find an extension based on its ID. The method updateExtension() adds or replaces an extension. The package moreover contains abstract classes for extensible and extension elements, as well as some basic extension implementations. Extensions are essentially annotations in the form of key-value pairs, where the key is an ID and the value can be of a custom type. As can be seen from Figure 3.6, automata, states and transitions are extensible in this way. Table 3.3 summarizes all currently available extension types in the EA

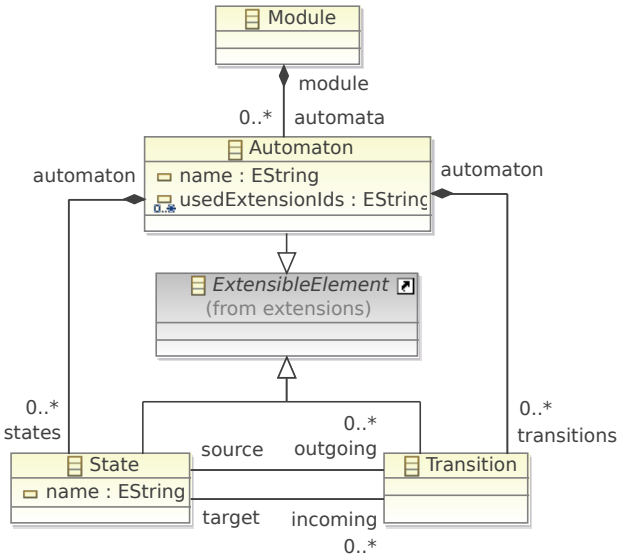


Figure 3.6: EA meta-model, package *cwi.ea.automata*

framework. Note that it is also possible to define dependencies between extensions, e.g. the port names extension for transitions always requires the port names extension for automata as well.

3.3.2 Extension providers

Extensions, i.e., instances of *IExtension* are mere data objects that are used as annotations in automata models. The life cycle of these data objects is managed using so-called *extension providers*. Extension providers are classes that implement the interface *IExtensionProvider* and that are registered in the EA runtime. For each of the extension types in Table 3.3 there exists a separate extension provider. The EA runtime uses these extension provides for the following tasks:

- instantiating default extensions,
- parsing and pretty-printing of extensions,
- pairwise joining of extensions,
- validating extensions.

Optionally, an extension provider can customize the graphical representation of states, transitions or automata to display their extensions. Figure 3.8 depicts the constraint automaton for the *FIFO1* in the graphical EA editor.

Validation of extensions is useful to automatically detect problems in the graphical editor. For instance, validation is used to ensure that data constraints in a transition refer only to enabled ports, or that state memory cells are always properly initialized.

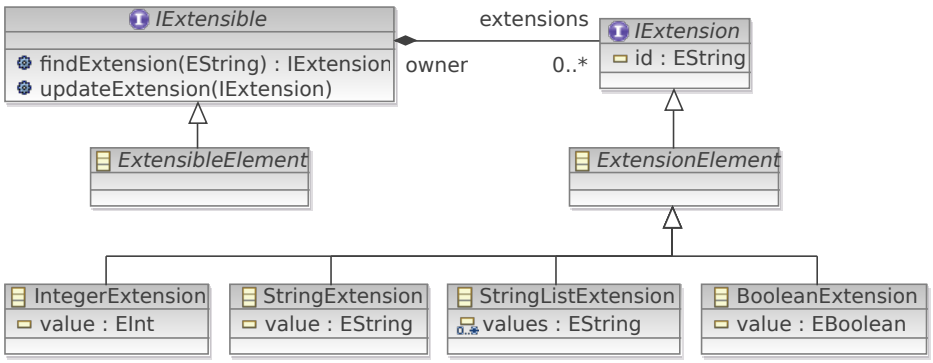


Figure 3.7: EA meta-model, package *cwi.ea.extensions*

The EA runtime also uses extension providers to compute the product or join of automata in a compositional manner. Every extension provider contains a method that checks whether two extension objects can be joined together and, if that is the case, returns the joint extension. However, different automata types usually come with different product definitions. We, therefore, define product implementations also using an extensible mechanism.

3.3.3 Product providers

Product providers are classes that implement the functionality to join two automata, defined in the interface *IProductProvider*. Product providers can be implemented specifically for a particular automata type, or in a generic way. In the latter case, the product provider uses the extension providers to join two extensions. This has the advantage that the product implementation is completely reusable.

There are three different product implementations available in the EA tools. All automata types that we consider are used as semantics for Reo. The informal semantics of Reo describes truly concurrent, as well as interleaved behavior. Therefore, we have implemented a default product implementation that computes the join of two automata of the same type by:

(i) pairwise joining the automaton extensions,

(ii) computing the cartesian product of the state sets and pairwise joining the state extensions,

(iii) computing the cartesian product of the transition sets and pairwise joining of the transition extensions,

(iv) joining all transitions with dummy τ-transitions in the other automaton.

Extension type	Data type	Enable for	Automaton types
Initial states	*boolean*	states	PA,CA,CAM,QCA,TCA,RA,IA,QIA
Port names (automata)	*string list*	automata	PA,CA,CAM,QCA,TCA,RA,IA,QIA
Port names (transitions)	*string list*	transitions	PA,CA,CAM,QCA,TCA,RA
Intensional port names	*custom*	transitions	IA,QIA
Data constraints	*custom*	transitions	CA,CAM,TCA,QCA
State memory	*string list*	states	CAM
Clocks	*string list*	automata	TCA
Clock guards	*custom*	transitions	TCA
Clock updates	*string list*	transitions	TCA
Clock state invariants	*custom*	states	TCA
Cost algebras	*custom*	automata	QCA
Cost values	*custom*	transitions	QCA
Guards	*custom*	transitions	RA

Table 3.3: list of available automata extension types

Step (iii) computes the concurrent firing of two transitions, whereas step (iv) takes care of the interleaved firing semantics. This product implementation is generic since it uses the corresponding extension providers and makes no assumption on the automaton type. The pairwise joining of extensions is the key to generic product definitions. Examples for pairwise joining of extensions include:

- logical 'and' for initial state extensions,
- set union for port name extensions,
- conjunction for data constraints.

It is important to note that the pairwise joining of extensions may not succeed because extension specific conditions are not fulfilled. In that case, the two automata, states or transitions cannot be joined. An example of such a condition can be found in the premise of rule 3.2 in the definition of the join operator for constraint automata. This particular condition is implemented in the extension provider for port name extensions on transitions.

The default product implementation described above can yield unreachable states in the product automaton. One reason for this is that it is oblivious to the concept of initial states – it treats them just like any other extension. However, for efficiency reasons, it can be beneficial to compute the product state space by starting from the initial states and traversing the automata along their transitions. If two transitions cannot be joined because an extension specific requirement is not fulfilled, the product implementation can discard the rest of the current path. Thereby, no unreachable states are computed. We have implemented such a product definition in the EA tools. It is generic like the default product, except that it requires and interprets the initial state extensions.

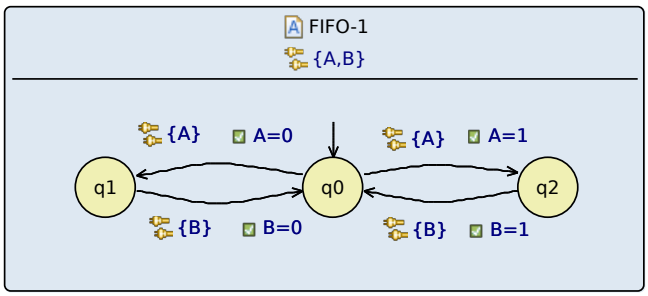


Figure 3.8: constraint automaton for *FIFO1* in the graphical EA editor

There exists a third product implementation which is not compatible with the generic ones described above. This product has been specifically implemented for (quantitative) intensional automata. It includes a refinement operator which has been defined specifically for this type of automata. Moreover, Reo automata [18] also require an additional synchronization operator which has not been implemented yet in the EA framework.

All product implementations can be invoked programmatically and from the graphical EA editor.

3.3.4 From Reo to automata models

Although it provides tools for semantical models of Reo connectors, the Extensible Automata framework does not have any dependencies on ECT's core tools for Reo. To derive semantical automata models from Reo diagrams we have implemented an additional framework in the plug-in *cwi.reo2ea*. Given a Reo connector or network and a target automaton type, the runtime of this framework performs the following steps to derive an automaton of that type:

(i) map every primitive in the connector or network to a primitive automaton,

(ii) join all primitive automata using a product implementation, and

(iii) hide internal ports (optional).

The primitive automata are defined in templates. The mapping from primitives to automata is performed based on the type of the primitive. If the primitive is user-defined, the conversion tool looks for semantical annotations in the primitive or tries to resolve its type to a known primitive. This way, pre-defined as well as user-defined primitives are supported by the conversion.

Since all of our product definitions are associative and commutative, the order in which we join them does not matter. However, for performance reasons, the joining is done in a binary, bottom-up fashion. Automata that have a larger set of overlapping

```
1  public interface Sink<T> extends Port<T> {
2
3    /**
4     * Write data to this port
5     * @param o Object to be written
6     */
7    void write(T data) throws InterruptedException;
8
9    /**
10    * Write data to this port with a timeout
11    * @param o the object to be written
12    * @param timeout in nanoseconds
13    */
14   void write(T o, long timeout) throws InterruptedException,
15                                        TimeoutException;
16 }
```

Listing 3.1: interface 'Sink' of the constraint automata runtime

port names are joined first, since they potentially contain more synchronous behavior and therefore yield a smaller product automaton.

The conversion from Reo to automata models can be invoked from the graphical Reo editor and programmatically. The conversion tool was written by Ziyan Maraikar and Young-Joo Moon.

3.3.5 CA runtime and code generation

Constraint automata not only allow formal reasoning about Reo connectors, but they can be also used to execute them as finite state machines. In this approach, a connector is first converted into a constraint automaton which is then executed by an interpreter engine or generated code, which serve as a centralized coordinator.

A constraint automaton specifies the set of ports that must be active to trigger a transition from a state. Once a transition is triggered, synchronous data transfer specified by its constraint occurs between the active ports. The blocking semantics of ports is implemented using so-called *synchronization points*, which are equivalent to Hoare's CSP channels [54]. Synchronization points can be implemented using common concurrency primitives such as mutexes and condition variables. Components communicate with the coordinator using basic *read* and *write* operations on their ports. The interface for sink ports is given in Listing 3.1. Source ports are defined analogously. Both the generated code and the interpreter depend on a lightweight and self-contained runtime library which provides an implementation of synchronization points, using language-specific synchronization primitives. The code generator and interpreter support user-defined predicates in constraints which are needed to

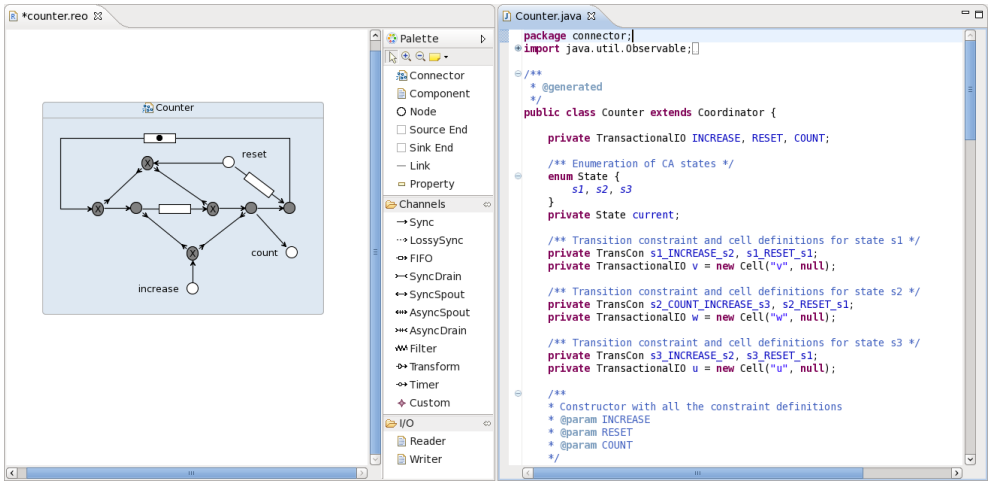


Figure 3.9: Java code generator in ECT

implement *Filter* channels. Similarly, data transformations using the *Transform* channel are also supported.

Currently the code generator produces Java code, but the entire code generation framework is retargetable. Defining a new code generation target involves implementing a set of code generation templates and porting the runtime library which contains the implementation of synchronization points. An experimental support for C as target platform is also available. The code generation can be invoked either from the graphical automata editor or the Reo editor itself. In the latter case, a constraint automaton is first derived from the Reo connector, as described above. An example of generated Java code for the counter connector is shown in Figure 3.9. The constraint automata runtime and code generation have been implemented by Ziyan Maraikar and are located in the package *cwi.ea.runtime*.

3.4 Conclusions

Port automata, as introduced in the beginning of this chapter, can be considered as the most basic semantical model for Reo. Although they abstract away data constraints, they already capture some of the key concepts required for defining channels and other primitives, i.e., synchrony, mutual exclusion and state. Moreover, they are compositional. Thus, the semantics of a connector or network can be derived using the join and hiding operators for port automata.

Port automata have been introduced as an abstraction of constraint automata [10] and offer a possibility for better comparison with other semantical models, such as

Reo automata [18]. Because of their simplicity, we also use port automata in Chapter 4 for an encoding into the mCRL2 specification language, and in Chapter 7 in the context of dynamic reconfigurations.

In recent work (cf. [55]) we have also shown that port and constraint automata are in fact sufficiently expressive to model context-dependency in Reo (cf. [26, 18, 29]). We discuss this result further in Section 4.3.4 and show in Section 4.4.2 how it can be used to validate context-dependent connectors in the Vereofy model checker, which is based on the constraint automata semantics of Reo.

In Section 3.2, we have shown using a decomposition scheme that in the port automata model, only two primitives ($\mathcal{FIFO}1$ and $\mathcal{XOR}$) are required for decomposing an arbitrary port automaton. Moreover, this decomposition scheme produces a very compact representation of the network and can thus be used for connector synthesis.

The Extensible Automata framework in ECT provides a powerful means to define automata models in an extensible way. Various automata formats are supported and can be automatically derived from Reo models. These automata models can be used for further analysis or can serve as a basis for generating executable code.

3.5 Discussion

The EA framework as presented in Section 3.3 provides a unifying implementation of a number of different automata-based semantics for Reo. The rather large number of different semantical (mostly automata-based) models for Reo can be justified by the argument that each of these models serves a particular modeling aspect (e.g. context-dependent, timed or probabilistic behavior) or improves some of the drawbacks of another model.

However, we believe that a unified semantical model for Reo can be of great use, since it would fix the formal semantics and remove inconsistencies between the different models. Furthermore, it is the authors believe that basic models such as port or constraint automata provide in fact already enough expressiveness for a core semantics of Reo. Supporting evidence is the result of [55], which shows that context-dependency in Reo, the key motivation for the coloring semantics [26] and Reo automata [18], can in fact be expressed in these basic models.

In the area of stochastic analysis for Reo, two main modeling approaches can be distinguished, i.e., the rather low-level quantitative intensional automata [4], and the more recently developed stochastic Reo automata [75]. In the opinion of the author of this thesis, it is still an open question which of these two approaches is more suitable. On the one hand, stochastic Reo automata are much more compact and there are more theoretical results on their correctness and compositionality readily available. On the other hand, quantitative intensional automata have the advantage that they are much closer to the Markovian models that serve as the final stochastic semantics in both approaches.

In this chapter, we have discussed only automata-based models for Reo. A Petri net semantics for Reo was considered, e.g., in [84]. However, Petri nets have a rather local

notion of synchronization, i.e. the *propagation* of synchrony (or context-dependency, for that matter) in Reo cannot be properly modeled in Petri nets, at least not in a compositional manner. For example, a natural choice to model a *Sync* channel as a Petri net is to use a simple transition, connecting an input place A with an output place B. Such a transition indeed models a synchronous data transfer from A to B. However, this synchrony is lost as soon this channel is composed with another channel, e.g. a *Sync* channel between B and C, modeled using another transition.

The lack of compositionality of Petri net semantics for Reo is the main motivation for us to use automata-based or process algebra models (e.g. the encoding in mCRL2 that we introduce in Chapter 4). Both approaches provide a more suitable notion of synchronization which is, moreover, compositional. In a nutshell, supporting evidence for the claim that automata-based and process-algebraic models provide a solid basis for formal modeling and verification in Reo are the existence of:

(i) theoretical results on their correctness and compositionality (see Chapter 3 and 4),

(ii) efficient formal verification techniques using model checkers (see Chapter 4),

(iii) engine implementations based on these models (see Section 3.3.5 and 5.5).

In conclusion, we believe that when it comes to formal modeling and analysis, simplicity and compositionality of a semantic model should be valued higher than a (seemingly) larger expressive power.

Chapter 4

Verification by model checking

In this chapter, we discuss verification of Reo connectors and networks using model checking. We do this by encoding Reo primitives into the mCRL2 specification language. Our encoding is compositional and based on the constraint automata semantics. We use the mCRL2 toolset to generate and visualize the corresponding state space and to validate behavioral properties. We show the correctness of the encoding with respect to the port automata semantics and discuss an alternative encoding that supports context-dependency. Finally, we present our tool support in ECT including additional analysis tools for qualitative and quantitative behavioral properties.

4.1 Overview

Verification of connector and network behavior can be done in ECT by generating animations and automata models. However, these two techniques do not scale very well and thus must be used with caution in larger applications. Furthermore, they do not provide any means for automatic validation of system properties. This can be crucial for ensuring correctness of an application.

We therefore propose another way to verify the behavior of connectors. We encode the behavior of Reo primitives in mCRL2, a behavioral specification language that is based on the process algebra ACP, with additional support for data and time [49]. Specifications in this language can be analyzed using an extensive set of model checking, simulation and visualization tools available in the mCRL2 toolset. A specification can also be converted into a labeled transition system (LTS) in various formats and subsequently used as input for external model checking tools, in particular CADP [43]. Both mCRL2 and CADP have proven their suitability for analyzing large scale applications of industrial strength. The mCRL2 specification language also supports algebraic data types and user-defined functions, which are essential features for a data-aware analysis of connectors. This includes full support for filtering and transforming structured data elements produced by components or services. Another important aspect of this approach is its compositionality: we encode Reo primitives as well as the join

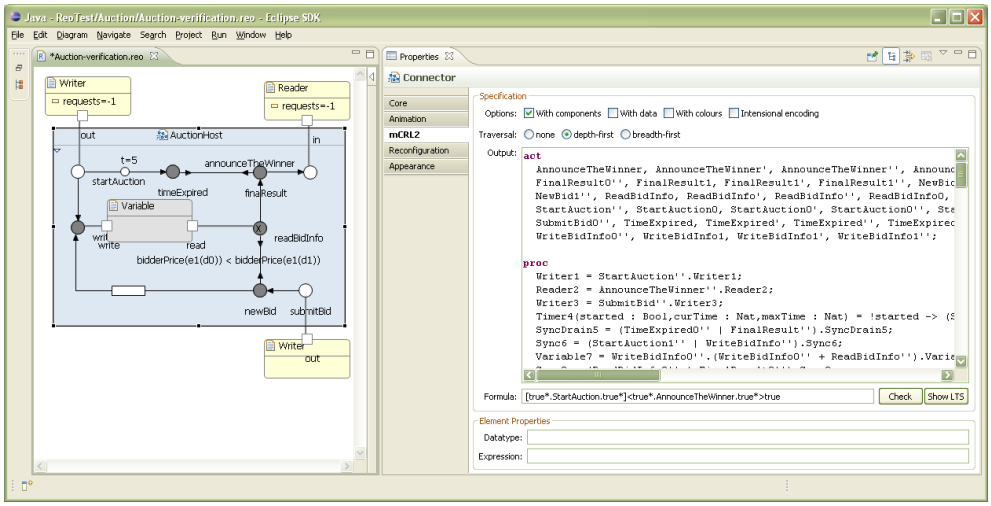


Figure 4.1: an auction process in Reo and the derived mCRL2 specification

and hiding operator in mCRL2. The generation of the state space of the whole system can then be efficiently handled by mCRL2 itself. Our encoding is based on the constraint and port automata semantics of Reo. To support context dependency, we also provide an alternative encoding based on the coloring semantics [26]. Furthermore, we show how timed Reo channels can be modeled as well.

An encoding such as we propose here is of theoretical value, since it gives an insight into the expressiveness of the original model. However, our goal is to provide an actual tool that scales well and that is of practical use. We therefore implemented a conversion tool that takes Reo models as input and produces mCRL2 code. This conversion is highly customizable and optimized for an efficient processing by the mCRL2 tools. The specifications are generated fully automatically and do not require any manual refinement. Just like the other analysis tools in ECT, the mCRL2 converter is tightly integrated with the rest of development environment. This includes invocation of mCRL2 toolchains through a simple user interface.

Organization

The rest of this chapter is organized as follows. In Section 4.2 we recall the basic concepts of the mCRL2 specification language. In Section 4.3 we present our encoding of Reo in mCRL2 and prove its correctness. In Section 4.4 we discuss the general tool support for verification in ECT, but with emphasis on the mCRL2 conversion tool. A screenshot of this tool is shown in Figure 4.1. We discuss optimizations and a small case study and compare the verification approach using mCRL2 with other existing solutions. Section 4.5 and 4.6 contain related work and conclusions.

4.2 The mCRL2 specification language

In this section, we give a brief overview of the mCRL2 specification language and toolset. For more details we refer to [49] and to the mCRL2 website [69].

4.2.1 Actions

The most basic notion in mCRL2 are actions, which represent atomic events and can be parameterized with data. A distinguishing feature between mCRL2 and its predecessor μCRL is its support for multiactions, which allow modeling of truly concurrent behavior. Multiactions are constructed from other actions or multiactions using the synchronization operator $|$, e.g. the multiaction $a|b$ represents a concurrent firing of the atomic actions a and b. A special τ-construct is used to refer to an internal unobservable action.

4.2.2 Processes

Processes in mCRL2 are defined by process expressions, which are essentially compositions of other processes and atomic (multi)actions at the lowest level. The basic operators for defining processes in mCRL2 are:

- *deadlock* or *inaction* δ, which does not display any behavior;
- *alternative composition*, written as $P + Q$, which represents a non-deterministic choice between the processes P and Q;
- *sequential composition*, written as $P \cdot Q$, which means that Q is executed after P, given that P terminates;
- *conditional* or *if-then-else* construct, written as $c \rightarrow P \diamond Q$, where c is a data expression that evaluates to true or false;
- *summation* $\Sigma_{d:D} P$, where P is a process expression in which the data variable d may occur, used as a for-all quantifier over a data domain D;
- the '*at*'-operator $a@t$, indicating that the action a happens at time t;
- *parallel composition* or *merge*, written as $P \parallel Q$, which interleaves and synchronizes the actions of P with those of Q;
- *restriction* $\nabla_V(P)$, where V specifies which actions from P are allowed to occur;
- *encapsulation* $\partial_H(P)$, where H is a set of action that are not allowed to occur;
- *renaming* $\rho_R(P)$, where R is a set of renamings of the form $a \mapsto b$, meaning that every occurrence of action a in P is replaced by the action b;
- *communication* $\Gamma_C(P)$, where C is a set of communications of the form $a_0|...|a_n \mapsto c$, which means that every group of actions $a_0|...|a_n$ within a multiaction is replaced by c;
- *hiding* $\tau_M(P)$, which hides all actions in M in all multiactions in P.

4.2.3 Data types

The mCRL2 language provides a number of built-in data types such as Booleans, natural and positive numbers, integers and real numbers. All standard arithmetic operations for them are predefined. Custom data type definitions in mCRL2 allow users to declare new sorts, constructors and functions. A structured data type is declared in mCRL2 in the following way:

$$S = \textbf{struct}\ \ c_1(p_1^1{:}S_1^1, \ldots, p_1^{k1}{:}S_1^{k1})?r_1\ \ |\ldots|\ \ c_n(p_n^1{:}S_n^1, \ldots, p_n^{kn}{:}S_n^{kn})?r_n$$

This construct defines the type S together with constructors $c_i : S_i^1 \times \ldots \times S_i^{ki} \to S$, projections $p_i^j : S \to S_i^j$, and recognizers $r_i : S \to \mathit{Bool}$. Various examples of data type definitions can be found in the mCRL2 language reference.

4.2.4 Tools

The mCRL2 toolset contains a large number of tools that can be used to verify systems specified in the mCRL2 language. The toolset includes a tool for converting an mCRL2 specification into a so-called linear process specification (LPS), which is a compact, symbolic representation of the system that can be used for subsequent analysis. A linear process specification can be used directly for model checking, or converted into a labeled transition system (LTS), if it is finite. The mCRL2 toolset contains utilities to compare and minimize labeled transitions systems based on various notions of equivalence, such as strong and branching bisimulation or trace equivalence. A tool for visualization of labeled transition systems is also included.

For model checking purposes, system properties are specified as formulas in a variant of the modal μ-calculus, extended with regular expressions, data and time. In combination with an LPS, such a formula is transformed into a parameterized Boolean equation system (PBES) and can be solved with the tools from the mCRL2 toolset. Model checking of labeled transition systems is not supported by mCRL2. However, the CADP [43] toolbox can be used for this purpose.

4.3 Encoding Reo in mCRL2

Our encoding reflects the constraint automata semantics of Reo. For every channel we define a process based on two atomic actions that model the data flow events at its respective ends. Analogously, we map a node to a process and corresponding actions for all adjacent channel ends. An important aspect of our encoding is that data constraints are translated faithfully. We therefore extend the actions corresponding to channel and node ends with data parameters. In the context of a given connector, we assume a global datatype, which we model as a sort *Data* in mCRL2, which makes sense since most of the channel types in Reo are agnostic to the actual type of data items they carry. Based on such a global type, we can use the summation operator in mCRL2 to define data constraints imposed by channels. These are already the ingredients required for an encoding of the basic channel types.

$Sync = \Sigma_{d:Data}\ A(d)\|B(d) \cdot Sync$
$LossySync = \Sigma_{d:Data}\ (A(d)\|B(d) + A(d)) \cdot LossySync$
$SyncDrain = \Sigma_{d_1,d_2:Data}\ A(d_1)\|B(d_2) \cdot SyncDrain$
$AsyncDrain = \Sigma_{d:Data}\ (A(d) + B(d)) \cdot AsyncDrain$
$FIFO(b{:}DataFIFO) = \Sigma_{d:Data}(isEmpty(b) \rightarrow A(d) \cdot FIFO(full(d)) \diamond B(e(b)) \cdot FIFO(empty))$
$Filter = \Sigma_{d:Data}\ (pred(d) \rightarrow A(d)\|B(d) \diamond A(d)) \cdot Filter$
$Transform = \Sigma_{d:Data}\ A(d)\|B(func(d)) \cdot Transform$
$Merger = \Sigma_{d:Data}\ (A(d)\|C(d) + B(d)\|C(d)) \cdot Merger$
$Replicator = \Sigma_{d:Data}\ A(d)\|B(d)\|C(d) \cdot Replicator$

Table 4.1: mCRL2 encoding for channels and nodes

The encodings for the basic channel types, together with *Merger* and *Replicator* primitives for encoding nodes, are summarized in Table 4.1. Note that the encoding of the *SyncDrain* requires a double summation, since it allows to drain two arbitrary data items synchronously. The *Filter* and *Transform* use additional data expressions, i.e., a predicate *pred* for filtering, and a user-defined function *func* for transforming data items. As in the constraint automata model, the *LossySync* non-deterministically loses or passes data in the mCRL2 version. The encoding of the *FIFO1* channel includes a parameter for modeling the state of its buffer, for which we need to introduce a new datatype, defined as follows:

$$DataFIFO = \textbf{struct}\ empty?isEmpty \mid full(e{:}Data)?isFull$$

An element of this data type allows to specify whether the buffer of the *FIFO1* is empty or full, and if it is full, the value stored in it. We have argued already that nodes can be encoded using basic *Merger* and *Replicator* primitives, as included in Table 4.1. However, in our tool we generate a single process for a given node by examining the number of incoming and outgoing channel ends. Similarly, our tool can directly encode nodes with exclusive router semantics. In practice, a third type of node comes in handy, which implements a tupling of incoming data items, as described informally in Section 2.2.2. We refer to this type of node as a *tuple* node here. To encode it in mCRL2 we need to extend our global data type with tuples. In the following, we assume that the global datatype is the union of n user-defined datatypes, which we refer to as $\mathcal{D}_1, \ldots, \mathcal{D}_n$. In practice, they are defined by the coordinated components or services. Given that, we define the global datatype as:

$$Data = \textbf{struct}\ D_1\ (e_1{:}\mathcal{D}_1) \mid \ldots \mid D_n\ (e_n{:}\mathcal{D}_n) \mid tuple\ (p_1{:}Data, p_2{:}Data)$$

This definition allows us to instantiate elements of any basic type as well as binary tuples, thus forming tree-like structures. Note that this datatype is suitable for tuple

nodes that have two incoming ends only. In the general case, for every tuple node with k incoming ends a $tuple_k(p_1{:}Data, ..., p_k{:}Data)$ must be added to the definition. Based on this extended global data type we can encode a tuple node with two incoming ends A and B, and two outgoing end C and D as follows:

$$\begin{aligned} \mathit{TupleNode} = \Sigma_{d_1,d_2:Data}\big((A(d_1)|B(d_2)|C(\mathit{tuple}(d_1,d_2)) + \\ (A(d_1)|B(d_2)|D(\mathit{tuple}(d_1,d_2)))\big) \cdot \mathit{TupleNode} \end{aligned}$$

4.3.1 Join and hiding operations

After generating process definitions for all channels and nodes, we need to compose them into one joint process which models the whole connector. We do this in three steps:

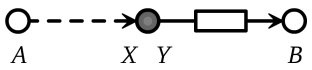
(i) forming the parallel composition of all channel and node processes,

(ii) synchronizing the actions for coinciding channel and node ends, and

(iii) hiding the internal actions (optional).

Step (ii) is achieved by an application of two mCRL2 operators: communication and blocking. The communication is used to merge a multiaction of two coinciding channel and node ends into one basic action. The blocking operator is then used to disallow unsynchronized events at the channel / node ends. The hiding operator can finally be used to abstract from internal data flow. We illustrate this composition using an example.

Example 4.1 (connector construction in mCRL2)**.** We consider the following simple connector which consists of two channels (*LossySync* and *FIFO1*) and three nodes:



For simplicity, we assume that the channel ends at A and B form the boundary of the connector, without explicit nodes. Thus, the system we want to model consists of two channels and one node which are encoded by the following three processes:

$$\begin{aligned} \mathit{LossySync} &= \Sigma_{d:Data}\ \big(A(d)|X_1(d) + A(d)\big) \cdot \mathit{LossySync} \\ \mathit{Node} &= \Sigma_{d:Data}\ X_2(d)|Y_2(d) \cdot \mathit{Node} \\ \mathit{FIFO}(b{:}\mathit{DataFIFO}) &= \Sigma_{d:Data}\ \mathit{isEmpty}(b) \rightarrow Y_1(d) \cdot \mathit{FIFO}\left(\mathit{full}(d)\right) \\ &\quad \diamond B(e(b)) \cdot \mathit{FIFO}(\mathit{empty}) \end{aligned}$$

Note that we use the projection e of the data type *DataFIFO* to access the data element stored in b. For obtaining the process for the connector as a whole, we first form the parallel composition of the three processes. Then we add communications for connected channel and node ends and block unsynchronized actions. In the last step

we hide the actions representing the data flow at the internal node. In the mCRL2 syntax, this reads as follows:

$$LossyFIFO = \tau_{\{X,Y\}}\left(\partial_{\{X_1,X_2,Y_1,Y_2\}}\left(\Gamma_{\{X_1|X_2\mapsto X,Y_1|Y_2\mapsto Y\}}\left(LossySync \,\|\, Node \,\|\, FIFO\right)\right)\right)$$

In practice, however, this direct approach of naively running all processes in parallel and then performing the communication, synchronization and optionally the hiding operator leads to a state space explosion during the linearization. To overcome this problem, we add processes one by one and immediately apply the composition and hiding operators. The *LossyFIFO* connector can thus be constructed also as follows:

$$LossyFIFO_1 = \tau_{\{X\}}\left(\partial_{\{X_1,X_2\}}\left(\Gamma_{\{X_1|X_2\mapsto X\}}\left(LossySync \;\|\; LossyFIFO_2\right)\right)\right)$$
$$LossyFIFO_2 = \tau_{\{Y\}}\left(\partial_{\{Y_1,Y_2\}}\left(\Gamma_{\{Y_1|Y_2\mapsto Y\}}\left(Node \;\|\; FIFO\right)\right)\right)$$

Here we first compose the node and the *FIFO*, synchronize and hide their connected ends yielding $LossyFIFO_2$, and then continue with the rest of the connector. The actual performance gain in the runtime for this optimization is discussed in Section 4.4.1. △

4.3.2 General port automata encoding

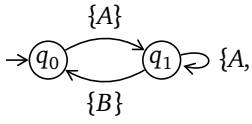
We now show the correctness of our mCRL2 encoding with respect to a simpler, but more general port automata encoding. However, the step from port to constraint automata is only of technical nature. The important points are that the constructed processes exhibit the same behavior and that join and hiding operations are preserved.

Let $PA = (Q, N, T, q_0)$ be an arbitrary port automaton. We interpret the elements of N as atomic actions and subsets as multiactions. For each state $q \in Q$ we now define the process $proc(q)$ by:

$$proc(q) = \sum_{q \xrightarrow{S} p} S \cdot proc(p) \tag{4.1}$$

where $S = \prod_{x \in S} x$ represents the multiaction composed from all ports in the sets. In this view, it comes natural to have for the synchronization $S_1|S_2$ of actions S_1 and S_2 the union of the underlying port names $S_1 \cup S_2$. States with no outgoing transitions are mapped to δ. The process $proc(q_0)$ corresponds to the initial process of the resulting specification.

Example 4.2 (mCRL2 port automata encoding)**.**



$$\Rightarrow \quad \begin{cases} proc(q_0) = A \cdot proc(q_1) \\ proc(q_1) = A|B \cdot proc(q_1) + B \cdot proc(q_0) \end{cases}$$

△

We can now verify the correctness of our mapping. We do this by showing bisimilarity of the states in the port automaton with the operational semantics of the derived

processes. The operational semantics of mCRL2 is defined in [49] using SOS rules in the style of Plotkin [78, 79].

THEOREM 4.3 (correctness of mCRL2 port automata encoding)**.** *Given an arbitrary port automaton $PA = (Q, N, T, q_0)$ and the encoding $proc(q)$ of a state $q \in Q$, defined in* (4.1)*, it holds that $q \sim proc(q)$.*

PROOF. The relation $R = \{\langle q, proc(q)\rangle \mid q \in Q\}$ is a strong bisimulation according to Definition 3.4:

(i) Suppose $q \xrightarrow{S} p$ in PA. Then by (4.1), $proc(q) = \ldots + S \cdot proc(p) + \ldots$ where this transition is mapped to the process expression $S \cdot proc(p)$. From the semantics of $\cdot$ we obtain $S \cdot proc(p) \xrightarrow{S} proc(p)$, and $\langle p, proc(p)\rangle \in R$.

(ii) Suppose $proc(q) \xrightarrow{S} P$. Then, by (4.1), this can be caused only by a term $S \cdot proc(p)$ in the definition of $proc(q)$ for some state $p \in Q$ and a transition $q \xrightarrow{S} p$. Therefore we have symmetrically $P = proc(p)$, and $\langle p, proc(p)\rangle \in R$. □

This shows that our simple encoding indeed preserves the port automata semantics. However, since our mapping also uses composition and hiding operators of mCRL2, we need to make sure that they correctly implement the join and hiding operators for constraint or port automata. We show this in the following.

THEOREM 4.4 (mCRL2 encoding of join)**.** *Let $PA_i = (Q_i, N_i, T_i, q_0^i)$ for $i \in \{1, 2\}$ be two port automata. We define two sets of port name renamings R_i as:*

$$R_i = \{x \mapsto \langle x, i\rangle \mid x \in N_1 \cap N_2\}$$

Let PA_i^R be the two port automata derived from PA_i by applying the renamings R_i,

- *$proc(q_i)$ the process corresponding to the state q_i in PA_i^R, and*
- *$proc(q_1, q_2)$ the process corresponding to the state $\langle q_1, q_2\rangle$ in $PA_1 \bowtie PA_2$.*

Further, we define B the set of blockings and C the set of communications as follows:

$$B = \{\langle x, i\rangle \mid x \in N_1 \cap N_2\}$$
$$C = \{\langle x, 1\rangle | \langle x, 2\rangle \mapsto x \mid x \in N_1 \cap N_2\}$$

Then the following equivalence holds:

$$proc(q_1, q_2) \sim \partial_B(\Gamma_C(proc(q_1) \,\|\, proc(q_2)))$$

PROOF. A detailed proof is given in Appendix A.1. □

In the following we show that the encoding of hiding is also correct.

THEOREM 4.5 (mCRL2 encoding of hiding)**.** *Let $PA = (Q, N, T, q_0)$ and $q \in Q$. Let q' the corresponding state in $PA\backslash M$ with $M \subseteq N$. Then the following holds:*

$$proc(q') \sim \tau_M\left(proc(q)\right)$$

PROOF. The semantics of τ_M is given in [49] by:

$$\frac{p \xrightarrow{S} p'}{\tau_M(p) \xrightarrow{\theta_M(S)} \tau_M(p')}$$

where θ_M removes all occurrences of elements in M from the multiaction S. This corresponds exactly to the hiding operator for constraint or port automata in Definition 3.3. Therefore, bisimilarity is easily established. □

4.3.3 Encoding of the coloring semantics

Constraint automata have one drawback: they lack support for context-dependent behavior, as for instance required for the *LossySync* channel. Intuitively, the *LossySync* should lose data items only if the party at the receiving end is not ready to accept it. This type of context-dependent behavior of connectors has mainly been studied by Clarke et al. and has led to various formal models. Most notably, intensional (constraint) automata [29], the coloring model [26], and most recently, Reo automata [18] have been introduced to grasp the concept of context-dependency.

Since context-dependency is an important feature of Reo, we consider it also in our verification approach using mCRL2. Mainly for its simplicity, we chose to encode the coloring semantics in mCRL2. However, since the concepts are very similar in all three models, it should be straight-forward to use a different model instead.

The coloring semantics

The basic idea of the coloring semantics is to associate flow and no-flow colors to primitive ends. Clarke et al. showed in [26] that one *flow* color and two *no-flow* colors are sufficient to model context-dependency as for instance required by the *LossySync*. The names and graphical representations of these colors are shown in Table 4.2.

The color *flow* represents ordinary data flow, just like in constraint automata. The other two colors model both *no-flow*. Additionally they encode a direction of the reason for the fact that no flow is possible. Intuitively, *no-flow-provide-reason* models the fact that the receiving or sending party is not ready to perform an I/O operation. Conversely, *no-flow-require-reason* says that the party is ready to accept data, but for some other reason the data flow is not possible. Valid behaviors of channels are then described as colorings of their respective ends. Table 4.3 depicts the colorings of the *Sync*, *LossySync* and *Merger* primitive.

Note that the colors are always read from the perspective of the adjacent nodes. For instance, in coloring (4.3) of the *Sync* the sink node at the right end provides a reason for no flow, whereas the source node on the left requires a reason. This models

Name	Symbol
flow	
no-flow-provide-reason	
no-flow-require-reason	

Table 4.2: colors and their graphical representations

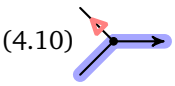
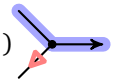
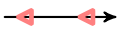
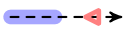
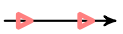
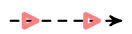
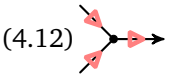
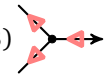
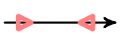
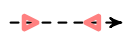
Sync	*LossySync*	*Merger*
(4.2)	(4.6)	(4.10)  (4.11) 
(4.3) 	(4.7) 	
(4.4) 	(4.8) 	(4.12)  (4.13) 
(4.5) 	(4.9) 	

Table 4.3: example colorings for some Reo primitives

the behavior where data is available at the source end but the receiver at the sink end is not ready to accept data. Similarly, in coloring (4.4) there is no flow, because there is no data available at the source end. Finally, coloring (4.5) models the situation where no data is available and the receiver is also not ready to accept any data.[1]

The encoding of the *LossySync* differs from the one of the *Sync* only in one coloring, i.e., coloring (4.7) where the sink node is not ready to accept data, but there is data available at the source end. In this situation the *LossySync* permits flow at the source end and loses the data item. Otherwise, no-flow behaviors are possible only when no data is available at the source end.

Nodes are encoded in the same way as channels in the coloring semantics. As usual, we can build nodes out of mergers and replicators. Table 4.3 also depicts the valid colorings of the *Merger* primitive.[2] An interesting fact here is that intuitively the colorings allow a propagation of no-flow reasons through the connector. We refer to [26] for a formal description of the composition operator in the coloring model.

Colorings in mCRL2

In mCRL2 we can encode color by simple data parameters of actions. Therefore, we need to introduce a new datatype:

$$Colored = \textbf{struct}\ flow(data{:}Data) \mid provide \mid require$$

[1] This behavior is implied by the so-called *flip rule* in [26].

[2] Colorings implied by the flip rule are not required for nodes, since the channels already allow a 'collision' of no-flow reasons, e.g. in coloring (4.5). This helps to remove equivalent colorings and, thus, to keep the number of valid colorings of a connector or a network small.

where *Data* is the global datatype as introduced before. The idea is that we explicitly model no-flow actions and wrap actual data items into flow actions. We use *provide* and *require* as abbreviations for respectively *no-flow-provide-reason* and *no-flow-require-reason*. With this setup the encoding of the primitives is straightforward. For instance, the *Sync*, *LossySync* and *Merger* primitives can be written as follows, where each line corresponds to a coloring in Table 4.3.

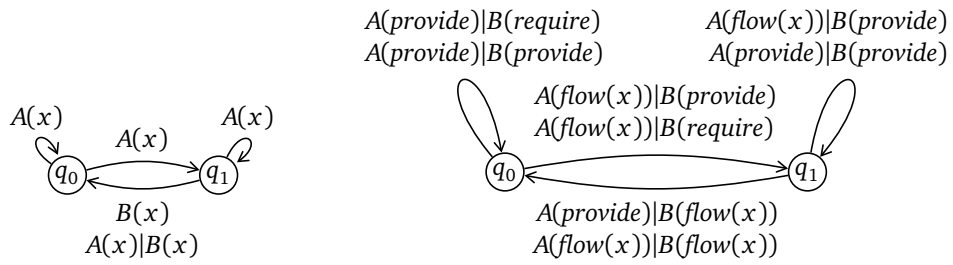
$$
\begin{aligned}
Sync = \big(\Sigma_{d:Data}\, & A(flow(d))|B(flow(d)) + && (4.2)\\
& A(require)|B(provide) + && (4.3)\\
& A(provide)|B(require) + && (4.4)\\
& A(provide)|B(provide)\big) \cdot Sync && (4.5)
\end{aligned}
$$

$$
\begin{aligned}
LossySync = \big(\Sigma_{d:Data}\, & A(flow(d))|B(flow(d)) + && (4.6)\\
& A(flow(d))|B(provide) + && (4.7)\\
& A(provide)|B(require) + && (4.8)\\
& A(provide)|B(provide)\big) \cdot LossySync && (4.9)
\end{aligned}
$$

$$
\begin{aligned}
Merger = \big(\Sigma_{d:Data}\, & A(flow(d))|B(provide)|C(flow(d)) + && (4.10)\\
& A(provide)|B(flow(d))|C(flow(d)) + && (4.11)\\
& A(require)|B(require)|C(provide) + && (4.12)\\
& A(provide)|B(provide)|C(require)\big) \cdot Merger && (4.13)
\end{aligned}
$$

Example 4.6 (colorings in mCRL2)**.** The *LossyFIFO* connector, which we have considered already in Example 4.1, is a standard example where context-dependency is required (cf. [26]). Figure 4.2 depicts the corresponding labeled transition systems for the basic constraint automata encoding (a), as well as the encoding based on the coloring semantics (b). For simplicity, we use the singleton set $Data = \{x\}$ as our data domain. The crucial point here is that in the initial state q_0, the constraint automata version can lose data (loop $A(x)$), which is unintended behavior. However, in the coloring encoding, there is no such behavior. Note that the loops in the colored version represent no-flow action only. △

Using the coloring semantics we can properly represent Reo's context-dependency in mCRL2. In contrast to [26], our encoding also reflects the state of the connectors and can further include data-dependency at the same time. Note also that even though the coloring encoding includes extra transitions for no-flow actions, the number of states is equal to its constraint automata version.



(a) constraint automata encoding

(b) coloring encoding

Figure 4.2: labeled transition systems for the *LossyFIFO* connector

4.3.4 Encoding context-dependency in port automata

The semantical models for Reo that support context-dependency, such as the coloring semantics used above or Reo automata [18], incorporate additional context information, i.e., the presence or absence of I/O request on the ports. In the coloring semantics this information is encoded using a second no-flow color. In Reo automata, the context is modeled using guards on transitions. This additional information is required to properly model context-dependent primitives, such as the *LossySync* channel, whose encoding in the port and constraint automata models can only approximate its intended behavior as a non-deterministic choice.

However, context information can in fact also be encoded in the basic port automata model. Instead of enriching the semantical model itself, we can simply encode context information using additional port names, which model the absence of dataflow. It goes without saying that we need to adapt our primitives for this purpose.

Example 4.7 (context-dependent *LossyFIFO* as a port automaton)**.** Figure 4.3 depicts an adaption of the $\mathcal{L}$ossy$\mathcal{S}$ync and the $\mathcal{FIFO}$ primitives in the port automata model, along with their composition using join and hiding. To incorporate context information for port X, we added one extra port, called $\overline{X}$, which models the absence of dataflow on X. The $\mathcal{L}$ossy$\mathcal{S}$ync and the $\mathcal{FIFO}$ now not only synchronize on X, but also on $\overline{X}$. The resulting port automaton on the right correctly models the $\mathcal{L}$ossy$\mathcal{FIFO}$ primitive, i.e., it permits to lose data at A in the full state q_{02}, but not in the empty state q_{01}. △

Thus, port and constraint automata are per se expressive enough to model context-dependent behavior. However, the assumption that a firing port name models the fact that dataflow occurs on that port must then be dropped. For the formal constructions of this encoding we refer to [55]. Note that this approach also enables model checking of context-dependent connectors using port or constraint automata based verification tools. We discuss an extension of the Vereofy model checker in Section 4.4.2.

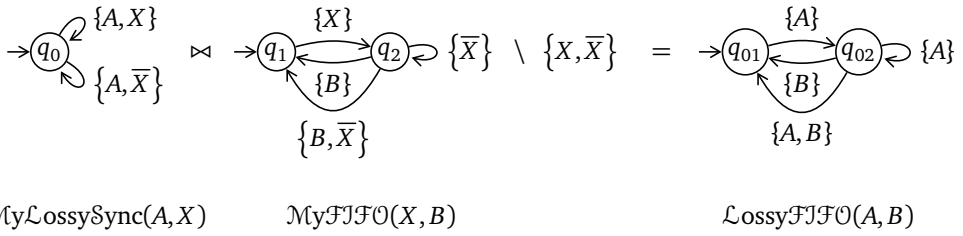


Figure 4.3: context-dependent *LossyFIFO* modeled as a port automaton

4.3.5 Encoding of Timed Reo

Timed Reo is an extension of Reo with support for timed behavior, which can be incorporated using special *Timer* channels. *Timer* channels are asynchronous channels with internal states: they consume any value of sort *Data*, delay the output for a predefined amount of time, and return a special 'timeout' value at their sink ends. For an encoding of these channels, a new data structure has to be added. For example, for specifying a *t-timer with off- and reset options*, reacting in a special way to 'reset' and 'off' data items, the following sort is needed:

$$DataTimer = \textbf{struct}\ reset?isReset \mid off?isOff \mid timeout \mid other(e{:}Data)?isOther$$

The channel behaves differently depending on whether the timer is switched on or off. In mCRL2, timed actions can be defined using the @ ('at') operator. Using this operator and the above data structure, the *t-timer with off- and reset options* can be specified as the following parameterized process:

$$\begin{aligned}
&Timer(isOff{:}Bool, x{:}Real, t{:}Real) = \\
&\quad isOff \rightarrow \big(\Sigma_{d:DataTimer}\ isOther(d) \rightarrow A(d) \cdot Timer(false, 0, t)\big) \\
&\qquad \diamond \Big((x < t) \rightarrow \big(\Sigma_{d:DataTimer} \\
&\qquad\qquad isReset(d) \rightarrow A(d) \cdot Timer(false, 0, t) + \\
&\qquad\qquad isOff(d) \rightarrow A(d) \cdot Timer(true, x, t)\ + \\
&\qquad\qquad tick@x \cdot Timer(false, x+1, t)\big) \\
&\qquad\qquad\quad \diamond B(timeout) \cdot Timer(true, x, t)\Big)
\end{aligned}$$

where *isOff*:*Bool* indicates whether the timer is off or on, x is the current time, t is the timer delay, A and B are source and sink ends of the channel, and the action *tick* occurring at time x represents the progress of time. Note that by hiding the *tick* action we can abstract from the time again and get an untimed model.

The operational model for time-aware Reo is given by Timed Constraint Automata (TCA) [2], which is essentially an extension of constraint automata with clock assignments and constraints. Timer channels are supported in ECT and are translated to mCRL2 as described above. For a detailed discussion we refer to [63].

4.4 Verification tools in ECT

In ECT, a number of verification tools can be used to analyze the behavior of Reo connectors. In this section, we give a brief overview of the available analysis tools. The mCRL2 conversion tool was written by this author of the thesis. All other tools presented here have been implemented by other researchers – either from the CWI or the TU-Dresden. Note also that in most cases third-party model checkers such as mCRL2 and PRISM are used as back-ends. However, all tools mentioned in the sequel are integrated in ECT and therefore provide powerful means for analyzing Reo connectors. Note that we do not further discuss the animation tool in ECT, since it does not cater for model checking or similar formal analysis. We now discuss the following verification tools in ECT:

- the mCRL2 conversion tool [65],
- the Vereofy model checker [9, 105],
- a tool for bounded model checking of timed constraint automata [59],
- a conversion tool to PRISM [67], and
- a discrete event simulator for Reo [55].

4.4.1 The mCRL2 conversion tool

The Reo to mCRL2 conversion tool is part of the Reo core tools in ECT. Using this conversion tool, an mCRL2 specification can be obtained automatically from a given connector, simply by selecting it in the graphical Reo editor. Screenshots of the tool are shown in Figure 4.1 and 4.4. The code generation can be customized using various options. For instance, enabling the option *with components* will add and incorporate process definitions for the components attached at the boundary of a connector. The option *with data* enables the data-aware encoding. If not enabled, data parameters and constraints are omitted. Furthermore, the option *with colors* can be used to add support for context-dependency as described in Section 4.3.3. Moreover, data types of components or services coordinated by Reo, as well as data constraints for data dependent channels such as the *Filter* or *Transform* channel can be defined using the same interface. Note that this information is saved as annotations in the Reo model, which are automatically merged in when generating the final mCRL2 specification. This way it is possible to regenerated the mCRL2 code at any point without manual changes.

The tool further includes an integration with mCRL2's model checking and state space visualization tools. Particularly, we use the `mcrl22lps` tool for generating linear process specifications from mCRL2 code, `lps2lts` and `ltsconvert` for generating and minimizing labeled transitions systems, `lps2pbes` for model checking formulas specified in modal μ-calculus, and finally `ltsgraph` for visualizing state spaces. As an alternative model checker back-end, CADP [43] can be used as well. However, mCRL2 is still required to compute a labeled transition system, which is subsequently model checked using CADP.

Runtime optimizations

In our encoding of Reo in mCRL2 every primitive (channels, nodes and components) is translated to a separate process, all of which are then run in parallel. Every primitive end corresponds to an action in this setting. Therefore, the derived specifications usually consist of a rather large number of processes and an even larger number of actions. However, the interaction among all these processes is rather local, e.g. a channel communicates only with its source and target nodes. Consequently, we observe a major performance loss in the generation of linear process specification with the `mcrl22lps` tool, caused by a state space explosion. To overcome this problem, we can construct the connector in a stepwise fashion by adding processes one by one and immediately applying the composition and hiding operators. We have explained this principle already in Example 4.1.

Using this adaption of the original approach, the intermediate state spaces remain relatively small and `mcrl22lps` can process them more efficiently. Furthermore, we can use the topology of the connector to determine an order for the processes that significantly improves the runtime of the linearization algorithm. In our experiments, a depth-first traversal on the connector graph yielded the best results.

EXAMPLE 4.8 (mCRL2 encoding benchmark)**.** Table 4.4 summarizes the runtimes of two example connectors:

(a) n parallel *FIFO1*s, the source ends of all of which are connected to a common exclusive router node, with their sink ends all connected to a common normal node. This connector implements an unordered buffer (a bag) of size n.

(b) n *FIFO1*s in a sequence, connected using normal nodes. This connector implements a *FIFO-n*.

n	naive	BFS	DFS
4	0.73	0.09	0.07
5	12.89	0.20	0.08
6	408.61	0.66	0.11
7		3.20	0.14
8		21.67	0.19
9		229.80	0.27
10			0.37
⋮			⋮
28			22.38
29			26.57
30			30.99

(a) parallel *FIFO1*s

n	naive	BFS/DFS
4	1.22	0.07
5	71.28	0.08
6		0.12
7		0.18
8		0.26
9		0.40
10		0.71
11		1.24
12		2.30
13		4.52
14		9.31
15		20.09

(b) sequential *FIFO1*s

Table 4.4: runtimes of the `mcrl22lps` tool in seconds

The benchmarks were performed on a desktop computer with a 2.40GHz quad-core CPU and 8GB memory, running Linux 2.6.27 and the development version of mCRL2 (revision 8013). The two tables show the number of *FIFO*s and the runtimes of `mcrl22lps` in seconds. The three columns show the results, respectively, for the naive encoding and two stepwise constructions based on a breadth-first search (BFS) and a depth-first search (DFS) on the connector graph. Empty cells mean that the computation took more than ten minutes. The example of n parallel *FIFO*s (a) shows that a breadth-first search can already improve the runtime. However, the growth is still exponential by a factor of 10 approximately, whereas the depth-first traversal yields much better results by a factor of around 1.2. In example (b), DFS and BFS coincide and have a factor of about 2. Note that in (b) the state space in fact grows exponentially, whereas in (a) it is linear in the number of *FIFO*s. △

The mCRL2 converter in ECT provides a powerful means for verifying data-, context- and time-dependent behavior. To the best of our knowledge, there exists no other modeling tool that provides such functionality for verifying connector behavior.

Case study

We briefly discuss a web-service example which illustrates the formal process modeling and verification using Reo and mCRL2. This example was studied in a research conducted in the EU COMPAS project [27]. The Reo connector that we consider here was generated from a business process specification in BPMN using a conversion tool in ECT, which we described in Section 2.2.1.

We consider a typical Loan Request scenario where the decision to approve or reject a client's application for a loan depends on the details of the request, say, the client's salary, the required loan amount and the period. A Reo connector for this scheme is shown in the upper part of Figure 4.4. In this model, a client's request is specified as a data item $request(amount{:}Pos, salary{:}Pos, period{:}Pos)$ provided by a *Writer* component. *FIFO1* channels represent basic (atomic) activities that constitute the process. The request is denied if the salary during the loan period is half of the required loan amount, and approved if it is three times bigger than this amount. These conditions are modeled using two *Filter* channels. Another *Writer* represents a bank clerk (Alice or Bob) who, in principle, may process the request. However, the condition of the filter channel $filter(login, authorized)$ allows only Alice to login into the system and process the request.

Actually, the process discussed contains a structural error. If the approval condition regarding the loan does not hold, the process deadlocks. Figure 4.5 shows the reduced LTS, minimized modulo branching bisimilarity [50], for different instances of the process, namely, for loan requests with the amount of 10.000, period of one year and client's salaries of 1.000, 2.000 and 3.000. The LTS for the first and second process instances contain deadlocks (states 10 and 5, respectively), automatically detectable by our set-up of model checking tools.

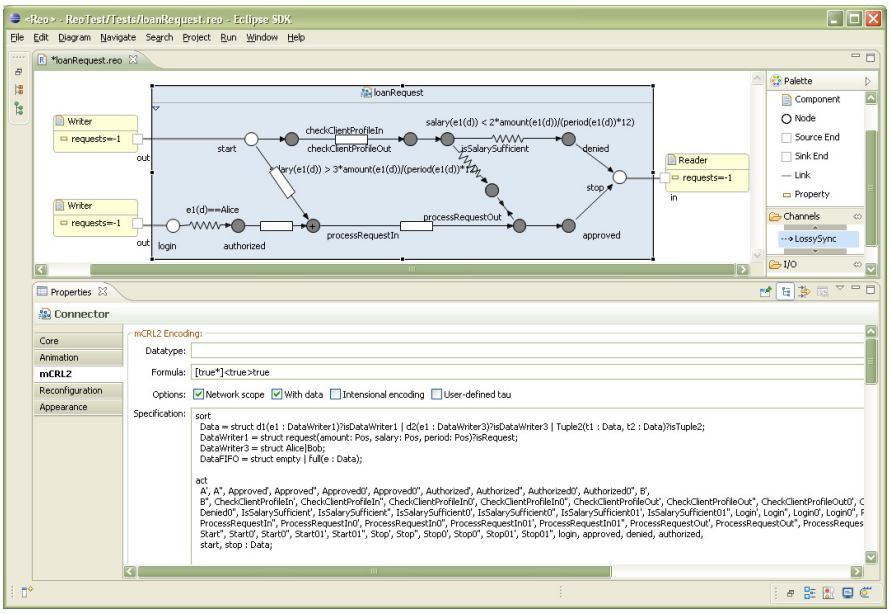
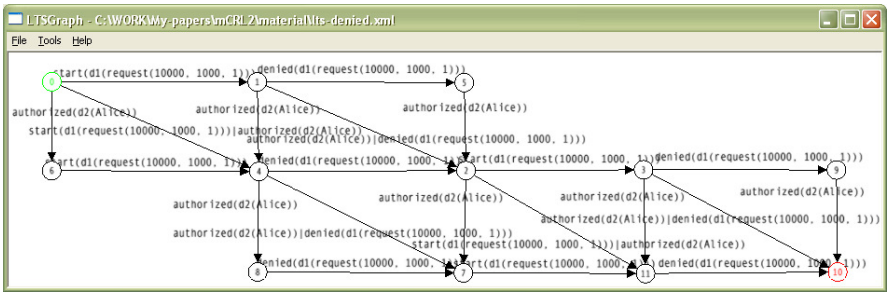
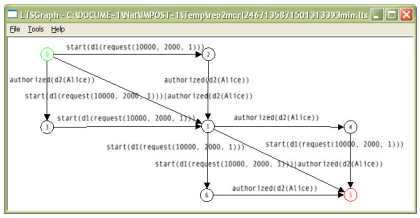


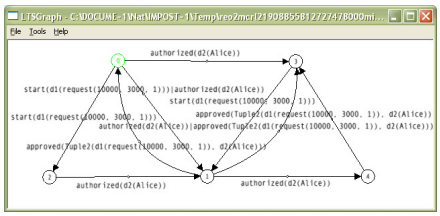
Figure 4.4: Reo to mCRL2 conversion plug-in



(a) salary=1.000, deadlock state 10



(b) salary=2.000, deadlock state 5



(c) salary=3.000, no deadlock

Figure 4.5: LTS for different instances of the Loan Request scenario

4.4.2 The Vereofy model checker

Vereofy [9, 105] is a model checking tool for the analysis of Reo connectors and is developed by the group of Christel Baier at the Technical University of Dresden. Vereofy uses two input languages, the Reo Scripting Language (RSL), and a guarded command language called the Constraint Automata Reactive Module Language (CARML) which are textual versions of Reo and constraint automata, respectively. Scripts in these languages can be generated from graphical Reo models and are used for the verification of temporal properties expressed in LTL and CTL-like logics, as well as bisimulation equivalence checks (cf. [16]). Connectors specified using RSL scripts can be also exported into the ECT Reo format, enabling graphical editing and animation.

Compared to the mCRL2-based approach, the main advantage of Vereofy is that it can generate counterexamples and show them as paths or even as animations in ECT's animation tool. This is possible using an Eclipse plug-in that integrates Vereofy with ECT. However, since the mCRL2 converter in ECT also supports analysis using CADP, it is in principle possible to extract counterexamples as well. This belongs to our future work. Note also that at present, timed behavior is not supported by the Vereofy model checker toolsuite. Moreover, Vereofy expects the user to define a global data domain eligible to all connectors and components in the model instead of generating it automatically as done in our approach. Also, Vereofy cannot handle recursive type definitions which we need to deal with, e.g., tuple nodes.

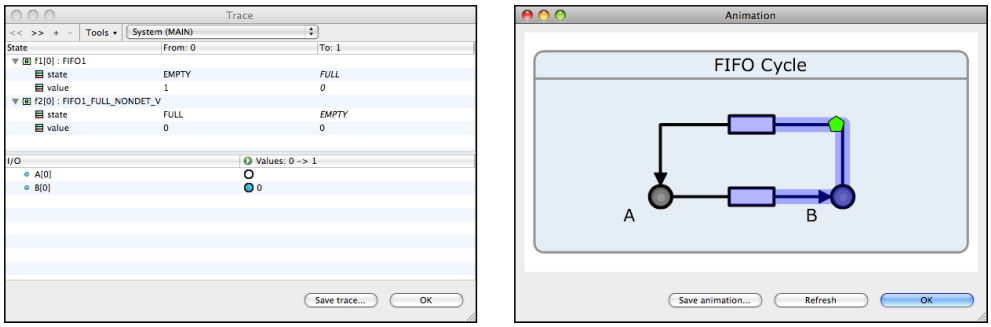


Figure 4.6: Vereofy trace explorer in ECT

Context-dependency in Vereofy

As indicated in Section 4.3.4 and formally shown in [55], context-dependency in Reo can also be encoded in basic two-color models, specifically using port or constraint automata. This encoding enables model checking of context-dependent connectors in Vereofy. For this purpose, we adapted Vereofy's library for built-in primitives. The new version of this library contains context-dependent variants of some of the basic Reo primitives and is shown in Listing B.1[3] in Appendix B.

[3] CD-Library and examples available at: `http://reo.project.cwi.nl/vereofy_CD.tar.gz`

```
1 #include "builtin"
2
3 // Non-deterministic LossyFIFO:
4 CIRCUIT LOSSY_FIFO_ND {
5   new LOSSY_SYNC_ND(A;B);
6   new FIFO1(B;C);
7   B = NULL;
8 }
```

```
1 #include "builtin_CD.carml"
2
3 // Context-dependent LossyFIFO:
4 CIRCUIT LOSSY_FIFO_CD {
5   new LOSSY_SYNC_CD(A,nB;B,nA);
6   new FIFO1_CD(B,nC;C,nB);
7   B = NULL; nB = NULL;
8 }
```

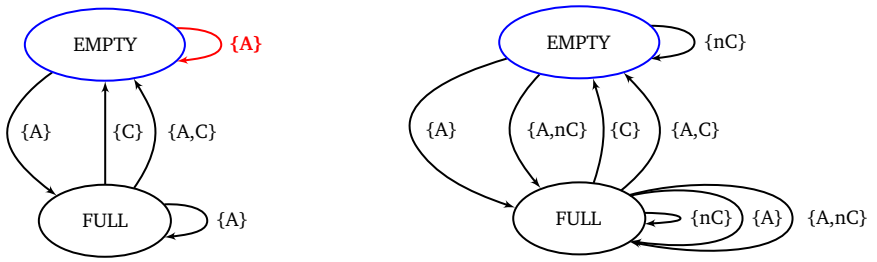


Figure 4.7: non-deterministic (left) vs. context-dependent (right) *LossyFIFO* in Vereofy

As an example, Figure 4.7 depicts a listing (top) of the non-deterministic and the context-dependent version of the *LossyFIFO* example, and two port automata generated from them using Vereofy (bottom). For simplicity we have hidden the internal node B, used a singleton set as data domain, and removed all data constraints in the generated automata. The port automata on the left and right correspond to the non-deterministic and the context-dependent versions, respectively. The latter uses the context-dependent primitives defined in Listing B.1. The crucial difference between the two is that the non-deterministic version contains an illegal transition via port A in the EMPTY state. This corresponds to the connector losing a data item in a situation where the *FIFO1* buffer is empty and should, in any case, accept the data item. In the context-depended version, however, this illegal transition does not exists.

4.4.3 Bounded model checking for timed constraint automata

A SAT-based approach [59] for bounded model checking of timed constraint automata [2] is implemented in the EA framework of ECT. In this approach, the behavior of a timed constraint automaton is represented as a formula in propositional logic with linear arithmetic, which can be analyzed by various SAT solvers. Since timed constraint automata provide operational semantics for Timed Reo, this approach can be used for model checking timed properties of Reo connectors. Timed constraint automata can be specified in the graphical EA editor. The conversion to propositional logic can be invoked from a context-menu entry, which generates code for the SAA-tRe [60] abstraction refinement model checker for timed automata. At the moment there is no tool for generating TCA from Reo connectors. The development of such a

plug-in for data-aware Reo will require tools for analyzing data constraints and functions used in *Filter* and *Transform* channels. The timed constraint automata extension and the conversion to propositional formulas was written by Stephanie Kemper and is described in detail in [58].

In our work on the mCRL2 semantics of Reo, we map each channel separately to a process and exploit the composition and hiding operators in mCRL2 to obtain a semantical model of the whole connector or network in terms of an LTS where transitions are labeled with names parametrized with data observed in these ports. Moreover, our approach can handle data manipulation using *Transform* channels with associated non-linear functions.

4.4.4 Stochastic analysis using PRISM

For performance evaluation, Reo connectors can be annotated with stochastic information such as data item arrival rates at boundary nodes and processing delays of channels. Moon et al. describe an approach where these Stochastic Reo models are translated to automata models which can subsequently be used for stochastic analysis. In particular, so-called quantitative intensional automata [4] (QIA) and stochastic Reo automata [75] (SRA) can be derived from Stochastic Reo models. In a second step, these automata models can be converted to continuos time Markov chains (CTMCs) which can then be fed into Matlab or PRISM [67] – a probabilistic symbolic model checker. PRISM can then be used, for instance, to analyze the blocking probability of a port or compute steady state probabilities.

The conversion tool is part of the EA framework in ECT and was written by Young-Joo Moon. For more details we refer to [73].

4.4.5 Stochastic analysis based on discrete event simulation

A second approach for stochastic analysis in ECT (cf. [106, 56]) is based on discrete event simulation and uses the coloring semantics as its underlying model. The advantage over the Markov chain based analysis approach is that the simulator does not impose the requirement of exponential distributions, but allows the use of general distributions for the arrival rates at nodes and, more importantly, for the processing delays in channels. Typical performance properties that can be analyzed are steady-state probabilities, channel and buffer utilizations, end-to-end delays, inter-arrival times at end points, loss ratios in *LossySync* channels and routing statistics in *Merger*s. In comparison to the automata-based approach using PRISM, the discrete event simulator also has the advantage that the state space can be computed on-the-fly, which permits to handle larger models.

The discrete event simulator for Reo was written by Oscar Kanters. A screenshot of the tool showing an instant messenger example in Reo (cf. Example 2.4) is depicted in Figure 4.8.

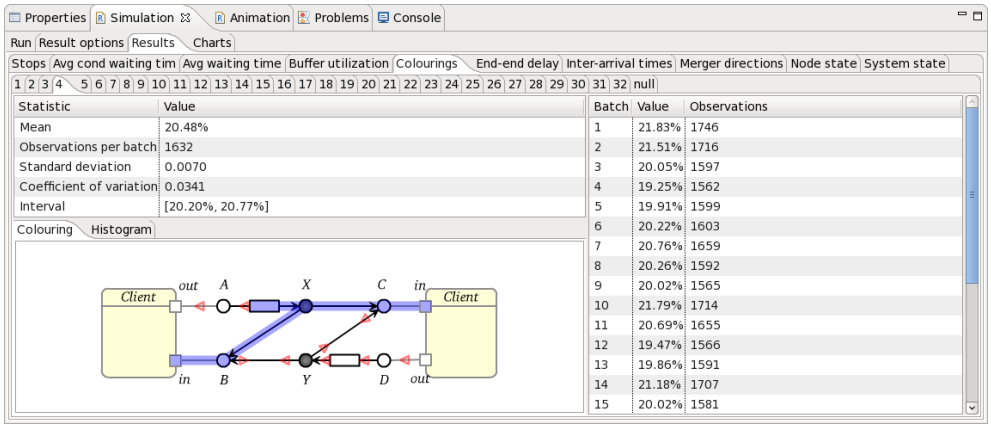


Figure 4.8: discrete event simulator in ECT

4.5 Related work

Khosravi et al. [61] establish a mapping of Reo to Alloy, a lightweight modeling language based on first-order relational logic. To check the correctness of a circuit, they express the desired properties in terms of assertions which are closely related to LTL and checked by the Alloy Analyzer. This approach deals with context dependency in Reo by defining special relations that enforce maximal progress in circuit execution. However, the actual values of data passed through the channels are not considered in this work. Moreover, the authors admit to have considerable problems with performance.

Bonsangue and Izadi [19] defined the semantics of context-dependent Reo connectors in terms of Büchi automata and generalized standard automata based model checking algorithms to enable verification of LTL formulas for Reo connectors. However, this work is purely theoretical and, currently, is not supported by any existing software tool.

4.6 Conclusions

In this chapter we have presented an approach for verifying Reo connectors using the mCRL2 model checker suite. Our mapping from Reo to the mCRL2 specification language is compositional and optimized for efficient processing. We have shown the correctness of our encoding with respect to the port automata semantics. However, our approach can also include data, based on the constraint automata model. Furthermore, we have encoded the coloring semantics for context-dependent behavior. A mapping of the *Timer* channel provides means for verifying timed behavior as well. To the best of our knowledge, the combination of these features is not supported by any other comparable tool.

The mCRL2 converter is implemented as a plug-in in ECT. We have also mentioned a number of other verification tools in ECT, e.g. the Vereofy model checker, a tool for bounded model checking of timed constraint automata, and stochastic analysis using PRISM and discrete event simulation. We have also indicated that context-dependency in Reo can be encoded in basic two-color models (formally shown in [55]) and extended the Vereofy model checker with context-dependency based on this observation. All in all, we have shown that ECT provides a powerful environment for modeling and analyzing component connector and networks based on the Reo coordination language.

Chapter 5

Reconfiguration by graph transformation

In this chapter, we introduce a rule-based approach for modeling reconfiguration of Reo connectors using graph transformation concepts. We show how confluence checks can be used to verify that reconfiguration rules do not interfere which each other. By additionally modeling the execution semantics of connectors using graph transformation we can analyze dynamic reconfiguration as well. In particular, the approach enables us to find unintended and potentially harmful interplay of execution and reconfiguration using model checking. We utilize the AGG tool for graph transformation and the Henshin framework for in-place model transformation.

5.1 Motivation

The language reference of Reo [1] describes channels as mobile entities. Their ends (and thereby the nodes they connect) can transparently be moved to other physical locations. Even more relevant for our work is the fact that channels can be created and destroyed, and nodes can be joined or split at runtime. In essence, Reo provides low-level operations for dynamically reconfiguring connectors.

The need for dynamic reconfiguration does not exclusively arise in highly adaptive environments such as peer-to-peer networks or multi-agent systems. In a simple component-based system, a typical scenario for dynamic reconfiguration is to apply hot-fixes to eliminate bugs that, for instance, cause security vulnerabilities. In the service-oriented setting, dynamic reconfiguration can be necessary if a used service suddenly becomes unavailable, its quality of service becomes unacceptable, or it changes its (behavioral) interfaces. In practice, reconfiguration tends to be more complex, especially when global changes are involved, e.g. when performing refactorings or adapting the system architecture itself. At the same time, the developers have only little control over the services that comprise the application, e.g., they usually cannot restart or reset a service.

In real-world applications, switching between a finite set of system configurations to accommodate the new requirements may not be feasible because the number of

components is often not fixed a priori. A standard example for a system with an unbounded number of components are peer-to-peer networks. In such applications, a rule-based approach for reconfiguration is better suited to perform structural changes in the system layout. Moreover, atomicity of complex reconfigurations must be ensured since they are usually performed as a series of low-level operations on primitives, e.g. the creation of a channel in Reo. When performing reconfiguration dynamically (at runtime), currently executing actions must be suspended or alternatively the reconfiguration has to be delayed until all pending actions are finished. Furthermore, when performing dynamic reconfiguration, two issues have to be dealt with: (i) the structural and logical integrity of the system has to be maintained, and (ii) it must be ensured that the system is in a consistent state after the reconfiguration (state transfer). For example, it should be guaranteed that the reconfiguration is not performed within a critical section (e.g. a transaction), and that the system is not brought into a deadlock. All these requirements are crucial in dynamically reconfigurable systems.

For the coordination language Reo we propose here a framework that provides theoretical and practical means (formalisms and implementations, respectively) for the definition, analysis and execution of dynamic reconfiguration. The underpinning formal framework for our approach is the theory of graph transformation.

5.2 Reconfiguration by graph transformation

In our approach we model possible reconfigurations of a network using a set of graph transformation rules. A graph transformation rule consists of a pattern which must be matched, and a template which describes the changes to be performed to the system. Additional application conditions (positive or negative) may restrict the applicability of a transformation rule further. A major advantage of this rewriting approach is that it allows to specify concretely in which situations and how a system should be changed, including possible dependencies that must be updated. Furthermore, these rules can not only be applied locally, but also globally, i.e., wherever the patterns match. Another important aspect of this approach is the granularity of changes that are made. Instead of sequentially performing low-level modifications on primitives, complex structural reconfigurations can be achieved in an atomic step.

In the previous chapter we elaborated the formal semantics of Reo connectors and networks. Their structural aspects, however, were treated rather informally. In order to correctly model and reason about reconfiguration, we need to introduce a formal model that captures the structural aspects of Reo connectors and networks as well. We will use a running example in this chapter, introduced below.

EXAMPLE 5.1 (resource mediator)**.** Figure 5.1 depicts a resource mediator network in Reo. Two worker components require mutually exclusive access to a resource, stored in the *FIFO1* channel in the middle. The workers can consume the resource via the source end *acquire*. The exclusive router node *C* ensures that only one of the workers can acquire the resource at a time. An administrative token is kept in a separate *FIFO1*. Once the worker has released the resource again, the copied resource is destroyed and

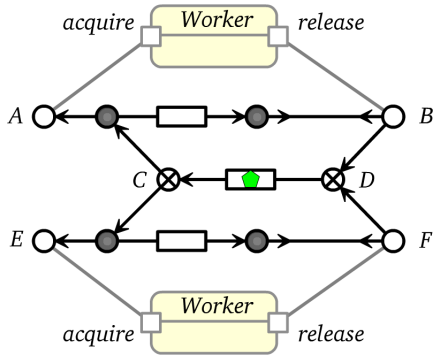


Figure 5.1: a resource mediator network in Reo

the released resource is stored again in the middle *FIFO1*. Note that node D is also an exclusive router because we want to be able to add more resources later. △

The example can be further extended by replacing the outgoing *Sync* channels from node C with *Filter* channels. The data constraints of the *Filter* channels can be used to define a pattern that the resource must match in order to be suitable for processing by a specific worker. Consider, for instance, that the resource is given in terms of a URI. If the workers can understand only certain protocols, such as *http*, *ftp* etc., the data constraint can ensure that the scheme of the URI matches that protocol. Moreover, we can also consider a scenario where the workers are allowed to modify the resource (either directly its data or the pointer that references the actual data). The resulting system can be compared to a so-called *tuple space*, which is a type of distributed shared memory and the central concept in the coordination language Linda [45].

5.2.1 Reo networks as typed hypergraphs

In the following, we formalize the structure of Reo connectors and networks using typed hypergraphs. Reo nodes are represented by vertices, whereas (hyper-)edges represent channels and components. Note that a hypergraph model can capture the notion of undirected (i.e. drain and spout) channels directly. An encoding into ordinary graphs with directed edges is in principle possible, but it is not a natural representation of this concept. Moreover, hyperedges are an appropriate model for components in our framework. Components have a fixed interface consisting of a number of source and target ends, and they are treated exactly like channels. The following definition formalizes the notion of hypergraphs.

Definition 5.2 (hypergraph)**.** A hypergraph $G = (V, E, s, t)$ consists of a set of vertices V, a set of edges E and source and target functions $s, t : E \to V^*$. ◊

The source function s models the source ends of a channel or a component. The target function t is used to model the sink ends. Whether a hyperedge $e \in E$ is a channel or component is merely based on the number of its source and sink ends. Note also that the ends are ordered since the codomain of both s and t is the set V^* of words over V. Since the graph transformation approach we use in the following is based on categorical concepts, we further need to recall the notion of hypergraph morphisms.

DEFINITION 5.3 (hypergraph morphism)**.** A hypergraph morphism $f:G_1 \to G_2$ is a pair of functions $f = (f_V, f_E)$ with $f_V:V_1 \to V_2$ and $f_E:E_1 \to E_2$, such that $f_V^* \circ s_1 = s_2 \circ f_E$ and $f_V^* \circ t_1 = t_2 \circ f_E$ where $f_V^*:V_1^* \to V_2^*$ is the free monoid morphism induced by f_V. ◊

Hypergraphs and their morphisms form a category, denoted by **HGraph** in the sequel. As pointed out above, hypergraphs naturally model the structure of connectors. However, for a complete representation, a typing mechanism is required to differentiate between multiple channel and component types. For this purpose, we consider hypergraphs together with morphisms into a fixed typegraph, denoted by *Reo*. This typegraph defines all allowed types of nodes, channels and components. Let $(G_i, type_i\colon G_i \to Reo)$ for $i \in \{1,2\}$ be two Reo graphs. Then a morphism $f\colon G_1 \to G_2$ is an ordinary hypergraph morphism, that additionally satisfies the equation $type_2 = f \circ type_1$. We denote the category of Reo graphs by $\mathbf{HGraph}_{Reo}$. Hence our final structural model for Reo connectors is typed hypergraphs. In our examples, the types of channels and nodes are indicated by their graphical representations, whereas the types of components are determined by their names.

The resource mediator network in Example 5.1 is formally modeled as a typed hypergraph that consists of six vertices (*A-F*) and in total seven hyperedges, modeling five channels and two components of type *Worker*. The exclusive routers *C* and *D* are modeled using an additional node type. However, we can also use the symbol ⊗ as an abbreviation for the hypergraph of the actual exclusive router in Example 2.2. We consider both encodings as valid models in our context.

5.2.2 Double pushout rewriting of Reo networks

We use the so-called double pushout (DPO) approach [28, 37] to graph transformation. The most important concept in the DPO approach is the rule-based graph rewriting, as schematically depicted in Figure 5.2.

A transformation rule consists of a left-hand side (LHS), a right-hand side (RHS) and a partial mapping between them. An application of a rule to a source graph results in a modified target graph. Note that this modification is performed *in-place*, i.e., the source graph is modified directly. This is an important criterion for applicability in dynamic reconfiguration scenarios, since at runtime it is in most cases not desirable to redeploy the complete system. Note also that a single rule can describe complex reconfigurations, which are performed in an atomic step. This is a major advantage of graph transformation over conventional approaches for reconfiguration, where a series of low-level operations must be performed and the consistency of the result has to be ensured through additional mechanisms. Moreover, rules may be extended

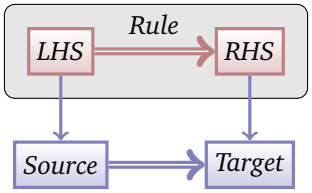


Figure 5.2: rule-based graph rewriting

with (negative) application conditions, for instance to make reconfiguration state- or context-dependent. If a single rule is not sufficient, a grammar consisting of multiple rules, which can be applied using additional control-flow mechanisms can be used. Thereby, graph transformation provides a powerful framework for realizing dynamic reconfiguration of graph-based connector models.

Formally, a transformation rule or *production* p in the DPO approach is defined as a span of injective morphisms –in our case– in the category $\mathbf{HGraph}_{Reo}$:

$$p = L \xleftarrow{\ell} K \xrightarrow{r} R$$

The left-hand side L defines the structural pattern that must be matched to apply the rule. The so-called *gluing graph* K contains all elements that are not removed by the rule and R additionally includes those elements that should be added to the graph, which in our case represents a Reo network. To reconfigure a given network M we have to apply a transformation rule p using a morphism $m{:}L \to M$, called *match*. This match defines in which part of the network, M, the reconfiguration should take place and ensures further that the required structural patterns exist. The actual reconfiguration of the network M using the rule p with the match m is formally defined as the following diagram where (1) and (2) are pushouts.

$$\begin{array}{ccccc} L & \xleftarrow{\ell} & K & \xrightarrow{r} & R \\ {\scriptstyle m}\downarrow & (1) & \downarrow & (2) & \downarrow \\ M & \xleftarrow{} & C & \xrightarrow{} & N \end{array}$$

Operationally, the network M is reconfigured by (i) removing the occurrence of $L \backslash \ell(K)$ in M, yielding the intermediate network C, and (ii) adding a copy of $R \backslash r(K)$ to C. This is the core of the double pushout approach, which has been applied to many high-level structures, such as typed attributed graphs, hypergraphs and Petri nets.

Remark 5.4 (reconfiguration rules)**.** In the following, we depict reconfiguration rules just by their left-hand and right-hand sides. The gluing graph K is defined as the intersection of the two (L and R) and the mappings ℓ and r are implicitly given by the labels and the relative positions of the nodes and edges. ▽

EXAMPLE 5.5 (reconfiguration rule *AddWorker*)**.** Figure 5.3 depicts the reconfiguration rule *AddWorker* for the resource mediator network. The rule matches an existing worker and resource, and creates a new worker and wires it to the connector. Note that it is still valid to consider the node C not as a vertex in the graph but just as an abbreviation for the exclusive router. This is possible because we can define, without much effort, a rule that adds another outgoing end to an exclusive router. △

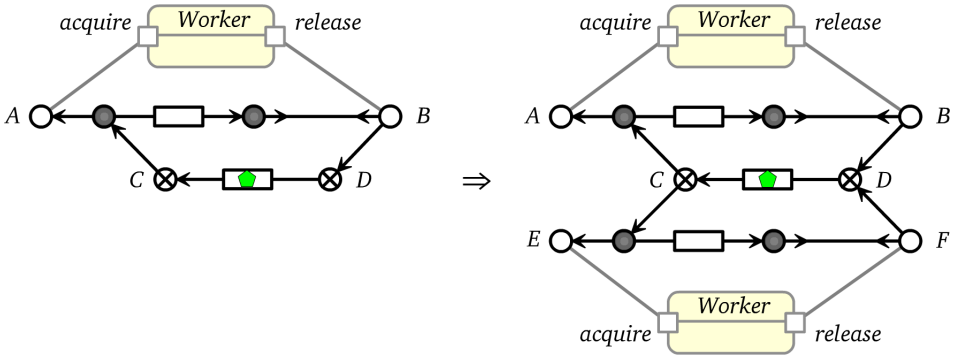


Figure 5.3: reconfiguration rule *AddWorker*

EXAMPLE 5.6 (reconfiguration rule *AddResource*)**.** So far, only a single resource was managed by the connector. To allow more than one resource we define the reconfiguration rule *AddResource*, depicted in Figure 5.4. This rule adds another *FIFO1* to the connector, which stores the new resource. We abstract here from the actual content of the resource. We assume that when this rule is applied in practice, the resource is given as a parameter to the rule, e.g. using a URI. Together with *AddWorker*, this rule enables us to dynamically add workers and resources to the basic network of Figure 5.1. The reconfigurable network now effectively ensures mutually exclusive access by n worker components to m resources. △

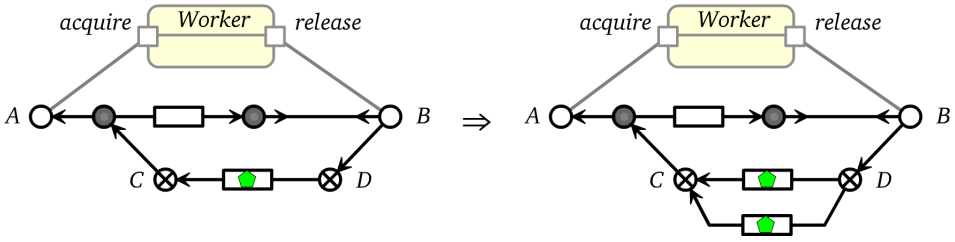


Figure 5.4: reconfiguration rule *AddResource*

5.2.3 Critical pair analysis in AGG

An important technique for formal verification of graph transformation systems is the so-called critical pair analysis, which originates in term rewriting and is used to statically check local confluence of transformation systems. The theory has been generalized to hypergraphs [80], term graphs [81] and to typed attributed graphs [51]. A critical pair is a pair of two conflicting rule applications in a minimal context. Essentially, in a critical pair one rule application disables the other rule application. The most common reason for a conflict is that one rule deletes an object which is matched by the other rule (so-called *delete-use* conflict).

Regarding reconfiguration, critical pair analysis is a useful tool for reasoning about sets of reconfiguration rules. As an example we have encoded the reconfiguration rules *AddWorker*, *AddResource* and *DelWorker*, *DelResource*, which are respectively defined as the inverses of the former rules, in the Attributed Graph Grammar (AGG) tool [87]. AGG implements the double pushout approach and has built-in support for critical pair analysis. The results of the analysis for our example are given in Table 5.1. From these results, it becomes evident that the rules *AddWorker* and *AddResource* are not in conflict with any rule. In fact, for *AddWorker* and *AddResource* there are no critical pairs at all, which means that all pairs of applications of these rules to the same graph are *parallel independent*, i.e., they can be applied in arbitrary order, yielding the same result. However, applications of *DelWorker* and *DelResource* can circumvent the applicability of the other rules.

first \ second	*AddWorker*	*AddResource*	*DelWorker*	*DelResource*
AddWorker	0	0	0	0
AddResource	0	0	0	0
DelWorker	2	2	16	3
DelResource	7	7	8	24

Table 5.1: results of critical pair analysis in AGG

The two critical pairs for *DelWorker* – *AddWorker* are shown in Figure 5.5. To make the encoding more compact, we have modeled the *Sync* channels as edges. The *FIFO1*s are encoded as vertices so that we can associate a boolean attribute for their states. Both critical pairs show essentially the same problem of our rules: since they all need a worker in their match, deleting a worker can destroy the original match. The parts that are deleted by *DelWorker* and matched by *AddWorker* are highlighted in the bottom parts in Figure 5.5.

We have to note here that critical pair analysis is a very expensive operation, since all possible overlappings of two rule patterns have to be generated. Even for our simple example the critical pair analysis with enabled type graph and multiplicity constraints in AGG v1.6.6 took already about 45 minutes to finish on a standard desktop computer. However, if applicable, critical pair analysis can give valuable insights about possibly unintended dependencies between multiple reconfiguration rules.

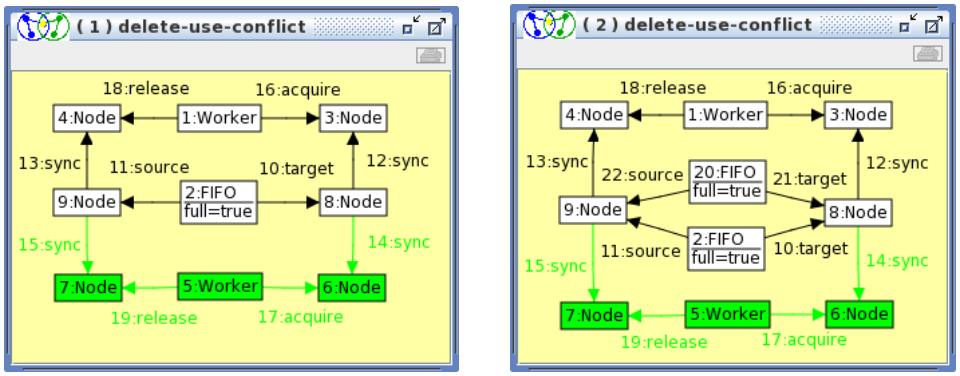


Figure 5.5: critical pairs for *DelWorker* – *AddWorker* generated by AGG

5.3 Modeling dynamic reconfiguration

As a complementary approach to critical pair analysis, model checking can be used as well for verifying reconfiguration. As a matter of fact, we show now that using a model checking approach, we are able to analyze dynamic reconfiguration, i.e., reconfiguration at runtime. A typical verification task for a dynamically reconfigurable system is to check whether there is a potentially harmful interplay of the execution, on the one hand, and the dynamic reconfiguration, on the other.

One way of analyzing the interplay of execution and dynamic reconfiguration is by encoding both behavioral aspects in the same formalism and then apply, e.g. model checking.[1] For our running example, we can refine the reconfiguration rules to explicitly incorporate state information. In fact, we have done this already for the *FIFO1* channels, since our rules contain information about whether they are full or empty. Additionally, we now use a boolean attribute *active* to store the state of the worker components. Figure 5.6 shows the refined reconfiguration rule *AddWorker*. Here, the variable x is used to maintain the state of the matched worker. To analyze the complete dynamics of the network, we additionally need to model its execution semantics. In general, this is a non-trivial task, since the execution rules must correctly model the behavior of the network even if it has been reconfigured already. However, for our simple example, the execution can be easily modeled. Figure 5.7 depicts the rule *StartWorker*, which models the action of assigning a resource to a worker and starting it. We define the rule *StopWorker* as its inverse. Now we have defined the complete dynamics (execution and reconfiguration) of the system using graph transformation rules. We can, thereby, analyze the network and the method we use to do that is model checking.

[1] Another approach is to integrate the formal models for execution and reconfiguration. We use this approach in Chapter 7, where we integrate the port automata semantics of Reo with the graph transformation model for reconfiguration.

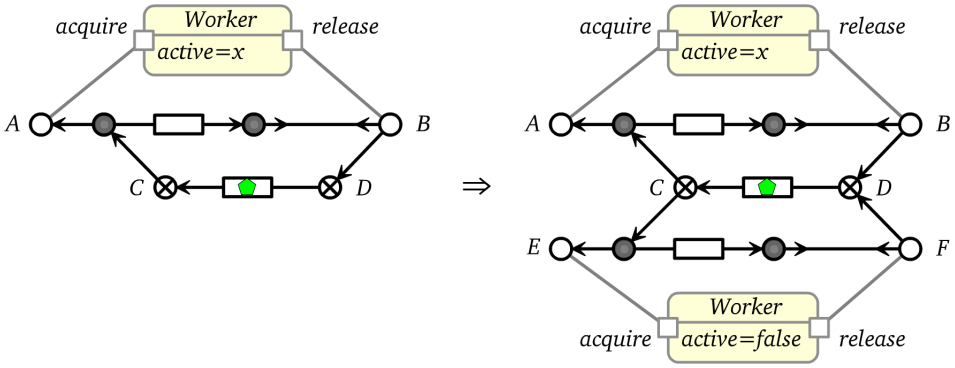


Figure 5.6: refined reconfiguration rule *AddWorker*

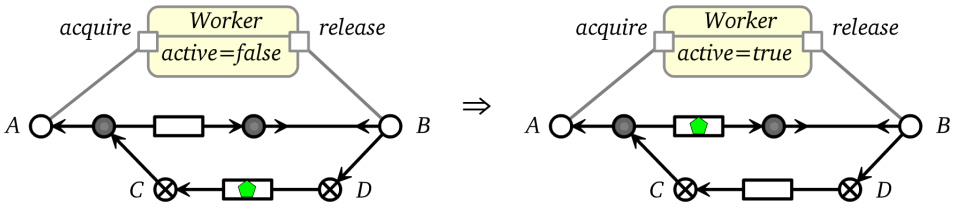


Figure 5.7: execution rule *StartWorker*

5.3.1 State space analysis in Henshin

Henshin [6, 53] is an in-place model transformation language for the Eclipse platform. Henshin is based on graph transformation concepts and provides means for defining endogenous as well as exogenous model transformations for the Eclipse Modeling Framework (EMF) [40]. The Henshin transformation language and toolset are currently developed in a joint effort of the CWI Amsterdam, the Technical University of Berlin and the Philipps-University Marburg. Henshin is an incubation project for EMF and is implemented in the context of the Eclipse Modeling Framework Technology (EMFT) [41] project.

One distinguishing feature of Henshin is its toolset for state space analysis, including a graphical state space exploration tool, which can be used to generate, visualize and model check state spaces. Once a state space for a set of transformation rules and initial states has been generated, Henshin provides interfaces to third-party model checkers including mCRL2 [49], CADP [43] and PRISM [67]. The first two enable a validation of temporal properties specified in the modal μ-calculus, whereas PRISM can be used for probabilistic analysis. Furthermore, Henshin supports checking of state invariants defined as OCL [76] constraints.

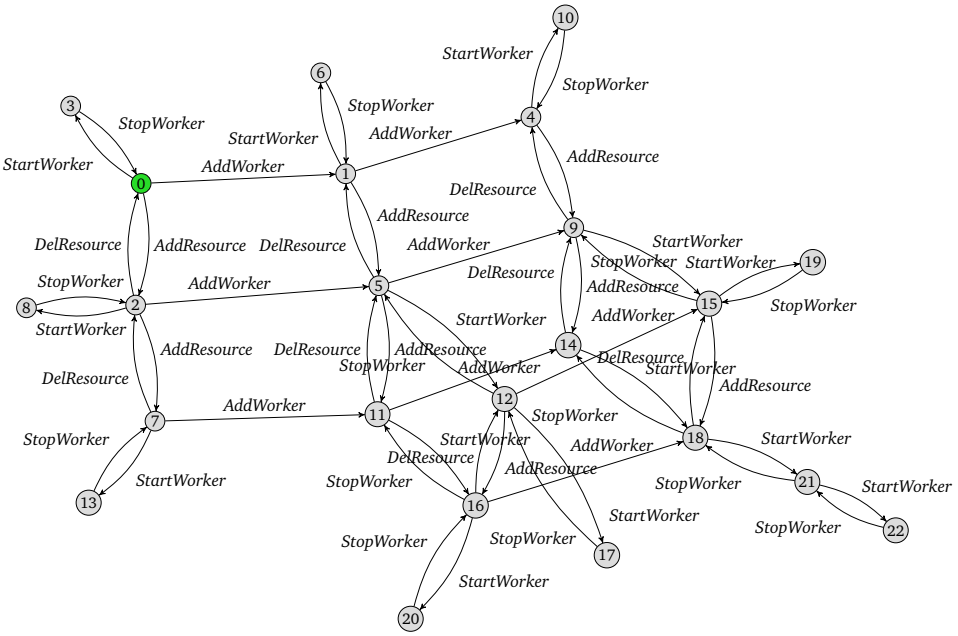


Figure 5.8: state space for resource mediator generated with Henshin

We have encoded our refined reconfiguration rules together with the execution rules *StartWorker* and *StopWorker* in Henshin, based on the Reo meta-model, which we have introduced in Section 2.2.2. To obtain a finite state space, we have added so-called negative application conditions (NACs) to the rules *AddWorker* and *AddResource*, which allow to give a bound for the maximum number of workers and resources in the network. We chose an upper bound of three workers / resources here. For better readability we have also left out rule *DelWorker*. As the initial state, we have chosen the resource mediator network with one resource and one inactive worker. We then used Henshin's graphical state space explorer to generate a labeled transition system where the states are essentially attributed graphs modeling the configurations of the network, and transitions are rule applications – in our case either execution or dynamic reconfiguration steps. We used Henshin's state space explorer to automatically layout the generated state space and exported it to LaTeX. The result is shown in Figure 5.8. Note that this state space is –for illustration– very small and that Henshin is capable of handling state spaces with millions of states.

Once a state space has been successfully generated, we can use Henshin's support for third-party model checkers. In the following, we consider:

(i) structural invariant checking with OCL,

(ii) qualitative model checking with CADP and mCRL2, and

(iii) probabilistic model checking with PRISM.

State invariant checking with OCL

OCL [76] is a constraint language for MOF [72] and EMF [40] based models. In state spaces generated with Henshin, every state corresponds to a distinct EMF model instance. Therefore, OCL can be used to validate structural state invariants.

In our example we can, for instance, verify that there are always enough empty *FIFO1*s for storing the resources in the network. We do this using the following OCL constraint, which we can directly check in Henshin's state space explorer:

- *FIFO.allInstances()→select(f | f.name='resource' and not f.full)→size()* ≥ *Worker.allInstances()→select(w | w.active)→size()*

This invariant is satisfied. Note that we need to mark those *FIFO1*s that are actually used for storing the resources. We use here the static method *allInstances()* to obtain all known instances of type *FIFO* and *Worker*, and compute a subset using the *select*-construct in OCL. An example for an invariant that is not satisfied is:

- *Worker.allInstances()→select(w | w.active)→size()* ≤ *1*

which states that there is at most one active worker in the network. The Henshin state space explorer finds as a counterexample a path into state 18, in which this constraint does not hold anymore.

Model checking with CADP and mCRL2

In Section 5.2.3 we used the critical pair analysis of AGG tool to verify the local confluence of applications of *AddWorker* and *AddResource*. This local confluence holds also for the refined version of the rules. In the state space the local confluence causes the typical diamond shape between *AddWorker* and *AddResource* transitions.

Using the CADP [43] and mCRL2 [49] model checker back-ends of Henshin, we can now also verify temporal properties specified as modal μ-calculus formulas. For instance, we can verify freedom of deadlock using the formula:

- $[true^*]\langle true\rangle true$

Furthermore, we can also confirm that after a *StartWorker* action there is always a *StopWorker* action using the following formula:[2]

- $\langle true^*.StartWorker.true^*.StopWorker\rangle true$

In general, the state space in Figure 5.8 is very symmetric and shows that there are no harmful interactions between reconfiguration and execution steps, e.g. the square of states 5-12-16-11 shows how adding and removing resources do not conflict with

[2]Note that the development version of Henshin also supports to model check parameterized actions, i.e., to quantify over nodes (cf. `http://wiki.eclipse.org/Henshin_Statespace_Explorer`). In our running example, this feature can be used to check that once a specific worker has been started, this worker is eventually stopped again.

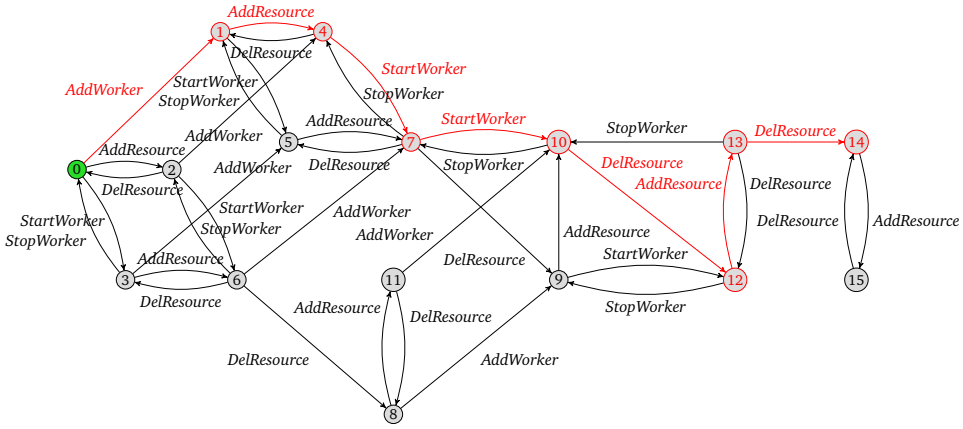


Figure 5.9: state space for the adapted reconfiguration rules

starting and stopping a worker. However, such a well-behaved system is not always a given. To illustrate the potentially harmful interactions of execution and reconfiguration actions, we have adapted our example by removing the state information of the *FIFO1*s and worker components in the LHS of the reconfiguration rules *AddWorker*, *AddResource* and *DelWorker*. One critical aspect in this modification is that *DelResource* may now delete empty *FIFO1*s whose tokens are still being used by a worker. For simplicity, we have set an upper bound of two workers and two resources now. The resulting labeled transition system is depicted in Figure 5.9.

The left part of the labeled transition system contains the 'cube' of adding / removing resources and starting / stopping workers, similar to the original version in Figure 5.8. The rest of the state space, however, is somewhat more asymmetric. In fact, it contains a connected island of states 14 and 15, from which the rest of the LTS is not reachable anymore. Moreover, the rules *StartWorker* and *StopWorker* are not enabled here anymore and we can see also that two workers have been started, but cannot be stopped anymore. To understand this problem we follow a trace into this region of the labeled transition system:

(i) initial state 0: one worker and one resource. The worker is inactive and the resource is stored in a *FIFO1*.

(ii) *AddWorker* $0 \rightarrow 1$: a second inactive worker is added.

(iii) *AddResource* $1 \rightarrow 4$: a second resource stored in a *FIFO1* is added.

(iv) $2\times$*StartWorker* $4 \rightarrow 7 \rightarrow 10$: both workers become active. The resources are removed from the two *FIFO1*s.

(v) *DelResource* $10 \rightarrow 12$: one of the empty *FIFO1*s is deleted.

(vi) *AddResource* $12 \rightarrow 13$: a new full *FIFO1* is added.

(vii) *DelResource* $13 \rightarrow 14$: the other empty *FIFO1* is deleted.

Thus, in states 14 and 15 there are respectively one and two *FIFO1*s in the network that already carry a resource. However, both workers are running already at this point and cannot release their resource anymore, which therefore results in a livelock.

Probabilistic analysis with PRISM

As mentioned already in the introduction, Henshin also supports probabilistic analysis using PRISM. For this purpose we need to annotate rules with positive real numbers which describe the rate of an exponentially distributed delay of its application. Following the approach of stochastic graph transformation, as introduced by Heckel et al. [52], the state space generated with Henshin together with the rule rates give rise to a Continuous Time Markov Chain (CTMC). The state space explorer in Henshin can produce such a CTMC in the input format of PRISM, thus allowing to perform stochastic analysis as well.

Rule	Rate	Rule	Rate	Rule	Rate
StartWorker	3600	*AddResource*	30	*AddWorker*	6
StopWorker	3600	*DelResource*	12	*DelWorker*	1

Table 5.2: rates for rule applications in the resource mediator network

For the example of the resource mediator network with upper bounds of respectively three workers and resources, we chose the rates shown in Table 5.2. For simplicity, we use integers here. We assume that starting and stopping a worker takes the same amount of time. We therefore associate the same rates to *StartWorker* and *StopWorker*. In general, we assume that reconfiguration steps occur much more scarcely than execution steps. Adding workers or resources is assumed to happen more often than deleting. This can be caused by a reconfiguration policy which adds workers and resources on demand, for instance based on the size of a task queue. We also assume that adding resources to the network happens more frequently than adding workers. In this example, we interpret the rates as execution or reconfiguration actions per hour. Thus, the mean time between starting workers, or between stopping workers, is one second. The mean time between adding new resources is 2 minutes and between deleting is 5 minutes. Finally, we assume that the average delay of adding new workers is 10 minutes and that workers are deleted again only once per hour.

Using a validation front-end in Henshin's state space explorer we are now able to perform some stochastic analysis with PRISM. We use the original version of the reconfiguration rules, as in Figure 5.8. However, this time we also include the rule *DelWorker*. As a standard test, we computed the steady-state probabilities of the generated CTMC. The rounded results are shown in Table 5.3, where we have omitted all states with a probability of less than 1%.

This shows that the configuration with three worker and three resources has with 76.8% the highest probability. Note that starting and stopping workers has no measurable effect on the average network configuration. The second most likely configura-

tion consists of two workers and three resources, or three worker and two resources. Note that the actual results computed by PRISM indicated a very small difference in the probabilities for these two network configurations. However, since this difference is very small, we did not reflect it in the probabilities in Table 5.3.

States	Probability
14, 18, 21, 22	19.2%
9, 11, 15, 16, 19, 20	3.2%

Configuration	Probability
3 workers, 3 resources	76.8%
2 workers, 3 resources or 3 workers, 2 resources	19.2%

Table 5.3: steady-state probabilities computed by PRISM

Observations

Even our very basic example shows that a careful design of reconfiguration rules is crucial. In particular, it is important that the reconfiguration rules contain constraints on the state information of the elements that are involved in the reconfiguration. We have shown that by modeling the reconfiguration and execution semantics as graph transformations we can analyze the complete dynamics of a network. State space analysis reveals potential problems and can be used to eliminate design failures. In particular, we can use model checking to verify structural and behavioral, and in fact even combined structural and behavioral invariants of a network. The analysis results can then be used to create a safe reconfiguration policy, which states precisely when and how a dynamic reconfiguration can be applied.

5.3.2 Transparent dynamic reconfiguration

The execution of a dynamic reconfiguration is not trivial. Assume a network of distributed components that communicate with each other via a centralized connector implementation, which we also refer to as the *coordinator* in the sequel. The components perform blocking I/O operations on their ports which implement synchronization point semantics for their corresponding coordinator ports, equivalent to Hoare's CSP channels. Additionally, I/O operations on ports may timeout. We have briefly discussed an implementation of this approach for Reo based on constraint automata in Section 3.3.5.

The coordinator monitors the I/O requests on its ports. When all synchronization and data constraints are fulfilled for one of the next possible transitions, the coordinator performs an atomic execution step in which data flow occurs between the components and/or memory cells of the coordinator. The I/O operations of the components involved in this step succeed and the components can spawn new I/O requests on their ports.

To perform a dynamic reconfiguration of such a system consisting of a set of active and autonomous components and a centralized coordinator, we need to be able to execute the reconfiguration while there are I/O requests pending on the ports. This is

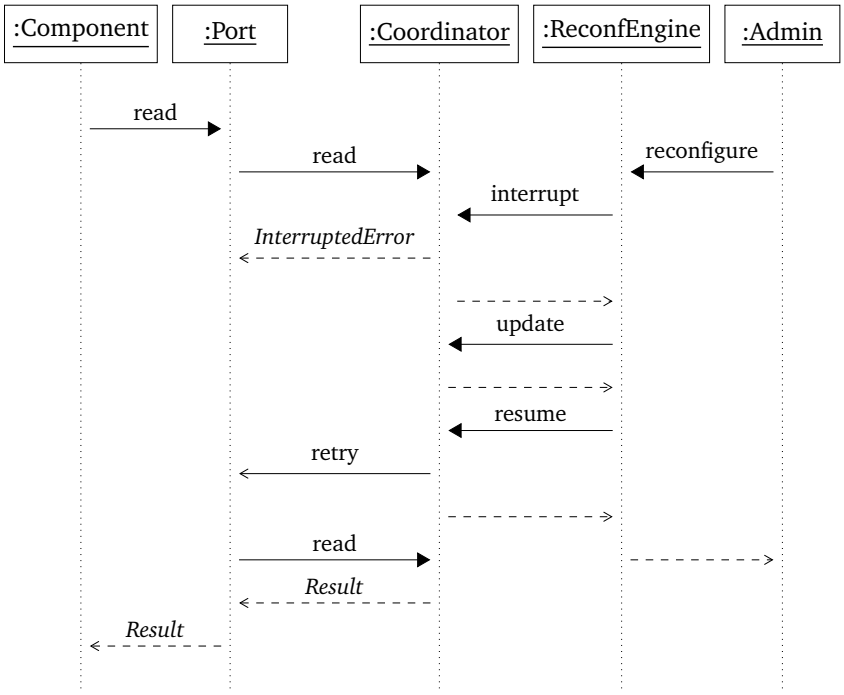


Figure 5.10: executing reconfigurations within a pending I/O operation

due to the fact that the coordination is done exogenously and that, thus, the coordinator has no control over the components. In our scenario, we consider an administrator that can decide at any point in time to reconfigure the connector. Note that we assume that the reconfiguration rules themselves contain conditions on the states of the involved primitives as described in the previous section. The reconfiguration can then be performed as shown in the message sequence chart in Figure 5.10.

The following parties are involved in the reconfiguration: the centralized coordinator, a number of autonomous components that communicate with the coordinator via ports, a reconfiguration engine that performs rule-based reconfigurations using graph transformation, and an administrator that initiates the reconfiguration. The following actions are shown in the message sequence chart:

(i) A component performs a blocking read operation on a port. The port redirects the read operation to the coordinator.

(ii) The administrator initiates a reconfiguration.

(iii) The reconfiguration engine interrupts all pending I/O request on the coordinator. All ports are suspended.

(iv) The reconfiguration engine reconfigures the connector and resumes the coordinator when finished.

(v) All ports are resumed and their pending I/O operations are now performed on the reconfigured coordinator.

(vi) The read operation of the component succeeds.

This approach has two major advantages: (i) a reconfiguration can be performed with pending I/O requests, and (ii) the components are oblivious to the reconfiguration. As far as they are concerned, they just perform normal I/O operations and do not notice that a reconfiguration occurred. Thus, our approach enables transparent dynamic reconfiguration.

In the message sequence chart in Figure 5.10, the actual reconfiguration action is abstracted in the update operation which the reconfiguration engine performs on the coordinator. In the centralized implementation of Reo described in Section 3.3.5, the whole connector is compiled into a constraint automaton first, which is then directly executed as a state-machine. A reconfiguration, in this case, is performed on the connector model, which is then recompiled into a constraint automaton and restarted. However, the state of the coordinator must be restored in this case. To that end, the contents of memory cells (*FIFO1* buffers) as well as the actual current state in the automaton must be restored. The latter is non-trivial, because it cannot be expected that there is any correlation between the constraint automaton before and after a reconfiguration. However, the best approximation is to restore the states of the primitives that are not changed by the reconfiguration and to use the default initial states of the newly created primitives. This can be implemented by storing the projections from the constraint automaton for the whole connector into the constraint automata of the primitives. This is essentially a method for handling the state transfer between the coordinator before and after the reconfigurations.

In distributed implementations such as the one described in [83], state transfer is not a problem, since state information is stored locally in the primitives. Thus, in the distributed case, reconfigurations can be performed directly on the deployed connector. However, performing reconfigurations on a distributed, and particularly, decentralized system requires a distributed scheme. We address this problem in Chapter 6.

5.4 Support for reconfiguration in ECT

To support reconfiguration in ECT we have extended the Reo meta-model with the possibility of associating reconfiguration (e.g. create and delete) actions with primitives and nodes. Additional validation code ensures the well-formedness of reconfiguration rules according to the formal model of rules in the double pushout approach. Through the extension of the Reo meta-model, it is possible to define reconfiguration rules in ordinary Reo files. Rules can have a local scope, or can be registered in a global registry. To edit reconfiguration rules, the so-called *properties view* in Eclipse can be used to define primitives and nodes augmented with reconfiguration actions. The graphical Reo editor automatically highlights parts of a connector that are augmented with reconfiguration actions. The graphical representation of a reconfiguration rule in the Reo editor is essentially an integrated view on its LHS and RHS. As an

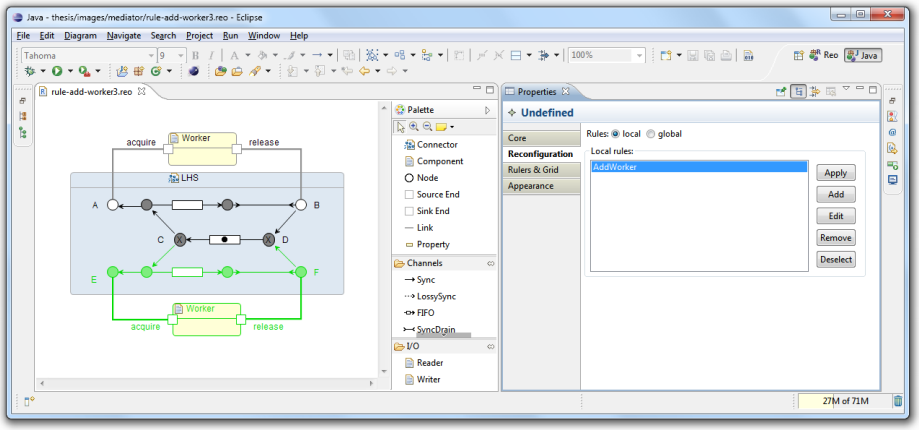


Figure 5.11: editing and executing reconfiguration rules in ECT

example, Figure 5.11 depicts the integrated notation of rule *AddWorker* in the Reo editor, together with the properties view. The green part of the network is the part that is added by the rule. We have implemented a prototypical reconfiguration engine which takes as input a reconfiguration rule, such as the one in Figure 5.11, and applies it to a given Reo network. The reconfiguration engine is essentially a DPO hypergraph transformation engine that natively operates on Reo connector / network models. This has the advantage that the underlying hypergraph model is more high-level and therefore a more natural encoding of Reo. However, since the Reo meta-model is defined using EMF, we can also translate reconfiguration rules directly into Henshin transformation rules and then use the far more efficient engine of Henshin. Reconfiguration rules can be applied directly in the graphical Reo editor or programmatically.

5.5 Dynamic reconfiguration in ReoLive

An prototypical implementation of transparent dynamic reconfiguration as described in Section 5.3.2 is the *ReoLive* web service[3]. ReoLive is essentially a wrapper for the constraint automata based interpreter engine of Reo. Deployment of connectors in ReoLive is simply done by uploading Reo files. The application automatically generates a constraint automaton from the Reo model which can then be executed on the ReoLive server. Components can connect and communicate with ReoLive using SOAP. Thus, ReoLive is a centralized implementation of Reo where the components are distributed. A prototypical implementation of transparent dynamic reconfiguration as described in Section 5.3.2 has been implemented in ReoLive. The execution of reconfiguration rules as well as starting, stopping, and monitoring the execution, can be done using a web interface in ReoLive. ReoLive was written by the author of this

[3]ReoLive web service: `http://reo.project.cwi.nl/live`

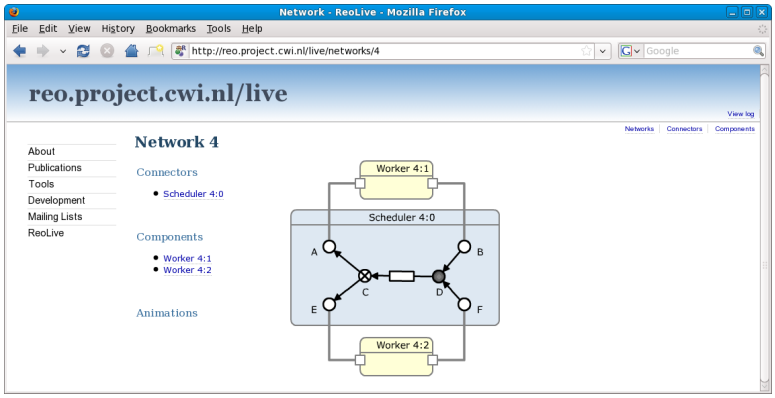


Figure 5.12: user front-end of the ReoLive web service

thesis and uses the constraint automata interpreter which was implemented by Ziyan Maraikar. A screenshot of the web front-end of ReoLive is shown in Figure 5.12.

5.6 Reconfigurable coordination of YAWL workflows

YAWL (*Yet Another Workflow Language*) [104] is a workflow language developed in a joint effort between the TU Eindhoven in the Netherlands and the Queensland University of Technology in Brisbane, Australia. YAWL is based on Petri nets and adds mechanisms to more directly support complex workflow patterns[4]. The implementation of YAWL[5] comprises a complete workflow management system, which is centralized though, i.e., a YAWL engine has full control over all process instances and does not interact with other engines.

To overcome the limitations inherent in centralized workflow systems, we implemented a prototype of a so-called *adapter task* for YAWL, which allows to coordinate using Reo different workflows, possibly running in different, distributed YAWL engines. Workflows in YAWL consist on the lowest level of atomic, executable entities, called *tasks*. In our approach, we implemented a specific type of task, viz. a web service, which performs a basic, blocking I/O operation at a Reo connector deployed in a ReoLive engine. A schematic overview of this approach is depicted in Figure 5.13. An adapter task essentially serves as a port between a Reo connector and YAWL workflow and allows to synchronize and exchange data between different YAWL workflows, possibly running in different engines. In this way, adapter tasks enable a seamless, yet simple integration of YAWL and Reo, without the need for changing the underlying implementations of YAWL or ReoLive. Moreover, ReoLive supports dynamic reconfiguration as described in the previous section. Thus, the integration with YAWL can benefit from this feature as well. For example, consider a workflow that, at a specific

[4]For a survey on workflow patterns we refer to [103].
[5]See `http://www.yawlfoundation.org`

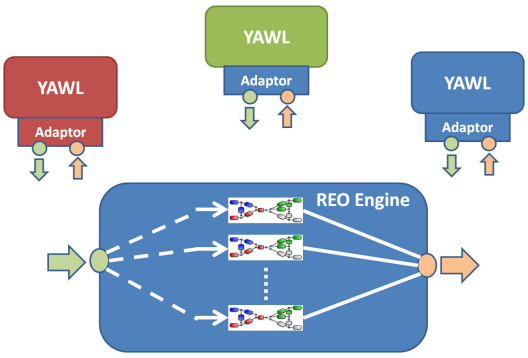


Figure 5.13: schematic overview of the integration of YAWL and Reo

point, requires data produced by some external workflows, possibly running in different YAWL engines. In this scenario, a simple Reo channel can be used to transfer the data from one of the producer tasks to the consumer task. Moreover, dynamic reconfiguration can be used here to wire the Reo channel to another producer task on certain events, e.g. when the current producer sends a special *end-of-data* signal. Even in this simple example, dynamic reconfiguration offers a powerful and also very natural means to realize adaptable coordination patterns. Moreover, this dynamic adaption can be done transparently, as described in Section 5.3.2, enabling a seamless integration and coordination of (in this case) YAWL workflows using Reo in a truely distributed and dynamically changing environment.

5.7 Related work

A graph transformation based reconfiguration approach for Reo in which dataflow events are used to trigger reconfiguration is considered in [91]. The approach is rather powerful and promises interesting applications. However, validation of such models (as presented here) is not considered.

A basic logic for reasoning about connector reconfiguration in Reo, including a model checking algorithm is considered in [25]. Different to our approach, the author uses a formalization of connectors, which is particularly not a graph model. Moreover, the reconfiguration operations are rather low-level and provide no means for rule-based reconfiguration.

In a more general setting, dynamic reconfiguration has been studied for a number of Petri net variants. For instance, open Petri nets [11] extend the original model by a notion of open places, representing an interface to their environment. As in our approach, reconfigurations of open Petri nets are modeled using double pushout rewriting. Behavior preserving reconfigurations of open Petri nets are studied in [12].

A typical application of (open) Petri nets are workflow nets [99] and interorganizational workflows [100]. Dynamic changes in workflow nets are discussed in [101]. The so-called *dynamic change bug* refers to a problem of ensuring a consistent system

state after a reconfiguration. This has some similarities to the notions of structural and behavioral invariants in our work. While in our approach invariants are application-specific and basically user-defined, the consistency for workflow nets ensures a basic notion of correctness, i.e., a reconfiguration is valid if the state after the reconfiguration could have been reached from the initial state. Their proposed solution is to calculate a safe change region and to allow reconfigurations only in these states. Related to the dynamic change bug, [102] introduces *inheritance-preserving* transformation rules for workflows which guarantee well-behaved reconfigurations.

Architectural Style Rewriting (ADR) [21] is a framework for modeling reconfigurable software architectures. This approach uses hyperedge replacement for defining hierarchical structures. Contrary to our approach, term rewriting is used for reconfigurations in ADR. Generally, the focus in ADR is on style-preservation, i.e., structural (and not behavioral) properties of reconfigurations.

5.8 Conclusions

Reconfiguration provides a powerful means for adapting a system configuration to accomodate for changes in the environment, component failure or insufficient quality of service. Due to the graph structure inherent in Reo networks, methods from graph transformation are well-suited to model and implement reconfiguration in Reo. Important aspects of this approach include rule-based definition and atomic execution of complex reconfiguration steps. For verification purposes, we have shown how the AGG system can be utilized to detect conflicting reconfiguration rules using critical pair analysis. The Henshin toolkit can be further used to generate state spaces and to do qualitative and quantitative model checking. We have also shown that in cases where the execution of Reo networks can be defined in terms of graph transformation rules as well, we can use Henshin's state space analysis tool to reason about dynamic reconfiguration. In particular, we have shown that naive implementations of reconfiguration rules may interfere with the execution semantics of the network and result in system failure. The analysis tools in AGG and Henshin can help to circumvent such design errors. Specifically, the rule-based approach for defining reconfigurations should be used in such a way that state information is included in the application conditions of reconfiguration rules. This ensures that a reconfiguration can be performed only in states where it is safe to reconfigure the network.

Regarding the execution of reconfigurations, we presented an approach in which networks are reconfigured transparently, i.e., without the knowledge or cooperation of the involved components. Thus, the components do not have to be sent into a stand-by mode or provide any extra functionalities for handling reconfiguration. A prototypical implementation of transparent reconfiguration is provided in the ReoLive web service. Reconfiguration rules can be defined directly in the graphical Reo editor in ECT. A naive implementation of a reconfiguration engine is also provided in ECT. For efficient execution of reconfiguration rules, we plan an automatic mapping into the Henshin format for in-place model transformation.

Chapter 6

Distributed networks and reconfiguration

In this chapter, we provide a formal framework for modeling distributed component connectors and their reconfiguration. We show how distribution can be used for logical structuring and encapsulation. In our approach, a reconfiguration of a distributed network is defined and executed locally. Therefore, no global knowledge is required to specify or carry out a reconfiguration. We utilize the theory of distributed graph transformation and present a new result on compositionality of flattening, which we also apply in Chapter 7.

6.1 Motivation

The need for considering distributed connectors and networks arises from two concerns. On the one hand, connectors are often structured into logically separate parts, each of which defines a specific subprotocol. On the other hand, connectors may be deployed on different physical locations in a distributed network. In both cases, the concept of distribution facilitates and promotes the use of black-boxed subconnectors in a larger context.

In this chapter, we therefore propose a framework for modeling reconfigurable, distributed Reo connectors. We consider connectors that are distributed over a network and are encapsulated, i.e., their internals are hidden from the outside world, and communicate only via their published interfaces. Connectors are linked together via the interfaces that they share. Reconfiguration of a network is achieved by reconfiguring its constituent connectors. Ultimately, reconfiguration is defined and performed locally (that is to say, in the scope of a single connector), although it can be either triggered from the inside or invoked from the outside. Reconfiguring a connector may involve a change in its interfaces and may require connectors in its neighborhood to reconfigure as well. This implies a need for synchronizing local reconfigurations into a consistent reconfiguration of the connector as a whole. It goes without saying that in a distributed setting, we cannot assume the existence of a (centralized) third party that monitors and coordinates local reconfigurations. Therefore, other mechanisms

should be in place to assure the consistency of a reconfigured network.

We utilize the well-studied framework of distributed graph transformation [86, 38] for modeling distributed connectors and reconfiguration here. Moreover, we show how reconfiguration can be defined and performed locally using a synchronization mechanism based on the notion of amalgamation [17, 28, 88]. Finally, we propose a distributed strategy to organize the stepwise reconfiguration of large networks.

6.2 Distributed graphs and Reo networks

To model reconfigurations of distributed Reo connectors, we use the framework of distributed graph transformation, as proposed by Taentzer [86]. A generalization from distributed graphs to a notion of distributed objects was considered by Ehrig et al. [38]. In the following, we recall the notions of this framework that are relevant to our present setting and apply them to Reo. We assume some familiarity with basic notions from category theory, most relevant here are the concepts of pushouts and functors. For a comprehensive introduction to category theory we refer to [68].

6.2.1 Distributed graphs

Distribution of graphs can be described by adding a second layer of abstraction, namely by modeling the topology of a system using a so-called *network graph*. The nodes in a network graph consist of *local graphs* and the edges are morphisms of local graphs. The idea is that a node models a physical or logical location of a local graph, whereas an edge indicates an occurrence of the source graph in the target graph. In particular, multiple outgoing edges from one local graph model the fact that the source graph is shared among the target graphs. Formally, we consider distributed graphs as defined in the following.

Definition 6.1 (distributed graph)**.** A *distributed graph* (N, D) consists of a graph N, called *network graph*, and a commutative functor $D\colon N \to \mathbf{Graph}$, where the graph N is interpreted as a category. ◊

The network graph N describes the topology of the network. The functor or *diagram* D associates to every node n in N a local graph $D(n)$ and to every edge $n \xrightarrow{e} n'$ in N a graph morphism $D(e)\colon D(n) \to D(n')$. Following [38], this functor is required to be commutative, i.e., for any two paths $p_1, p_2\colon n \xrightarrow{*} n'$ in N, it must hold that $D(p_1) = D(p_2)$. This arises from the assumption that the morphisms associated with edges represent the sharing of the local graphs.

Due to the categorical definition, the concept of distribution can be easily generalized to other structures as well, e.g. we can consider distributed typed hypergraphs for modeling distributed Reo networks by using diagrams $D\colon N \to \mathbf{HGraph}_{Reo}$, extending the model introduced in Chapter 5.

Example 6.2 (distributed mediator)**.** Figure 6.1 depicts a distributed Reo graph for the resource mediator network from Example 5.1. The network graph consists of four

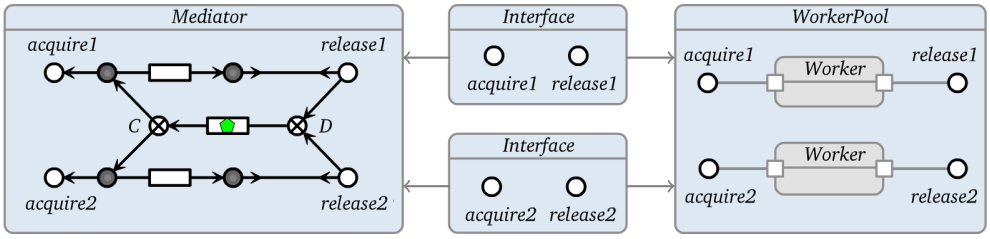


Figure 6.1: distributed version of the resource mediator network

nodes modeling the mediator connector, a worker pool and two interface nodes to connect the former. The local morphisms are the obvious inclusions. Since the interfaces are embedded into both the mediator and the worker pool, these two local connectors are considered to be connected along their shared interfaces. This model provides a logical separation of the mediator connector, on the one hand, and a worker pool, on the other. △

Note that the graphical representation we use here for distributed networks is borrowed from ECT. However, the means for structuring in ECT are not based on distributed graphs. The main difference is that the blue connector nodes in ECT are essentially hierarchical containers for nodes and primitives, whereas in our model there are only two levels of abstraction: the network and the local layer. We now recall the definition of morphisms for distributed graphs.

Definition 6.3 (distributed graph morphism)**.** For two distributed graphs (N_1, D_1) and (N_2, D_2), a morphism $f = (f_N, f_D)$: $(N_1, D_1) \to (N_2, D_2)$ consists of a graph morphism f_N: $N_1 \to N_2$ and a natural transformation f_D: $D_1 \to D_2 \circ f_N$. ◊

For brevity, we will just write f for the network morphism f_N. By definition, the natural transformation f_D assigns to every node n of the network graph N_1 a graph morphism f_n: $D_1(n) \to D_2(f(n))$ which is is called the *local* graph morphism of n. Furthermore, for every edge $n \xrightarrow{e} n'$ in N_1 the following diagram commutes.

$$\begin{array}{ccc} D_1(n) & \xrightarrow{D_1(e)} & D_1(n') \\ \downarrow{\scriptstyle f_n} & & \downarrow{\scriptstyle f_{n'}} \\ D_2(f(n)) & \xrightarrow{D_2(f(e))} & D_2(f(n')) \end{array} \tag{6.1}$$

Example 6.4 (distributed graph morphism / transformation rule)**.** An example morphism of distributed Reo graphs is shown in Figure 6.2. The morphism is injective on the local and the network layer. The mappings are indicated by the relative positions and channel types and node names. We can interpret this morphism also as a transformation rule in the double pushout approach. This rule creates a new *Interface* network node, extends the mediator connector, and wires the new interface with the connector. △

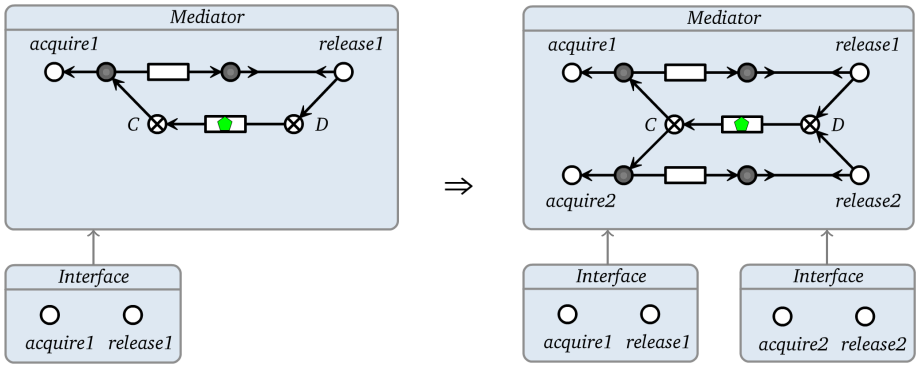


Figure 6.2: example morphism / transformation rule p_1 of distributed Reo graphs

Following the notation of [38], we denote the category of distributed Reo graphs as **Dis**(**HGraph**$_{Reo}$). The category **HGraph**$_{Reo}$can be embedded by using network graphs with a single node and no edges.

6.2.2 Extended typing for distributed Reo networks

We can extend the typing mechanism of **Dis**(**HGraph**$_{Reo}$) for a proper modeling of distributed Reo networks. When Reo networks are considered as distributed typed graphs, i.e., objects (N, D) with $D\colon N \to$ **HGraph**$_{Reo}$, only the local graphs are typed. Applied to Reo, typing information at the network level can be useful as well.

For instance, we can use the network type graph in Figure 6.3 for a more detailed specification of the mediator network example. This type graph defines three types of local connectors on the network layer: *Mediator*, *Interface* and *WorkerPool*. It, moreover, ensures that edges can exist only from interfaces to mediators and worker pools. It is not possible to connect mediators and worker pools directly, but only via an interface. However, this typing information is limited to the network layer only. Type constraints that relate the local and the network layer still cannot be expressed. For the mediator network, we would like to be able to enforce the additional constraint that interfaces may consist only of nodes. Furthermore, we want to distinguish two types of nodes: *acquire* and *release*. This additional type information helps to ensure that a node in an interface is mapped to the right node in a connector.

To model type constraints that relate the local and network layer we switch to a slightly different distributed graph model. We consider *typed distributed graphs*, as opposed to distributed typed graphs that we had before. In **Dis**(**HGraph**$_{Reo}$) we had a type graph *Reo* which describes the local layer only. We can now consider a distributed type graph *DisReo* which describes both layers at once. For the mediator example, we can define *DisReo* on the network as the network type graph in Figure 6.3. For each

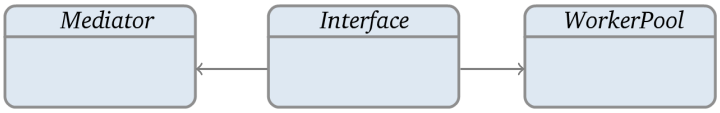


Figure 6.3: a network type graph for Reo

node in the network type graph we then define a local type graph. For the network node *Mediator* we choose the default type graph *Reo* with the modification that we consider three different node types: *acquire*, *release* and *internal*. This enables us to distinguish between interface and internal nodes. For the network node *Interface* we consider a local type graph with two node types: *acquire* and *release*, and no channels or other primitives. For the *WorkerPool* node we have again two node types for the interface and, furthermore, two hyperedges for modeling the worker components. The edges in the distributed type graph *DisReo* then are the obvious embeddings of the interface node types. Based on this distributed type graph we can then switch to the slice category $\mathbf{Dis}(\mathbf{HGraph})\backslash \mathit{DisReo} = \mathbf{Dis}(\mathbf{HGraph})_{\mathit{DisReo}}$. In this way we obtain a more expressive typing mechanism which helps us to ensure the consistency of our distributed network models.

6.3 Reconfiguring distributed networks

In order for the double-pushout approach to apply, we must make sure that the category of distributed Reo graphs $\mathbf{Dis}(\mathbf{HGraph})_{\mathit{DisReo}}$ has pushouts. As shown in [38] the category of distributed objects $\mathbf{Dis}(\mathbf{C})$ is (co)complete if $\mathbf{C}$ is (co)complete. We therefore know that $\mathbf{Dis}(\mathbf{HGraph})$ is also cocomplete. Since $\mathbf{Dis}(\mathbf{HGraph})_{\mathit{DisReo}}$ is just a slice category of $\mathbf{Dis}(\mathbf{HGraph})$ we immediatly derive that it is cocomplete, too, and that it consequently has pushouts and thus enables DPO graph transformations for modeling reconfigurations of distributed Reo networks.

In the following, we show that a reconfiguration can be also defined locally, i.e., in the scope of a single connector, and we discuss how these local reconfigurations can be synchronized. The key tool for synchronizing local reconfigurations will be the concept of amalgamation.

6.3.1 Local reconfigurations

The need for local reconfigurations arises from the distributed setting, where no global knowledge of the network is available. In the following, we consider local reconfiguration rules for the resource mediator example.

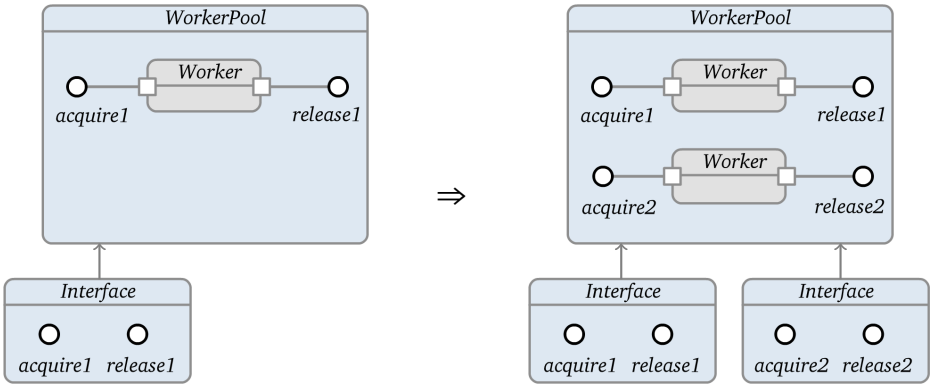


Figure 6.4: distributed reconfiguration rule p_2

Example 6.5 (local reconfiguration rules)**.** In the non-distributed version of the mediator network example, we used a single rule for adding a worker component to the network (cf. Example 5.5). In the distributed case, we want to model the reconfiguration of the mediator and the worker pool using separate local rules. The rule p_1 for adding a 'slot' to the mediator connector is shown in Figure 6.2. The rule p_2 is depicted in Figure 6.4 and adds a new worker component to the worker pool. Both rules p_1 and p_2 create a new interface node in the network and wire it automatically. These rules enable a black-box view on the mediator, on the one hand, and the worker pool, on the other. The key idea is that the network reconfiguration is splitted into local parts. The connectors, moreover, are not supposed to be accessed directly, but via their interfaces, which are reconfigured together with the actual connector. △

To reconfigure networks using local rules, we need a way to synchronize the local reconfigurations. As can be seen from the example above, applying the two local reconfiguration rules p_1 and p_2 naively to the network of Figure 6.1, does not give the desired result, because each rule creates a separate new interface, whereas we actually need just one that is shared by the two connectors.

6.3.2 Synchronizing local reconfigurations

Following [17] (see also [46]), we use the concept of amalgamation for synchronizing local reconfigurations. We first recall the basic definitions for amalgamated graph transformations.

The synchronization of two productions is achieved by identifying a common subproduction and gluing the productions along this subproduction. Let

$$p_i = L_i \xleftarrow{\ell_i} K_i \xrightarrow{r_i} R_i$$

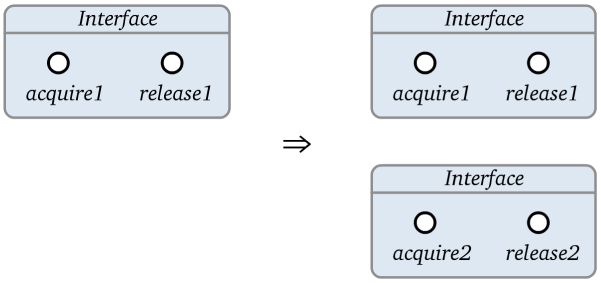


Figure 6.5: common subproduction p_0 for modeling the interface evolution

be two productions with $i \in \{0,1\}$. The production p_0, together with graph morphisms $in_L^1\colon L_0 \to L_1$, $in_K^1\colon K_0 \to K_1$, $in_R^1\colon R_0 \to R_1$, is called a *subproduction* of p_1, if in the following diagram (1) and (2) commute. Putting $in^1 = \langle in_L^1, in_K^1, in_R^1 \rangle$, we write $in^1\colon p_0 \to p_1$ for the embedding of p_0 into p_1.

$$\begin{array}{ccccc} L_0 & \xleftarrow{\ell_0} & K_0 & \xrightarrow{r_0} & R_0 \\ \downarrow{\scriptstyle in_L^1} & (1) & \downarrow{\scriptstyle in_K^1} & (2) & \downarrow{\scriptstyle in_R^1} \\ L_1 & \xleftarrow{\ell_1} & K_1 & \xrightarrow{r_1} & R_1 \end{array}$$

The productions p_1 and p_2 are called *synchronized* with respect to p_0, if p_0 is a subproduction of both p_1 and p_2, denoted by $p_1 \xleftarrow{in_1} p_0 \xrightarrow{in_2} p_2$.

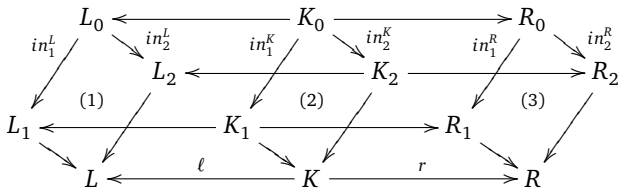
Example 6.6 (interface evolution)**.** A non-trivial (i.e., non-empty) subproduction p_0 of the reconfiguration rules p_1 and p_2 is depicted in Figure 6.5. The rule creates a new interface node in a network graph. As a general property of our reconfiguration approach, the common subproduction of two synchronized rules always describes an interface change of the involved connectors. By making explicit the change of interface due to an update of connectors, synchronized productions can properly describe reconfigurations in a distributed network. △

The execution of synchronized productions can be achieved using amalgamation. Given two synchronized productions $p_1 \xleftarrow{in_1} p_0 \xrightarrow{in_2} p_2$, the *amalgamated production*

$$p_1 \oplus_{p_0} p_2\colon L \xleftarrow{\ell} K \xrightarrow{r} R$$

is constructed by gluing p_1 and p_2 along p_0 using the pushouts (1), (2) and (3) in the diagram below, such that all squares commute. The morphisms ℓ and r are induced by the universal property of the pushout (2). Applying $p_1 \oplus_{p_0} p_2$ to a graph G yields a

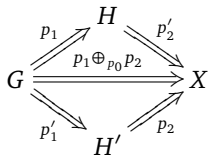
so-called *amalgamated derivation* $G \Rightarrow X$.



EXAMPLE 6.7 (amalgamated reconfiguration)**.** Amalgamation of the productions p_1 and p_2 in Figures 6.2 and 6.4 using the subproduction p_0 in Figure 6.5 yields the intended reconfiguration rule for the reconfiguring distributed mediator network in Figure 6.1. Note that the complete distributed network is reconfigured using local rules in one atomic step. △

Although it provides proper means for synchronizing local reconfigurations, in general, amalgamation is not well-suited for application in distributed networks. This is due to the fact that it requires (i) knowledge of all connectors and their reconfiguration rules, and (ii) a centralized entity that is aware of the whole network and that performs the reconfiguration in a non-local fashion.

The solution to this problem is an asynchronous execution of synchronized productions. The idea is to adapt the transformation rules in such a way that their sequential application yields the same result as the amalgamated rule:



Here, p_1' and p_2' are called the *remainders* of p_1 and p_2 w.r.t. p_0. This problem was solved by Boehm et al. in the so-called amalgamation theorem in [17], which can be seen as a generalization of the well-known parallelism theorem for graph grammars [28]. The authors present in [17] a construction of remainder rules and show that they in fact yield a decomposition of the amalgamated derivation. Unfortunately, the result was shown for node and edge labeled graphs only. A direct transfer to typed distributed hypergraphs, as we use them here, cannot be easily established and a generalization of the amalgamation theorem is out the scope of this thesis.

However, in our application to Reo, we can consider two concrete examples of an asynchronous application of local reconfiguration rules, such as the ones in Figure 6.2 and 6.4. The first case we consider is that a new interface node is created as in Figure 6.5, which is then automatically wired with the two connectors. For this kind of reconfiguration, we take $p_1' = p_1$ and derive p_2' from p_2 by adding the interface node to be created to the LHS, already. Thus, p_2' merely establishes the links between the interface and the connector. As a second example, we consider the reverse rule: removing an interface node and updating both connectors accordingly. In this situation

we adapt p_1 to p_1' by letting it destroy the link between the interface and the connector only. We then take $p_2' = p_2$ and thus the interface node is properly removed in the second step of the reconfiguration.

Our claim is that it is in principle possible to apply reconfiguration rules locally. Thus, a network can be reconfigured by a stepwise updating of its constituent connectors. In particular, the connectors can also be black boxes that reconfigure themselves. On the other hand, these local reconfigurations must be coordinated somehow, since the order of local reconfigurations and the choice of which connector updates the common interface is not clear. For this purpose, we discuss a strategy now.

6.3.3 Coordinating local reconfigurations

We outline a strategy for organizing local reconfigurations in a network. The central idea is that a reconfiguration is triggered locally at one of the connectors and that this creates a cascade of follow-up reconfigurations across the network.

Connectors may be equipped with synchronized reconfiguration rules, i.e., rules that describe how a connector itself is changed, and further, how its interfaces are updated. We also assume that a connector reconfigures itself triggered by an external request. For this purpose, it may publish the names of its reconfiguration rules. Connectors in the neighborhood can invoke these reconfiguration rules via their shared interface (through a communication channel that is not explicitly modeled here). When a rule is invoked, a connector performs the reconfiguration in three steps:

(i) Determine the interface where the request came from and the interfaces of those connectors in the neighborhood that also need to be updated.

(ii) Send reconfiguration requests to those connectors in the neighborhood that must be updated and block until they are reconfigured.

(iii) Do the local reconfiguration and reconfigure the interface, if necessary, only where the request came from.

We assume that there is an active party in the network that initiates the reconfiguration by invoking a rule on some connector. Every connector can handle only one reconfiguration request at a time. Hence, the request builds up a reconfiguration dependency tree in the network. The root of the tree is where the reconfiguration was initially invoked. The reconfiguration is then executed bottom-up, starting at the leaves until the root is also reconfigured.

Connectors may also respond to a reconfiguration request with a failure. In that case, the failure is forwarded in the network and all reconfigurations performed so far are rolled-back. This ensures atomicity of the reconfiguration. Thus, our approach enables a black-box view on connectors and provides a scheme for a localized definition and execution of reconfigurations, while still ensuring consistency and atomicity at the network level, without requiring a centralized entity, which is a prime assumption in distributed environments.

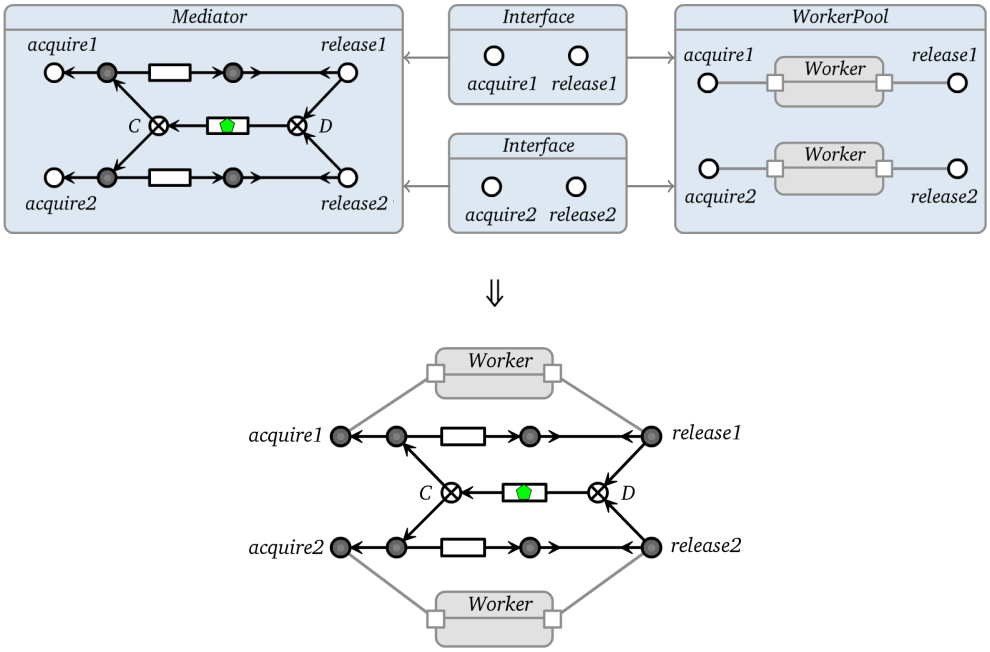


Figure 6.6: flattening of the resource mediator network

6.4 Flattening of distributed graphs

In this section, we consider a flattening operation for distributed networks. A well-known fact from the theory of distributed graph transformation is that a flattening of a distributed graph or object (N, D) can be achieved by considering the colimit of D [86] and that this extends to a functor F: **Dis**(**C**) $\to$ **C**, assuming **C** is cocomplete [32]. This definition is rather elegant since it defines the flattening operation in terms of a universal property, and not by an algorithm or referring to an operational semantics.

In the case of distributed graphs, the flattening essentially glues together all local graphs along their shared subgraphs. We can directly apply flattening to distributed Reo networks, as shown in the following example.

Example 6.8 (flattening of distributed networks)**.** Figure 6.6 depicts the flattening of the resource mediator network. The flattening glues together the mediator connector and worker pool along their shared interface nodes. △

Considering flattening of distributed graphs is in particular interesting when distribution is used for representing a logical partitioning of an otherwise flat system. The distributed graph model can be interpreted as a higher-level view of a flat structure. In this perspective, it is crucial to know whether flattening interacts well with composition. Composing two distributed objects in **Dis**(**C**) and flattening the result

should yield the same outcome as first flattening both distributed objects and then composing them in **C**. Formally, we need to show that the flattening functor F preserves pushouts, or more generally, colimits.

THEOREM 6.9 (flattening preserves colimits)**.** *Let* **C** *be a cocomplete category. The flattening functor* $F\colon \mathbf{Dis}(\mathbf{C}) \to \mathbf{C}$ *has a right adjoint and is therefore cocontinuous.*

PROOF. A detailed proof is given in Appendix A.2. □

Thus, we can use distributed graphs and graph transformation to describe logical partitioning of Reo networks which can –at the implementation level– still be realized as flat networks. In particular, composition and reconfiguration of distributed networks can be transparently carried out on the underlying flat network.

6.5 Related work

Modeling the distribution of systems via embedding interfaces represented as morphisms in a suitable category is also used for open Petri nets [11]. The explicit modeling of glue code and the exploitation of pushout constructions to deal with composition in this work is similar to ours.

In [21], Architectural Design Rewriting is proposed as a framework for modeling reconfigurable software architectures. This work deals with hierarchical, non-distributed architectures and uses hyperedge replacement, as opposed to algebraic graph transformations in our work. A general introduction to system modeling and system evolution using graph transformation techniques, including hierarchical and distributed approaches, can be found in [42].

Our approach to coordination achieved by Reo connectors and their dynamic reconfiguration fits in the framework of runtime software adaptation [107, 23] for component-based software engineering. Process algebraic treatments include [20, 31]. To accommodate dynamic reconfiguration, predicted behavioral changes combined with revision of message translation are captured by so-called contextual mappings. However, the focus in this work is not on distribution, which is a key aspect of our approach. A workflow language extension with the so-called configurable elements, e.g. for YAWL [104], is proposed in [47], with a semantics based on a variant of Petri-nets, called extended workflow nets (EWF-nets).

6.6 Conclusions

We presented a framework for modeling distributed Reo networks and their reconfigurations using the theory of distributed graph transformation. Our approach allows a black-boxed view on subnetworks for which reconfigurations can be defined and executed locally. Furthermore, we have showed how local reconfigurations can be synchronized in the absence of a centralized entity, which is a prime assumption in distributed environments. In the last part of this chapter we presented a new result regarding the compositionality of the flattening operation for distributed graphs. This

result is of a general nature, since we have shown it for a generic category **Dis**(**C**) of distributed objects. In Chapter 7, we use this result to show an enhanced notion of compositional semantics for so-called *distributed port automata* which are an integrated structural and behavioral model for, e.g., Reo networks and Petri nets.

Chapter 7

Distributed port automata

In this chapter, we introduce *distributed port automata* to establish a connection between the graph transformation based reconfiguration approach presented in Chapter 5 and the port automata model discussed in Chapter 3. We provide a generalized result on compositional semantics which can be applied to dynamic reconfiguration.

7.1 Overview

Analyzing the interplay of execution and reconfiguration can give valuable insights into the dynamics of a reconfigurable network. Key challenges in this area include (i) the problem of state transfer, i.e., to determine the state of the network after a reconfiguration, and (ii) to ensure consistency in terms of structural and/or behavioral invariants of the network.

In Section 5.3, we presented an approach to encode the execution semantics of Reo using graph transformation rules. However, this approach lacks compositionality and it is not clear how to apply it in general. Therefore, we seek for an alternative approach to unify the structural and behavioral aspects of Reo networks. A key challenge in this chapter is to integrate the graph transformation based approach for network reconfiguration, as presented in Chapter 5, and the port automata based semantics of networks, as discussed in Chapter 3. For this purpose, we recast the port automata model into a categorical setting, in which we consider port automata as objects and simulations between them as morphisms. Unlike the usual notion of simulation, our definition of port automata morphisms includes an additional inverse mapping of nodes. This allows us to define a rather liberal notion of simulation in which the automata do not strictly need to share the same set of port names. We show then that the category **PA** of port automata and their simulations is complete. In particular, pullbacks in **PA** can be considered as a generalized join operator in which automata are joined over a common subautomaton, which can be seen as shared behavioral interface or context.

We exploit again the framework of distributed graph transformation to show how

Reo networks can be modeled as distributed port automata – formally as objects in the category $\mathbf{Dis}(\mathbf{PA}^{op})$. This category combines both the structural and the behavioral aspects of networks. The primitives, i.e., the channel and nodes in a network, are modeled using primitive port automata (cf. Table 3.2) which are the network nodes in a distributed port automaton. Moreover, the network topology is modeled by the network graph itself. Thus, distributed port automata capture the structure of a network and the semantics of the primitives it is comprised of.

After the presentation of the distributed port automata model, we show how the flattening functor as discussed in Section 6.4 can be used to derive the semantics of complete networks. We show that the semantics of a network can be defined as the limit over the network graph. This can be seen as a generalization of the join operator defined as pullbacks in **PA**. Thus, the limit over a network graph provides us with a characterization of its semantics in terms of a universal construction. Finally, we show the compositionality of the semantics of distributed port automata. That is, a structural gluing of the network graphs using pushouts corresponds to a pullback to the semantical level. We use Theorem 6.9 for showing this result. An interesting corollary is that the structure and the semantics of distributed port automata (and therefore Reo) can be phrased in terms of a pair of adjoint functors.

As applications, we show how distributed port automata can be used to model Reo networks as well as Petri nets. Moreover, we show that reconfiguration can be modeled using DPO graph transformation at the network level in $\mathbf{Dis}(\mathbf{PA}^{op})$. The connection between the structural and the semantical level provides us with a framework in which we can reason about the semantics of dynamically reconfigurable networks. Specifically, we can solve the problem of state transfer and, furthermore, define a consistency condition for dynamic reconfigurations.

The results in this chapter are a generalization of the tailored approach in [95] and were first presented in [96]. As in the previous two chapters, we assume familiarity with basic notions from category theory, e.g. pullbacks and functors. For a comprehensive introduction to category theory we refer to [68].

7.2 The category of port automata

We use port automata as presented in Definition 3.6. To establish a categorical framework, we introduce a notion of simulations for port automata. A port automata simulation essentially consist of a mapping of states and transitions, and an inverse mapping of port names. We ensure consistency of firing events using a condition on the transitions of the port automata.

Definition 7.1 (port automata simulation)**.** Let $PA_1 = (Q_1, N_1, T_1, q_0^1)$ and $PA_2 = (Q_2, N_2, T_2, q_0^2)$ be two port automata. A simulation $f = (f_Q, f_N, f_T)$ consists of functions $f_Q\colon Q_1 \to Q_2$, $f_N\colon N_2 \to N_1$ and $f_T\colon T_1 \to T_2$, such that:

- $f_Q(q_0^1) = q_0^2$ and
- $f_T\left(q_1 \xrightarrow{S_1} p_1\right) = \left(q_2 \xrightarrow{S_2} p_2\right)$ implies $q_2 = f_Q(q_1)$, $p_2 = f_Q(p_1)$ and $S_2 = f_N^{-1}(S_1)$ ◊

Note that port names are mapped in the opposite direction and that the condition $S_2 = f_N^{-1}(S_1)$ ensures consistency of firing events on all shared port names.

Example 7.2 (port automata simulation)**.** An example of a port automata simulation is depicted in Figure 7.1. States q_0, q_2 are both mapped to p_0, and q_1 is mapped to p_1. The port name function is the inclusion map in the opposite direction. The transition via $\{B, C\}$ in the source is mapped to the transition via $\{B\}$. The transition via $\{C\}$ corresponds to the τ-step in the target automaton. △

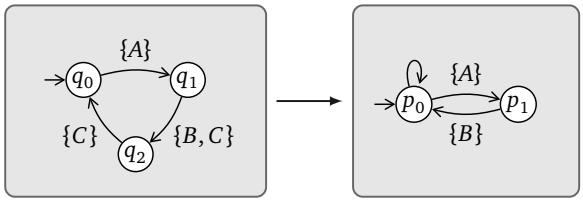


Figure 7.1: a simulation of port automata

Note that we model τ-transitions explicitly here. We define identity and composition of port automata simulations component-wise. The following lemma states that port automata and simulations form a category.

Lemma 7.3 (category of port automata)**.** *Port automata and port automata simulations give rise to a category, denoted by* **PA**.

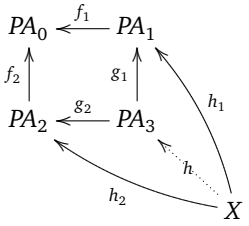
Proof. We verify the consistency condition of firing events for the identity: $S = id_N^{-1}(S)$. Similarly, for the composition of $f : PA_1 \to PA_2$ and $g : PA_2 \to PA_3$ we have:

$$(g \circ f)_N^{-1}(S_1) = \left(f_N \circ g_N\right)^{-1}(S_1) = g_N^{-1}\left(f_N^{-1}\left(S_1\right)\right) = g_N^{-1}(S_2) = S_3 \qquad \square$$

The port automaton with one state, an empty port name set and a single τ-transition is the final object in **PA**, denoted by **1**. If there is a morphism between two port automata PA_1 and PA_2, we may also write $PA_1 \succeq PA_2$ for short. Similarly, if there exists a (categorical) isomorphism, we denote this by $PA_1 \cong PA_2$. Note that this notion of behavioral equivalence is stronger than the equivalence based on bisimulations as in Definition 3.4.

We define the composition of port automata now using pullbacks in **PA**. This generalizes the join operator of Definition 3.2, since it allows to join two automata over a common automaton, which can be seen as a shared context. Our new composition operator is derived and phrased in terms of a universal property. Observe that, unlike the join operator for port automata in Definition 3.2, the composition using pullbacks is based on mere synchronization only. Interleavings of actions are modeled by synchronizations of an action and a τ-transition in the other automaton. Consequently, the join operator based on pullbacks is simpler since only one rule is required for composing transitions.

THEOREM 7.4 (pullbacks of port automata)**.** *The category* **PA** *has pullbacks and they can be constructed component-wise in* **Set***. For a cospan $PA_1 \to PA_0 \leftarrow PA_2$ the pullback object is $PA_3 = (Q_3, N_3, T_3, q_0^3)$ with:*



- $Q_3 = Q_1 \times_{Q_0} Q_2$ *(pullback in* **Set***)*
- $N_3 = N_1 +_{N_0} N_2$ *(pushout in* **Set***)*
- $q_0^3 = \langle q_0^1, q_0^2 \rangle$
- T_3 *is defined by the following rule:*

$$\frac{f_{1,T}(q_1 \xrightarrow{S_1} p_1) = f_{2,T}(q_2 \xrightarrow{S_2} p_2) \quad S_0 = f_{1,N}^{-1}(S_1) = f_{2,N}^{-1}(S_2) \quad S_3 = S_1 +_{S_0} S_2}{\langle q_1, q_2 \rangle \xrightarrow{S_3}_3 \langle p_1, p_2 \rangle} \tag{7.1}$$

PROOF. Due to the component-wise construction in **Set**, we need to show only that g_1, g_2 and h are valid **PA**-morphisms, i.e., we need to check the consistency condition for the firing ports. For g_1 we have by construction: $g_{1,N}^{-1}(S_3) = g_{1,N}^{-1}(S_1 +_{S_0} S_2) = S_1$, and analogously for g_2. For h, assume there is a transition via S in X that is, w.l.o.g., mapped to a transition via S_i in PA_i with $i = 1, 2$. We need to show that there is a corresponding transition via S_3 in PA_3. Since the h_i are by assumption valid **PA**-morphisms, we have: $h_{i,N}^{-1}(S) = S_i$. Moreover, $f_1 \circ h_1 = f_2 \circ h_2$ and thus: $f_{1,N}^{-1}(S_1) = f_{2,N}^{-1}(S_2)$. Therefore, the premise of rule (7.1) is fulfilled and the transition exists. Since $h_i = g_i \circ h$ we also know that $h_N^{-1}(S) = S_3$ and thus the consistency condition holds also for h. □

Note that $f_1(q_1 \xrightarrow{S_1} p_1) = f_2(q_2 \xrightarrow{S_2} p_2)$ also implies $f_1(q_1) = f_2(q_2)$ and $f_1(p_1) = f_2(p_2)$. Rule (7.1) for constructing the transition relation can be compared with rule (3.2) of the classical join operator. Note, however, that the original operator allows to join any two transitions, whereas the pullback requires that the images of the transitions and states coincide in the context automaton PA_0. The semantics of the original join operator can be recovered if PA_0 consists of exactly one state and one τ-transition (but not necessarily empty port names set).

EXAMPLE 7.5 (pullback of port automata)**.** An example of a port automata pullback is depicted in Figure 7.2. The state maps are indicated by state names. Note that the pullback automaton in the bottom right actually includes more states and transitions which are not shown here since they are unreachable. The automata in this pullback diagram can be modeled using *FIFO1* channels. In fact, all four automata correspond to Reo connectors and the whole pullback corresponds to a structural gluing of these connectors, as shown in the pushout of Reo graphs in Figure 7.3. We discuss how to derive the port automata semantics from a structural model of Reo connectors in Section 7.3.4. △

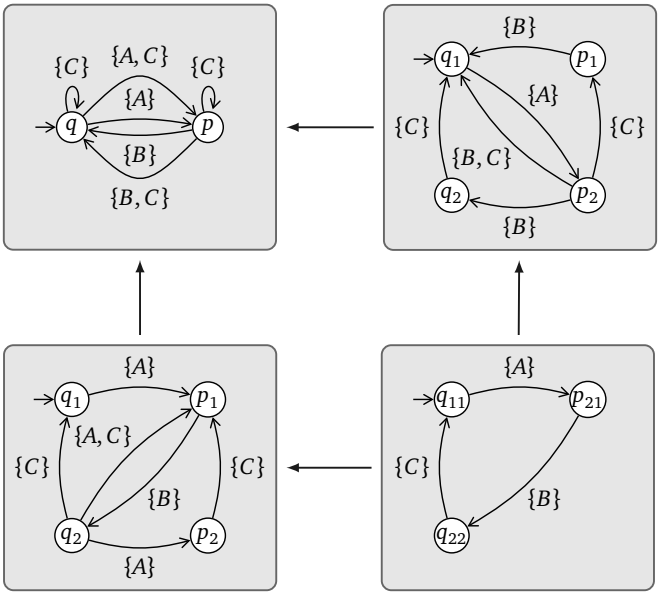


Figure 7.2: a pullback of port automata

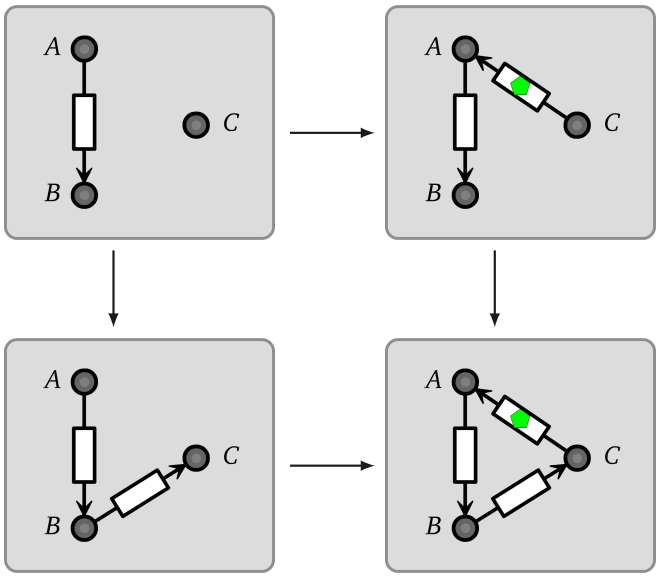


Figure 7.3: a pushout of Reo connectors

We use the default notation for pullbacks of port automata, i.e., $PA_3 = PA_1 \times_{PA_0} PA_2$. The product of two port automata is the special case where the interface automaton is the final object, i.e., $PA_1 \times PA_2 = PA_1 \times_{\mathbf{1}} PA_2$. As mentioned above, this notion of composition generalizes the join operation from Definition 3.2 and [10] since it allows composition along a common interface automaton. In the original approach, automata are joined only along a common set of port names, whereas we allow now the interface to be stateful as well. The states of the interface automaton can be seen as a shared context of the two automata to be composed.

The categorical construction using pullbacks furthermore includes the morphisms into the original port automata and thereby relates them with the result using simulations. Note also, that we have indirectly shown that **PA** has general limits, since it has pullbacks and a final object. In the following lemma, we state a basic compatibility result for port automata morphisms, which is a direct consequence of the universal property of pullbacks.

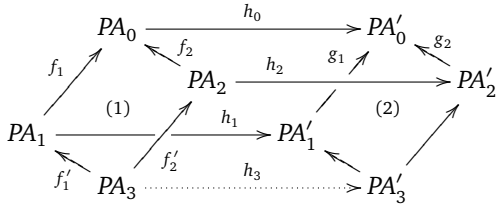
LEMMA 7.6 (compatibility with simulations)**.** *Given two cospans of port automata simulations* $PA_1 \xrightarrow{f_1} PA_0 \xleftarrow{f_2} PA_2$ *and* $PA'_1 \xrightarrow{g_1} PA'_0 \xleftarrow{g_2} PA'_2$*, then:*

(i) for three morphisms $h_i \colon PA_i \to PA'_i$ *with* $i \in \{0,1,2\}$*:*

- *if* $h_0 \circ f_1 = g_1 \circ h_1$ *and* $h_0 \circ f_2 = g_2 \circ h_2$ *then* $\left(PA_1 \times_{PA_0} PA_2\right) \succeq \left(PA'_1 \times_{PA'_0} PA'_2\right)$.

(ii) if $PA_1 \succeq PA'_1$ *and* $PA_2 \succeq PA'_2$ *then* $\left(PA_1 \times_{PA_0} PA_2\right) \succeq \left(PA'_1 \times PA'_2\right)$.

PROOF. Consider the diagram below where (1) and (2) are pullbacks of the given automata. The precondition of *(i)* states that the top-face and the back-face commute and (1) commutes as it is a pullback. Hence $g_1 \circ h_1 \circ f'_1 = g_2 \circ h_2 \circ f'_2 \colon PA_3 \to PA'_0$. The morphism h_3 is then uniquely defined by the pullback (2) and hence $PA_3 \succeq PA'_3$. For *(ii)* we take $PA'_0 = \mathbf{1}$ and automatically obtain the precondition of *(i)*.



□

7.3 The category of distributed port automata

Example 7.5 in the previous section already indicated that there appears to be a connection between the graph-based model of connectors and networks, and their semantical counterpart in the category of port automata. In this section, we provide a categorical model which integrates the structural aspects, i.e., the topology of a network, and the semantics of the primitives it is comprised of. As it turns out, we can

use again the theory of distributed graph transformation as presented in the more general setting of distributed objects in [38]. Specifically, we consider the following category.

DEFINITION 7.7 (category of distributed port automata)**.** The category of *distributed port automata* is defined as $\mathbf{Dis}(\mathbf{PA}^{op})$. ◊

Note that we consider $\mathbf{Dis}(\mathbf{PA}^{op})$ and not $\mathbf{Dis}(\mathbf{PA})$, because the **PA**-semantics is contravariant to the graph structure of networks, as also indicated in Example 7.5, where a pushout of networks corresponds to a pullback of their respective port automata. However, before investigating on the properties of $\mathbf{Dis}(\mathbf{PA}^{op})$, we first show how to encode Reo networks and Petri nets as distributed port automata to give an intuition about this model.

7.3.1 Encoding of Reo networks

The encoding of Reo in $\mathbf{Dis}(\mathbf{PA}^{op})$ is straightforward. The idea is that every primitive in a network is represented by its port automaton (cf. Table 3.2). In the same way, each node can be modeled as a simple stateless port automaton. Note, however, that the only semantics of a node here is mere synchronization. In particular, we do not model the merger semantics of nodes here. However, as argued also in other approaches (see, e.g., [26]), mergers can also be modeled as primitives and therefore the expressiveness is not affected.

Formally, we identify each primitive in a network with its port automaton and associate with every node X the stateless port automaton with one port name X and two transitions: one labeled $\{X\}$ and the other one a τ-transition. The port automata for all nodes and primitives are now considered as vertices in a network graph N. For every pair of a node and a connected primitive, we create an edge between their corresponding vertices in the network graph. The edge points toward the port automaton of the primitive. However, it corresponds to the **PA**-simulation in the opposite direction, which maps all transitions of the primitive that involve the connected node X to the self loop transition $\{X\}$, and all other transitions to τ. The reason for inverting the edges is, formally, the fact that we consider the category $\mathbf{Dis}(\mathbf{PA}^{op})$. Informally, this inversion can be motivated by arguing that the edges in the network graph model primarily structural mappings of the node names, which are contravariant for port automata simulations.

EXAMPLE 7.8 (Reo networks as distributed port automata)**.** Figure 7.4a depicts a Reo connector consisting of three *FIFO1*s and three nodes. Figure 7.4b shows the corresponding distributed port automata model. Note that the edges in the distributed port automaton correspond to *inverse* simulations. An easy way to remember the direction of the edges is to look at the port name map (inclusions here) and not the state map of the simulations. △

7.3.2 Encoding of Petri nets

Petri nets can also be modeled directly as distributed port automata. However, since we want the primitive port automata to be finite, we need to enforce an upper bound for the capacity of places. With this requirement the encoding is again straightforward. Essentially, transitions in a Petri net play the role of nodes in our encoding of Reo. Similarly, places are modeled as primitives with buffer semantics. However, Petri net places have bag semantics, as opposed to *FIFO* channels in Reo.

EXAMPLE 7.9 (Petri nets as distributed port automata)**.** Figure 7.5 depicts a Petri net and its corresponding distributed port automaton. In this example, all places have a capacity of 1. Similar to the distributed port automata encoding of full *FIFO1* channels in Reo, we model the markings of a Petri net place by adapting the initial state in its corresponding port automaton. Note also that, in this example, the upper and lower right places synchronize on B. △

Encoding of both Reo networks and Petri nets are one-to-one mappings, which include the semantics of all of their primitives, and moreover, contain their structural information, i.e., the topology of the networks.

7.3.3 Composing distributed port automata

In compliance with the DPO approach to graph transformation [28], we define composition of distributed port automata using pushouts. Therefore, we need the following lemma.

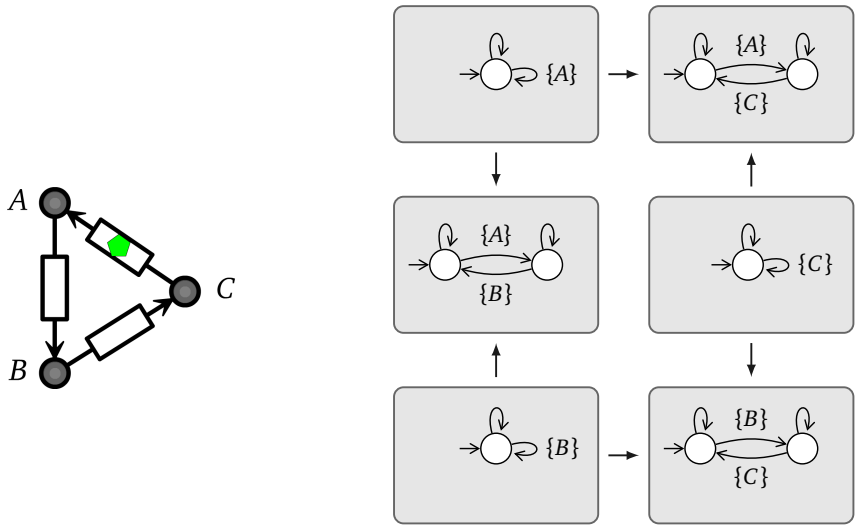
LEMMA 7.10. $\mathbf{Dis}(\mathbf{PA}^{op})$ *is cocomplete.*

PROOF. $\mathbf{PA}$ is complete since it has pullbacks and a final object. Thus, $\mathbf{PA}^{op}$ and therefore also $\mathbf{Dis}(\mathbf{PA}^{op})$ are cocomplete. □

In Figure 7.3 we show how pushouts can be used to glue Reo networks along a common subnetwork. Note again that this gluing is of a pure structural nature. In Section 7.3.1, particularly in Figure 7.4, we see how Reo networks can be encoding as distributed port automata. An important aspect of this encoding is that the topology of the network is modeled by the network graph of the distributed port automaton. Moreover, each local port automaton models the semantics of its respective primitive in the network.

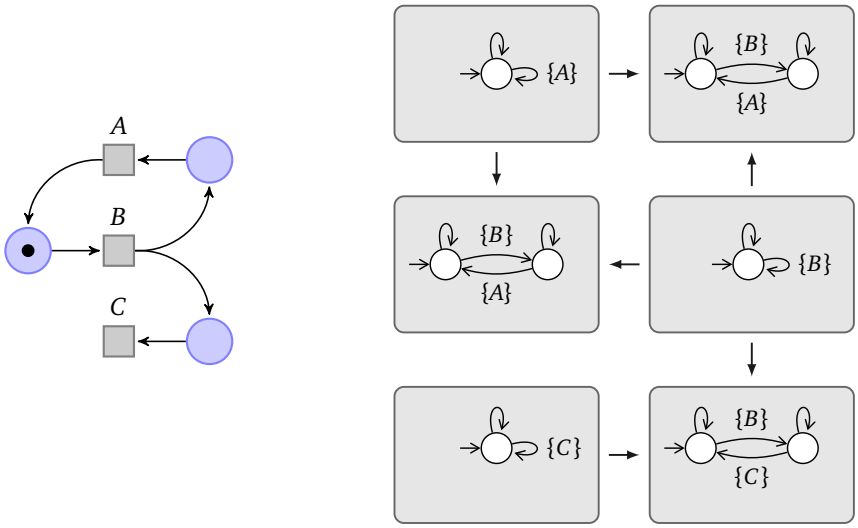
Since $\mathbf{Dis}(\mathbf{PA}^{op})$ is cocomplete, we can compose distributed port automata using pushouts. Relevant here is the fact that usually the semantics of primitives is fixed. Therefore, the local morphisms (port automata simulations) are isomorphisms. This is, for instance, the case for Reo networks and Petri nets. In this situation, the primitive port automata in network nodes are not changed when composing two distributed port automata using pushouts. Thus, the composition is performed only on the network level and is of purely structural nature.

The case where a local port automata morphism is not an isomorphism has applications as well. Essentially, it allows to model a refinement of primitives. For instance,



(a) graphical representation (b) distributed port automata model

Figure 7.4: a Reo network modeled as distributed port automaton



(a) graphical representation (b) distributed port automata model

Figure 7.5: a Petri net modeled as distributed port automaton

it is possible to refine a buffer with bag semantics to one with *FIFO* semantics. The only formal requirement is that the target automaton simulates the source automaton, i.e., there exists a port automata simulation that is contravariant to the structural mapping.

7.3.4 Semantics of distributed port automata

Distributed port automata contain the semantics of each primitive in the network and, moreover, the topology of the network. The semantics of a network is given by the port automaton that can be derived by joining all primitive port automata in the network along their shared subautomata. Essentially, we should be able to derive the semantics of a network such as the one in Figure 7.4b by repeatedly constructing pullbacks in **PA** along the reverse simulations between the primitive port automata.

More directly, for a distributed port automaton $(N, D) \in \mathbf{Dis}(\mathbf{PA}^{op})$ (cf. Definition 6.1) we can construct the port automaton that corresponds to its network semantics as the colimit of the diagram D. The colimit of D glues together all shared node names. Moreover, since we have reversed the arrows, it corresponds to a limit in **PA**. As a special case, the semantics of a network with the shape of a span ($\bullet \leftarrow \bullet \rightarrow \bullet$) corresponds to a pullback of the primitive port automata, as presented in Theorem 7.4.

As we have discussed in Section 6.4, the colimit over a distributed graph or object can be interpreted as a flattening operation, which, moreover, extends to a flattening functor $F\colon \mathbf{Dis}(\mathbf{C}) \to \mathbf{C}$. In the case of distributed port automata, i.e., in the category $\mathbf{Dis}(\mathbf{PA}^{op})$, the flattening using colimits can thus be used to define the composite behavior of networks in terms of a semantics functor.

Definition 7.11 (semantics functor)**.** Let $F\colon \mathbf{Dis}(\mathbf{PA}^{op}) \to \mathbf{PA}^{op}$ the flattening functor for distributed port automata. By reversing the arrows, this induces the following contravariant functor:

$$Sem\colon \mathbf{Dis}(\mathbf{PA}^{op}) \to \mathbf{PA}$$

called the *semantics functor* for distributed port automata. ◊

The following example shows the contravariance of the semantics functor.

Example 7.12 (semantics functor)**.** Figure 7.6 depicts a morphism of distributed port automata together with the induced morphism between the semantical objects. The transitions via $\{C\}$ in the lower right automaton are mapped to τ-transitions in the lower left automaton. We have omitted all self-loop τ-transitions for clarity here. Similarly, the transition via $\{A, C\}$ is mapped to the transition $\{A\}$. Note that the consistency condition for firing events (see Definition 7.1) holds here. △

The following theorem states that the semantics of distributed port automata is compositional, i.e., it is compatible with composition of distributed port automata using pushouts, or more generally with colimits.

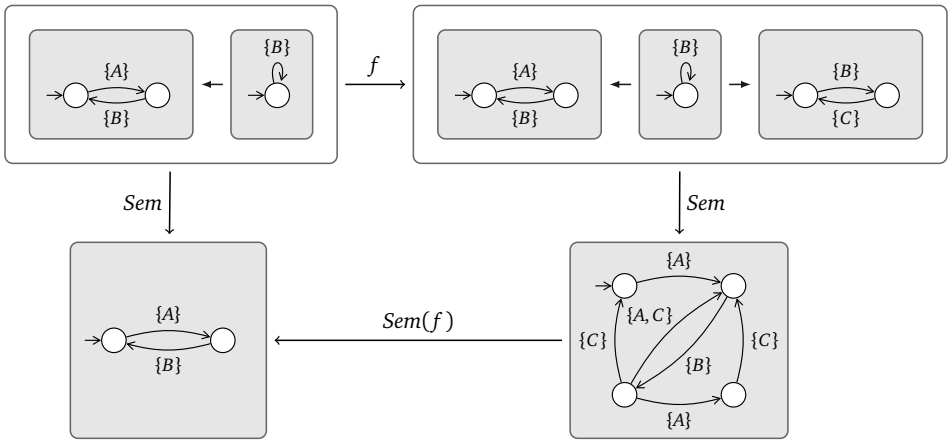


Figure 7.6: a distributed port automata morphism and its semantics

THEOREM 7.13 (compositional semantics)**.** *The semantics functor Sem*: **Dis**(**PA**^{op}) → **PA** *is compositional in the following sense: it maps colimits of distributed port automata to limits of port automata.*

PROOF. This holds since the flattening functor is cocontinuous (cf. Theorem 6.9). □

Thus, we have shown that a structural gluing of networks, which is realized as a pushout of the network graph, has a corresponding semantical join operation, i.e., a pullback of the respective port automata. Furthermore, Theorem 6.9 shows that the structure and the semantics of distributed port automata form a pair of adjoint functors.

7.4 Towards dynamic reconfiguration

Since distributed port automata constitute a combined model which captures both the structure of a connector or a network and the semantics of the primitives it is comprised of, distributed port automata can be used for problems occurring in the area of dynamic reconfiguration. We have stated already, that it is important to determine the state of a network after a dynamic reconfiguration, which we referred to as the problem of *state transfer*. Additionally, it is crucial to ensure that the new network state is indeed a valid one, which we referred to as the problem of *state consistency*.

In Section 5.3 we presented an approach to encode the execution semantics as graph transformation rules, which enabled us to analyze their interplay with dynamic reconfiguration. However, we have argued that this approach is not compositional. In general, it is not clear how to encode the semantics of a reconfigurable network with graph transformation rules for all possible configurations.

On the other hand, distributed port automata lay the grounds for reasoning about state transfer and consistency. To illustrate this, we revisit Example 7.5 which shows how a pushout of Reo networks (Figure 7.3) on the structural level corresponds to a pullback of port automata (Figure 7.2) on the semantical level. We now interpret this as an application of a simple reconfiguration rule which adds a full *FIFO1* between the matched nodes C and A. Essentially, we assume that Figure 7.3 is the right part of a DPO diagram. In this view, the corresponding pullback of port automata can be seen as an application of a 'semantical reconfiguration rule' in the category of port automata.

In such an approach, we can deduce the effect of an application of a purely structural reconfiguration rule on the connector semantics. For instance, assume that the connector to be reconfigured (lower left automaton in Figure 7.2) is in its initial state q_1, in which both *FIFO1*s are empty. The image of q_1 in the left-hand side of the rule (upper left automaton) is state q. Moreover, we assume that the *FIFO1* to be added by the rule is initially full, i.e., the selected state in the right-hand side of the rule is q_1 (in the upper right automaton). This information is sufficient to deduce the state of the connector after the reconfiguration. The pullback construction given in Theorem 7.4 yields as new target state q_{11} in the resulting automaton (lower right automaton). Thus, we can determine the state after a reconfiguration, which provides us with a means to solve the problem of state transfer.

Now assume that before the reconfiguration, the *FIFO1*(A,B) and *FIFO1*(B,C) are already full, i.e., the automaton in the upper right part is in state p_1 and the automaton in the lower left part is in state p_2. Both states are mapped to state p in the upper left automaton, and therefore correspond to state p_{12} in the lower right automaton, which is not shown because it is unreachable from the initial state. In this particular state, all three *FIFO1*s are full and the connector would run into a deadlock. Therefore, it is crucial to check in which state the connector currently is, before reconfiguring it. In essence, this is the problem of state consistency. In the distributed port automata approach, we can characterize reconfigurations which yield consistent connector states by demanding that the target state of the reconfigured connector must be reachable from the initial (or the current) state.

Thus, distributed port automata provide a formal framework for analyzing the interplay between the execution of a network on the one hand, and its reconfiguration on the other. In particular, our model provides a means for reasoning about state transfer and state consistency in dynamic reconfiguration scenarios.

7.5 Related work

A marking graph semantics of Petri nets is considered in [77]. Similarly to our approach, the authors show the compositionality of this semantics using a pair of adjoint functors. A compositional semantics for open Petri nets based on deterministic processes is considered in [11]. The automata semantics for Petri nets considered [34] is more restrictive than our port automata model, since their concurrent actions imply interleaved semantics. The authors show that there is a coreflection between a

category of Petri nets and their automata model.

The compositionality for constraint automata [10] states that the semantics of a connector can be computed out of the semantics of its constituent primitives. Our notion of compositionality is more general since it works with arbitrary gluings of connector graphs. In fact, we generalize the join operation of [10] by allowing to join two automata along a common context automaton.

7.6 Conclusions and future work

In this chapter, we have presented distributed port automata as an integrated structural and behavioral model for reconfigurable networks. We have shown how Reo networks and Petri nets can be directly encoded in this model and that it contains both the topology of the network and the semantics of the primitives it is comprised of. Furthermore, we argued that the flattening functor known from distributed graph transformation can be used to derive the semantics of a distributed port automaton. Moreover, we have shown that a structural gluing on the network level of distributed port automata has a corresponding composition operation on the semantical level. Thus, we have established a result on compositional semantics which can be applied to reconfiguration modeled as graph transformations. As concrete applications we discussed the problems of state transfer and consistency in the area of dynamic reconfiguration.

However, here we have considered only composition and not transformation of distributed port automata. Reconfiguration based on DPO transformation requires more general results, e.g. regarding the existence and construction of pushout complements. To apply the full theory of algebraic graph transformation requires to show that $\mathbf{Dis}(\mathbf{PA}^{op})$ is an (adhesive) high-level replacement category [39]. This, however, is out of the scope of this thesis and belongs to our future work.

Chapter 8

Conclusions and further directions

Adapting software at runtime using dynamic reconfiguration introduces new major challenges in their design and verification. In this thesis, we have studied formal modeling and verification techniques for static as well as dynamically reconfigurable component connectors.

We have shown in this thesis that model checking of connectors provides a powerful means for verifying their behavior. In our approach, using the mCRL2 specification language, we have demonstrated how to analyze static connectors including their data-, context- and time-dependent behavior. Moreover, we have argued that the theory of graph transformation offers a means for formal modeling of reconfigurable connectors, because it can naturally express both their structural and their behavioral aspects. An important feature of the graph transformation theory is its rule-based rewriting approach which, as we have shown in this thesis, can be used to formally describe dynamic reconfiguration in terms of *reconfiguration rules*. Using model checking and the so-called critical pair analysis known from rewrite systems, our reconfigurable connector models can be inspected for potentially harmful interplay of dynamic reconfiguration, on the one hand, and the execution itself, on the other. Moreover, we have shown how quantitative properties of reconfigurable connectors, e.g. steady state probabilities, can be analyzed as well.

Since the assumption of a centralized coordinator constitutes a big imposition on the type of system to be modeled, we have extended our approach to distributed reconfigurable connectors as well. To avoid ad hoc approaches, we have used the theory of distributed graph transformation for this purpose. As a new general result in the field of distributed graph transformation, we have shown that the flattening operation for distributed graphs is compositional. This result was also crucial in our *distributed port automata* model, which we have introduced to study issues arising in the area of dynamic reconfiguration. Specifically, we have shown how the problems of state transfer and state consistency can be solved in our set-up. For this purpose we have combined a semantical automata model with the graph transformation based approach for reconfiguration.

The practical relevance of the methods considered in this thesis has been made ev-

ident by providing sophisticated tool support. The Eclipse Coordination Tools (ECT) which have been implemented in the context of this thesis provide an integrated environment for modeling and analysis of component connectors in Reo. For the formal analysis approaches considered in this thesis we have used a number of verification tools (partially as back-ends of ECT), i.e., AGG [87], CADP [43], Henshin [6], mCRL2 [49] and PRISM [67].

Further research directions mainly involve areas where different modeling techniques are being combined. Examples include the use of (combined) stochastic and timed models in Reo itself, but also in the area of graph transformation systems. Similarly, the combination of automata-based behavioral models and graph transformation-based reconfigurations, as we have considered in our distributed port automata model, introduces a number of challenges. For instance, if the LTS induced by the reconfiguration rules is finite, it may be desirable to have an unfold operation that combines the execution semantics of the involved primitive port automata with the reconfiguration semantics of the network into a single LTS, similarly to our approach in Chapter 5 where we encoded the execution semantics also as graph transformation rules. Moreover, to apply the full theory of algebraic graph transformation requires to show that the category of distributed port automata, as introduced in Chapter 7 is an (adhesive) high-level replacement category [39]. Aside from these rather technical issues, we believe that the established formal modeling approaches, e.g. using automata models, process algebra or graph transformation, are on their own not powerful enough to study all problems in the area of dynamically reconfigurable systems. In this thesis, we therefore combined different modeling and verification techniques. However, it is not clear yet whether the challenges arising in dynamically reconfigurable systems can be solved solely using traditional formal methods.

Bibliography

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

[2] F. Arbab, C. Baier, F. de Boer, and J. Rutten. Models and temporal logical specifications for timed component connectors. *Software and Systems Modeling*, 6:59–82, 2007.

[3] F. Arbab, C. Baier, F. S. de Boer, J. J. M. M. Rutten, and M. Sirjani. Synthesis of Reo circuits for implementation of component-connector automata specifications. In *7th International Conference on Coordination Models and Languages (Coordination'05)*, volume 3454 of *Lecture Notes in Computer Science*, pages 236–251. Springer-Verlag, 2005.

[4] F. Arbab, T. Chothia, R. Mei, S. Meng, Y.-J. Moon, and C. Verhoef. From coordination to stochastic models of QoS. In *11th International Conference on Coordination Models and Languages (Coordination'09)*, volume 5521 of *Lecture Notes in Computer Science*, pages 268–287. Springer-Verlag, 2009.

[5] F. Arbab, T. Chothia, S. Meng, and Y.-J. Moon. Component connectors with QoS guarantees. In *9th International Conference on Coordination Models and Languages (Coordination'07)*, volume 4467 of *Lecture Notes in Computer Science*, pages 286–304. Springer-Verlag, 2007.

[6] T. Arendt, E. Bierman, S. Jurack, **C. Krause**, and G. Taentzer. Henshin: Advanced concepts and tools for in-place EMF model transformations. In *13th International Conference on Model Driven Engineering Languages and Systems (MoDELS'10)*, volume 6394 of *Lecture Notes in Computer Science*, pages 121–135. Springer-Verlag, 2010.

[7] C. Baier, T. Blechmann, J. Klein, and S. Klüppelholz. Formal verification for components and connectors. In *Formal Methods for Components and Objects: 7th International Symposium (FMCO'08), Revised Lectures*, pages 82–101. Springer-Verlag, 2009.

[8] C. Baier, T. Blechmann, J. Klein, and S. Klüppelholz. A uniform framework for modeling and verifying components and connectors. In *11th International Conference on Coordination Models and Languages (Coordination'09)*, volume 5521 of *Lecture Notes in Computer Science*, pages 247–267. Springer-Verlag, 2009.

[9] C. Baier, T. Blechmann, J. Klein, S. Klüppelholz, and W. Leister. Design and verification of systems with exogenous coordination using Vereofy. In *Leveraging Applications of Formal Methods, Verification, and Validation - 4th International Symposium on Leveraging Applications, (ISoLA'10)*, volume 6416 of *Lecture Notes in Computer Science*, pages 97–111. Springer-Verlag, 2010.

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

[11] P. Baldan, A. Corradini, H. Ehrig, and R. Heckel. Compositional semantics for open Petri nets based on deterministic processes. *Mathematical Structures in Computer Science*, 15:1–35, 2005.

[12] P. Baldan, A. Corradini, H. Ehrig, R. Heckel, and B. König. Bisimilarity and behaviour-preserving reconfigurations of open Petri nets. *CoRR*, abs/0809.4115, 2008.

[13] E. Biermann, K. Ehrig, C. Ermel, **C. Koehler**, and G. Taentzer. The EMF model transformation framework. In *Applications of Graph Transformations with Industrial Relevance (AGTIVE'07)*, volume 5088 of *Lecture Notes in Computer Science*, pages 566–567. Springer-Verlag, 2008.

[14] E. Biermann, K. Ehrig, **C. Koehler**, G. Kuhns, G. Taentzer, and E. Weiss. EMF model refactoring based of graph transformation concepts. In *3rd International Workshop on Software Evolution through Transformations (SETra'06)*, volume 3. Electronic Communications of the EASST, Sept. 2006.

[15] E. Biermann, K. Ehrig, **C. Koehler**, G. Kuhns, G. Taentzer, and E. Weiss. Graphical definition of in-place transformations in the Eclipse Modeling Framework. In *9th International Conference on Model Driven Engineering Languages and Systems (MoDELS'06)*, volume 4199 of *Lecture Notes Notes in Computer Science*, pages 425–439. Springer-Verlag, 2006.

[16] T. Blechmann and C. Baier. Checking equivalence for Reo networks. *Electronic Notes in Theoretical Computer Science*, 215:209–226, 2008.

[17] P. Boehm, H.-R. Fonio, and A. Habel. Amalgamation of graph transformations: a synchronization mechanism. *Journal of Computer and System Sciences*, 34:377–408, 1987.

[18] M. Bonsangue, D. Clarke, and A. Silva. Automata for context-dependent connectors. In *11th International Conference on Coordination Models and Languages (Coordination'09)*, volume 5521 of *Lecture Notes in Computer Science*, pages 184–203. Springer-Verlag, 2009.

[19] M. Bonsangue and M. Izadi. Automata based model checking for Reo connectors. In *3rd International Conference on Fundamentals of Software Engineering (FSEN'09)*, volume 5961 of *Lecture Notes in Computer Science*, pages 260–275. Springer-Verlag, 2010.

[20] A. Brogi, J. Cámera, C. Canal, J. Cubo, and E. Pimentel. Dynamic contextual adaptation. *Electronic Notes in Theoretical Computer Science*, 175:81–95, 2007.

[21] R. Bruni, A. L. Lafuente, U. Montanari, and E. Tuosto. Style-based architectural reconfigurations. *Bulletin of the EATCS*, 94:181–180, 2008.

[22] R. Bruni, I. Lanese, and U. Montanari. A basic algebra of stateless connectors. *Theoretical Computer Science*, 366(1):98–120, 2006.

[23] C. Canal, J. Murillo, and P. Poizat. Software adaptation. *L'Object*, 12:9–31, 2006.

[24] B. Changizi, N. Kokash, and F. Arbab. A unified toolset for business process model formalization. In *7th International Workshop on Formal Engineering approaches to Software Components and Architectures (FESCA'10)*. Tool demonstration paper, 2010.

[25] D. Clarke. A basic logic for reasoning about connector reconfiguration. *Fundamenta Informaticae*, 82(4):361–390, 2008.

[26] D. Clarke, D. Costa, and F. Arbab. Connector colouring I: Synchronisation and context dependency. *Science of Computer Programming*, 66(3):205–225, 2007.

[27] COMPAS. Compliance-driven Models, Languages, and Architectures for Services (EU project). `http://www.compas-ict.eu/`.

[28] A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel, and M. Löwe. *Handbook of Graph Grammars and Computing by Graph Transformation*, chapter Algebraic approaches to graph transformation I: Basic concepts and double pushout approach, pages 163–245. World Scientific, 1997.

[29] D. Costa. *Formal Models for Context Dependent Connectors for Distributed Software Components and Services*. PhD thesis, Vrije Universiteit Amsterdam, The Netherlands, 2010.

[30] Credo. Modeling and analysis of evolutionary structures for distributed services (EU project). `http://projects.cwi.nl/credo/`.

[31] J. Cubo, G. Salaün, J. Cámara, C.Canal, and E. Pimentel. Context-based adaptation of component behavioural interfaces. In *9th International Conference on Coordination Models and Languages (Coordination'07)*, volume 4467 of *Lecture Notes in Computer Science*, pages 305–323. Springer-Verlag, 2007.

[32] J. de Lara and G. Taentzer. Modelling and analysis of distributed simulation protocols with distributed graph transformation. In *5th International Conference on Application of Concurrency to System Design (ACSD'05)*, pages 144–153, Washington, DC, USA, 2005. IEEE Computer Society.

[33] E. Drexler. *Engines of Creation: The Coming Era of Nanotechnology*. Anchor, Oct. 1987.

[34] M. Droste and R. M. Shortt. From Petri nets to automata with concurrency. *Applied Categorical Structures*, 10(2):173–191, 2002.

[35] Eclipse platform. `http://www.eclipse.org`.

[36] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. Fundamental theory for typed attributed graphs and graph transformation based on adhesive HLR categories. *Fundamenta Informaticae*, 74(1):31–61, 2006.

[37] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of Algebraic Graph Transformation*. EATCS Monographs in Theoretical Computer Science. Springer-Verlag, 2006.

[38] H. Ehrig, F. Orejas, and U. Prange. Categorical foundations of distributed graph transformation. In *3rd International Conference on Graph Transformation (ICGT'06)*, volume 4178 of *Lecture Notes in Computer Science*, pages 215–229. Springer-Verlag, 2006.

[39] H. Ehrig, J. Padberg, U. Prange, and A. Habel. Adhesive high-level replacement systems: A new categorical framework for graph transformation. *Fundamenta Informaticae*, 74:1–29, 2006.

[40] EMF. The Eclipse Modeling Framework. `http://www.eclipse.org/emf`.

[41] EMFT. Eclipse Modeling Framework Technology. `http://www.eclipse.org/modeling/emft`.

[42] G. Engels and R. Heckel. Graph transformation as a conceptual and formal framework for system modeling and model evolution. In *Twenty-Seventh International Colloquium on Automata, Languages and Programming (ICALP'00)*, volume 1853 of *Lecture Notes in Computer Science*, pages 127–150. Springer-Verlag, 2000.

[43] H. Garavel, R. Mateescu, F. Lang, and W. Serwe. CADP 2006: A toolbox for the construction and analysis of distributed processes. In *Computer Aided Verification, 19th International Conference (CAV'07)*, volume 4590 of *Lecture Notes in Computer Science*, pages 158–163. Springer-Verlag, 2007.

[44] D. Gelernter and N. Carriero. Coordination languages and their significance. *Commun. ACM*, 35(2):97–107, 1992.

[45] D. Gelernter, N. Carriero, S. Chandran, and S. Chang. Parallel programming in Linda. In *International Conference on Parallel Processing (ICPP)*, pages 255–263, 1985.

[46] U. Golas, H. Ehrig, and A. Habel. Multi-amalgamation in adhesive categories. In H. Ehrig, A. Rensink, G. Rozenberg, and A. Schürr, editors, *ICGT*, volume 6372 of *Lecture Notes in Computer Science*, pages 346–361. Springer-Verlag, 2010.

[47] F. Gottschalk, W. M. P. van der Aalst, M. H. Jansen-Vullers, and M. La Rosa. Configurable workflow models. *Journal of Cooperative Information Systems*, 17:177–221, 2008.

[48] I. Grabe, M. M. Jaghoori, B. K. Aichernig, C. Baier, T. Blechmann, F. S. de Boer, A. Griesmayer, E. B. Johnsen, J. Klein, S. Klüppelholz, M. Kyas, W. Leister, R. Schlatte, A. Stam, M. Steffen, S. Tschirner, L. Xuedong, and W. Yi. Credo methodology: Modeling and analyzing a peer-to-peer system in Credo. *Electronic Notes in Theoretical Computer Science*, 266:33–48, 2010.

[49] J. F. Groote, A. H. J. Mathijssen, M. A. Reniers, Y. S. Usenko, and M. J. van Weerdenburg. The formal specification language mCRL2. In *Methods for Modelling Software Systems*. IBFI, Schloss Dagstuhl, 2007.

[50] R. van Glabbeek and P. Weijland. Branching time and abstraction in bisimulation semantics. *Journal of the ACM*, 43:555–600, 1996.

[51] R. Heckel, J. M. Küster, and G. Taentzer. Confluence of typed attributed graph transformation systems. In *1st International Conference on Graph Transformation (ICGT'02)*, volume 2505 of *Lecture Notes in Computer Science*, pages 161–176. Springer-Verlag, 2002.

[52] R. Heckel, G. Lajios, and S. Menge. Stochastic graph transformation systems. *Fundamenta Informaticae*, 74(1):63–84, 2006.

[53] Henshin. `http://www.eclipse.org/modeling/emft/henshin`.

[54] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21:666–677, 1978.

[55] S.-S. Jongmans, **C. Krause**, and F. Arbab. Encoding context-sensitivity in Reo into non-context-sensitive semantic models. In *13th International Conference on Coordination Models and Languages (Coordination'11)*, Lecture Notes in Computer Science (to appear). Springer-Verlag, 2011.

[56] O. Kanters. QoS analysis by simulation in Reo. Master's thesis, CWI Amsterdam and Vrije Universiteit Amsterdam, The Netherlands, 2010.

[57] S. Kemper. Compositional construction of real-time dataflow networks. In *12th International Conference on Coordination Models and Languages (Coordination'10)*, volume 6116 of *Lecture Notes in Computer Science*, pages 92–106. Springer-Verlag, 2010.

[58] S. Kemper. *Modelling and Analysis of Real-time Coordination Patterns*. PhD thesis, Leiden University (to be submitted), The Netherlands, 2011.

[59] S. Kemper. SAT-based verification for timed component connectors. *Science of Computer Programming (to appear)*, 2011.

[60] S. Kemper and A. Platzer. SAT-based abstraction refinement for real-time systems. In *Formal Aspects of Component Software, Third International Workshop (FACS'06)*, volume 182 of *ENTCS*, pages 107–122, 2007.

[61] R. Khosravi, M. Sirjani, N. Asoudeh, S. Sahebi, and H. Iravanchi. Modeling and analysis of Reo connectors using Alloy. In *10th International Conference on Coordination Models and Languages (Coordination'08)*, volume 5052 of *Lecture Notes in Computer Science*, pages 169–183. Springer-Verlag, 2008.

[62] N. Kokash, **C. Krause**, and E. P. de Vink. Data-aware design and verification of service compositions with Reo and mCRL2. In *2010 ACM Symposium on Applied Computing (SAC'10)*, pages 2406–2413. ACM, 2010.

[63] N. Kokash, **C. Krause**, and E. P. de Vink. Time and data-aware analysis of graphical service models in Reo. In *8th IEEE International Conference on Software Engineering and Formal Methods (SEFM'10)*, pages 125–134. IEEE Computer Society, 2010.

[64] N. Kokash, **C. Krause**, and E. P. de Vink. Verification of context-dependent channel-based service models. In *Formal Methods for Components and Objects: 8th International Symposium (FMCO'09)*, volume 6286 of *Lecture Notes in Computer Science*, pages 21–40. Springer-Verlag, 2010.

[65] N. Kokash, **C. Krause**, and E. P. de Vink. Reo+mCRL2: A framework for model-checking dataflow in service compositions. *Formal Aspects of Computing*, 2011. Special Issue on Selected SEFM'10 papers (under review).

[66] K. Krohn and J. Rhodes. Algebraic theory of machines I: Prime decomposition theorems for finite semigroups and machines. *Transactions of the American Mathematical Society*, 116:450–464, 1965.

[67] M. Kwiatkowska, G. Norman, and D. Parker. PRISM: Probabilistic Symbolic Model Checker. In *12th International Conference on Modelling Tools and Techniques for Computer and Communication System Performance Evaluation (TOOLS'02)*, Lecture Notes in Computer Science, pages 200–204. Springer-Verlag, 2002.

[68] S. M. Lane. *Categories for the Working Mathematician (Graduate Texts in Mathematics)*. Springer-Verlag, 2nd edition, 1998.

[69] mCRL2. `http://www.mcrl2.org`.

[70] R. Milner. Calculi for synchrony and asynchrony. *Theoretical Computer Science*, 25:267–310, 1983.

[71] R. Milner. *Communication and concurrency*. Prentice Hall International, 1989.

[72] MOF. Meta Object Facility (MOF) Core. `http://www.omg.org/spec/MOF`.

[73] Y.-J. Moon. *Stochastic Models for Quality of Service of Component Connectors*. PhD thesis, Leiden University (to be submitted), The Netherlands, 2011.

[74] Y.-J. Moon, A. Silva, **C. Krause**, and F. Arbab. A compositional semantics for stochastic Reo connectors. In *9th International Workshop on the Foundations of Coordination Languages and Software Architectures (FOCLASA'10)*, volume 30 of *Electronic Proceedings in Theoretical Computer Science*, pages 93–107, 2010.

[75] Y.-J. Moon, A. Silva, **C. Krause**, and F. Arbab. A compositional model to reason about end-to-end QoS in stochastic Reo connectors. *Science of Computer Programming (to appear)*, 2011.

[76] OCL. The Object Constraint Language. `http://www.omg.org/technology/documents/formal/ocl.htm`.

[77] J. Padberg, H. Ehrig, and G. Rozenberg. Behavior and realization construction for Petri nets based on free monoid and power set graphs. In *Unifying Petri Nets, Advances in Petri Nets*, volume 2128 of *Lecture Notes in Computer Science*, pages 230–249. Springer-Verlag, 2001.

[78] G. D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, University of Aarhus, 1981.

[79] G. D. Plotkin. The origins of structural operational semantics. *Journal of Logic and Algebraic Programming*, 60–61:3–15, 2004.

[80] D. Plump. Hypergraph rewriting: critical pairs and undecidability of confluence. In *Term Graph Rewriting: Theory and Practice*, pages 201–213. John Wiley and Sons Ltd., 1993.

[81] D. Plump. Critical pairs in term graph rewriting. In *19th International Symposium on Mathematical Foundations of Computer Science (MFCS'94)*, volume 841 of *Lecture Notes in Computer Science*, pages 556–566. Springer-Verlag, 1994.

[82] B. Pourvatan, M. Sirjani, H. Hojjat, and F. Arbab. Automated analysis of Reo circuits using symbolic execution. *Electronic Notes in Theoretical Computer Science*, 255:137–158, 2009.

[83] J. Proença. *Deployment of Distributed Component Based Systems*. PhD thesis, Leiden University, The Netherlands, 2011.

[84] J. Scholten. *Mobile channels for exogenous coordination of distributed systems: semantics, implementation and composition*. PhD thesis, Leiden University, The Netherlands, 2007.

[85] A. Silva. A specification language for Reo connectors. In *4rd International Conference on Fundamentals of Software Engineering (FSEN'10)*, Lecture Notes in Computer Science. Springer-Verlag (to appear), 2011.

[86] G. Taentzer. Distributed graphs and graph transformation. *Applied Categorical Structures*, 7:431–462, 1999.

[87] G. Taentzer. AGG: A graph transformation environment for modeling and validation of software. In *Application of Graph Transformations with Industrial Relevance (AGTIVE'03)*, volume 3062 of *Lecture Notes in Computer Science*, pages 446–453. Springer-Verlag, 2004.

[88] G. Taentzer and M. Beyer. Amalgamated graph transformations and their use for specifying AGG. In *Dagstuhl Seminar on Graph Transformations in Computer Science*, volume 776 of *Lecture Notes in Computer Science*, pages 380–394. Springer-Verlag, 1994.

[89] **C. Koehler**, F. Arbab, and E. P. de Vink. Reconfiguring distributed Reo connectors. In *Recent Trends in Algebraic Development Techniques: 19th International Workshop (WADT'08)*, volume 5486 of *Lecture Notes in Computer Science*, pages 221–235. Springer-Verlag, 2009.

[90] **C. Koehler** and D. Clarke. Decomposing port automata. In *2009 ACM Symposium on Applied Computing (SAC'09)*, pages 1369–1373, New York, NY, USA, 2009. ACM.

[91] **C. Koehler**, D. Costa, J. Proença, and F. Arbab. Reconfiguration of Reo connectors triggered by dataflow. In *7th International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT'08)*, volume 10 of *Electronic Communications of the EASST*, 2008.

[92] **C. Koehler**, A. Lazovik, and F. Arbab. ReoService: coordination modeling tool. In *5th International Conference on Service Oriented Computing (ICSOC'07)*, volume 4749 of *Lecture Notes in Computer Science*, pages 625–626. Springer-Verlag, 2007.

[93] **C. Koehler**, A. Lazovik, and F. Arbab. Connector rewriting with high-level replacement systems. In *6th International Workshop on the Foundations of Coordination Languages and Software Architectures (FOCLASA'07)*, volume 194 of *Electronic Notes in Theoretical Computer Science*, pages 77–92, Amsterdam, The Netherlands, 2008. Elsevier Science Publishers B.V.

[94] **C. Koehler**, H. Lewin, and G. Taentzer. Ensuring containment constraints in graph-based model transformation approaches. In *6th International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT'07)*, volume 6 of *Electronic Communications of the EASST*, 2007.

[95] **C. Krause**. Integrated structure and semantics for Reo connectors and Petri nets. In *2nd Interaction and Concurrency Experience Workshop (ICE'09)*, volume 12 of *Electronic Proceedings in Theoretical Computer Science*, page 57, 2009.

[96] **C. Krause**. Distributed port automata. In *10th International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT'11)*, Electronic Communications of the EASST (to appear), 2011.

[97] **C. Krause**, C. Krause, and E. P. de Vink. Action-based analysis of discrete regulatory networks with short-term stimuli. In *8th Conference on Computational Methods in Systems Biology (CMSB'10)*, pages 66–75, New York, NY, USA, 2010. ACM.

[98] **C. Krause**, Z. Maraikar, A. Lazovik, and F. Arbab. Modeling dynamic reconfigurations in Reo using high-level replacement systems. *Science of Computer Programming*, 76(1):23–36, 2011. Selected papers from the 6th International Workshop on the Foundations of Coordination Languages and Software Architectures (FOCLASA'07).

[99] W. M. P. van der Aalst. The application of Petri nets to workflow management. *Journal of Circuits, Systems, and Computers*, 8(1):21–66, 1998.

[100] W. M. P. van der Aalst. Interorganizational workflows: An approach based on message sequence charts and Petri nets. *Systems Analysis - Modelling - Simulation*, 34(3):335–367, 1999.

[101] W. M. P. van der Aalst. Exterminating the dynamic change bug: A concrete approach to support workflow change. *Information Systems Frontiers*, 3(3):297–317, 2001.

[102] W. M. P. van der Aalst and T. Basten. Inheritance of workflows: an approach to tackling problems related to change. *Theoretical Computer Science*, 270(1–2):125–203, 2002.

[103] W. M. P. van der Aalst and A. H. M. ter Hofstede. Workflow patterns: On the expressive power of (Petri-net-based) workflow languages. In *Fourth International Workshop on Practical Use of Coloured Petri Nets and the CPN Tools*, pages 1–20. Technical Report DAIMI PB-560, 2002.

[104] W. M. P. van der Aalst and A. H. M. ter Hofstede. YAWL: Yet Another Workflow Language. *Information Systems*, 30(4):245–275, 2005.

[105] Vereofy. `http://www.vereofy.de`.

[106] C. Verhoef, **C. Krause**, O. Kanters, and R. van der Mei. Simulation-based performance analysis of channel-based coordination models. In *13th International Conference on Coordination Models and Languages (Coordination'11)*, Lecture Notes in Computer Science (to appear). Springer-Verlag, 2011.

[107] D. Yellin and R. Strom. Protocol specifications and component adaptors. *ACM Transactions on Programming Languages and Systems*, 19:292–333, 1990.

Appendix A

Proofs

A.1 Proof for Theorem 4.4

PROOF. We show that the relation

$$R = \{\langle proc(q_1, q_2), \partial_B(\Gamma_C(proc(q_1) \parallel proc(q_2)))\rangle\}$$

is a strong bisimulation according to Definition 3.4. Note again that $proc(q_1, q_2)$ is the process corresponding to a state in the joint automaton $PA_1 \bowtie PA_2$, and that $proc(q_i)$ corresponds to a state in the automaton with renamed ports PA_i^R.

- Suppose $proc(q_1, q_2) \xrightarrow{S} P$. From Lemma 4.3 we know that $proc(q_1, q_2) \sim \langle q_1, q_2\rangle$. Therefore there must exist a corresponding transition in $PA_1 \bowtie PA_2$. Then one of the three cases from Definition 3.2 for the join operator applies:

  (i) There exists $q_1 \xrightarrow{S} p_1$ in PA_1 with $S \cap N_2 = \emptyset$ and we have $P = proc(p_1, q_2)$. Since none of the port names in S are part of the renaming, the transition is the same in PA_1^R. For PA_1^R we have again $q_1 \sim proc(q_1)$ and thus:

  $$proc(q_1) \xrightarrow{S} proc(p_1)$$

  Moreover, from the semantics of $\parallel$ we derive for the joint process:

  $$proc(q_1) \parallel proc(q_2) \xrightarrow{S} proc(p_1) \parallel proc(q_2)$$

  We know further that ∂_B and Γ_C have no impact here, since only the port names from $N_1 \cap N_2$ are changed. Therefore we obtain as expected:

  $$\partial_B(\Gamma_C(proc(q_1) \parallel proc(q_2))) \xrightarrow{S} \partial_B(\Gamma_C(proc(p_1) \parallel proc(q_2)))$$

  (ii) There exists $q_2 \xrightarrow{S} p_2$ in PA_2 with $S \cap N_1 = \emptyset$. Symmetrical case.

(iii) There exist transitions $q_1 \xrightarrow{S_1} p_1$ in PA_1 and $q_2 \xrightarrow{S_2} p_2$ in PA_2 with $S = S_1 \cup S_2$, $S_1 \cap N_2 = S_2 \cap N_1$. Hence $P = proc(p_1, p_2)$. Analogously to case (i) we obtain:

$$proc(q_1) \parallel proc(q_2) \xrightarrow{S_1^R \cup S_2^R} proc(p_1) \parallel proc(p_2)$$

where S_i^R is the port name set obtained by applying the renaming R_i to S_i. By applying the communications Γ_C, the original port name set S for this transition is recovered. The blocking operator ∂_B has no impact here since $S_1 \cap N_2 = S_2 \cap N_1$. Therefore here we also get the expected step:

$$\partial_B(\Gamma_C(proc(q_1) \parallel proc(q_2))) \xrightarrow{S} \partial_B(\Gamma_C(proc(p_1) \parallel proc(p_2)))$$

- Now suppose $\partial_B(\Gamma_C(proc(q_1) \parallel proc(q_2))) \xrightarrow{S} P$. The semantics of $\parallel$ is given by:

$$\begin{aligned} proc(q_1) \parallel proc(q_2) = {} & proc(q_1) \between proc(q_2) + \\ & proc(q_2) \between proc(q_1) + \\ & proc(q_1) \mid proc(q_2) \end{aligned}$$

where $\between$ is the so-called *left merge* operator of the process algebra ACP. We therefore have to distinguish again three cases, from which the first two are symmetrical.

(i) There exist P_1 with $proc(q_1) \xrightarrow{S'} P_1$ and $P = \partial_B(\Gamma_C(P_1 \parallel proc(q_2)))$. Note that because of the renamings due to Γ_C we do not know whether $S = S'$. However, since $proc(q_1) \sim q_1$ this step must correspond to a transition

$$q_1 \xrightarrow{S_1^R} p_1 \text{ in } PA_1^R \quad \text{and} \quad q_1 \xrightarrow{S} p_1 \text{ in } PA_1$$

and thus we know $S' = S_1^R$. From the definition of the blocking set B we know that S cannot contain any shared port names, because otherwise $\partial_B(\Gamma_C(proc(q_1) \parallel proc(q_2)))$ could not have performed a step in the first place. Thus, $S = S_1^R$ and $S \cap N_2 = \emptyset$. Hence, the first case of the join operator in Definition 3.2 applies and we have:

$$\langle q_1, q_2 \rangle \xrightarrow{S} \langle p_1, q_2 \rangle \text{ in } PA_1 \bowtie PA_2$$

and because of the bisimilarity with $proc(q_1, q_2)$ also:

$$proc(q_1, q_2) \xrightarrow{S} proc(p_1, q_2)$$

(ii) There exists P_2 with $proc(q_2) \xrightarrow{S'} P_2$. Symmetrical case.

(iii) There exists P_1, P_2 with $proc(q_1) \xrightarrow{S_1'} P_1$ and $proc(q_2) \xrightarrow{S_2'} P_2$ and furthermore $P = \partial_B(\Gamma_C(P_1 \parallel P_2))$. We know again that S_1' and S_2' are renamed port name sets $S_i' = S_i^R$ and that they correspond to transitions:

$$q_1 \xrightarrow{S_1} p_1 \text{ in } PA_1 \quad \text{and} \quad q_2 \xrightarrow{S_2} p_2 \text{ in } PA_2$$

From the definition of C we know $S = S_1 \cup S_2$. From the definition of B we know $S_1 \cap N_2 = S_2 \cap N_1$. Hence, the third case of the join applies and:

$$proc(q_1, q_2) \xrightarrow{S} proc(p_1, p_2) \qquad \square$$

A.2 Proof for Theorem 6.9

PROOF. We consider the functor $G\colon \mathbf{C} \to \mathbf{Dis}(\mathbf{C})$ which maps an object $X \in \mathbf{C}$ to the distributed object $(1, (1 \mapsto X)) \in \mathbf{Dis}(\mathbf{C})$ and a morphism $f\colon X \to X'$ to $(id_1, (f))$, where 1 is the terminal object in **Graph**. We have $F \dashv G$ since there is a bijective correspondence

$$\Phi_{X,Y}\colon \hom_{\mathbf{C}}(FY, X) \to \hom_{\mathbf{Dis}(\mathbf{C})}(Y, GX)$$

that is natural in $X \in \mathbf{C}$ and $Y = (N, D) \in \mathbf{Dis}(\mathbf{C})$. The flattening functor F associates the colimit to a distributed object. Thus, FY is the colimit of the diagram D together with $\mathbf{C}$-morphisms $(y_n\colon D(n) \to FY)_{n \in N}$. Now, for a $\mathbf{C}$-morphism $h\colon FY \to X$ we have the $\mathbf{Dis}(\mathbf{C})$-morphism $\Phi_{X,Y}(h) = (!_N, (h \circ y_n)_{n \in N})\colon Y \to GX$ where $!_N\colon N \to 1$ is the terminal map for N in **Graph**. The mapping $\Phi_{X,Y}$ is bijective since all $\mathbf{Dis}(\mathbf{C})$-morphisms $Y \to GX$ are of the above form.

Now let $f\colon X \to X'$ be a $\mathbf{C}$-morphism and $g\colon Y' \to Y$ a $\mathbf{Dis}(\mathbf{C})$-morphism. The morphism $Gf\colon GX \to GX'$ is given as above. $Fg\colon FY' \to FY$ is the unique morphism into the colimit object FY. Now we need to show the following naturality condition:

$$Gf \circ \Phi_{X,Y}(h) \circ g \quad \stackrel{!}{=} \quad \Phi_{X',Y'}(f \circ h \circ Fg) \quad : \quad Y' \to GX'$$

We write $Y' = (N', D')$ and $g = (g_{N'}, (g_m)_{m \in N'})$. Moreover, let $(y'_m\colon D'(m) \to FY')_{m \in N'}$ be the $\mathbf{C}$-morphisms into the colimit of Y'. We now exploit the componentwise composition of $\mathbf{Dis}(\mathbf{C})$-morphisms:

$$\begin{aligned} Gf \circ \Phi_{X,Y}(h) \circ g &= \big(id_1, (f)\big) \circ \big(!_N, (h \circ y_n)_{n \in N}\big) \circ \big(g_{N'}, (g_m)_{m \in N'}\big) \\ &= \Big(id_1 \circ !_N \circ g_{N'}, (f \circ h \circ y_{g_{N'}(m)} \circ g_m)_{m \in N'}\Big) \\ &= \Big(!_{N'}, (f \circ h \circ y_{g_{N'}(m)} \circ g_m)_{m \in N'}\Big) \\ &= \Big(!_{N'}, (f \circ h \circ Fg \circ y'_m)_{m \in N'}\Big) \qquad \text{(A.1)} \\ &= \Phi_{X',Y'}(f \circ h \circ Fg) \end{aligned}$$

Equality (A.1) holds since Fg is the unique morphism into the colimit FY. $\square$

Appendix B

Listings

B.1 Vereofy library for context-dependent primitives

```
1  // Context-dependent Sync channel:
2  MODULE SYNC_CD {
3    in: A; in: nB;
4    out: B; out: nA;
5    -[ { A, B} & #A==#B ]->;
6    -[ {nA,nB} ]->;
7    -[ {   nB} ]->;
8  }
9
10 // Context-dependent LossySync channel:
11 MODULE LOSSY_SYNC_CD {
12   in: A; in: nB; out: B; out: nA;
13   -[ {A, B} & #A==#B ]->;
14   -[ {A,nB}          ]->;
15   -[ {  nB}          ]->;
16 }
17
18 // Context-dependent FIFO1 channel:
19 MODULE FIFO1_CD {
20   in: A; in: nB;
21   out: B; out: nA;
22   var: enum {EMPTY,FULL} state:=EMPTY;
23   var: Data value;
24   state==EMPTY -[ {nB}    ]-> state:=EMPTY;
25   state==EMPTY -[ {A}     ]-> state:=FULL & value:=#A;
26   state==EMPTY -[ {A,nB} ]-> state:=FULL & value:=#A;
27   state==FULL -[ {nA}     ]-> state:=FULL;
```

```
28    state==FULL −[ {nB}       ]−> state:=FULL;
29    state==FULL −[ {nA,nB}  ]−> state:=FULL;
30    state==FULL −[ {B,nA} & #B==value ]−> state:=EMPTY & value:=ANY;
31  }
32
33  // Context-dependent Reo node:
34  CIRCUIT NODE_CD <m,n> {
35
36    // Base node:
37    for (i=0; i<m; i=i+1) { new SYNC(source[i]; A); }
38    for (i=0; i<n; i=i+1) { new SYNC(A; sink[i]); }
39
40    // Context node:
41    for (i=0; i<n; i=i+1) { new SYNC(source[m+i]; nA); }
42    for (i=0; i<m; i=i+1) { new SYNC(nA; sink[n+i]); }
43
44    // Synchronizations:
45    for (i=0; i<m; i=i+1) {
46      for (j=0; j<n; j=j+1) {
47        if (i!=j) {
48          new SYNC(source[i]; sink[n+j]);
49        }
50      }
51    }
52  }
```

Appendix C

Summary

The dynamicity inherent in today's software is on the one hand a key feature which allows to build robust applications in dynamically changing, distributed environments. On the other hand, it introduces new major challenges in the design and formal analysis of such software. In this thesis, we therefore study formal methods for static as well as dynamically reconfigurable component-based software. Dynamic reconfiguration has broad applications, ranging from service-oriented software to embedded systems as used, e.g., in the automotive industry.

In this thesis, we propose to use the exogenous coordination language Reo for integrating and orchestrating distributed and potentially heterogeneous software components. Based on automata and process algebra models for the semantics of Reo, we show how the methodology of model checking can be applied to verify the behavior of component connectors modeled in Reo. Specifically, we use the mCRL2 specification language and model checking tool to analyze static component connectors. This approach enables us to validate some of the advanced modeling features of Reo, including data-, context- and time-dependent behavior.

As a formal approach for modeling structural change, i.e., reconfiguration of component connectors, we utilize the well-studied theory of algebraic graph transformation. We argue that the application of graph transformation in this field has major advantages over ad hoc reconfiguration approaches. Graph transformation can naturally express both structural and behavioral aspects of software systems. Moreover, the rule-based rewriting approach in graph transformation allows to model reconfiguration at a high level of abstraction and with an atomic execution semantics. The existence of formal analysis techniques and an extensive tool support were additional motivation for the use of graph transformation.

In this thesis, we show how to formally model and analyze dynamic reconfiguration scenarios in Reo using graph transformation. Using model checking and the so-called critical pair analysis known from rewrite systems, our reconfigurable connector models can be inspected for potentially harmful interplay of dynamic reconfiguration, on the one hand, and the execution itself, on the other. Additionally, we demonstrate how quantitative properties of reconfigurable connectors, e.g. steady state probabili-

ties, can be analyzed in our framework as well.

To remove the limitations of centralized coordination models, we extend our reconfiguration approach to distributed connectors. For this purpose, we use the theory of distributed graph transformation. As a new general result in the field of distributed graph transformation, we show that the flattening operation for distributed graphs is compositional. We argue that this result is relevant for transparent implementations of distributed connector models.

In the last chapter of this thesis, we show how to integrate an automata-based semantic model of Reo with our graph transformation based reconfiguration approach. We argue that this is a key ingredient to analyze problems in the domain of dynamic reconfiguration, such as state transfer and state consistency. For showing that the semantics is compositional in this approach, we reuse our earlier result on compositionality of the flattening operation for distributed graphs.

To highlight the practical relevance of the methods proposed in this thesis we provide an extensive tool support. The Eclipse Coordination Tools (ECT) which have been implemented in the context of this thesis provide an integrated environment for modeling and analysis of component connectors in Reo. For the formal analysis approaches considered in this thesis we use a number of verification tools (partially as back-ends of ECT), i.e., AGG, CADP, Henshin, mCRL2 and PRISM.

Summarizing, this thesis proposes formal models, verification techniques and tools to cope with the problems arising in the area of dynamically reconfigurable component connectors.

Appendix D

Samenvatting (dutch)

Het dynamische karakter dat inherent is aan hedendaagse software is een essentiële eigenschap die het mogelijk maakt om robuuste applicaties te schrijven voor dynamische, gedistribueerde omgevingen die onderhevig zijn aan verandering. Maar ook stelt het ons voor nieuwe uitdagingen in het ontwerp en de formele analyse van zulke software. In dit proefschrift bestuderen we daarom formele methoden voor zowel statisch als dynamisch herconfigureerbare component-based software. Dynamische herconfiguratie is breed toepasbaar: van service-oriented software tot embedded systems.

In dit proefschrift stellen we voor om de exogene coördinatie taal Reo te gebruiken voor de integratie en orkestratie van gedistribueerde en potentieel heterogene software componenten. Middels automaten en procesalgebraïsche modellen voor de semantiek van Reo laten we zien hoe de model checking methodologie kan worden toegepast om het gedrag van in Reo gemodelleerde component connectors te verifiëren. In het bijzonder gebruiken we de mCRL2 specificatietaal en model checking tool voor de analyse van component connectors. Deze benadering maakt het mogelijk om enkele geavanceerde modelleer aspecten van Reo te valideren, waaronder data-, context-, en tijd-afhankelijk gedrag.

We gebruiken de breed bestudeerde theorie der algebraïsche graaftransformaties als een formele benadering om de veranderingen in de structuur, i.e., de herconfiguratie, van component connectors te modelleren. We beargumenteren dat het toepassen van graaftransformaties op dit vlak zeer grote voordelen biedt ten opzichte van een adhoc aanpak van herconfiguratie. Bovendien maken graaftransformaties die gebaseerd zijn op rule-based rewriting het mogelijk om herconfiguraties te definiëren op een hoog abstractieniveau en met een atomaire executie semantiek. Het bestaan van technieken ten behoeve van formele analyses en de uitgebreide tool ondersteuning vormen een aanvullende motivatie voor het gebruik van graaftransformaties.

In dit proefschrift laten we zien hoe, middels graaftransformaties, scenario's voor dynamische herconfiguratie kunnen worden gemodelleerd en geanalyseerd in Reo. Middels model checking en zogenaamde crictical pair analyse, bekend van rewrite systems, kunnen herconfigureerbare connector modellen geïnspecteerd worden op

een potentieel schadelijke wisselwerking tussen dynamische herconfiguratie enerzijds en de executie zelf anderzijds. Daarnaast tonen we aan hoe ook kwantitatieve eigenschappen van herconfigureerbare connectors, zoals steady state probabilities, geanalyseerd kunnen worden in ons framework.

Om de beperkingen van gecentraliseerde coördinatiemodellen te verwijderen breiden we de manier waarop we herconfiguratie benaderen uit naar gedistribueerde connectors. We gebruiken hier de theorie van gedistribueerde graaftransformaties voor en tonen aan dat de flattening operatie voor gedistribueerde grafen compositioneel is. Dit is een nieuw algemeen resultaat op het gebied van gedistribueerde graaftransformaties. We beargumenteren dat dit resultaat relevant is voor de transparante implementatie van gedistribueerde connectormodellen.

In het laatste hoofdstuk van dit proefschrift laten we zien hoe het op automaten gebaseerde semantische model van Reo geïntegreerd kan worden in onze, op graaftransformaties gebaseerde, aanpak van herconfiguratie. We beargumenteren dat dit een essentieel onderdeel is voor de analyse van problemen in het domein van dynamische herconfiguratie, zoals state transfer en state consistency. We hergebruiken ons eerdere resultaat omtrent de compositionaliteit van de flattening operatie op gedistribueerde grafen om te laten zien dat de semantiek van deze aanpak compositioneel is.

Om aan te tonen dat de methoden die we in dit proefschrift voorstellen ook relevant zijn in de praktijk bieden we uitgebreide tool ondersteuning aan. De Eclipse Coordination Tools (ECT), die geïmplementeerd zijn in de context van dit proefschrift, verschaffen een geïntegreerde omgeving voor het modelleren en analyseren van component connectors in Reo. We gebruiken (gedeeltelijk als back-ends voor ECT) een aantal verificatieprogramma's, namelijk AGG, CADP, Henshin, mCRL2, en PRISM, om de verschillende manieren waarop we formele analyse benaderen in dit proefschrift te ondersteunen.

Samenvattend stelt dit proefschrift voor om formele modellen, verificatietechnieken en tools te gebruiken om het hoofd te bieden aan de problemen die ontstaan op het gebied van dynamisch herconfigureerbare component connectors.

Titles in the IPA Dissertation Series since 2005

E. Ábrahám. *An Assertional Proof System for Multithreaded Java -Theory and Tool Support-* . Faculty of Mathematics and Natural Sciences, UL. 2005-01

R. Ruimerman. *Modeling and Remodeling in Bone Tissue*. Faculty of Biomedical Engineering, TU/e. 2005-02

C.N. Chong. *Experiments in Rights Control - Expression and Enforcement*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-03

H. Gao. *Design and Verification of Lock-free Parallel Algorithms*. Faculty of Mathematics and Computing Sciences, RUG. 2005-04

H.M.A. van Beek. *Specification and Analysis of Internet Applications*. Faculty of Mathematics and Computer Science, TU/e. 2005-05

M.T. Ionita. *Scenario-Based System Architecting - A Systematic Approach to Developing Future-Proof System Architectures*. Faculty of Mathematics and Computing Sciences, TU/e. 2005-06

G. Lenzini. *Integration of Analysis Techniques in Security and Fault-Tolerance*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-07

I. Kurtev. *Adaptability of Model Transformations*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-08

T. Wolle. *Computational Aspects of Treewidth - Lower Bounds and Network Reliability*. Faculty of Science, UU. 2005-09

O. Tveretina. *Decision Procedures for Equality Logic with Uninterpreted Functions*. Faculty of Mathematics and Computer Science, TU/e. 2005-10

A.M.L. Liekens. *Evolution of Finite Populations in Dynamic Environments*. Faculty of Biomedical Engineering, TU/e. 2005-11

J. Eggermont. *Data Mining using Genetic Programming: Classification and Symbolic Regression*. Faculty of Mathematics and Natural Sciences, UL. 2005-12

B.J. Heeren. *Top Quality Type Error Messages*. Faculty of Science, UU. 2005-13

G.F. Frehse. *Compositional Verification of Hybrid Systems using Simulation Relations*. Faculty of Science, Mathematics and Computer Science, RU. 2005-14

M.R. Mousavi. *Structuring Structural Operational Semantics*. Faculty of Mathematics and Computer Science, TU/e. 2005-15

A. Sokolova. *Coalgebraic Analysis of Probabilistic Systems*. Faculty of Mathematics and Computer Science, TU/e. 2005-16

T. Gelsema. *Effective Models for the Structure of pi-Calculus Processes with Replication*. Faculty of Mathematics and Natural Sciences, UL. 2005-17

P. Zoeteweij. *Composing Constraint Solvers*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2005-18

J.J. Vinju. *Analysis and Transformation of Source Code by Parsing and Rewriting*.

Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2005-19

M.Valero Espada. *Modal Abstraction and Replication of Processes with Data*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2005-20

A. Dijkstra. *Stepping through Haskell*. Faculty of Science, UU. 2005-21

Y.W. Law. *Key management and link-layer security of wireless sensor networks: energy-efficient attack and defense*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-22

E. Dolstra. *The Purely Functional Software Deployment Model*. Faculty of Science, UU. 2006-01

R.J. Corin. *Analysis Models for Security Protocols*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-02

P.R.A. Verbaan. *The Computational Complexity of Evolving Systems*. Faculty of Science, UU. 2006-03

K.L. Man and R.R.H. Schiffelers. *Formal Specification and Analysis of Hybrid Systems*. Faculty of Mathematics and Computer Science and Faculty of Mechanical Engineering, TU/e. 2006-04

M. Kyas. *Verifying OCL Specifications of UML Models: Tool Support and Compositionality*. Faculty of Mathematics and Natural Sciences, UL. 2006-05

M. Hendriks. *Model Checking Timed Automata - Techniques and Applications*. Faculty of Science, Mathematics and Computer Science, RU. 2006-06

J. Ketema. *Böhm-Like Trees for Rewriting*. Faculty of Sciences, VUA. 2006-07

C.-B. Breunesse. *On JML: topics in tool-assisted verification of JML programs*. Faculty of Science, Mathematics and Computer Science, RU. 2006-08

B. Markvoort. *Towards Hybrid Molecular Simulations*. Faculty of Biomedical Engineering, TU/e. 2006-09

S.G.R. Nijssen. *Mining Structured Data*. Faculty of Mathematics and Natural Sciences, UL. 2006-10

G. Russello. *Separation and Adaptation of Concerns in a Shared Data Space*. Faculty of Mathematics and Computer Science, TU/e. 2006-11

L. Cheung. *Reconciling Nondeterministic and Probabilistic Choices*. Faculty of Science, Mathematics and Computer Science, RU. 2006-12

B. Badban. *Verification techniques for Extensions of Equality Logic*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2006-13

A.J. Mooij. *Constructive formal methods and protocol standardization*. Faculty of Mathematics and Computer Science, TU/e. 2006-14

T. Krilavicius. *Hybrid Techniques for Hybrid Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-15

M.E. Warnier. *Language Based Security for Java and JML*. Faculty of Science, Mathematics and Computer Science, RU. 2006-16

V. Sundramoorthy. *At Home In Service Discovery*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-17

B. Gebremichael. *Expressivity of Timed Automata Models*. Faculty of Science, Mathematics and Computer Science, RU. 2006-18

L.C.M. van Gool. *Formalising Interface Specifications*. Faculty of Mathematics and Computer Science, TU/e. 2006-19

C.J.F. Cremers. *Scyther - Semantics and Verification of Security Protocols*. Faculty of Mathematics and Computer Science, TU/e. 2006-20

J.V. Guillen Scholten. *Mobile Channels for Exogenous Coordination of Distributed Systems: Semantics, Implementation and Composition*. Faculty of Mathematics and Natural Sciences, UL. 2006-21

H.A. de Jong. *Flexible Heterogeneous Software Systems*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2007-01

N.K. Kavaldjiev. *A run-time reconfigurable Network-on-Chip for streaming DSP applications*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-02

M. van Veelen. *Considerations on Modeling for Early Detection of Abnormalities in Locally Autonomous Distributed Systems*. Faculty of Mathematics and Computing Sciences, RUG. 2007-03

T.D. Vu. *Semantics and Applications of Process and Program Algebra*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2007-04

L. Brandán Briones. *Theories for Model-based Testing: Real-time and Coverage*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-05

I. Loeb. *Natural Deduction: Sharing by Presentation*. Faculty of Science, Mathematics and Computer Science, RU. 2007-06

M.W.A. Streppel. *Multifunctional Geometric Data Structures*. Faculty of Mathematics and Computer Science, TU/e. 2007-07

N. Trčka. *Silent Steps in Transition Systems and Markov Chains*. Faculty of Mathematics and Computer Science, TU/e. 2007-08

R. Brinkman. *Searching in encrypted data*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-09

A. van Weelden. *Putting types to good use*. Faculty of Science, Mathematics and Computer Science, RU. 2007-10

J.A.R. Noppen. *Imperfect Information in Software Development Processes*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-11

R. Boumen. *Integration and Test plans for Complex Manufacturing Systems*. Faculty of Mechanical Engineering, TU/e. 2007-12

A.J. Wijs. *What to do Next?: Analysing and Optimising System Behaviour in Time*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2007-13

C.F.J. Lange. *Assessing and Improving the Quality of Modeling: A Series of Empirical Studies about the UML*. Faculty of Mathematics and Computer Science, TU/e. 2007-14

T. van der Storm. *Component-based Configuration, Integration and Delivery*.

Faculty of Natural Sciences, Mathematics, and Computer Science,UvA. 2007-15

B.S. Graaf. *Model-Driven Evolution of Software Architectures*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2007-16

A.H.J. Mathijssen. *Logical Calculi for Reasoning with Binding*. Faculty of Mathematics and Computer Science, TU/e. 2007-17

D. Jarnikov. *QoS framework for Video Streaming in Home Networks*. Faculty of Mathematics and Computer Science, TU/e. 2007-18

M. A. Abam. *New Data Structures and Algorithms for Mobile Data*. Faculty of Mathematics and Computer Science, TU/e. 2007-19

W. Pieters. *La Volonté Machinale: Understanding the Electronic Voting Controversy*. Faculty of Science, Mathematics and Computer Science, RU. 2008-01

A.L. de Groot. *Practical Automaton Proofs in PVS*. Faculty of Science, Mathematics and Computer Science, RU. 2008-02

M. Bruntink. *Renovation of Idiomatic Crosscutting Concerns in Embedded Systems*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2008-03

A.M. Marin. *An Integrated System to Manage Crosscutting Concerns in Source Code*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2008-04

N.C.W.M. Braspenning. *Model-based Integration and Testing of High-tech Multidisciplinary Systems*. Faculty of Mechanical Engineering, TU/e. 2008-05

M. Bravenboer. *Exercises in Free Syntax: Syntax Definition, Parsing, and Assimilation of Language Conglomerates*. Faculty of Science, UU. 2008-06

M. Torabi Dashti. *Keeping Fairness Alive: Design and Formal Verification of Optimistic Fair Exchange Protocols*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2008-07

I.S.M. de Jong. *Integration and Test Strategies for Complex Manufacturing Machines*. Faculty of Mechanical Engineering, TU/e. 2008-08

I. Hasuo. *Tracing Anonymity with Coalgebras*. Faculty of Science, Mathematics and Computer Science, RU. 2008-09

L.G.W.A. Cleophas. *Tree Algorithms: Two Taxonomies and a Toolkit*. Faculty of Mathematics and Computer Science, TU/e. 2008-10

I.S. Zapreev. *Model Checking Markov Chains: Techniques and Tools*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-11

M. Farshi. *A Theoretical and Experimental Study of Geometric Networks*. Faculty of Mathematics and Computer Science, TU/e. 2008-12

G. Gulesir. *Evolvable Behavior Specifications Using Context-Sensitive Wildcards*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-13

F.D. Garcia. *Formal and Computational Cryptography: Protocols, Hashes and Commitments*. Faculty of Science, Mathematics and Computer Science, RU. 2008-14

P. E. A. Dürr. *Resource-based Verification for Robust Composition of Aspects*.

Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-15

E.M. Bortnik. *Formal Methods in Support of SMC Design*. Faculty of Mechanical Engineering, TU/e. 2008-16

R.H. Mak. *Design and Performance Analysis of Data-Independent Stream Processing Systems*. Faculty of Mathematics and Computer Science, TU/e. 2008-17

M. van der Horst. *Scalable Block Processing Algorithms*. Faculty of Mathematics and Computer Science, TU/e. 2008-18

C.M. Gray. *Algorithms for Fat Objects: Decompositions and Applications*. Faculty of Mathematics and Computer Science, TU/e. 2008-19

J.R. Calamé. *Testing Reactive Systems with Data - Enumerative Methods and Constraint Solving*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-20

E. Mumford. *Drawing Graphs for Cartographic Applications*. Faculty of Mathematics and Computer Science, TU/e. 2008-21

E.H. de Graaf. *Mining Semi-structured Data, Theoretical and Experimental Aspects of Pattern Evaluation*. Faculty of Mathematics and Natural Sciences, UL. 2008-22

R. Brijder. *Models of Natural Computation: Gene Assembly and Membrane Systems*. Faculty of Mathematics and Natural Sciences, UL. 2008-23

A. Koprowski. *Termination of Rewriting and Its Certification*. Faculty of Mathematics and Computer Science, TU/e. 2008-24

U. Khadim. *Process Algebras for Hybrid Systems: Comparison and Development*. Faculty of Mathematics and Computer Science, TU/e. 2008-25

J. Markovski. *Real and Stochastic Time in Process Algebras for Performance Evaluation*. Faculty of Mathematics and Computer Science, TU/e. 2008-26

H. Kastenberg. *Graph-Based Software Specification and Verification*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-27

I.R. Buhan. *Cryptographic Keys from Noisy Data Theory and Applications*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-28

R.S. Marin-Perianu. *Wireless Sensor Networks in Motion: Clustering Algorithms for Service Discovery and Provisioning*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-29

M.H.G. Verhoef. *Modeling and Validating Distributed Embedded Real-Time Control Systems*. Faculty of Science, Mathematics and Computer Science, RU. 2009-01

M. de Mol. *Reasoning about Functional Programs: Sparkle, a proof assistant for Clean*. Faculty of Science, Mathematics and Computer Science, RU. 2009-02

M. Lormans. *Managing Requirements Evolution*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2009-03

M.P.W.J. van Osch. *Automated Model-based Testing of Hybrid Systems*. Faculty of Mathematics and Computer Science, TU/e. 2009-04

H. Sozer. *Architecting Fault-Tolerant Software Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-05

M.J. van Weerdenburg. *Efficient Rewriting Techniques*. Faculty of Mathematics and Computer Science, TU/e. 2009-06

H.H. Hansen. *Coalgebraic Modelling: Applications in Automata Theory and Modal Logic*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2009-07

A. Mesbah. *Analysis and Testing of Ajax-based Single-page Web Applications*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2009-08

A.L. Rodriguez Yakushev. *Towards Getting Generic Programming Ready for Prime Time*. Faculty of Science, UU. 2009-9

K.R. Olmos Joffré. *Strategies for Context Sensitive Program Transformation*. Faculty of Science, UU. 2009-10

J.A.G.M. van den Berg. *Reasoning about Java programs in PVS using JML*. Faculty of Science, Mathematics and Computer Science, RU. 2009-11

M.G. Khatib. *MEMS-Based Storage Devices. Integration in Energy-Constrained Mobile Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-12

S.G.M. Cornelissen. *Evaluating Dynamic Analysis Techniques for Program Comprehension*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2009-13

D. Bolzoni. *Revisiting Anomaly-based Network Intrusion Detection Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-14

H.L. Jonker. *Security Matters: Privacy in Voting and Fairness in Digital Exchange*. Faculty of Mathematics and Computer Science, TU/e. 2009-15

M.R. Czenko. *TuLiP - Reshaping Trust Management*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-16

T. Chen. *Clocks, Dice and Processes*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2009-17

C. Kaliszyk. *Correctness and Availability: Building Computer Algebra on top of Proof Assistants and making Proof Assistants available over the Web*. Faculty of Science, Mathematics and Computer Science, RU. 2009-18

R.S.S. O'Connor. *Incompleteness & Completeness: Formalizing Logic and Analysis in Type Theory*. Faculty of Science, Mathematics and Computer Science, RU. 2009-19

B. Ploeger. *Improved Verification Methods for Concurrent Systems*. Faculty of Mathematics and Computer Science, TU/e. 2009-20

T. Han. *Diagnosis, Synthesis and Analysis of Probabilistic Models*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-21

R. Li. *Mixed-Integer Evolution Strategies for Parameter Optimization and Their Applications to Medical Image Analysis*. Faculty of Mathematics and Natural Sciences, UL. 2009-22

J.H.P. Kwisthout. *The Computational Complexity of Probabilistic Networks*. Faculty of Science, UU. 2009-23

T.K. Cocx. *Algorithmic Tools for Data-Oriented Law Enforcement*. Faculty of Mathematics and Natural Sciences, UL. 2009-24

A.I. Baars. *Embedded Compilers*. Faculty of Science, UU. 2009-25

M.A.C. Dekker. *Flexible Access Control for Dynamic Collaborative Environments*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-26

J.F.J. Laros. *Metrics and Visualisation for Crime Analysis and Genomics*. Faculty of Mathematics and Natural Sciences, UL. 2009-27

C.J. Boogerd. *Focusing Automatic Code Inspections*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2010-01

M.R. Neuhäußer. *Model Checking Nondeterministic and Randomly Timed Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2010-02

J. Endrullis. *Termination and Productivity*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2010-03

T. Staijen. *Graph-Based Specification and Verification for Aspect-Oriented Languages*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2010-04

Y. Wang. *Epistemic Modelling and Protocol Dynamics*. Faculty of Science, UvA. 2010-05

J.K. Berendsen. *Abstraction, Prices and Probability in Model Checking Timed Automata*. Faculty of Science, Mathematics and Computer Science, RU. 2010-06

A. Nugroho. *The Effects of UML Modeling on the Quality of Software*. Faculty of Mathematics and Natural Sciences, UL. 2010-07

A. Silva. *Kleene Coalgebra*. Faculty of Science, Mathematics and Computer Science, RU. 2010-08

J.S. de Bruin. *Service-Oriented Discovery of Knowledge - Foundations, Implementations and Applications*. Faculty of Mathematics and Natural Sciences, UL. 2010-09

D. Costa. *Formal Models for Component Connectors*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2010-10

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

A. Morali. *IT Architecture-Based Confidentiality Risk Assessment in Networks of Organizations*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-06

M. van der Bijl. *On changing models in Model-Based Testing*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-07

C. Krause. *Reconfigurable Component Connectors*. Faculty of Mathematics and Natural Sciences, UL. 2011-08