Centrum voor Wiskunde en Informatica

REPORTRAPPORT

SEN

Software Engineering

Software ENgineering

Design and implementation of an editor and simulators for constraint automata in the context of Reo

Hok Kwan Kan

REPORT SEN-E0512 NOVEMBER 2005

CWI is the National Research Institute for Mathematics and Computer Science. It is sponsored by the Netherlands Organization for Scientific Research (NWO).
CWI is a founding member of ERCIM, the European Research Consortium for Informatics and Mathematics.

CWI's research has a theme-oriented structure and is grouped into four clusters. Listed below are the names of the clusters and in parentheses their acronyms.

Probability, Networks and Algorithms (PNA)

Software Engineering (SEN)

Modelling, Analysis and Simulation (MAS)

Information Systems (INS)

Copyright © 2005, Stichting Centrum voor Wiskunde en Informatica
P.O. Box 94079, 1090 GB Amsterdam (NL)
Kruislaan 413, 1098 SJ Amsterdam (NL)
Telephone +31 20 592 9333
Telefax +31 20 592 4199

ISSN 1386-369X

Design and implementation of an editor and simulators for constraint automata in the context of Reo

ABSTRACT

The coordination language Reo offers a flexible framework for compositionally constructing software systems out of components through connectors. These connectors not only connect components with each other, but also exogenously coordinate the interactions between them. The connectors themselves are compositionally built out of simpler connectors, where the simplest connectors are a set of user-defined channels with well-defined behavior. Formal semantics can be given to Reo using constraint automata and timed data streams. Constraint automata can be seen as an extension of the finite automata, where the language accepted by an automaton is specified by using timed data streams. In this thesis we describe the design and implementation of a tool for constraint automata consisting of an editor and three simulators. The constraint automata editor allows users to visually construct and modify constraint automata. The first simulator for constraint automata acts as a language acceptor of timed data streams. The second one simulates a constraint automaton as a Reo connector where the input is defined as timed data streams. The third simulator simulates a constraint automaton as a Reo connector where components are attached to. These components deliver the input at real-time instead of using predefined timed data streams.

1998 ACM Computing Classification System: D.1.7,D.2.2,D.2.6,D.2.11,D.3.4,I.6.8

Keywords and Phrases: constraint automata, simulators, visual programming, modular design, Reo

Note: The author carried out this work at CWI as an assignment for a Masters degree in Computer Science at the Delft University of Technology

Design and Implementation of an Editor and Simulators for Constraint Automata in the Context of Reo

Hok Kwan Kan

Design and Implementation of an Editor and Simulators for Constraint Automata in the Context of Reo

Master's Thesis in Computer Science

Software Engineering group
Faculty of Electrical Engineering, Mathematics, and Computer Science
Delft University of Technology

Hok Kwan

18th October 2005

Author
Hok Kwan Kan

Title
Design and Implementation of an Editor and Simulators for Constraint Automata in the Context of Reo

MSc presentation
31st October 2005

Graduation Committee

Prof. Dr. A. van Deursen (chair)	Delft University of Technology
Prof. Dr. F. Arbab	Centrum voor Wiskunde en Informatica
Dr. N.K. Diakov	Centrum voor Wiskunde en Informatica
Ir. C. Pronk	Delft University of Technology

Abstract

The coordination language Reo offers a flexible framework for compositionally constructing software systems out of components through connectors. These connectors not only connect components with each other, but also exogenously coordinate the interactions between them. The connectors themselves are compositionally built out of simpler connectors, where the simplest connectors are a set of user-defined channels with well-defined behavior.
Formal semantics can be given to Reo using constraint automata and timed data streams. Constraint automata can be seen as an extension of the finite automata, where the language accepted by an automaton is specified by using timed data streams.
In this thesis we describe the design and implementation of a tool for constraint automata consisting of an editor and three simulators. The constraint automata editor allows users to visually construct and modify constraint automata. The first simulator for constraint automata acts as a language acceptor of timed data streams. The second one simulates a constraint automaton as a Reo connector where the input is defined as timed data streams. The third simulator simulates a constraint automaton as a Reo connector where components are attached to. These components deliver the input at real-time instead of using predefined timed data streams.

Acknowledgments

This thesis reports on the results of the MSc project I carried out at the Software Engineering group of the Computer Science department at Delft University of Technology. The work was conducted within the SEN3 Group at CWI in Amsterdam. CWI stands for Centrum voor Wiskunde en Informatica, which is the National Research Institute for Mathematics and Computer Science in the Netherlands. The SEN3 Group at CWI performs research in the area of coordination languages.

I learned about the coordination language Reo from a guest lecture delivered by Prof. Dr. F. Arbab at the TU Delft. Reo immediately caught my interest, because of its simplicity and expressiveness. When offered the opportunity to do a project at CWI on constraint automata, which are related to Reo, I took it after little consideration.

Before continuing with my MSc thesis, I would like to use this opportunity to thank a few people. I would like to thank my daily supervisor at the CWI Dr. N.K. Diakov for his support and answering my questions throughout this MSc project. I would like to thank my supervisor at the TU Delft Ir. C. Pronk for his patience and constructive comments on my thesis. My thanks also go to Prof. Dr. F. Arbab for this opportunity to work on this subject and to the people of the SEN3 group who made my stay at the CWI an enjoyable and interesting time. Next, I would like to thank my friends who gave me the needed distractions at set times. Last but certainly not least, I would like to thank my family for their support during my years at the university.

Hok Kwan Kan

Delft, The Netherlands
18th October 2005

Contents

Preface v

1 Introduction 1
- 1.1 Background 1
- 1.2 Objective and Motivation 2
- 1.3 Related work 2
- 1.4 Approach 4
- 1.5 Thesis Outline 5

2 Reo 7
- 2.1 Basic Concepts 7
- 2.2 Channels 8
  - 2.2.1 Sync 8
  - 2.2.2 SyncDrain 9
  - 2.2.3 SyncSpout 9
  - 2.2.4 FIFO and FIFO*n* 9
  - 2.2.5 LossySync 10
- 2.3 Join 10
  - 2.3.1 Source node: replicate 10
  - 2.3.2 Sink node: merge 11
  - 2.3.3 Mixed node: replicate + merge 11
- 2.4 Hide 11
- 2.5 Examples 12
  - 2.5.1 Asynchronous Drain 12
  - 2.5.2 Regulated Reads 13
  - 2.5.3 Barrier Synchronization 13
  - 2.5.4 Exclusive and Inclusive Router 14

3 Constraint Automata 17
- 3.1 Finite Automata 17
- 3.2 TDS-language 18
  - 3.2.1 Stream 18
  - 3.2.2 Data Stream 19

3.2.3 Time Stream 19
3.2.4 Timed Data Stream 19
3.3 Defining Constraint Automata 20
3.3.1 Ports 20
3.3.2 Transitions 20
3.3.3 Name-data-assignments 20
3.3.4 Data Constraints 20
3.3.5 Definition of Constraint Automata 21
3.3.6 Deterministic vs Non-deterministic 22
3.4 TDS-language and Constraint Automata 22

4 Modeling Reo by Constraint Automata 25
4.1 Channels 25
4.1.1 Sync 25
4.1.2 SyncDrain and SyncSpout 26
4.1.3 FIFO1 26
4.1.4 LossySync 27
4.2 Join 27
4.2.1 Product-construction 28
4.2.2 Merger 30
4.3 Hide 30
4.4 Parameterized Constraint Automata 31

5 Requirements and Analysis 35
5.1 MSc Assignment 35
5.2 Existing Reo and Constraint Automata Tools 36
5.3 Operating System and Programming Language 36
5.4 Global overview 37
5.4.1 Integrated Tool GUI (IT-GUI) 37
5.4.2 Constraint Automaton Model (CAM) 40
5.4.3 CA-Editor 41
5.4.4 Layout Engine 43
5.4.5 Load/Save Constraint Automaton Model (LSCAM) 44
5.4.6 TDS-Language Acceptor Simulator (TDSLAS) 46
5.4.7 Reo Connector Simulator with TDS (RCSwTDS) 47
5.4.8 Timed Data Stream Model (TDSM) 50
5.4.9 Load/Save Timed Data Stream (LSTDS) 50
5.4.10 Reo Connector Simulator with Components (RCSwC) . . 50
5.4.11 Component 53
5.4.12 Load Component 53
5.4.13 Constraint Automaton Engine (CA-Engine) 54
5.5 Requirements for the IT-GUI 55
5.6 Requirements for the CA-Editor 55
5.7 Requirements for the TDSLAS 56

5.8 Requirements for the RCSwTDS . . . 56
5.9 Requirements for the RCSwC . . . 56
5.10 Requirements for the CA-Engine . . . 57

6 Design and Implementation **59**
6.1 Programming Language . . . 59
6.2 Architectural Overview . . . 59
6.3 CA-Editor . . . 60
6.3.1 CAM . . . 61
6.3.2 LSCAM . . . 66
6.3.3 GUI CA-Editor and CA-Editor Control . . . 67
6.3.4 Input Parsers . . . 70
6.3.5 Layout Engine . . . 71
6.4 IT-GUI . . . 72
6.5 CA-Engine . . . 73
6.6 TDSLAS . . . 75
6.6.1 Model-View-Controller Design Pattern . . . 76
6.6.2 TDSM and LSTDS . . . 76
6.6.3 GUI TDS-Editor . . . 77
6.6.4 TDS-Editor Control . . . 80
6.6.5 GUI TDSLAS . . . 80
6.6.6 TDSLAS Control . . . 82
6.6.7 Simulation Coloring . . . 84
6.7 RCSwTDS . . . 84
6.7.1 GUI RCSwTDS . . . 85
6.7.2 RCSwTDS Control . . . 87
6.7.3 Simulation Coloring . . . 88
6.8 RCSwC . . . 88
6.8.1 GUI RCSwC . . . 88
6.8.2 Load Component and Component . . . 89
6.8.3 Python Component . . . 90
6.8.4 RCSwC Control . . . 91
6.8.5 Simulation Coloring . . . 94

7 Conclusions **97**
7.1 Contributions . . . 97
7.2 Summary . . . 98
7.3 Future Work . . . 98
7.4 Personal Experience . . . 99

A User's Manual **103**
A.1 Installation . . . 103
A.1.1 System Requirements . . . 103
A.1.2 Installing and Running . . . 103

A.2 Constraint Automata Editor 104
A.2.1 Add State 106
A.2.2 Modify State Properties 106
A.2.3 Add Transition 107
A.2.4 Modify Transition Properties 108
A.2.5 Modify Constraint Automaton Properties 108
A.2.6 Perform Layout 109
A.2.7 Exercise 112
A.3 Timed Data Stream Language Acceptor Simulator 112
A.3.1 Timed Data Stream Editor 114
A.3.2 Simulator 114
A.3.3 Exercise 114
A.4 Reo Connector Simulator with Timed Data Streams 115
A.4.1 Simulator 115
A.4.2 Exercise 115
A.5 Reo Connector Simulator with Components 116
A.5.1 Implementing Components 118
A.5.2 Implementing Python Components 119
A.5.3 Exercise 119

B Developer's Manual **121**

Chapter 1

Introduction

In the first section of this chapter we give a short introduction to the background of this thesis, the coordination language Reo and constraint automata, followed by a description of the MSc project assignment and its motivation. Next, the work of research related to ours is discussed. The following section describes our approach to the MSc project. The chapter concludes with an overview of the remainder of this thesis.

1.1 Background

The current methodology in software development is still based on a modular design, where an application consists of modules. These modules offer functionalities via well-defined interfaces, which the modules use to refer directly to each other. These direct references result in tight dependencies between the modules, which prevent replacing a module without too much additional work or the reuse of modules.
This in contrast to methodologies used in other, mature engineering disciplines, such as electrical engineering, where it is common to build new applications out of reusable components, which are easily replaceable.
The potential of such an approach is also recognized by the software development community. This is the reason for the upcoming interest in software components in the last decades. On the contrary to the tightly coupled approach discussed above, it is expected from software components that they are independent from each other and the application environment where they are deployed. A well-known definition of software components is given by Szyperski [9]:

> "Software components are binary units of independent production, acquisition, and deployment that interact to form a functioning system."

Reo[1] is a channel-based coordination language, which provides connectors for connecting components. The simplest connector in Reo is a user-defined channel with well-defined behavior. More complex connectors are compositionally built

out of the simpler ones. The behavior of every connector imposes a specific coordination pattern on the components that perform I/O operations through that connector in such way that a component is unaware that it is being coordinated together with other components. Thus, Reo enables components to interact with each other, while keeping their independency of each other.
It is shown that constraint automata[3] can be used as an operational model for Reo connectors. The states of a constraint automaton represent the possible configurations of a Reo connector (e.g. the content of a FIFO-channel), while the transitions going out of a state express the possible data flows and its effect on the configuration.

1.2 Objective and Motivation

The assignment of this MSc project is the design and implementation of a tool that serves as an editor and simulator for constraint automata. This tool is useful for the research on constraint automata, but also for the development of the coordination language Reo itself, since constraint automata can act as an operational semantic model for Reo.
A visual editor for constraint automata has certain advantages for the research on constraint automata. It reduces, for example, the amount of work, because with an editor it is possible to create constraint automata on the computer instead of drawing them by hand on paper or the whiteboard. Visualizing a large constraint automaton with many states is difficult, because one loses the overview easily. Since in the editor the visual layout of a constraint automaton is not fixed, the user can rearrange the layout as he prefers. Therefore, dealing with large constraint automata becomes simpler, especially in combination with an automatic layout generator. Another useful feature of having an editor is the ability to save and load constraint automata to and from some persistent data storage, allowing the user to continue previous work.
The behavior of a constraint automaton can be simulated by hand. However, this approach can be quite time consuming, especially for large constraint automata. Using a simulator saves a lot of time. Simulation by hand is also more prone to error than doing this with a simulator. Hence, a simulator facilitates the research on the behavior of complex constraint automata.
A simulator can serve as a tool for teaching and demonstration purposes. Figure 1.1 illustrates the position of the work of this MSc project within the overall tool framework of Reo and constraint automata (labeled “CA simulator”).

1.3 Related work

Since the objective is to create a tool for constraint automata, which can be used as an operational model for Reo connectors, the work of this thesis relates to automata-based modeling theory and automata-based modeling tools.

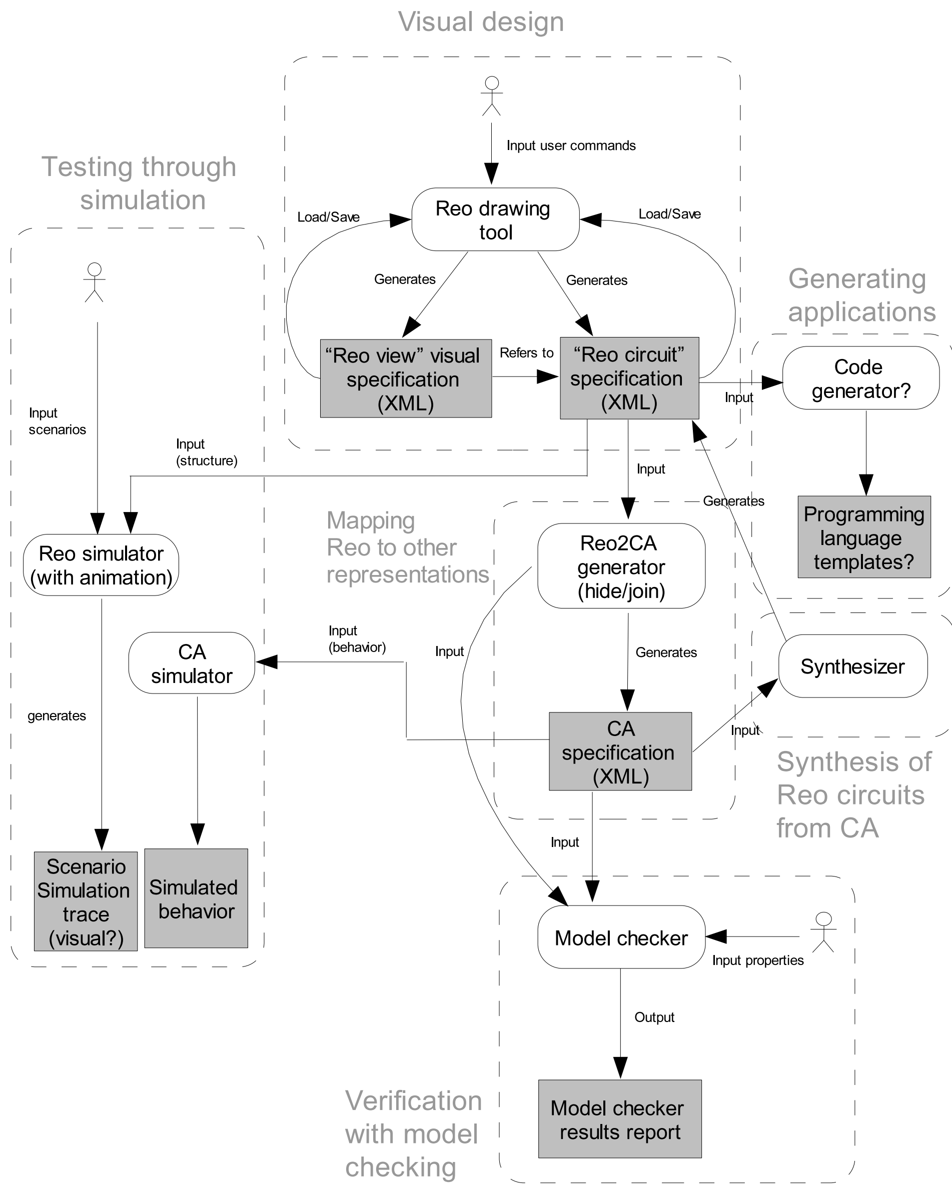

Figure 1.1: The overall tool frame work of Reo and constraint automata.

Well-known automata-based modeling theories are I/O-automata, timed port automata and interface automata. The major differences and similarities are summarized as follows[3]:

- Transitions in I/O-automata are labeled with action names, while transitions in timed port automata and constraint automata are data-dependent. On the contrary to timed port automata where transitions are labelled with specific data values, constraint automata transitions are labelled with data constraints (boolean expressions for the data values).
- I/O-automata and timed port automata follow a strictly time-synchronous approach, while constraint automata do not. This difference becomes important when constraint automata are composed together. The composition of the two constraint automata allows transitions when data occur at the input/output ports that the resulting automaton inherits from only one of the automata (because at that point in time, there is no suitable data on any of its corresponding ports). Such transitions do not exist in the "one-to-many composition" of timed port automata.
- Interface automata use the notion of input enabledness as I/O-automata and timed port automata. Constraint automata do not have this notion. In fact, they do not even distinguish between input and output ports.

Several tools exist for the different modeling theories. UPPAAL[1], for example, is an integrated tool environment for modeling, validation and verification of real-time systems modeled as networks of timed automata. UPPAAL is quite extensive, but an important drawback is the fact that the project is closed-source, which prevents us from extending or modifying it to our needs.
There is also a project where simulation and verification tools are being implemented for I/O automata[2]. However, there is no tool to construct and modify an I/O-automata visually. Since it is our intension to build a visual editor for constraint automata, this project is not useful for our purposes.
Some editor and simulator tools exist for finite automata. However, it is expected that they are not useful to us, because it would require too much time to adapt them for constraint automata.

1.4 Approach

First, we perform a literature study to acquire the background knowledge on Reo and constraint automata, and become familiar with the concepts and terminology. Next, we analyze the assignment and its requirements, decomposing them and investigating potential problems. Finally, we describe the design and implementation of the tool.

[1]http://http://www.uppaal.com/
[2]http://theory.csail.mit.edu/tds/ioa/

1.5 Thesis Outline

The remainder of the thesis consists of 6 chapters, which are structured in the following way. Chapter 2 describes Reo, while chapter 3 introduces constraint automata. Chapter 4 discusses how constraint automata can be used as an operational semantic model for Reo. Chapter 5 presents our analysis of the assignment and its requirements. Chapter 6 describes the design and implementation of the tool. In chapter 7 we present our conclusions, summarizing our work, discussing in which directions future work could be done and reflecting on the personal experiences.

Chapter 2

Reo

Reo is a channel-based exogenous coordination language, which connect and coordinate different activities through connectors. Complex connectors are compositionally built out of simpler ones. The most primitive connectors are a set of channels with well-defined behavior provided by the user.
This chapter discusses Reo in detail. The first section introduces some basic concepts of Reo, followed by an explanation about channels in Reo and the way they connect components with each other. The next section demonstrates how to make more complex connectors composed of simple connectors by the join operation. The following section describes the hide operation, an abstraction mechanism in Reo. The last section shows some examples of Reo, demonstrating its expressive power. The majority of this chapter is based on [1] and [4]

2.1 Basic Concepts

The emphasis in Reo is not on the computational entities it coordinates, but on the connectors. A computational entity is referred to as a component instance. A component instance consists of at least one active entity. How the internal of a component instance is organized is not relevant for Reo. It can consist of processes, threads or even agents. What is of importance is the inter-component-instance communication that takes place.
A component instance has some ‘contact points’, which it can use to exchange information with its environment. These contact points are referred to as the ports of the component instance. These ports allow a component instance to connect to Reo connectors. Reo connectors are explained in further detail in the following sections. Furthermore, we assume that ports are unidirectional. A component instance can use an input port to receive and an output port to send information.
In the Object Oriented world a difference is made between objects and classes. A class is a blueprint of an object. It describes the structure and the behavior of an object. An object itself is an instance of a class. The same analogy can be applied to component instances. The blueprint of a component instance is called

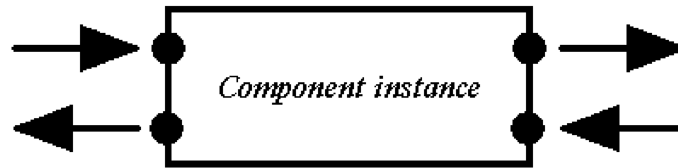

Figure 2.1: Component instance with 4 ports.

component, an abstract type that describes the properties of its instances.

2.2 Channels

In Reo component instances are connected to each other through Reo connectors. Such a connector is a composition of simpler connectors, where the most primitive connector is a channel. A channel has two channel ends. There are two types of channel ends, a source channel end and a sink channel end. Through the source channel end a channel accepts data, while on the sink channel end data leaves the channel. Although the channel ends are directed, the channel itself does not have to be directed. Thus, a channel can have two source channel ends, or two sink channel ends. A channel can have a pattern, which acts as a filter that limits the set of data items that is allowed to go through the channel.

Reo itself does not provide pre-defined channels. On the contrary, a user may define own channels, as long as the behavior is well-defined. Although this is true, there are channel types from which it is expected that they will be part of a 'standard library', because they will be used regularly. We discuss several of these channels in the next subsections.

2.2.1 Sync

A channel of type Sync has a source end and a sink end. Data items are only transferred if simultaneously a write and a take operation take place at the source end, respectively the sink end. Figure 2.2 shows a Sync channel.

Figure 2.2: Sync channel.

2.2.2 SyncDrain

A SyncDrain is a channel where both channel ends are source ends. Write operations can take place at both end. However, the write operations can only succeed simultaneously. All data items that are written are lost. The SyncDrain is depicted in figure 2.3.

Figure 2.3: SyncDrain channel.

2.2.3 SyncSpout

The SyncSpout is the opposite of the SyncDrain. Instead of source ends, both channel ends are sink ends. They both have the same synchronous character. Data items are sent if both sides simultaneously want to take. The data items that are transferred are random data items. It is possible to apply a pattern such that the data items are selected from a restricted set, e.g. the numbers from the interval [1, 10]. A SyncSpout is shown in figure 2.4.

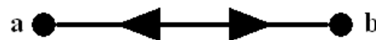

Figure 2.4: SyncSpout channel.

2.2.4 FIFO and FIFO*n*

A FIFO channel has a source and a sink end, but also an unbounded buffer. The source end always accepts data items, which are stored in the buffer. Take operations at the sink end succeed only if data items are available in the buffer. The data items leaving the buffer are in FIFO order (in the same order as they entered the buffer). On the contrary to the previous channel types, a FIFO channel is asynchronous. The FIFO channel type is drawn as the Sync channel, but with a rectangle in the middle representing the buffer (figure 2.5).

Figure 2.5: FIFO channel.

The FIFO*n* channel has, in contrast to FIFO, a bounded buffer. The name indicates the buffer size, e.g. FIFO1 has a buffer of size 1. A FIFO or FIFO*n* channel can

also be initialized with some data items already available in the buffer.

2.2.5 LossySync

A LossySync Channel has a source end and a sink end. The source end always accepts a data item, but it is only transferred to the sink end if a take operation is present. As long as write and take operations take place simultaneously, this type of channel acts like a Sync channel. Otherwise the data item is lost. The LossySync is depicted in figure 2.6.

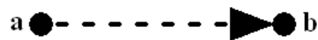

Figure 2.6: LossySync channel.

2.3 Join

In the previous section a few primitive connectors, channels, with interesting behavior have been introduced. They are used to compositionally build more complex connectors. These complex connectors are made by applying the Reo join operation. This is the joining of multiple channel ends at one node. Three types of nodes can be distinguished: source nodes, sink nodes and mixed nodes.

2.3.1 Source node: replicate

A source node is a node where only source channel ends coincide. Figure 2.7 shows a source node where three source channel ends coincide. A write operation at a source node succeeds only if all the source channel ends accept the data item. The source node acts like a replicator.

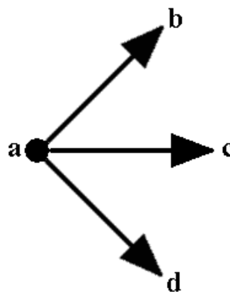

Figure 2.7: Source node.

Using figure 2.7 as an example, this means that a write operation at point **a** succeeds only if all the source channel ends of the channels indicated by **b**, **c** and **d** accept the data item. If this situation occurs, then the data item is transferred to all source channel ends.

2.3.2 Sink node: merge

If the channel ends that coincide at a node are all sink channel ends, then this node is called a sink node. An example of a sink node is shown in figure 2.8. A data item is only transferred if simultaneously a take and at least one write operation take place. If data items are offered by multiple sink ends, then one of them is chosen non-deterministically. The sink node acts like a merger.

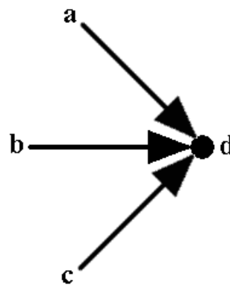

Figure 2.8: Sink node.

2.3.3 Mixed node: replicate + merge

A mixed node is, as the name already indicates, a mixture of a source node and a sink node. Thus, source channel ends coincide at a mixed node as well as sink channel ends (figure 2.9). The behavior of a mixed node is the combination of a source node and sink node. Data items are only transferred if simultaneously at least one sink node offers a data item and all the source nodes accept this data item. If so, then this data item is transferred from the sink node to all the source nodes. If multiple source nodes want to write, then one of them is non-deterministically chosen. Thus, a mixed node acts like a replicator as well as a merger.
There is one important difference between mixed nodes and source and sink nodes. Reo allows component instances to directly write to source nodes and directly take from sink nodes, but with mixed nodes this is not permitted.

2.4 Hide

Complex connectors can be built by composing the primitive channels using the join operation. These connectors can be used to connect (the ports of) component

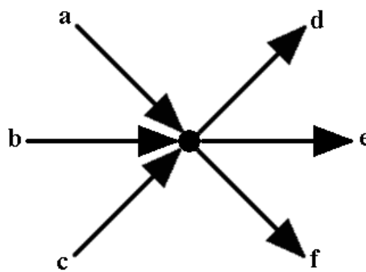

Figure 2.9: Mixed node

instances with each other. After building such a connector, usually only the external behavior is of interest, not how the internal of this connector is organized. Therefore, Reo introduces an abstraction mechanism, the hide operation.

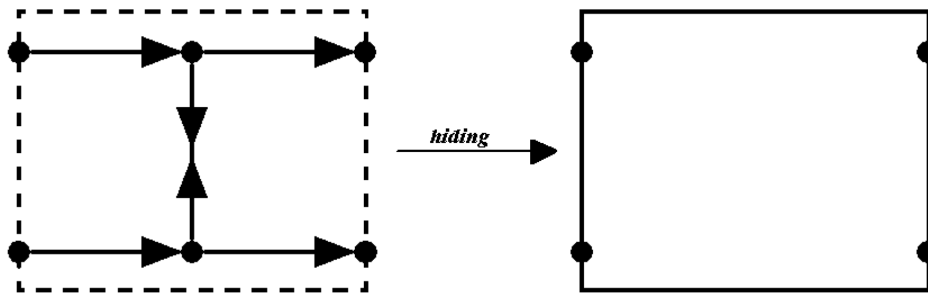

Figure 2.10: Hiding.

In figure 2.10 a connector is shown at the left side. By applying the hide operation, the topology of the nodes (and edges) is hidden and cannot be modified anymore. This results in a connector with a number of input and output ports, shown at the right.

2.5 Examples

In this section a few examples demonstrate the expressive power of connector composition in Reo.

2.5.1 Asynchronous Drain

An AsyncDrain channel has two source channel ends and the data items written to the channel are lost, similar to the SyncDrain, but on the contrary to the SyncDrain the writes have to take place asynchronously. Such a channel can be defined by the

user, but it is also possible to construct the AsyncDrain out of the already existing channels (figure 2.11).

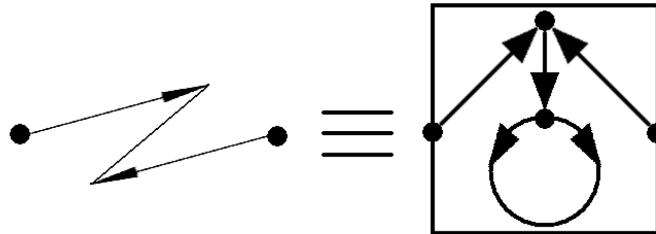

Figure 2.11: AsyncDrain channel.

2.5.2 Regulated Reads

Data items flowing from one component to another through a Sync channel can easily be regulated using the construction illustrated in figure 2.12. Data items go from **a** to **b** only if a take operation is present at **c**. For instance, a component connected to **c** is able to regulate the flow from **a** to **b**. On the other hand, one can say that **b** also regulates the flow from **a** to **c**.

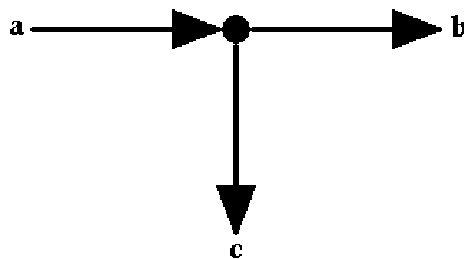

Figure 2.12: Regulated Reads through takes.

It is possible to use write operations instead of take operations to regulate the flow. This can easily be accomplished by replacing one of the Sync channels by a SyncDrain channel (figure 2.13).

2.5.3 Barrier Synchronization

A barrier synchronization, where all channels are only allowed to transfer data items at the same time, can be realized in Reo using the construction shown in figure 2.14. To accomplish the opposite, thus preventing synchronization between the channels, the SyncDrain needs to be replaced by an AsyncDrain.

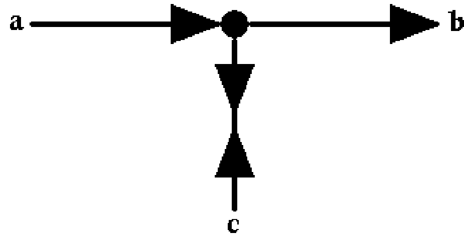

Figure 2.13: Regulated Reads through writes.

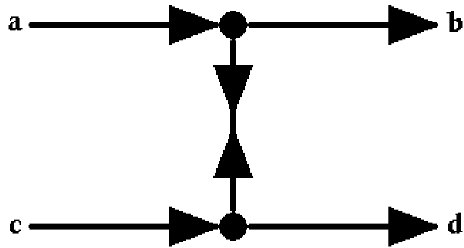

Figure 2.14: Barrier Synchronization.

2.5.4 Exclusive and Inclusive Router

An exlusive router is depicted in figure 2.15. The behavior of an exlusive router is such that a data item written to **a** is only transferred to either **b** or **c**, but never to both.
The behavior of an inclusive router is such that a data item written to **a** flows to **b**, or **c** or to both (figure 2.16).

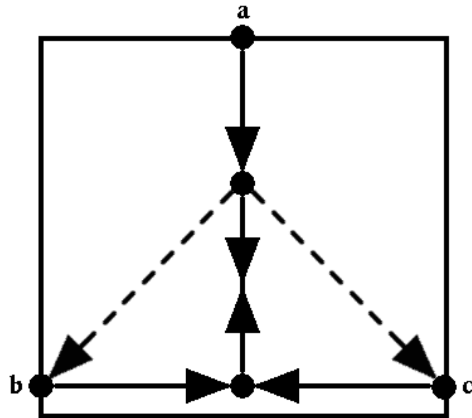

Figure 2.15: Exclusive Router.

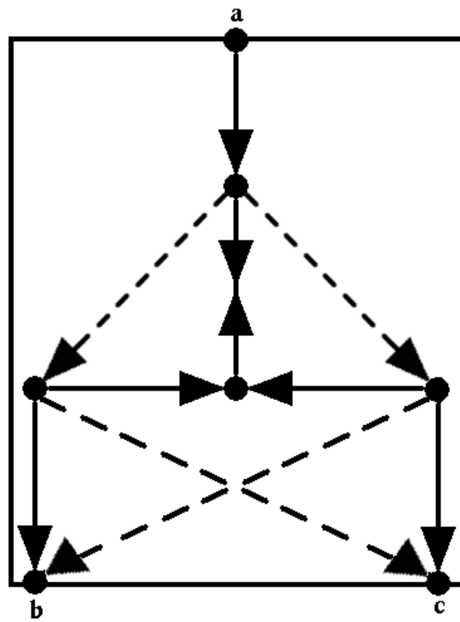

Figure 2.16: Inclusive Router.

Chapter 3

Constraint Automata

Automata are mathematical models, which are used to model operations of many systems. In this chapter we look at automata as very simple computer programs. These computer programs receive some input and can generate two possible outputs:

- **'accept'**, the input is accepted
- **'reject'**, the input is rejected

This chapter discusses a special type of automata, the constraint automata, but first a short introduction to a very basic automaton is given, the finite automaton. The first section is based on [8], while the remainder of the chapter about the constraint automata is mainly based on [3].

3.1 Finite Automata

The finite automata is one of the most basic and well-known automata. Figure 3.1 depicts a finite automaton.

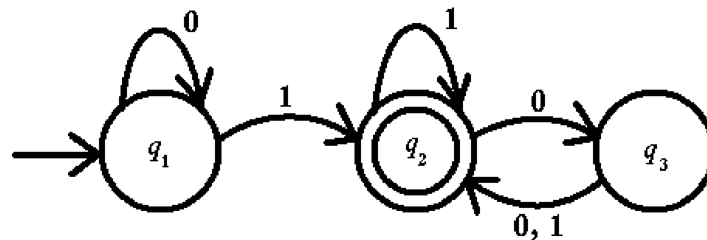

Figure 3.1: Finite automaton.

The automaton is represented using a state diagram. The state q_1, which is indicated by an arrow pointing at it from nowhere, is the initial state. The state q_2 with

a double circle is the accept state. The arrows between the states are transitions. The label on an arrow indicates on which input this transition takes place.
The finite automaton begins in the initial state and moves from state to state dependent on the input it reads. The final state, the state in which the automaton is when reading the last input, determines the output. If the final state is an accept state, then the automaton produces the output 'accept', otherwise 'reject' is output. For example, when the string '0100' is fed to the automaton depicted in figure 3.1, it goes through the following sequence of states: q_1 q_1 q_2 q_3 q_2. Ending in state q_2, which is an accept state, the automaton accepts the input.
A finite automaton is formally defined as a 5-tuple $(Q, \sum, \delta, q_0, F)$, where

- Q is a finite non-empty set of states,
- $\sum$ is the alphabet, a finite set of symbols which act as the input for the automaton.,
- $\delta : Q \times \sum \longrightarrow Q$ is the transition function,
- $q_0 \in Q$ is the initial state,
- $F \subseteq Q$ is the set of accepting states.

Suppose A is the set that contains all strings automaton M accepts, then we call A the **accepted language** of automaton M, also written as $\mathcal{L}(M) = A$.

3.2 TDS-language

This section defines the notion of timed data streams, which act as the input for constraint automata. They are also used to describe the language of constraint automata, the TDS-language.

3.2.1 Stream

A stream is an infinite sequence over a set. Let V be the set of elements from which the stream consists. Then the set of all streams is defined by:

$$V^\omega = \{\alpha \,|\, \alpha : \{0, 1, 2, ...\} \rightarrow V\}$$

A stream and its elements are denoted as follows:

$$\alpha = (\alpha(0)\,, \alpha(1)\,, \alpha(2)\,, ...) \qquad \text{where} \quad \alpha\,(k) \in V \quad \text{for } k \geq 0$$

The first element $\alpha(0)$ of the stream α is called the initial value of α. The stream derivative α' of α is defined as:

$$\alpha' = (\alpha(1)\,, \alpha(2)\,, \alpha(3)\,, ...)$$

Higher order derivatives $\alpha^{(k)}$ are defined as follows:

$$\begin{aligned}\alpha(0) &= \alpha \\ \alpha^{(k+1)} &= \left(\alpha^{(k)}\right)' \quad \text{for } k \geq 0\end{aligned}$$

The following equation can be derived with the previous definitions:

$$\alpha^{(k)}(n) = \alpha(n+k) \quad \text{for } k \geq 0$$

3.2.2 Data Stream

Using the definition of streams, it is possible to define data streams and time streams. A data stream is a sequence over a fixed, non-empty, finite set $Data$, where the elements of this set $Data$ are uninterpreted data elements. The set DS of all data streams over $Data$ is defined as:

$$DS = Data^{\omega}$$

3.2.3 Time Stream

For time streams sequences over the set $\mathbb{R}_+$ are used, $\mathbb{R}_+^{\omega}$. The non-negative real numbers represent the time points. For convenience, the relations $<$ and $\leq$ are defined for $a, b \in \mathbb{R}_+^{\omega}$ as follows:

$$\begin{aligned}a < b &\equiv \forall n \geq 0,\ a(n) < b(n) \\ a \leq b &\equiv \forall n \geq 0,\ a(n) \leq b(n)\end{aligned}$$

All elements of a time stream are strictly smaller than their successor. Thus, the time stream consists of increasing time moments. It is assumed that the time elements go to infinity. This assumption prevents the so-called Zeno paradox, where infinite number of actions may take place within a finite time interval. The set TS of all time streams is defined as:

$$TS = \left\{ a \in \mathbb{R}_+^{\omega} \mid a < a' \text{ and } \lim_{k \to \infty} a(k) = \infty \right\}$$

3.2.4 Timed Data Stream

A timed data stream is defined as:

$$TDS = DS \times TS$$

Thus, a timed data stream $\langle \alpha, a \rangle$ is a pair of streams, where data stream $\alpha \in DS$ and time stream $a \in TS$. The intuitive meaning of a TDS is that data element $\alpha(n)$ occurs at time point $a(n)$.

3.3 Defining Constraint Automata

3.3.1 Ports

Timed data streams are used as the input for constraint automata. The constraint automaton can have more than one input, on the contrary to finite automaton. Therefore, a constraint automaton has one or more ports, which are identified by names from a name-set $\mathcal{N}ames$, e.g. $\mathcal{N}ames = \{A_1, \dots, A_n\}$. Each port denoted in $\mathcal{N}ames$ is associated with a timed data stream.

3.3.2 Transitions

Constraint automata have transitions just like the finite automata, but in contrast to finite automata, each transition of a constraint automaton is labeled with a pair N, g. N is a non-empty subset of $\mathcal{N}ames$ indicating which ports are active during the transition, meaning at which ports data is being observed. g is a data constraint, which imposes restrictions on the transition, on the data observed in the timed data streams of the active ports.

3.3.3 Name-data-assignments

A name-data-assignment for $\emptyset \neq N \subseteq \mathcal{N}ames$ is a function $\delta : N \to Data$ assigning data items to names in N. Notations like $\delta = [data\,(A) = d_A : A \in N]$ are used to describe the assignment of a value $d_A \in Data$ to any TDS-name $A \in N$.

3.3.4 Data Constraints

Data constraints have the following grammar:

$$g ::= true \mid false \mid data(A) = d \mid g_1 \vee g_2 \mid g_1 \wedge g_2$$

where $A \in \mathcal{N}ames$ and $d \in Data$. The notation $DC\,(N, Data)$ (or DC) is used to describe the set of data constraints. Often derived DC's are used such as $data\,(A) \neq d$ and $data\,(A) = data\,(B)$, which respectively stands for

$$\bigvee_{d' \in Data \setminus \{d\}} \left(data\,(A) = d'\right) \text{ and } \bigvee_{d \in Data} \left(data\,(A) = d\right) \wedge \left(data\,(B) = d\right)$$

The satisfaction relation $\models$ is used for interpreting DC's over name-data-assignments, for example in the following way:

$$\begin{aligned}
[data\,(A) = d_1, data\,(B) = d_2, data\,(C) = d_1] &\models data\,(A) = data\,(C) \\
[data\,(A) = d_1, data\,(B) = d_2, data\,(C) = d_1] &\not\models data\,(A) \neq data\,(B) \\
&\quad \text{if } d_1 \neq d_2
\end{aligned}$$

Satisfiability and validity, logical equivalence $\equiv$ and logical implication $\leq$ of DCs are defined as:

$g_1 \equiv g_2$	iff for all name-data-assignments δ:	$\delta \models g_1 \Longleftrightarrow \delta \models g_2$
$g_1 \leq g_2$	iff for all name-data-assignments δ:	$\delta \models g_1 \Longrightarrow \delta \models g_2$

3.3.5 Definition of Constraint Automata

A constraint automaton is formally defined as a tuple $\mathcal{A} = (Q, \mathcal{N}ames, \longrightarrow, Q_0)$ where

- Q is a set of states,
- $\mathcal{N}ames$ is a finite set of names, which is used to identify the ports,
- $\longrightarrow$ is a subset of $Q \times 2^{\mathcal{N}ames} \times DC \times Q$, called the transition relation,
- $Q_0 \subseteq Q$ is the set of initial states.

For convenience, $(q, N, g, p) \in\longrightarrow$ is written as $q \xrightarrow{N,g} p$. It is required that $(1)\, N \neq \emptyset$ and $(2)\, g \in DC\,(N, Data)$.

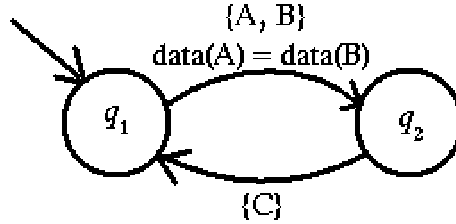

Figure 3.2: Constraint automaton.

We explain the operational behavior of constraint automata using the constraint automaton depicted in figure 3.2 as an example. This constraint automaton has three ports A, B and C. It starts in its initial state q_1 and waits until data is observed at some of its ports. Suppose data item $d1$ occurs at port A and at port B, while the other ports are not active, in this case port C. This event triggers to check the data constraints of the transition $q_1 \xrightarrow{N,g} q_2$. Because $[data\,(A) = d_1, data\,(B) = d_1] \models data\,(A) = data\,(B)$ holds, it moves to the state q_2.
If instead different data items would occur at port A and port B, for instance data item $d1$ respectively $d2$, then no transitions were possible, because no data constraints could be fulfilled. This would cause the constraint automaton to reject.
Knowing this behavior the requirements (1) and (2) can be explained as follows. Condition (1) says a transition can only take place if data occurs at at least one of its ports. Condition (2) states that the automaton can only put requirements on the data that is being observed (not on data that may occur in the future).

3.3.6 Deterministic vs Non-deterministic

The definition above allows for non-deterministic constraint automata. Suppose a constraint automaton resides in state q. Then for a nonempty subset N of $\mathcal{N}ames$ and a given data-name-assignments δ, there may be several transitions

$$q \xrightarrow{N,g_1} q_1,\ q \xrightarrow{N,g_2} q_2,\ \ldots \quad \text{with } \delta \models g_i, \quad i = 1,\ 2,\ \ldots$$

A constraint automaton is called deterministic if for every state q, every N and for every name-data-assignment δ there is at most one transition

$$q \xrightarrow{N,g} q' \text{ with } \delta \models g$$

A non-deterministic constraint automaton can always be transformed in a deterministic constraint automaton that accepts the same language.

3.4 TDS-language and Constraint Automata

This section defines when a language is accepted by a constraint automaton. Consider a constraint automaton $\mathcal{A} = (Q, \mathcal{N}, \longrightarrow, Q_0)$ with two ports A and B, $\mathcal{N} = \{A, B\}$. Further, the timed data streams $\langle \alpha, a \rangle$ and $\langle \beta, b \rangle$ are associated with respectively port A and port B. Then the language accepted by this automaton $\mathcal{A}$ is defined as follows:

$$\mathcal{L}_{TDS}(\mathcal{A}) = \bigcup_{q_0 \in Q_0} \mathcal{L}_{TDS}(\mathcal{A}, q_0)$$

$\mathcal{L}_{TDS}(\mathcal{A}, q)$ denotes the language accepted by the state q of the constraint automaton $\mathcal{A}$:

$$\begin{aligned} \mathcal{L}_{TDS}(\mathcal{A}, q) \quad = \quad & \{(\langle \alpha, a \rangle, \langle \beta, b \rangle) \in TDS \times TDS \mid (\langle \alpha, a \rangle, \langle \beta, b \rangle) \\ & \text{is a timed run for } (\mathcal{A}, q)\} \end{aligned}$$

$(\langle \alpha, a \rangle, \langle \beta, b \rangle)$ is called a timed run for $(\mathcal{A}, q)$ iff there exists a transition $q \xrightarrow{N,g} {}^{-}q$ such that

$$\begin{array}{llll} & a_0 < b_0 & \wedge & N = \{A\} \wedge [data(A) = \alpha_0] \models g \wedge \\ & & & \left(\langle \alpha', a' \rangle, \langle \beta, b \rangle\right) \in \mathcal{L}_{TDS}\left(\mathcal{A}, {}^{-}q\right), \\ \text{or} & b_0 < a_0 & \wedge & N = \{B\} \wedge [data(B) = \beta_0] \models g \wedge \\ & & & \left(\langle \alpha, a \rangle, \langle \beta', b' \rangle\right) \in \mathcal{L}_{TDS}\left(\mathcal{A}, {}^{-}q\right), \\ \text{or} & a_0 = b_0 & \wedge & N = \{A, B\} \wedge [data(A) = \alpha_0, data(B) = \beta_0] \models g \wedge \\ & & & \left(\langle \alpha', a' \rangle, \langle \beta', b' \rangle\right) \in \mathcal{L}_{TDS}\left(\mathcal{A}, {}^{-}q\right) \end{array}$$

The intuitive meaning of the definitions above is as follows. The data items that appear as first are selected out of the data streams of the TDS. This is done using

the order specified by the time stream of the TDS. Subsequently, these data items are assigned to the associated ports, which thus become active. Next, the constraint automaton tries to make a transition with the data items observed at the ports. If an infinite sequence of such transitions exists (starting from one of the initial states), then this timed data stream is accepted by the constraint automaton.

Figure 3.3: Constraint automaton (of a Sync channel).

Using the constraint automaton depicted in figure 3.3 as an example and call it A, then the accepted TDS-language can be defined as
$\mathcal{L}_{TDS}(A) = \{(\langle\alpha, a\rangle \langle\beta, b\rangle) \in TDS \mid \alpha = \beta \wedge a = b\}$

Chapter 4

Modeling Reo by Constraint Automata

This chapter discusses how constraint automata can be used as an operational semantic model for Reo. First, the relation between Reo channels and constraint automata is explained. Followed by an explanation how the join and hide operation on Reo channels can be modeled in constraint automata. The last section discusses the parameterized constraint automata, which simplify the notation of the ordinary constraint automata.

4.1 Channels

Channels in Reo are related to constraint automata by the TDS-language. The next subsection explains this relation in detail using the Sync channel as an example. The following subsection shows the other basic channels, but less extensively.

4.1.1 Sync

The behavior of the Sync channel can be captured by a TDS-language. Data items that occur at each channel end and their corresponding time points can be 'recorded' by a timed data stream. Suppose the timed data streams $\langle \alpha, a \rangle$ and $\langle \beta, b \rangle$ are used for respectively channel end **a** and **b**. The behavior of a Sync channel is such that data items are only transferred when simultaneously a write and read operation take place. Thus, when a write operation of data element d_1 at the source channel end at time point t_1 succeeds, the same data element will appear at the other side instantaneously at time point t_1. Figure 4.1 illustrates how the timed data streams look like.
The behavior of the channel holds for every data item that goes through the Sync channel. Thus $\alpha(k) = \beta(k)$ and $a(k) = b(k)$ for all $k \geq 0$. In general the behavior of the Sync channel can be described by the following TDS-language:

$$\{(\langle \alpha, a \rangle, \langle \beta, b \rangle) \in TDS \times TDS \mid \alpha = \beta \wedge a = b\}$$

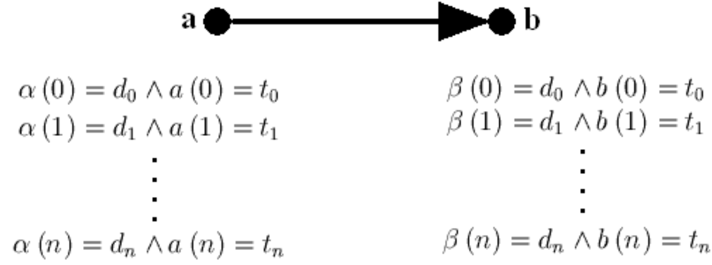

Figure 4.1: Sync channel with Timed Data Stream.

This TDS-language is the accepted TDS-language of the constraint automaton depicted in figure 3.3.
Hence, a channel induces a TDS-language, which is again the accepted TDS-language of a certain constraint automaton. On the other hand, you can also say that a constraint automaton describes the behavior of a channel. Note, however, that constraint automata only speak about ports and do not distinguish between input and output ports.

4.1.2 SyncDrain and SyncSpout

The TDS-language of the SyncDrain is the same as for the SyncSpout, because both only care about the timing at which the data is observed and not the data itself. The TDS-language is

$$\{(\langle\alpha, a\rangle, \langle\beta, b\rangle) \in TDS \times TDS \mid a = b\}$$

The constraint automata of both the channels are also the same (figure 4.2).

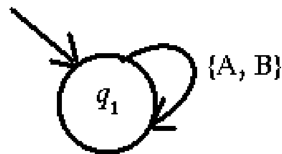

Figure 4.2: Constraint automaton for SyncDrain and SyncSpout.

4.1.3 FIFO1

The TDS-language of the FIFO1 channel is

$$\left\{(\langle\alpha, a\rangle, \langle\beta, b\rangle) \in TDS \times TDS \mid \alpha = \beta \wedge a < b < a'\right\}$$

To keep the constraint automaton of the FIFO1 channel simple, the $Data$-set consists only of one element (figure 4.3).

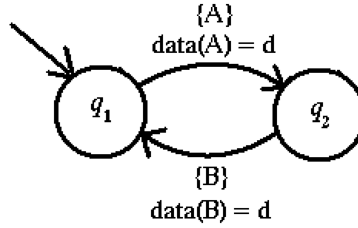

Figure 4.3: Constraint automaton for FIFO1.

4.1.4 LossySync

The TDS-language of the LossySync is as follows:

$$\{(\langle \alpha, a \rangle, \langle \beta, b \rangle) \in TDS \times TDS \mid \; = \beta = L(\alpha, a, b)\}$$

$$L(\alpha, a, b) = \begin{cases} \alpha(0) \bullet L(\alpha', a', b') & \text{if } b(0) \leq a(0) \leq b(1) \\ L(\alpha', a', b) & \text{otherwise} \end{cases}$$

The corresponding constraint automaton is shown in figure 4.4.

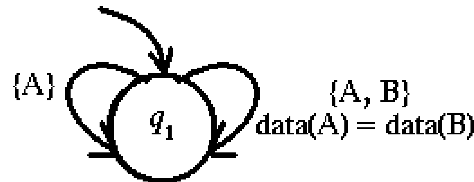

Figure 4.4: Constraint automaton for LossySync.

4.2 Join

The join operation of Reo can be modeled at the constraint automata level. The join of a source node with another node (of arbitrary type) is realized by a product construction. The join of sink nodes is modeled using a merger.
However, at the constraint automata level not all types of nodes can be joined together. This does not raise any problems, because the focus is at static Reo connectors. Therefore, it can be assumed that complex connectors are built such that the join operation is applied in a specific order. Thus, first the sink nodes are joined and then the resulting node is joined with the source nodes.

4.2.1 Product-construction

Suppose there are two Reo connectors that are modeled by the constraint automata $\mathcal{A}_1$ and $\mathcal{A}_2$ with respectively node-sets $\mathcal{N}_1$ and $\mathcal{N}_2$. A join operation need to be performed at the node-pairs $\langle B_i, \overline{B}_i \rangle \in \mathcal{N}_1 \times \mathcal{N}_2$ for $i = 1, \ldots, k$, where at least one node of the node-pair is a source node. For simplification, it is assumed that the nodes are renamed such that $B_i = \overline{B}_i$ for $i = 1, \ldots, k$ and that the automata do not have other common nodes. Thus, the join need to be performed at the nodes $B \in \mathcal{N}_1 \cap \mathcal{N}_2$. The join can be performed at the constraint automata level using the product-construction.

The product of the constraint automata $\mathcal{A}_1 = (Q_1, \mathcal{N}ames_1, \longrightarrow_1, Q_{0,1})$ and $\mathcal{A}_2 = (Q_2, \mathcal{N}ames_2, \longrightarrow_2, Q_{0,2})$ is defined as follows:

$$\mathcal{A}_1 \bowtie \mathcal{A}_2 = (Q_1 \times Q_2, \mathcal{N}ames_1 \cup \mathcal{N}ames_2, \longrightarrow_1, Q_{0,1} \times Q_{0,2})$$

where $\longrightarrow$ is defined as:

$$\frac{q_1 \xrightarrow{N_1,g_1}_1 p_1, q_2 \xrightarrow{N_2,g_2}_2 p_2, N_1 \cap \mathcal{N}ames_2 = N_2 \cap \mathcal{N}ames_1}{\langle q_1, q_2 \rangle \xrightarrow{N_1 \cup N_2, g_1 \wedge g_2} \langle p_1, p_2 \rangle}$$

$$\frac{q_1 \xrightarrow{N,g}_1 p_1, N \cap \mathcal{N}ames_2 = \emptyset}{\langle q_1, q_2 \rangle \xrightarrow{N,g} \langle p_1, q_2 \rangle}$$

$$\frac{q_2 \xrightarrow{N,g}_1 p_2, N \cap \mathcal{N}ames_1 = \emptyset}{\langle q_1, q_2 \rangle \xrightarrow{N,g} \langle q_1, p_2 \rangle}$$

Figure 4.5 illustrates how a FIFO2 is constructed out of two FIFO1 channels by the product-construction.

The join operation is also defined at the constraint automata language level. Suppose there are two TDS-languages $L_1 = l_1 (A, B)$ with node-set $\mathcal{N} = \{A, B\}$ and $L_2 = L_2 (B, C)$ with node-set $\mathcal{N} = \{B, C\}$), then the join is defined as:

$$L_1 \bowtie L_2 = \{(\langle \alpha, a \rangle, \langle \beta, b \rangle, \langle \gamma, c \rangle) : (\langle \alpha, a \rangle, \langle \beta, b \rangle) \in L_1 \wedge (\langle \beta, b \rangle, \langle \gamma, c \rangle) \in L_2\}$$

The correctness of the product can be proven, which states that the accepted TDS-language of the product-automaton is equivalent to the join of accepted TDS-languages of the constraint automata separately.

(a) $\mathcal{L}_{TDS} (\mathcal{A}_1 \bowtie \mathcal{A}_2) = \mathcal{L}_{TDS} (\mathcal{A}_1) \bowtie \mathcal{L}_{TDS} (\mathcal{A}_2)$

(b) If $\mathcal{N}ames_1 = \mathcal{N}ames_2$
then $\mathcal{L}_{TDS} (\mathcal{A}_1 \bowtie \mathcal{A}_2) = \mathcal{L}_{TDS} (\mathcal{A}_1) \cap \mathcal{L}_{TDS} (\mathcal{A}_2)$

The proof is not given here, but can be found in [3].

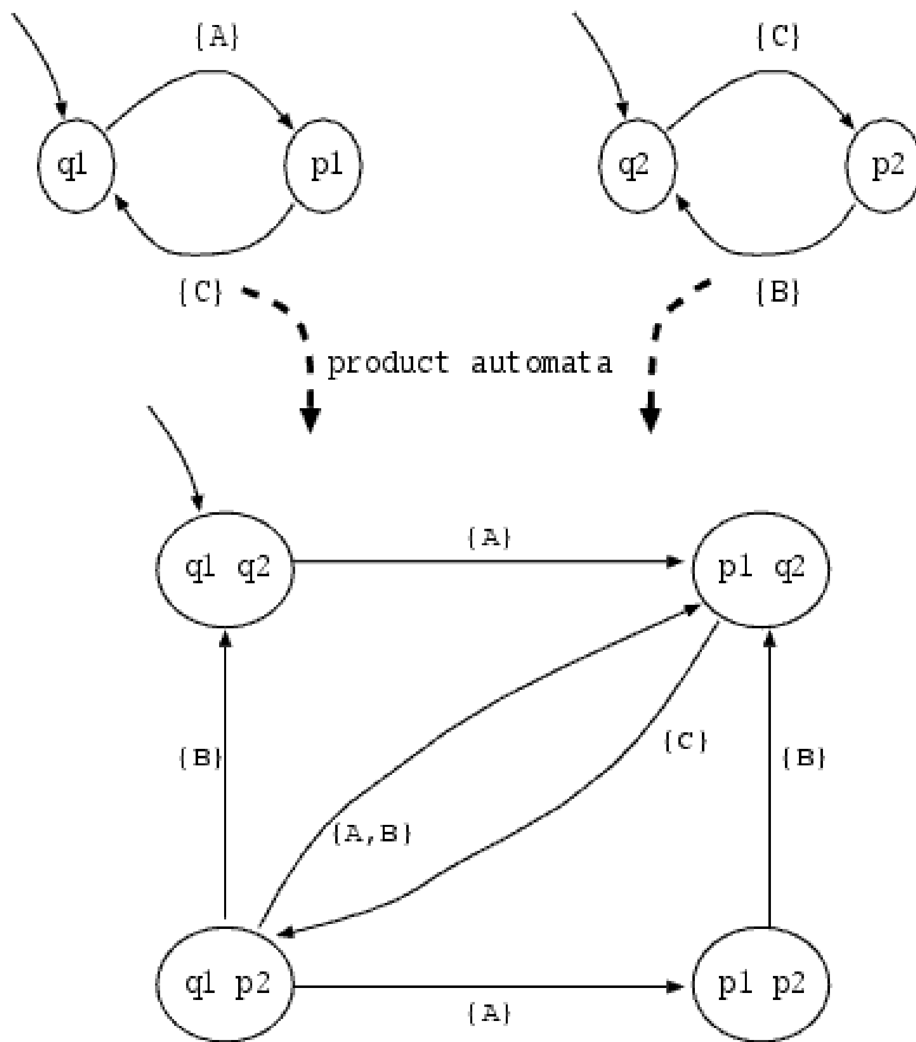

Figure 4.5: Join of two FIFO1 channels.

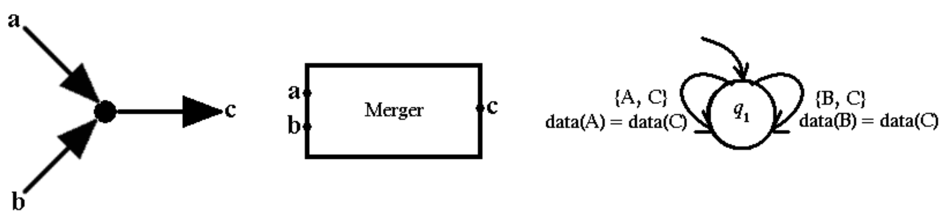

Figure 4.6: Merger

4.2.2 Merger

Sink nodes cannot be joined directly as is done in the previous section. Instead an intermediary 'component' is needed, a merger. The merger is shown in figure 4.6 along with the constraint automaton describing its behavior.
The join at the Reo level can be performed at the automata level by applying the product-construction to the merger constraint automaton with two other constraint automata that also contain the nodes A and B. If needed the node C can again be joined with another sink node using the merger construction or it can be joined with a source node to create a mixed node.

4.3 Hide

The hide operation in Reo makes the internal organization of a connector not observable anymore from the outside. For TDS-languages this is realized by existential quantification. Thus, hiding of a name (node) C in a TDS-language $L(C, A_1, \ldots, A_n)$ means that existential quantification over this C-component needs to be applied. For instance, hiding the name C in the language
$L = L(C, A, B)$ is realized as:

$$\exists C[L] = \{(\langle\alpha, a\rangle, \langle\beta, b\rangle) : \exists TDS\,\langle\gamma, c\rangle \text{ with } (\langle\gamma, c\rangle, \langle\alpha, a\rangle, \langle\beta, b\rangle) \in L\}$$

At the constraint automata level hiding removes all information about a certain port. Suppose the port C needs to be hidden, then the hiding operation for constraint automata is as follows. Let $\mathcal{A} = (Q, \mathcal{N}ames, \longrightarrow, Q_0)$ be a constraint automaton and $C \in \mathcal{N}ames$. Then the constraint automaton after hiding is defined as

$$\exists C[\mathcal{A}] = (Q, \mathcal{N}ames \setminus \{C\}, \longrightarrow_C, Q_{0,C})$$

Let $\rightsquigarrow^*$ be the transition relation such that $q \rightsquigarrow^* p$ if and only if there exists a finite path

$$q \xrightarrow{\{C\},g_1}_1 q_1 \xrightarrow{\{C\},g_2}_1 q_2 \xrightarrow{\{C\},g_3}_1 \ldots \xrightarrow{\{C\},g_n}_1 q_n$$

where $q_n = p$ and $g_1, \ldots, g_n$ are satisfiable. Then the set $Q_{0,C}$ of initial states is as follows

$$Q_{0,C} = Q_0 \cup \{p \in Q : q_0 \rightsquigarrow^* p \text{ for some } q_0 \in Q_0\}$$

The transition relation $\longrightarrow_C$ is given by

$$\frac{q \rightsquigarrow^* p, p \xrightarrow{N,g} r, \overline{N} = N \setminus \{C\} \neq \emptyset, \overline{g} = \exists C[g]}{q \xrightarrow{\overline{N},\overline{g}}_C r}$$

where $\exists C[g] = \bigvee_{d \in Data} g[data(C)/d]$. $g[data(C)/d]$ denotes the data constraint where all occurrences of $data(C)$ in g are syntactically replaced by d. This

comes down to replacing every atom $data(C) = d'$ with $true$ if $d = d'$ and with $false$ if $d \neq d'$.
The correctness of hiding can be proven, which says how the relation between constraint automata and TDS-languages is affected by hiding:

(a) $\exists C [\mathcal{L}_{TDS}(\mathcal{A})] \subseteq \mathcal{L}_{TDS}(\exists C [(A)])$
(b) If $\mathcal{A}$ is finite and does not contain a cycle $q_0 \overset{\{N_1\},g_1}{\longrightarrow} q_1 \overset{\{N_2\},g_2}{\longrightarrow} \ldots \overset{\{N_k\},g_k}{\longrightarrow} q_k$ where $g_1, \ldots, g_k$ are satisfiable and $C \notin N_1 \cup \ldots \cup N_k$ then $\exists C [\mathcal{L}_{TDS}(\mathcal{A})] = \mathcal{L}_{TDS}(\exists C [\mathcal{A}])$

The proof can be found in [3].

Hiding can be intuitively explained as follows. First, the transition relation $q \rightsquigarrow^* p$ is defined. This transition relation denotes all states that are reachable from state q using transitions depending only on the data observed at port C.
If hiding of C takes place, then every state p indicated by $q \rightsquigarrow^* p$ can automatically be reached from state q, because the guards of these transitions only depend on data observed at port C. However, these guards are not taking into account anymore, because of the hiding of C.
If state q is an initial state, then every state p indicated by $q \rightsquigarrow^* p$ becomes an initial state, because they can automatically be reached. Therefore, the set of initial states of the resulting constraint automata after hiding is extended with these states.
A transition from state q to r in the resulting constraint automata is possible when a certain state p can be automatically entered from q (because of the hiding) and a transition from p to r exists where in this transition all occurrences related to port C are removed or replaced.
Figure 4.5 shows the constraint automaton for a FIFO2 channel created by joining two constraint automata of FIFO1 channels. Figure 4.7 shows the constraint automaton of the FIFO2 channel after hiding port C.

4.4 Parameterized Constraint Automata

In the previous examples of the constraint automata, the data items of the set $Data$ were not taken into account. This was done to keep the constraint automata simple. If these data items were modeled, then this could have led to a constraint automaton with a huge number of states. To show how the size of the $Data$ set influences the number of states, we use the constraint automaton of the FIFO1 channel as an example.
Figure 4.8 depicts the constraint automaton of the FIFO1 channel where just one type of data item is allowed to go through the channel. The set $Data$ consists of one data item d_1. State q_1 represents the channel when the buffer is empty. The situation where the buffer contains a data item d_1 is represented by state q_2.

Figure 4.9 shows the constraint automaton of the same FIFO1 channel, but in this case two data items are allowed to go through the channel. The set $Data$ consists of the data items d_1 and d_2. The states q_1 and q_2 represents the same situation as before. However, an extra state is added. This state q_3 represents the buffer when it contains data item d_2. Thus, for each data item a state is needed to represent the content of the buffer. This could result in a huge number of states, especially when this constraint automaton is to be joined with others. To prevent such a state explosion, the parameterized constraint automaton is introduced.
The parameterized constraint automaton uses state variables. In the situation of the FIFO1 channel, one state with a state variable x could be used to model the content of the buffer, instead of using multiple states where each state represents the content of the buffer with another data item (figure 4.10).
A parameterized constraint automaton is formally defined as a tuple

$$\mathcal{P} = (Loc, Var, v, \mathcal{N}ames, \rightsquigarrow, Loc_0, init)$$

where

- Loc is a set of locations,
- $Loc \subseteq Loc$ is a set of initial locations,
- Var is a set of variables,
- $v : Loc \rightarrow 2^{Var}$ assigns to any location a (possibly empty) set of variables,
- $init$ is a function that assigns to any initial location $l \in Loc_0$ a condition for the variables.

$v(l)$ denotes the variable set of a location l. For the constraint automaton illustrated in figure 4.10, $v(q_2) = \{x\}$ states that x is a state variable of location q_2. q_1 having no state variables is denoted by $v(q_1) = \{\}$.

The transition from location l to $\bar{l}$ is denoted as $l \overset{N,h}{\rightsquigarrow}_X \bar{l}$ where

- l and $\bar{l}$ are locations,
- N is a name-set, a non-empty subset of $\mathcal{N}ames$,
- h a (parameterized) data constraint for N with the form "$d_A = expr$" where $expr$ is an operation on $Data$, d_B for $B \in N$ and the variables $x \in v(l)$. For instance, $d_A = d_1 + d_B + x$,
- X is a function that assigns a name $A \in N$ to each variable $\overline{x} \in v\left(\bar{l}\right) \setminus v(l)$ and possibly to the variables in $v\left(\bar{l}\right) \cap v(l)$. Intuitively $X(\overline{x}) = A$ stands for the assignment $\overline{x} := d_A$.

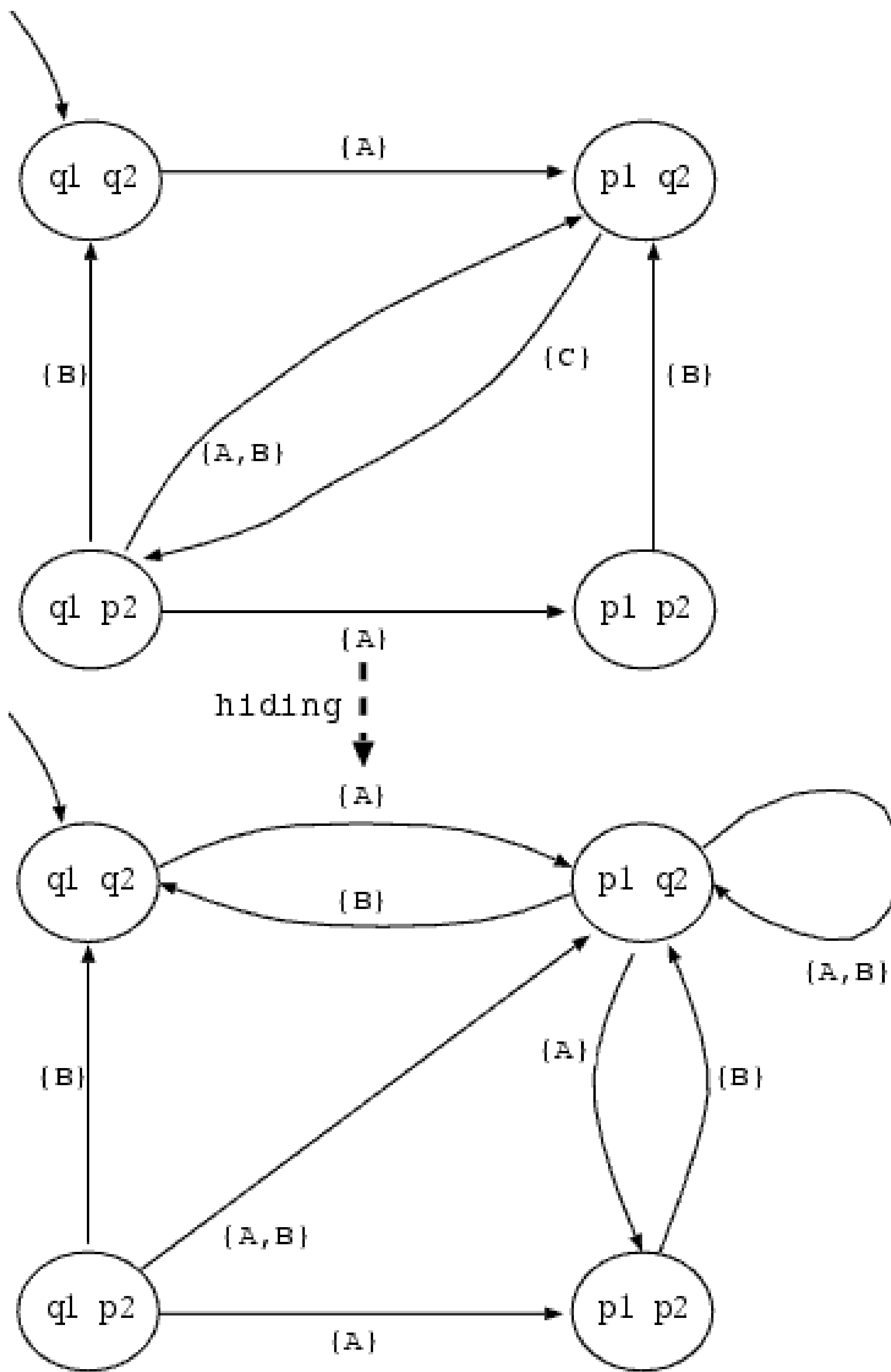

Figure 4.7: Hide of port C after the join of two FIFO1 channels.

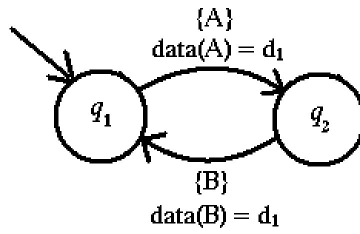

Figure 4.8: Constraint automaton of FIFO1 channel with one data item.

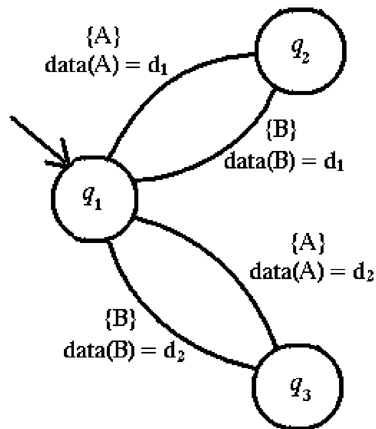

Figure 4.9: Constraint automaton of FIFO1 channel with two data items.

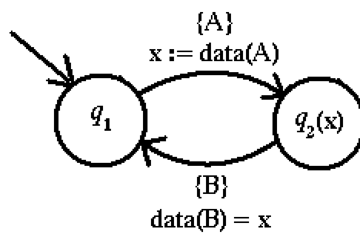

Figure 4.10: Parameterized constraint automaton of FIFO1 channel.

Chapter 5

Requirements and Analysis

This chapter collects and structures the requirements that the constraint automata editor and the simulator have to meet. In order to do this, we first analyse the assignment. Hereafter we look into existing tools and try to see which operating systems need to be supported and which programming language is best suited. Next, we decompose the functionality of the tool into several smaller parts. We analyze the parts and their dependencies. We investigate the possible solutions of the potential problems we have identified thus far. All this is done as a preparation for the next phase, the design and implementation.

5.1 MSc Assignment

The assignment of this MSc thesis is to design and implement a tool, which consists of two parts:

- an editor for constraint automata,
- a simulator for constraint automata.

With the constraint automata editor the user should be able to construct a constraint automaton visually. The user should then be able to simulate the constraint automaton with the simulator.

We distinguish three different kinds of simulators. The first simulator acts as a language acceptor of timed data streams. This means that at every time point of the timed data streams a transition should be made. Otherwise the simulator rejects the input. We call this simulator "TDS-Language Acceptor Simulator".

The second simulator uses constraint automata to simulate the behavior of Reo connectors. The input of a connector is specified by timed data streams. This simulator will be called "Reo Connector Simulator with TDS".

The third simulator behaves like the second simulator, but instead of using timed data streams as input the user should be able to attach 'components' to the simulator. These components then generate the input for the simulator at real-time

by calling standard Reo operations (e.g. take, write) on the I/O ports of the constraint automata simulator. We call this simulator “Reo Connector Simulator with Components”.

5.2 Existing Reo and Constraint Automata Tools

Currently two tools exist for Reo and constraint automata, SAFA[1] and Swiss Watch[1]. SAFA is a tool for converting Reo circuits to constraint automata. The programming language used to implement this tool is Java. Furthermore, XML is used to provide persistent data structures for both Reo circuits and constraint automata.
The second tool, Swiss Watch, is an editor for constraint automata, but it is still in an early development stage. The tool has difficulties, for example visualizing transitions from a state to itself in a convenient way. The constraint automata editor to be implemented should have a better GUI than Swiss Watch.
Swiss Watch does not validate the input of state names and data constraints, which can lead to a corrupt constraint automaton. This should be handled in a better way by the constraint automata editor to be developed.
Besides the editor functionality Swiss Watch is able to perform the join and hide operation on constraint automata. Like the SAFA tool, Swiss Watch is programmed in Java and uses XML to provide the persistent data structure for constraint automata.

5.3 Operating System and Programming Language

The standard operating systems deployed at the computers at CWI are MS Windows and Linux. Some people use the Mac operating system. In order to support as many platforms as possible a suitable programming language is needed. The most obvious programming language is Java, because a Java Virtual Machine exists for all these operating systems.
The existing Reo and constraint automaton tools are already implemented in Java. To keep the possibility that in the future some features of these tools will be integrated with the tools developed during this MSc project, it seems best to use Java. If Java would be chosen as the programming language the choice remains which version of Java would be best. Java 1.5, which is the latest version, seems a good candidate as it has a lot of new useful features compared to the previous version. It supports, for example, generics, typesafe enumeration and autoboxing.
Another important addition is the java.util.concurrent package. This package (and its subpackages) contains a lot of classes that make it much easier to develop multithreaded applications. This can especially be useful for the “Reo Connector Simulator with Components” as it is expected that this will be multithreaded.

[1]For more information contact Marjan Sirjani (Marjan.Sirjani@cwi.nl).

5.4 Global overview

On the basis of the assignment we decompose the desired tool into 6 main subsystems:

- Integrated Tool GUI (IT-GUI),
- Constraint Automata Editor (CA-Editor),
- TDS-Language Acceptor Simulator (TDSLAS),
- Reo Connector Simulator with TDS (RCSwTDS),
- Reo Connector Simulator with Components (RCSwC),
- Constraint Automaton Engine (CA-Engine).

Figure 5.1 shows the structure of these subsystems together with their dependencies. In the following sections we discuss the structure of these systems.

5.4.1 Integrated Tool GUI (IT-GUI)

The Integrated Tool GUI is the graphical user interface of the overall tool. Through the IT-GUI the user will be able to open:

- the CA-Editor,
- the TDSLAS,
- the RCSwTDS,
- the RCSwC.

The IT-GUI offers flexibility to the user, because the user can edit and simulate a constraint automaton within the same tool. Without an integrated environment the user has to save the constraint automaton to a file from the constraint automata editor application and load it into a constraint automata simulator application.

To simulate multiple constraint automata simultaneously, we could join all the constraint automata to be simulated into one constraint automaton (figure 5.2.a). However, the join could lead to a state explosion. To prevent this, the simulator should act on each constraint automaton individually (figure 5.2.b). To support this, the tool needs a multiple document interface (MDI), such that multiple constraint automata can be loaded into the tool. Each document window contains a CA-Editor, where the user can edit and modify a constraint automaton.

Java provides all the packages to create a MDI application. However, to implement the whole MDI application using these packages will be quite time consuming, because you still have to "glue" everything together by yourself.

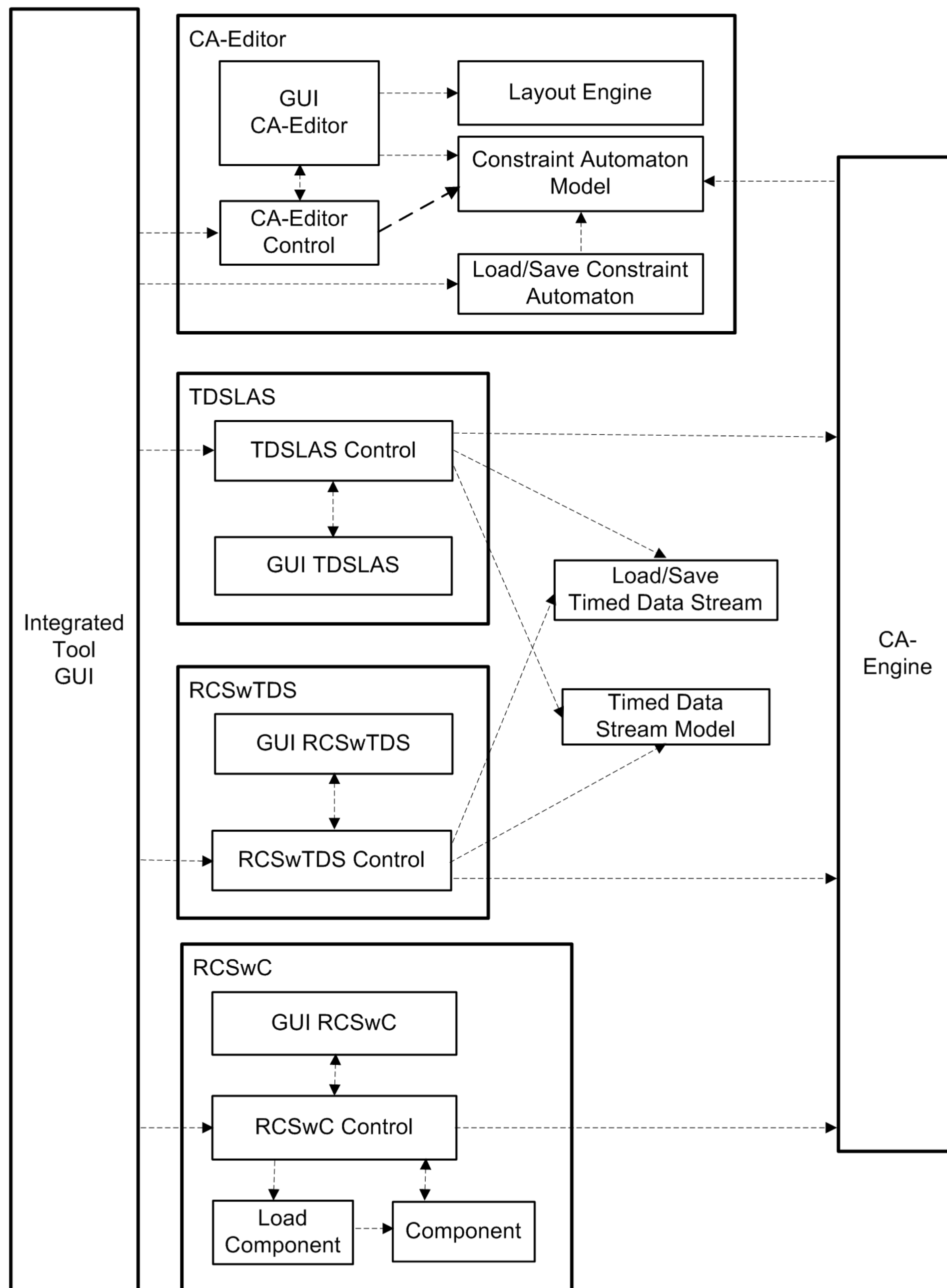

Figure 5.1: Overview of the tool. Dashed arrows indicate dependencies.

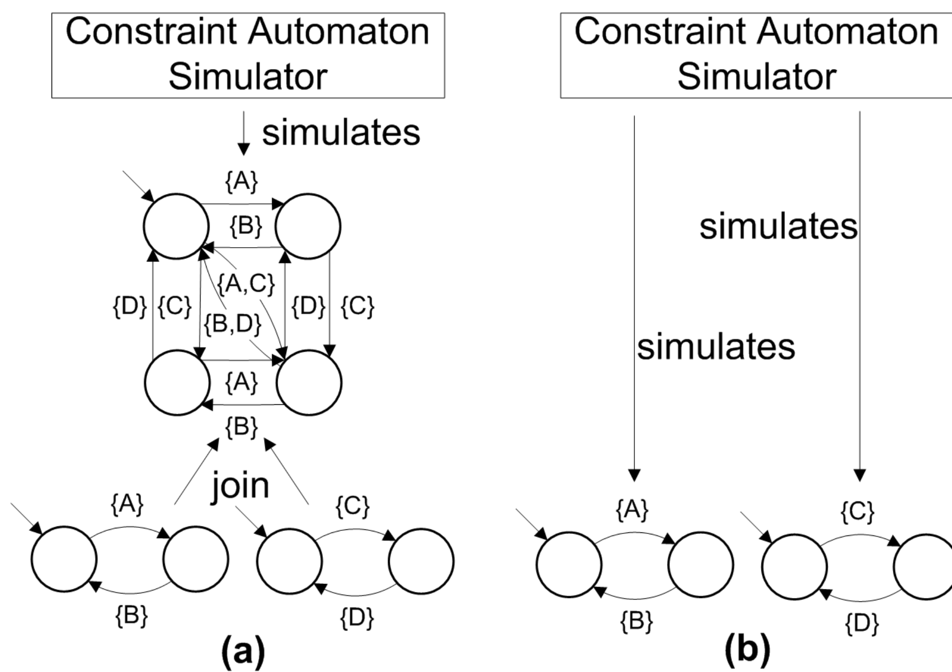

Figure 5.2: (a) Simulating constraint automata by simulating the joined constraint automaton. (b) Simulating constraint automata by simulating each constraint automaton individually.

A better alternative is the Java MDI Application Framework[1]. This framework provides a skeleton for MDI applications, therefore simplifying the development of a MDI application. It is expected that using this framework will save a considerable amount of time. The features of the Java MDI Application Framework are:

- A framework which offers the standard functionality that is expected from MDI applications and which simplifies MDI application development.
- GUI implementation using Swing, including a full-featured user interface with menus, toolbars, status bars, file selectors, dialogs, ...
- Data-Action-View paradigm: Data and View are separated. Action objects are used to synchronize them.
- Support for multiple views per document.
- Support for nested views as well as nested data objects.
- Modular File I/O system for loading, saving and exporting documents.
- Undo/Redo functionality.
- Clipboard functions (cut, copy, paste).
- Printing subframework with preview capability.
- Internationalization support using ResourceBundles.

5.4.2 Constraint Automaton Model (CAM)

The Constraint Automaton Model represents a constraint automaton at the software level. It contains the data structures to store:

- the topology of a constraint automaton, e.g. states and transitions,
- properties of states, e.g state name,
- properties of transitions, e.g. data constraints.

The CAM provides the following methods to modify the data structures:

- add and remove states,
- add and remove transitions,
- modify state properties,
- modify transition properties.

[1] http://jmdiframework.sourceforge.net/

5.4.3 CA-Editor

The CA-Editor allows the user to construct and modify a constraint automaton visually. The user should be able to:

- add and remove states,
- add and remove transitions,
- modify state properties,
- modify transition properties.

Through the IT-GUI the user is able to open the CA-Editor. When the CA-Editor is started, it creates an empty CAM and shows the GUI CA-Editor. The sequence diagram depicted in figure 5.3 shows the interactions within the CA-Editor.

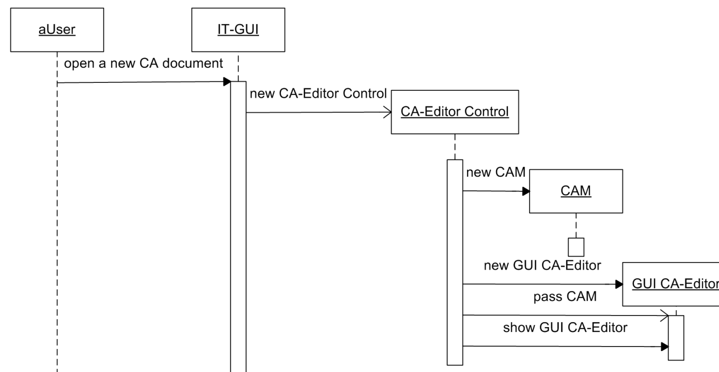

Figure 5.3: Sequence diagram of the user starting the GUI CA-Editor through the IT-GUI.

The GUI CA-Editor is the graphical user interface of the CA-Editor, therefore responsible for visualizing the CAM on the screen as a state diagram. All actions executed in the GUI-CA Editor are propagated to and handled by the CA-Editor Control.
The CA-Editor Control is responsible for that all actions executed in the GUI CA-Editor are reflected in the CAM, such that the visual representation of the constraint automaton is always consistent with the corresponding data representation in the CAM (figure 5.4).
The consistency should also hold when actions are performed on the CAM. Thus, if the CAM is changed programmatically, it notifies the GUI CA-Editor such that it updates the visual representation accordingly (figure 5.5).

Since a constraint automaton can be visualized as a graph, graph visualization libraries can be used to implement the GUI of the editor. Several non-commercial libraries exist:

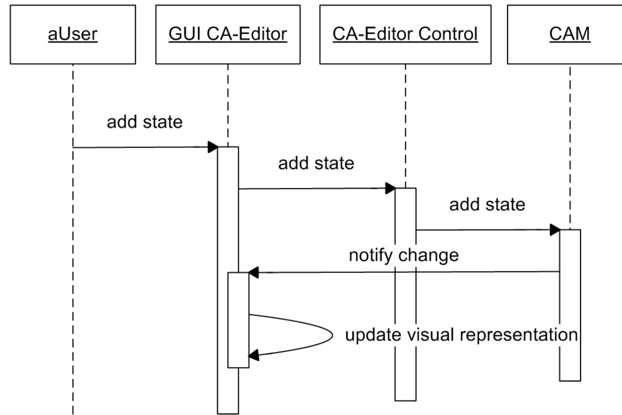

Figure 5.4: Sequence diagram of the user adding a state through the GUI CA-Editor.

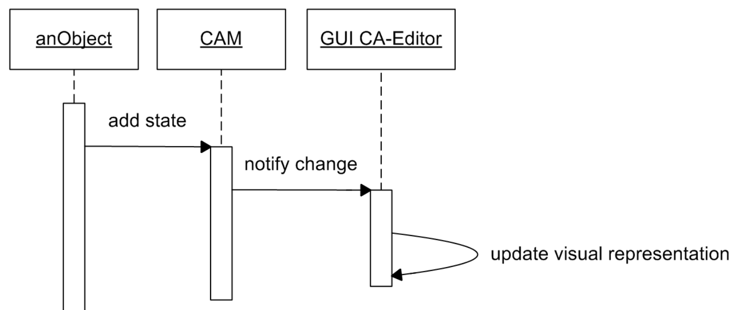

Figure 5.5: Sequence diagram of the CAM programmatically being changed.

- JGraph[2],
- Grappa [3] which is a Java library of a subset of GraphViz[4],
- JHotDraw[5].

Another option is to use an already existing automata editor/simulator and adapt it to our needs. The following editors/simulators exist for finite automaton that have a decent GUI: JFLAP[6] and Visual Automata Simulator[7].
JFLAP does not visualize multiple transitions between two states well, because the lines representing the transitions coincide with each other. Both JFLAP and Visual Automata Simulator do not have automatic layout generation for states and transitions. To adapt the current applications and implement these functionalities may prove difficult, especially because they were not built with these features in mind in the first place.
JGraph, Grappa and JHotDraw are all capable of visualizing multiple transitions between two states, since one can programmatically ensure that the lines representing the transitions do not coincide.
Another possibility is to use a generator. Grace[8] is a generator for graph editors in Java. One specifies a mapping from the model domain to the graph domain, for example "classes A and B represent nodes" and "classes X and Y represent edges". With such specifications Grace generates an editor. The generated editor itself makes sure that the corresponding model is always consistent with all the actions performed in the graphical user interface (and vice versa).
Adapting existing application may prove too difficult, because of some of the requirements, e.g. automatic layout generation, multiple document interface. The graph visualization libraries support the implementation of the GUI CA-Editor, but one still has to implement the CA-Editor Control, which can be quite time consuming. Therefore, Grace seems the most promising approach, because Grace is able to generate the GUI CA-Editor and the CA-Editor Control as well.

5.4.4 Layout Engine

The GUI CA-Editor depends on the Layout Engine to automatically layout the states and transitions. This feature is especially useful when loading a constraint automaton into the editor from a file, because the file format does not keep the screen positions.
The GUI CA-Editor passes the CAM to the Layout Engine. Using the topology stored in the CAM, the Layout Engine calculates and sets the positions of the states

[2] http://www.jgraph.com/
[3] http://www.research.att.com/ john/Grappa/
[4] http://www.graphviz.org/
[5] http://www.jhotdraw.org/
[6] http://www.cs.duke.edu/~rodger/tools/jflap/
[7] http://www.cs.usfca.edu/~jbovet/vas.html
[8] http://www.doclsf.de/grace/

and the transitions (figure 5.6).

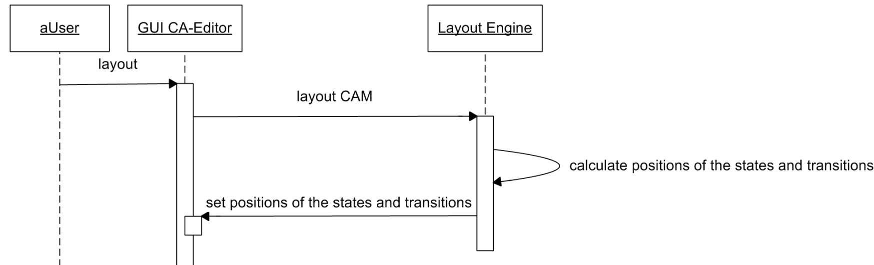

Figure 5.6: Sequence diagram of user starting the automatic layout.

Of all the graph visualization libraries discussed in section 5.4.3, only JGraph and Grappa have access to a layout engine. The JGraph library has a built-in layout engine. Grappa does not have a built-in layout engine, but it has methods to externally call the Graphviz layout engine. If we choose, for the implementation of the GUI CA-Editor, a library or solution other than JGraph and Grappa, then we could use the layout functionality of JGraph or Graphviz externally. This will however be less convenient and will result in a longer implementation time, because internal data structures have to be converted to and from the data structures of JGraph or Graphviz.

5.4.5 Load/Save Constraint Automaton Model (LSCAM)

The GUI CA-Editor depends on the LSCAM to load and save constraint automata to and from some persistent data storage (figure 5.7 and figure 5.8 respectively). The user has access to the LSCAM subsystem through the Integrated Tool GUI.

Since the existing tools for Reo and constraint automata use XML to provide the persistent data structure for constraint automata, it is required that the same Constraint Automaton XML (CA-XML) will be used within our MSc project. Besides this, XML has the following advantages:

- XML allows a developer to create his own data structures
- XML is text-based, thus easier to read and to document.
- tools, e.g. parsers, are widely available for handling XML
- it is possible to validate XML using for example an XML Schema

The programming language Java has defined a Java API for XML Processing, JAXP. Since Java 1.4 the Xerces implementation of JAXP is included in the Java

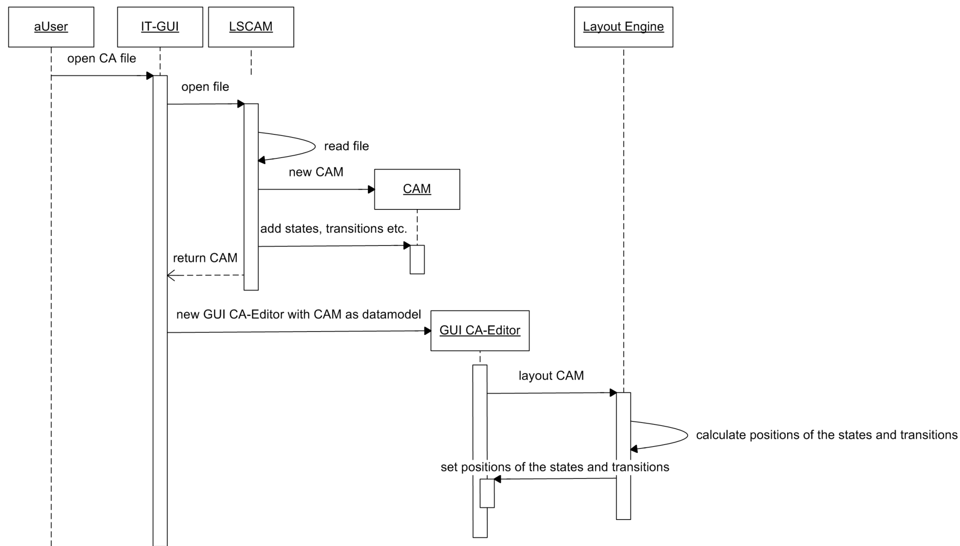

Figure 5.7: Sequence diagram of the user loading a constraint automaton from a file.

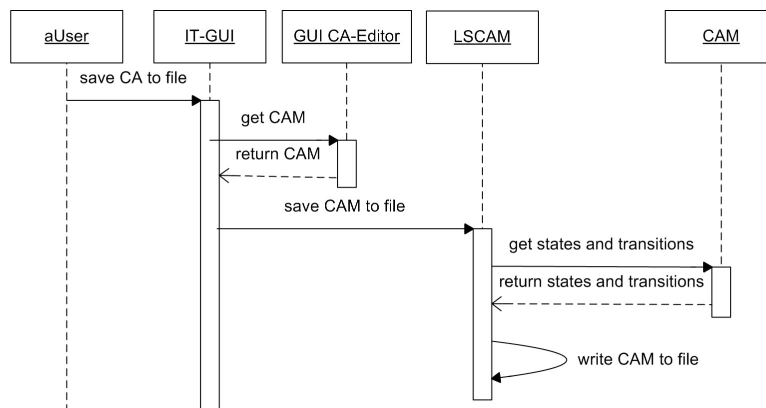

Figure 5.8: Sequence diagram of the user saving a constraint automaton to a file.

library. Therefore, no third party library is necessary to handle XML if Java is used.
For saving, the GUI CA-Editor passes the corresponding CAM of the constraint automaton it is currently visualizing to the LSCAM. Subsequently the LSCAM converts the CAM to CA-XML and saves the CA-XML to a file (figure 5.8).
By loading, the LSCAM uses the CA-XML file to build the CAM and returns this to the GUI CA-Editor, which then visualizes the CAM on the screen (figure 5.7).

5.4.6 TDS-Language Acceptor Simulator (TDSLAS)

The “TDS-Language Acceptor Simulator” behaves like a standard automata simulator, a language acceptor. The user creates or loads timed data streams and specifies through which ports they are fed to the simulator. The TDSLAS tries to accept a timed data tuple of the timed data streams and moves to the next constraint automaton state. A timed data tuple is accepted when the TDSLAS is able to make a transition with this tuple. If a timed data tuple is not accepted, the TDSLAS rejects the input and stops (figure 5.9).

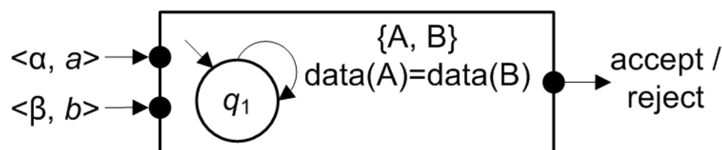

Figure 5.9: The TDS-language Acceptor Simulator.

The implementation of the TDSLAS is straight forward. Given a constraint automaton and timed data streams, the TDSLAS checks whether a timed run exists. At each time step t the TDSLAS creates a set of name-data-assignments of the time data tuples at time point t of the timed data streams. The TDSLAS accepts a time data tuple when an outgoing transition from the current state is possible with the corresponding set of name-data-assignments. If so, the TDSLAS updates the current state and continues with time step $t + 1$, otherwise the TDS-language is rejected.
The TDSLAS relies on the CA-Engine to provide the following functionalities:

- return all initial states,
- return all outgoing transitions from a specific state,
- evaluate (the data constraints of) a transition against a set of name-data-assignments.

The CA-Engine is discussed in further detail in section 5.4.13.
Through the IT-GUI the user is able to start the TDSLAS. First, the TDSLAS Control is created. Subsequently, the TDSLAS Control creates and shows the graphical

user interface of the simulator, the GUI TDSLAS. This process is shown in the sequence diagram depicted in figure 5.10.

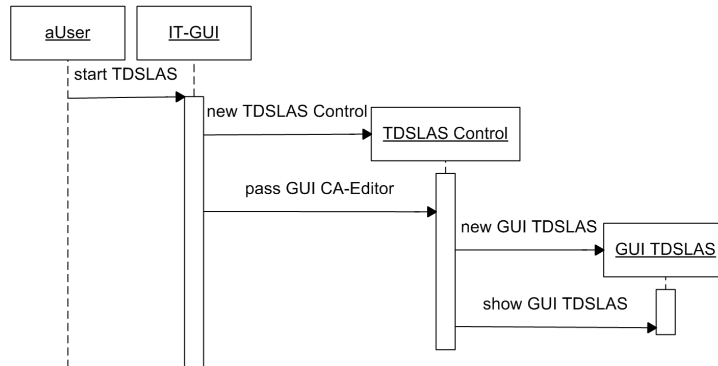

Figure 5.10: Sequence diagram of the user starting the TDSLAS.

All the actions that the user performs in the GUI TDSLAS are propagated to and handled by the TDSLAS Control. Through the GUI TDSLAS the user is able to edit (figure 5.11), save and load (figure 5.16 and 5.17) timed data streams.

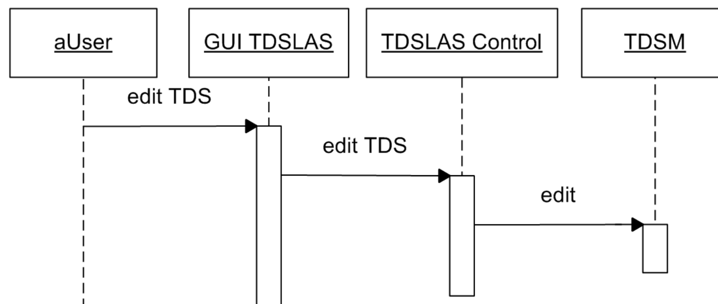

Figure 5.11: Sequence diagram of the user editing the timed data streams through the GUI TDSLAS.

The user is able to start the simulation and interact with the simulator through the GUI TDSLAS (figure 5.12).

5.4.7 Reo Connector Simulator with TDS (RCSwTDS)

The “Reo Connector Simulator with TDS” simulates a constraint automaton as a Reo connector where the input is specified by timed data streams. The behavior of the RCSwTDS resembles the behavior of a Reo connector in two ways.
First, the RCSwTDS does not reject the input if a timed data tuple is not accepted. The RCSwTDS delays not accepted timed data tuples and combines them with the timed data tuples of the next step. The combination of all time data tuples is the

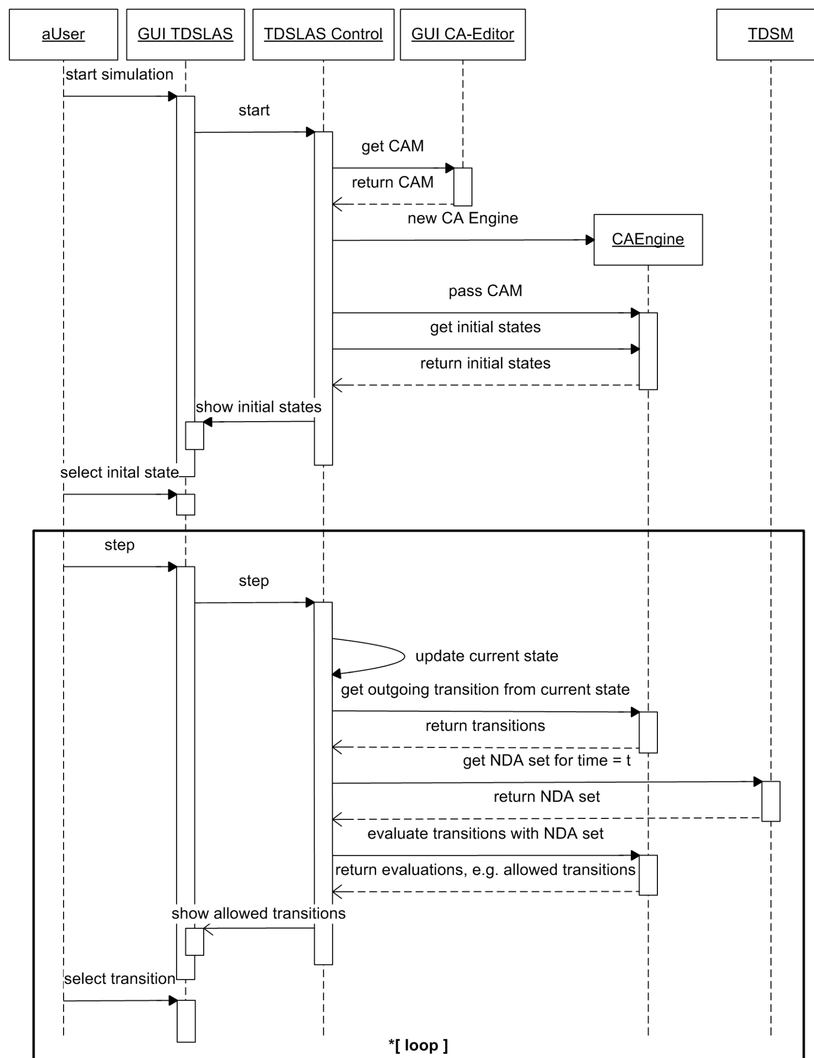

Figure 5.12: Sequence diagram of the user interacting with the TDSLAS.

'observed' time data tuple. This process continues until the 'observed' time data tuple becomes such that a transition can be made. This behavior is just like a Reo connector, which delays write or take operations until the conditions become such that these pending operations can be completed. Figure 5.13 illustrates a run of timed data streams for the constraint automaton shown in figure shown in figure 4.6.

Second, in the RCSwTDS the ports of a constraint automaton can act as input ports or as output ports. The user creates or loads timed data streams and specifies through which input ports they are fed to the RCSwTDS. Using the data at the input ports the RCSwTDS tries to make transitions and generates the data for the

	TDS				Observed TDS			
	A	**B**	**C**		**A**	**B**	**C**	
0	d_1			**0**	d_1			not accepted
1		d_2		**1**	d_1	d_2		not accepted
2			d_1	**2**	d_1	d_2	d_1	accepted: $data(A) = data(C)$
3				**3**		d_0		not accepted
4			d_2	**4**		d_2	d_2	accepted: $data(B) = data(C)$

Figure 5.13: A RCSwTDS run of timed data streams with the 'merger' constraint automaton.

output ports (figure 5.14).

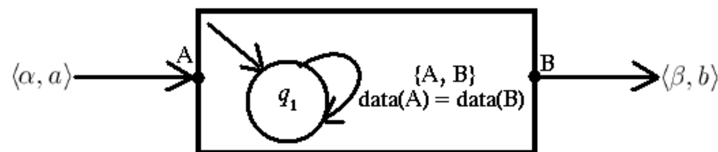

Figure 5.14: The Reo Connector Simulator with TDS.

The main difficulty with this simulator is that, in contrast to Reo connectors, constraint automata do not distinguish between input and output ports. The 'input' ports of the constraint automaton can be fed with timed data streams, but the timed data streams at the 'output' ports need to be generated by the RCSwTDS using the input data and the data constraints. However, the data constraint of a transition is a propositional formula that only states under which conditions a transition is allowed, but it does not say how data items are assigned/transferred from one port to another.

To overcome this difficulty, we use a different perspective to look at this problem. Timed data streams are given to the simulator as input, but the timed data streams can be incomplete, they contain 'gaps'. The simulator tries to complete the timed data stream using the data constraints to generate the missing data items. These gaps can be interpreted as pending take operations, while the data items filling the gaps can be seen as the data for completion of these take operations (figure 5.15).

The CA-Engine should provide the service to generate the missing data items during evaluation of the data constraints. Further, the RCSwTDS needs the same services from the CA-Engine as the TDSLAS. The CA-Engine is discussed in further detail in section 5.4.13.

The user is able to start the RCSwTDS through the IT-GUI. First, the RCSwTDS Control is created. Subsequently, the RCSwTDS Control creates and shows the GUI RCSwTDS. Through the GUI RCSwTDS the user is able to edit, save and load timed data streams, start and interact with the simulator. The TCSwTDS Control handles all the actions the user performs in the GUI RCSwTDS. The sequence

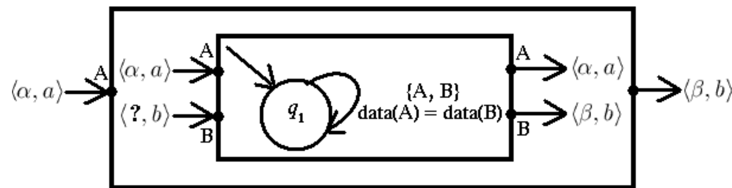

Figure 5.15: The RCSwTDS interpreting the completion of take operations in Reo as the completion of incomplete timed data stream in constraint automata.

diagrams of the RCSwTDS are omitted, because they are similar to the ones of the TDSLAS.

5.4.8 Timed Data Stream Model (TDSM)

The Timed Data Stream Model represents a timed data stream at the software level. Both TDSLAS and RCSwTDS depend on the TDSM. The TDSM provides the following methods for editing and accessing the timed data streams:

- add and remove time data tuples,
- return data from a time point t.

5.4.9 Load/Save Timed Data Stream (LSTDS)

The TDSLAS and the RCSwTDS depend on the Load/Save Timed Data Stream to load and save timed data streams to and from some persistent data storage. The user has access to the LSTDS subsystem through the GUI TDSLAS and GUI RC-SwTDS.
By loading, the LSTDS builds the TDSM using the TDS file and returns the TDSM to the TDSLAS or RCSwTDS (figure 5.16).
TDSLAS and RCSwTDS pass the TDSM to be saved to the LSTDS, which saves the TDSM to a file (figure 5.17).

Because it was required that XML will be used for constraint automata, XML will also be used to provide persistent data structure for timed data streams (TDS-XML).

5.4.10 Reo Connector Simulator with Components (RCSwC)

The Reo Connector Simulator with Components behaves as the RCSwTDS, but it does not use ‘predefined’ timed data streams as input. Instead the user attaches components to the simulator (figure 5.18). The components try to perform take and

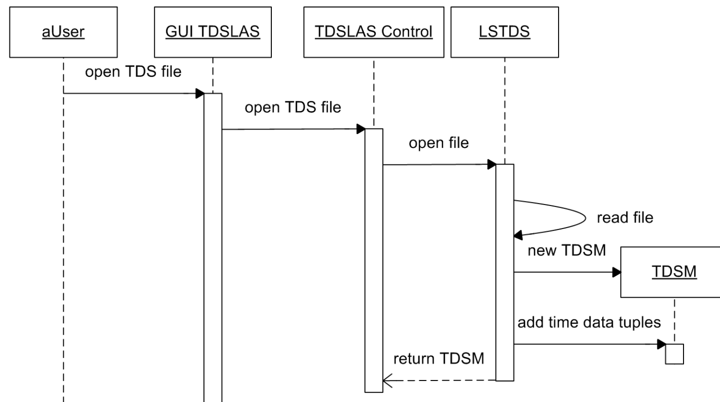

Figure 5.16: Sequence diagram of the user loading the timed data streams.

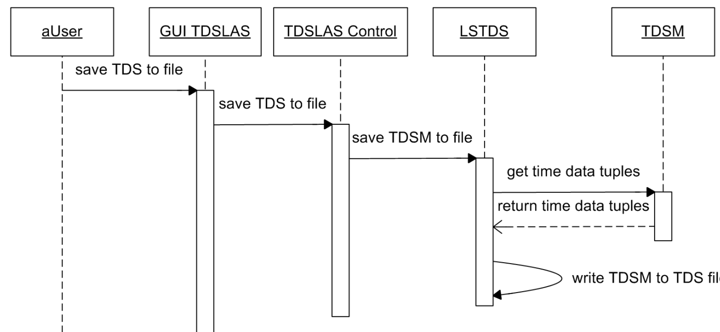

Figure 5.17: Sequence diagram of the user saving the timed data streams.

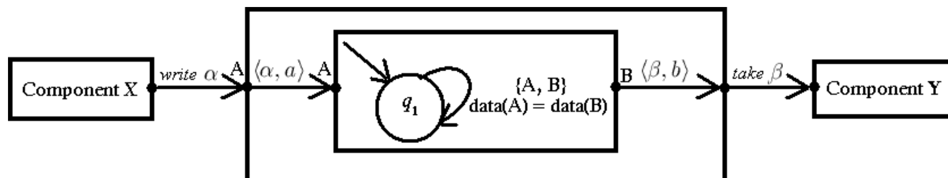

Figure 5.18: The Reo Connector Simulator with Components.

write operations on the ports. These pending operations can be interpreted as dynamically created 'timed data streams'. The RCSwC tries to complete these pending operations based on the underlying constraint automaton. Figure 5.19 depicts a sequence diagram that shows how the simulator interacts with the components.
Here the same problem arises as with the RCSwTDS, a constraint automaton does not distinguish between input and output ports. We use the same approach as with the RCSwTDS, interpreting the completion of take operations in Reo as the

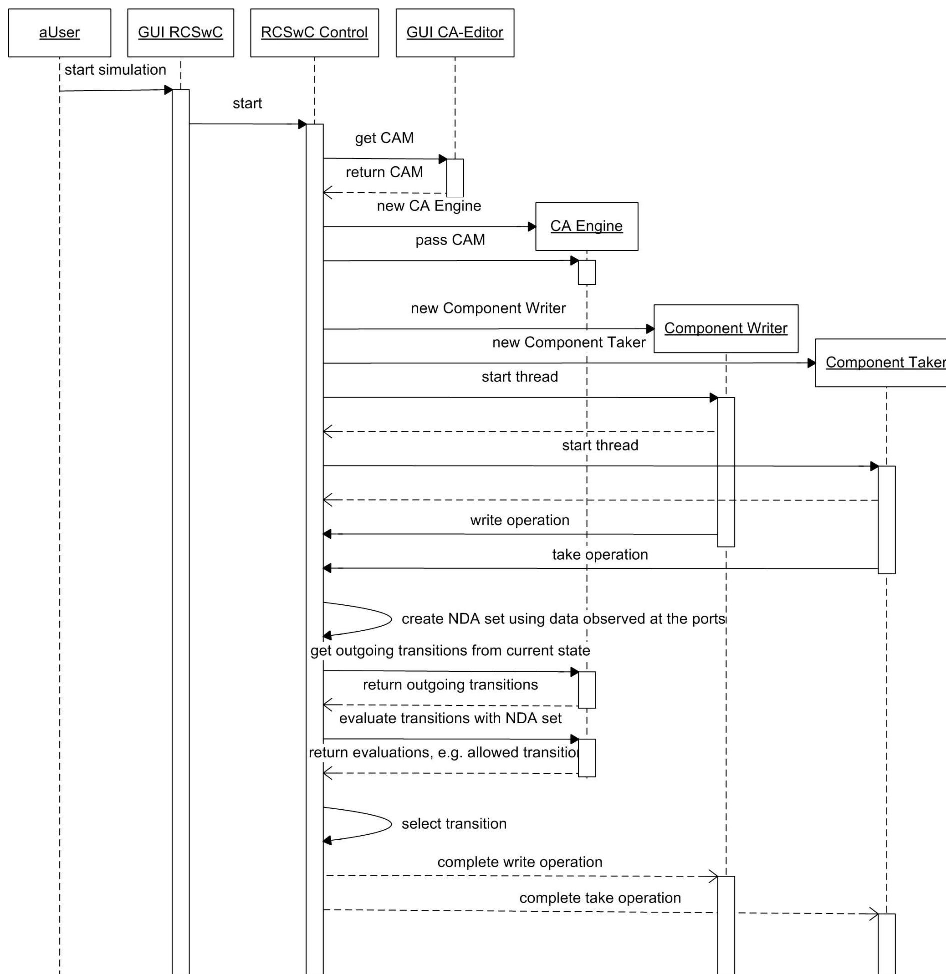

Figure 5.19: Sequence diagram of the interaction between the RCSwC and the components.

completion of incomplete timed data stream in constraint automata (figure 5.20).

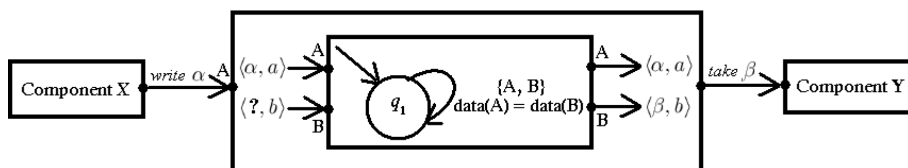

Figure 5.20: The RCSwC interpreting the completion of take operations of components in Reo as the completion of incomplete timed data stream in constraint automata.

Through the IT-GUI the user is able to start the RCSwC. First, the RCSwC Control is created. Subsequently, the RCSwC Control creates and shows the GUI RCSwC. Through the GUI RCSwC the user is able to load components and constraint automata, and connect component ports with constraint automaton ports. All the interactions of the user with the GUI RCSwC are propagated to and handled by the RCSwC Control.

5.4.11 Component

The components and component instances in the context of the RCSwC are respectively implemented as classes and objects in Java. An API should be provided that allows the objects to send and receive data to and from the simulated Reo connector.
Due to the compilation process of Java classes this approach is not flexible, therefore it is recommended to use a scripting language for implementing the behavior of the simulated components. The approach of using scripted components does not necessarily interfere with the approach of components written in Java. A Java object can be used as a wrapper around a scripting engine. This way it is possible to support 'Java'-components as well as scripted components. Additionally this approach enables the support of multiple scripting languages (by creating an appropiate wrapper for each language).
Many scripting engines written in Java exist. The following is by no means an exhaustive list:

- Rhino[9], a JavaScript engine,
- Jython[10], a Python engine for Java,
- Jess[11], a rule engine for the scripting language Jess for Java,
- Beanshell[12], a scripting engine for Java,
- JRuby[13], a Ruby engine for Java.

5.4.12 Load Component

The Load Component loads a component from some persistent data storage into the memory. The user is able to access Load Component through the GUI RCSwC (figure 5.21).

[9] http://www.mozilla.org/rhino/
[10] http://www.jython.org/
[11] http://herzberg.ca.sandia.gove/jess/
[12] http://www.beanshell.org/
[13] http://jruby.sourceforge.net/

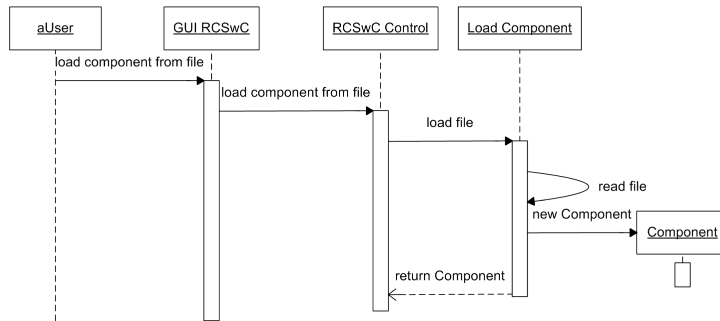

Figure 5.21: Sequence diagram of the user loading a component from a file.

5.4.13 Constraint Automaton Engine (CA-Engine)

The Constraint Automaton Engine depends on a CAM to provide the following functionalities needed by a simulator:

- return all the initial states,
- return all outgoing transitions from a certain state,
- evaluate data constraints with a set of name-data-assignments.

The CA-Engine is separated from the simulators such that different kind of simulators can be build using the same engine.

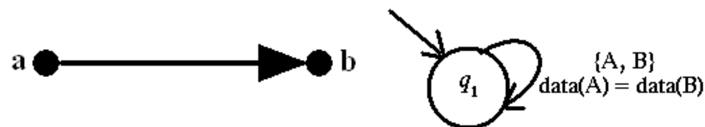

Figure 5.22: Sync channel and the corresponding constraint automaton

In the following we address the issue that constraint automata do not distinguish between input and output. Suppose the Sync channel is to be simulated by a constraint automaton (figure 5.22). If port A is associated with a timed data stream, then the timed data stream of port B should be generated by the constraint automaton. However, the data constraint $data\,(A) = data\,(B)$ is a propositional formula and not an assignment. Thus, the constraint automaton itself is not directional, as it does not say anything how data items flow through the Reo connector.
The constraint automaton engine should be able to handle this situation. If port A and B are active, but only at port A a data item is present, then a data item should be assigned to port B such that the data constraint $data\,(A) = data\,(B)$ is evaluated

to true. One apparent solution is to treat the data constraints in a Prolog like manner where free variables are bound to values when possible.
Several Java libraries exist that act as an interface to Prolog, e.g. InterProlog[14], K-Prolog[15], JPL[16]. Since we strive for a platform independent application, we do not prefer this approach, because the Prolog engine itself is not a Java application. Prolog libraries that are completely written in Java are, for example, JIProlog[17], jProlog[18] and Prolog Cafe[19]. JIProlog is not an option since it is a commercial library. jProlog is a very outdated library from 1997 and is not mature enough (some of the files are marked as version 0.1). The latest release of Prolog Cafe is version 0.9.1 on 24th February 2004. Therefore, Prolog Cafe seems the most promising library.

5.5 Requirements for the IT-GUI

We summarize the following refined requirements to the IT-GUI. The user should be able to:

- open *multiple* CA-Editors,
- start the TDSLAS,
- start the RCSwTDS,
- start the RCSwC.

5.6 Requirements for the CA-Editor

We summarize the following refined requirements to the CA-Editor. The user should be able to:

- add and remove states,
- add and remove transitions,
- modify state properties, e.g. state names,
- modify transition properties, e.g. data constraints,
- save and load constraint automata to and from XML,
- automatically layout the constraint automaton on the screen.

[14]http://www.declarativa.com/interprolog/
[15]http://www.kprolog.com/
[16]http://www.swi-prolog.org/packages/jpl/
[17]http://www.ugosweb.com/jiprolog/
[18]http://www.cs.kuleuven.ac.be/ bmd/PrologInJava/
[19]http://kaminari.scitec.kobe-u.ac.jp/PrologCafe/

5.7 Requirements for the TDSLAS

We summarize the following refined requirements to the TDSLAS. The user should be able to:

- create timed data streams,
- save and load timed data stream to and from TDS-XML files,
- simulate a constraint automaton with timed data streams.

5.8 Requirements for the RCSwTDS

We summarize the following refined requirements to the RCSwTDS. The user should be able to:

- create timed data streams,
- save and load timed data stream to and from TDS-XML files,
- simulate a constraint automaton with incomplete timed data streams
- generate the complete timed data streams.

5.9 Requirements for the RCSwC

We summarize the following refined requirements to the RCSwC. The user should be able to:

- load and execute components,
- connect component-ports to Reo-connector-ports,
- simulate a constraint automaton as a Reo connector while components are writing and reading to the simulated Reo connector.

Additionally, the following should be provided with the simulator:

- an Java API for implementing components,
- support for components written in a scripting language.

5.10 Requirements for the CA-Engine

We summarize the following requirements to the CA-Engine. The CA-Engine should be able to:

- return all initial states,
- return all outgoing transitions from a certain state,
- evaluate data constraints against a set of name-data-assignments in a Prolog like manner.

Chapter 6

Design and Implementation

In this chapter we discuss the design and implementation of the tool and elaborate on the technological choices we made. We present the software architecture of the tool using the Unified Modeling Language (UML). The number of classes and methods is usually too large to view in one class diagram. Therefore, we display the tool architecture using several class diagrams and in each class diagram we show only the relevant classes and methods.

6.1 Programming Language

We choose Java as the programming language, because applications written in Java can run on both MS Windows and Linux, the standard operating systems deployed within the CWI.
The choice for Java helps the later integration of the features of the already existing Reo and constraint automaton tools (also written in Java) with our tool.
We choose for Java version 1.5, because this version has many new expressive language features compared with the previous versions and better support for the development of multithreaded applications, as discussed in section 5.3.

6.2 Architectural Overview

In this section we present a high level introduction to the software architecture of the tool. The tool is decomposed in the following main packages:

- **cwi.reo.itgui**, contains the classes for the Integrated Tool GUI,
- **cwi.reo.caeditor**, contains the classes for the CA-Editor,
- **cwi.reo.tdslas**, contains the classes for the TDS Language Acceptor Simulator,
- **cwi.reo.rcswtds**, contains the classes for the Reo Connector Simulator with TDS,

- **cwi.reo.rcswc**, contains the classes for the Reo Connector Simulator with Components,
- **cwi.reo.caengine**, contains the classes for the CA-Engine.

These packages and their dependencies are shown in the UML package diagram in Figure 6.1. The decomposition of the tool into packages corresponds closely to the decomposition shown in figure 5.1.

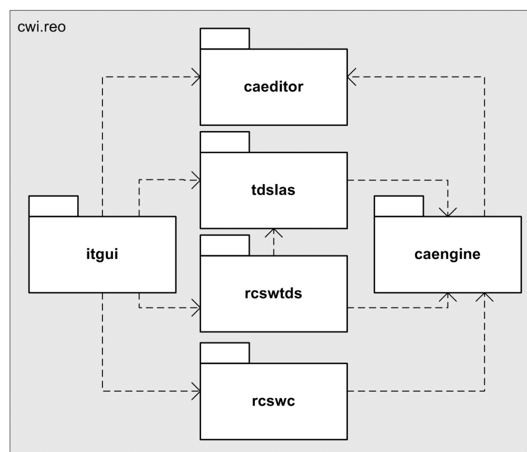

Figure 6.1: The UML package diagram of the tool.

6.3 CA-Editor

In this section we discuss how the CA-Editor is designed and implemented. The classes for the CA-Editor are located in the **cwi.reo.caeditor** package, which is again decomposed into several subpackages. The decomposition is depicted in the UML package diagram in figure 6.2.

- **cwi.reo.caeditor.cam**, contains the classes for the Constraint Automaton Model (CAM).
- **cwi.reo.caeditor.gui**, contains the classes for the GUI CA-Editor.
- **cwi.reo.caeditor.layoutengine**, contains the classes for the Layout Engine.
- **cwi.reo.caeditor.dcparser**, contains the classes for parsing strings representing data constraints.
- **cwi.reo.caeditor.namesparser**, contains the classes for parsing strings representing names (of a names set).

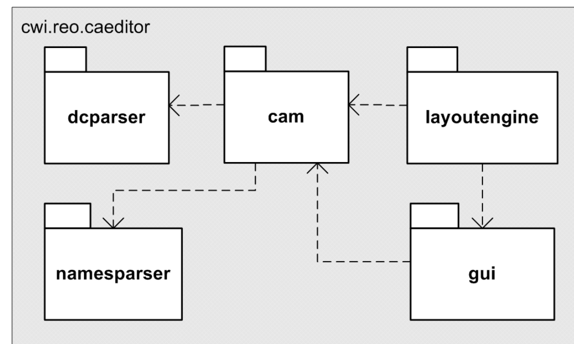

Figure 6.2: The UML package diagram of the CA-Editor.

6.3.1 CAM

In this subsection we discuss the design and implementation of the CAM. All the classes for the CAM are located in the **cwi.reo.caeditor.cam** package.

Constraint automaton

The data structures representing a constraint automaton are shown in the class diagram in figure 6.3. The following classes have been defined:

- **ConstraintAtomaton**, represents the overall constraint automaton,
- **Transition**, represents a transition,
- **State**, represents a state.

To be able to model a state as an initial state, we introduce the following classes:

- **InitialTransition**, each **State** object that is the target state of **InitialTransition** is an initial state,
- **InitialState**, acts as the source state for **InitialTransition**.

Figure 6.4 shows the relation between the components of a constraint automaton and the classes representing the constraint automaton.

We generalize the two state and two transition classes by introducing the abstract classes **AbstractState** and **AbstractTransition**. This provides **ConstraintAutomaton** with a generic view such that it can work with different state and transition classes without having to recognize the exact individual subtype.

Using this design for constraint automata, we can easily extend toward the parameterized constraint automaton. States in parameterized constraint automata have

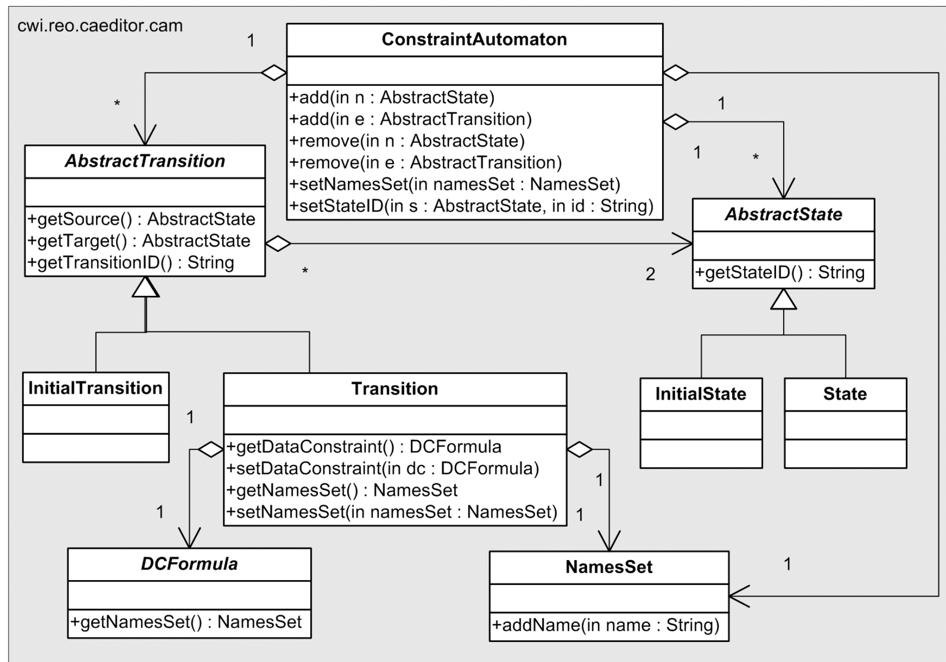

Figure 6.3: The class diagram of the data structures representing a constraint automaton.

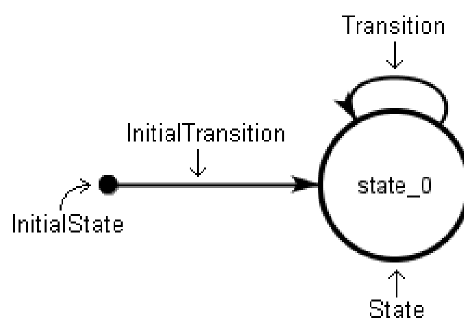

Figure 6.4: Relation between the components of a constraint automaton and the classes representing the constraint automaton.

state variables and transitions are labeled with assignments for these state variables. Extending the current design to support parameterized constraint automata can easily be realized by adding extra information to the classes.

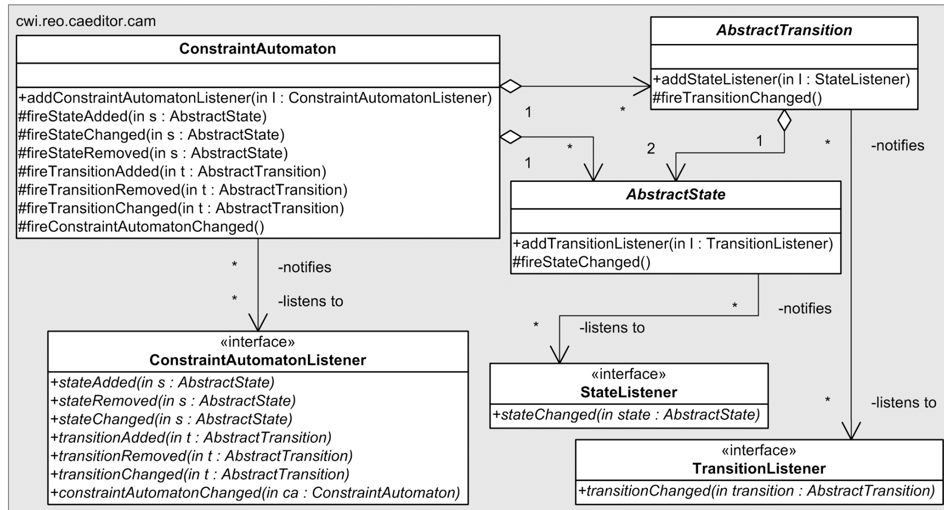

Figure 6.5: The class diagram of the constraint automaton classes and the listener interfaces.

The constraint automaton, transition and state classes implement the Observer design pattern[6], which allows them to notify others about changes, e.g. the addition or removal of states, the modification of transition properties. The IT-GUI, for example, needs to get notified of changes in the constraint automaton, because the save button (for saving constraint automata) becomes only enabled when the constraint automaton has been changed. The following listener interfaces are defined:

- **ConstraintAutomatonListener**, listener interface for **ConstraintAutomaton**,
- **TransitionListener**, listener interface for **Transition**,
- **StateListener**, listener interface for **State**.

The class diagram in figure 6.5 shows the relations between the constraint automaton classes and the listener interfaces. The sequence diagram in figure 6.6 illustrates the interaction between a **ConstraintAutomaton** object and a concrete **ConstraintAutomatonListener**.

Data Constraint

The data structures representing a data constraint are shown in the class diagram in figure 6.7. Each operator that can occur in a data constraint formula is represented

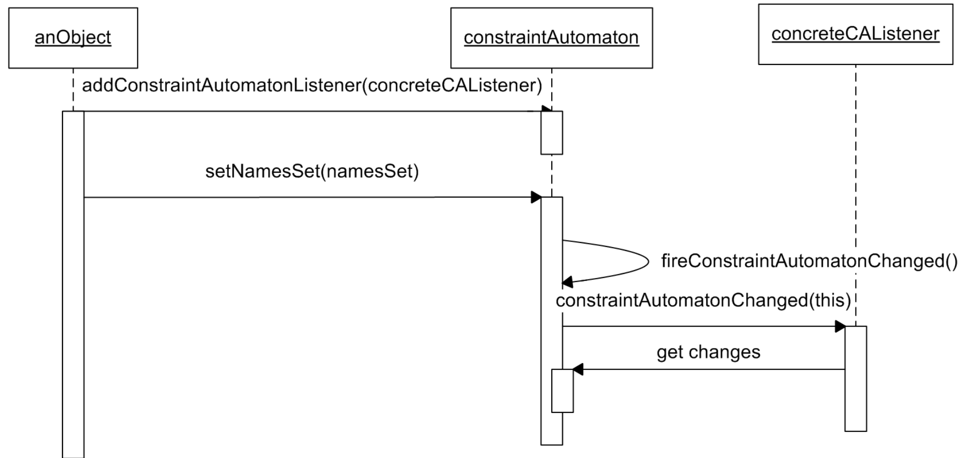

Figure 6.6: The sequence diagram showing the interaction between a **ConstraintAutomaton** object and a concrete **ConstraintAutomatonListener**.

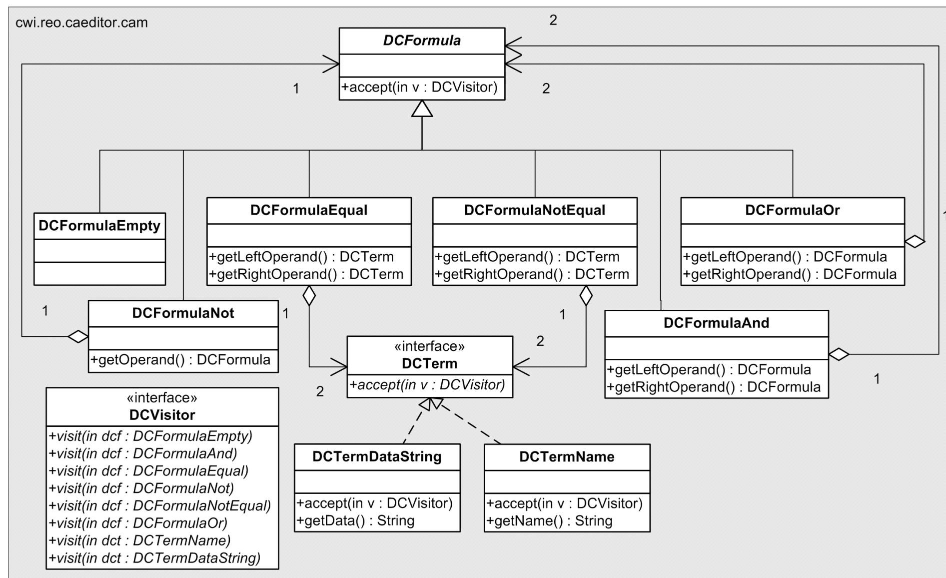

Figure 6.7: The class diagram of data constraint formulae.

by a class:

- **DCFormulaEqual**, represents the equal operator,
- **DCFormulaNotEqual**, represents the not equal operator,
- **DCFormulaNot**, represents the not operator,
- **DCFormulaAnd**, represents the and operator,
- **DCFormulaOr**, represents the or operator.

There is separate class for representing the empty data constraint formula, **DCFormulaEmpty**.
The equal and not equal operators have operands that are terms. The following types of terms have been defined:

- **DCTermName**, represents the name of a name-data-assignment,
- **DCTermDataString**, represents a data item of type string.

Data constraint formulae can be both individual data constraint formulae (e.g. **DCFormulaEqual**) and compositions of data constraint formulae (e.g. **DCFormulaAnd**). By applying the Composite design pattern[6], introducing the abstract class **DCFormula**, we can treat all data constraint formula objects in the composite structure uniformly, since the abstract class allows us to ignore the difference between individual and compositions of data constraint formulae.

To be able to add new types of terms easily in the future, we define the interface **DCTerm** that is placed between **DCFormulaEqual**, **DCFormulaNotEqual** and concrete term classes. The interface **DCTerm** provides a generic view, which enables introducing new types of terms without having to recognize the exact individual subtype.

With these classes an abstract syntax tree can be build that represents a data constraint formula. An example is shown in figure 6.8.

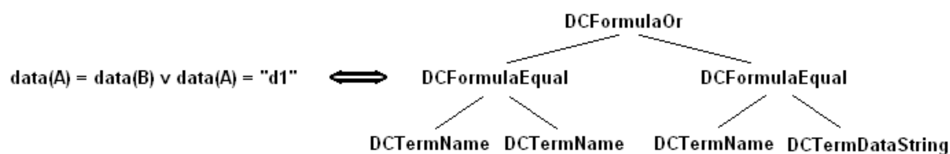

Figure 6.8: The abstract syntax tree of a data constraint formula.

To perform operations on the abstract syntax tree, we apply the Visitor design pattern[6]. This design pattern offers the flexibility of defining new operations

over a structure without changing the structure itself. The **DCVisitor** defines the interface of a visitor for a data constraint formula. Figure 6.9 depicts the sequence diagram showing how a concrete **DCVisitor** object traverses an abstract syntax tree of **DCFormula** and **DCTerm** objects.

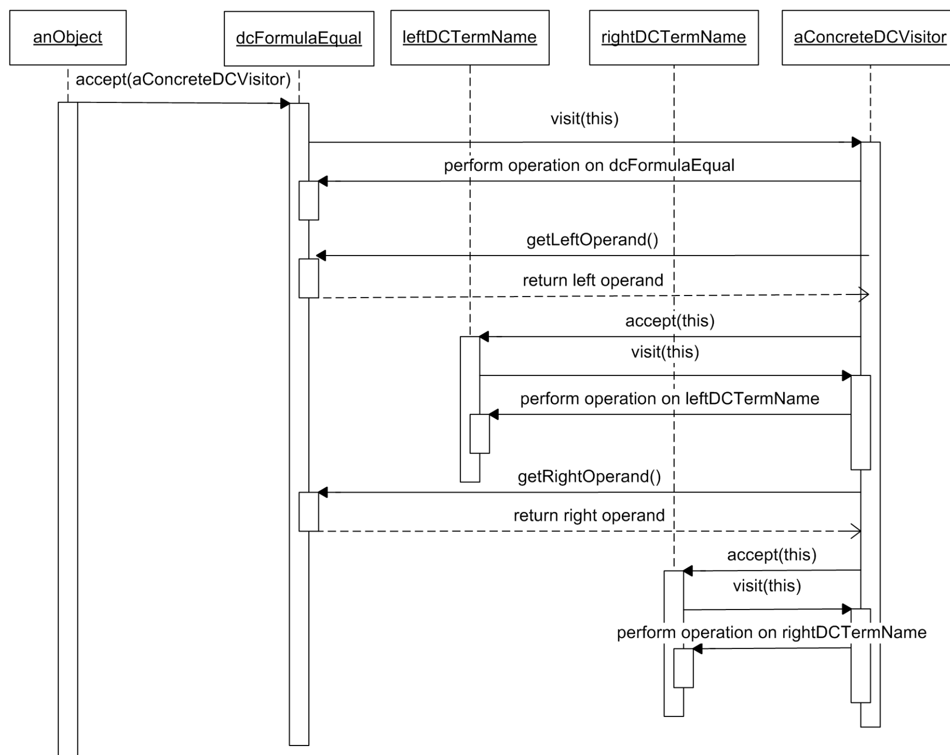

Figure 6.9: The sequence diagram showing the interaction between a **DCVisitor** and some **DCFormula** and **DCTerm** objects.

Names set

The class **NamesSet** represents a set of names. It is part of **ConstraintAutomaton** and **Transition**.

6.3.2 LSCAM

The class responsible for loading and saving a constraint automaton to and from a CA-XML file is **CAXMLFileIOModule**. It has access to all the classes of the CAM. The structure and relations are depicted in figure 6.10. The **CAXMLFileIOModule** class is part of the **cwi.reo.caeditor.cam** package.

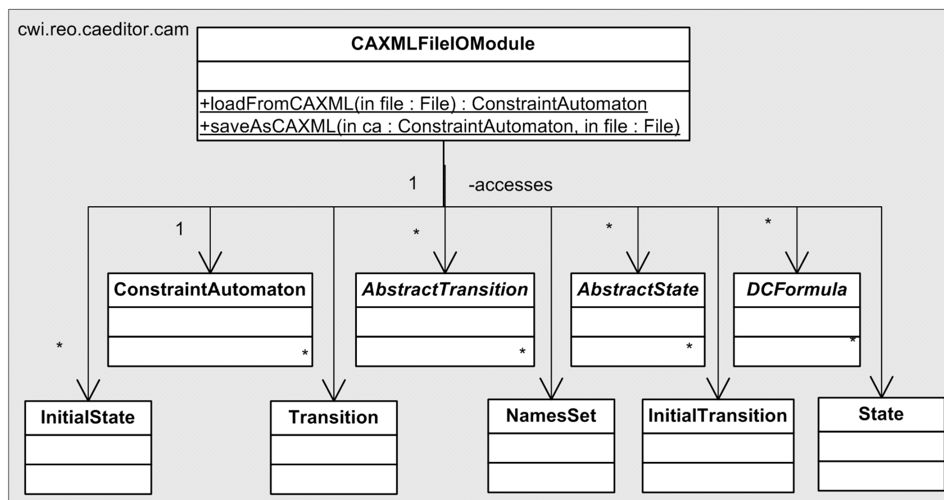

Figure 6.10: The class diagram of the **CAXMLFileIOModule**

6.3.3 GUI CA-Editor and CA-Editor Control

In this subsection we discuss how the GUI CA-Editor and CA-Editor Control are designed and implemented. All the classes, interfaces and subpackages implementing the GUI CA-Editor and the CA-Editor Control are located in the **cwi.reo.caeditor.gui** package:

- **graceeditor** package,
- **gracefigures** package,
- **ConstraintAutomatonView** class,
- **ConstraintAutomatonViewListener** interface,
- **PropertySheet** class.

The class diagram in figure 6.11 shows an overview of **cwi.reo.caeditor.gui** package.

graceeditor package

For the implementation of GUI CA-Editor and CA-Editor Control we choose for Grace, because Grace is able to generate both on the basis of a specification (as discussed in section 5.4.3). Hence, Grace generates a graph editor that is able to:

- visualize a constraint automaton on the screen,
- maintain consistency between the visual representation and the data representation of a constraint automaton.

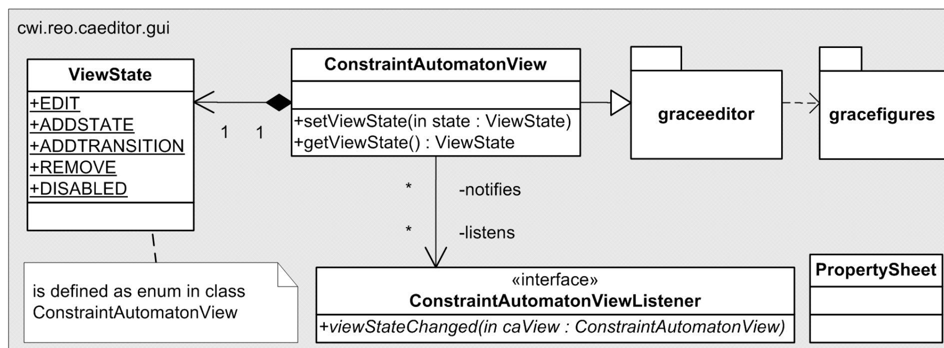

Figure 6.11: The class diagram of GUI CA-Editor

The specification consists of three parts. First, one has to describe the mapping from the application domain to the graph domain, thus the mapping from classes to graph, nodes and edges. In order for the Grace editor classes to recognize the roles of the application domain classes as such and to interact with them, the application domain classes need to implement the interfaces **Graph**, **Node**, **Edge**. Figure 6.12 shows how this is done for the classes of the CAM.
Second, the presentation style needs to be specified, a description how each component of the graph domain is visualized on the screen (figure 6.13). Grace provides the developer with a set of *figures*, e.g. boxes and circles for nodes, straight lines and Bezier curves for edges. This set also contains figures for displaying text, e.g. labels for nodes and edges.
Third, the interactions must to be specified, describing the behavior of the editor when it interacts with the user. For example, clicking with the mouse on a node, subsequently dragging to and releasing on another node should create an edge between those nodes. The complete specification language of Grace can be found in [7]. All the Grace generated classes are placed in the **graceeditor** package.

gracefigures package

The basic set of figures that Grace provides for specifying the presentation style does not suffice to express constraint automata. For example, a transition can have multiple data constraint formulae, which we want to visualize as an edge label where each data constraint formula is on a separate line, while Grace only supports single line edge labels. Therefore, we extend this set with the package **gracefigures** containing the following custom figures:

- multiline text label for nodes,
- multiline text label for edges,
- concatenated Bezier curves figure for edges.

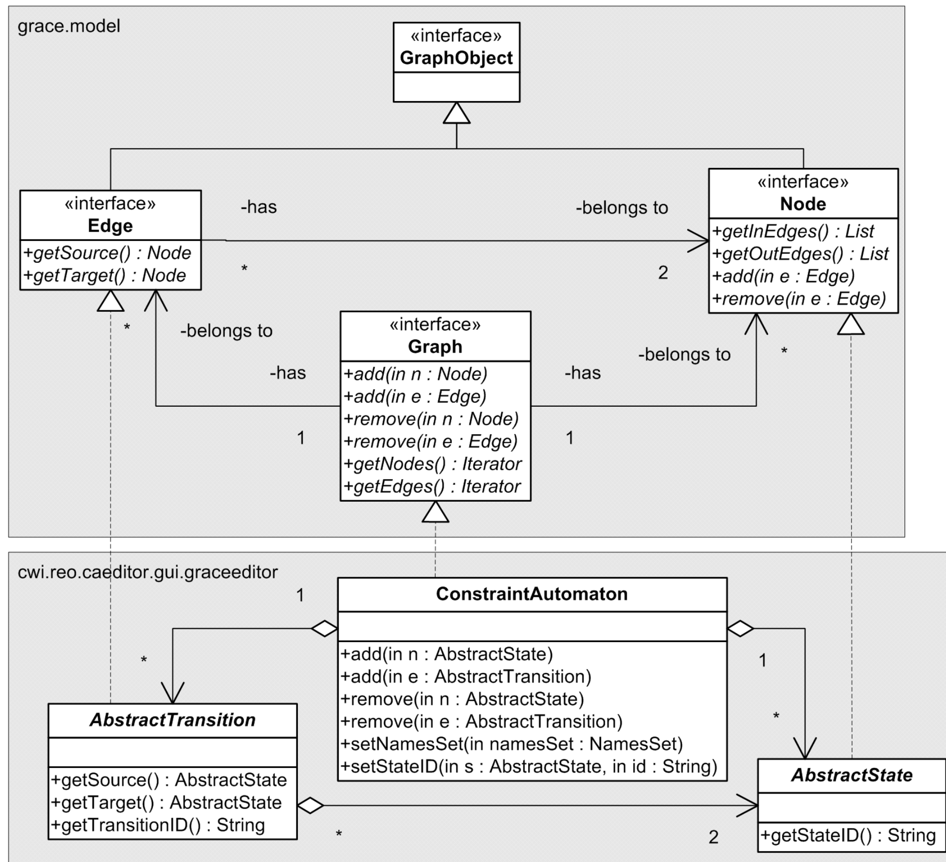

Figure 6.12: Integration of the application domain classes with the Grace editor classes.

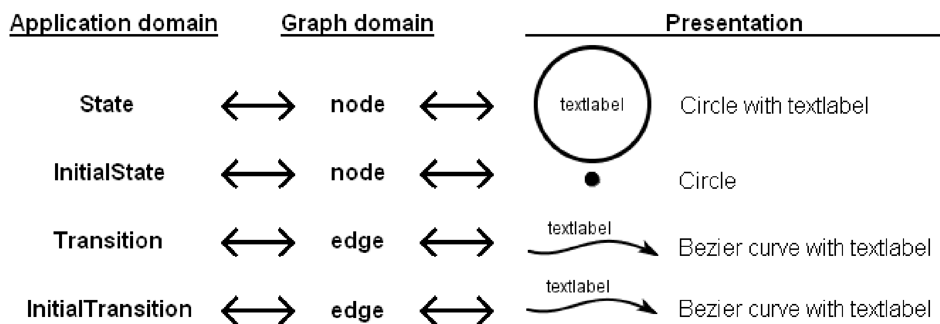

Figure 6.13: The specification of the GUI CA-Editor

Grace provides a Bezier curve figure for edges, but this figure only consists of one single Bezier curve. The custom edge figure that we implement supports one Bezier curve, but also supports multiple concatenated Bezier curves. The concatenated Bezier curves figure for edges is added, because we plan to use Graphviz as the layout engine and Graphviz describes the layout of an edge by one or more Bezier curves.

ConstraintAutomatonView

The Grace generated editor does not have scrolling abilities. The **ConstraintAutomatonView** class extends the Grace generated editor and adds scrolling abilities to the editor. Thus, if part of the visualized constraint automaton is placed outside the viewing area, then a scrollbar appears, which the user can move to change the viewing area.

ConstraintAutomatonViewListener

The **ConstraintAutomatonView** can be in different states, e.g. 'add state', 'add transition' or remove state. To allow **ConstraintAutomatonView** to notify others about changes of its state, we apply the Observer design pattern[6], introducing the listener interface **ConstraintAutomatonViewListener**. Mainly the IT-GUI will listen to these changes to update itself to show the user in which state the editor currently is, e.g. whether it is in an "add state" or "add transition" state.

PropertySheet

The **PropertySheet** class is a panel that can be used to show the properties of constraint automata, states and transitions, e.g. state name, transition data constraints. Through the **PropertySheet** the user can also modify these properties.

6.3.4 Input Parsers

The **DCParser** and **NamesParser** package contains all the classes for parsing data constraints and names respectively. Parsing is the conversion of input to internal data structures. In our case we need to parse:

1. a data constraint string to an abstract syntax tree of **DCFormula**,
2. a names string to a **NamesSet** object.

Instead of implementing these two parsers, we use a parser generator, a tool that reads a description of a language and converts it to a program that can read and analyze that language. JavaCC[1] is a popular parser generator for use with Java applications. In addition to the description of the language, the actions that need to

[1]http://javacc.dev.java.net/

```
digraph a_graph
{
    rankdir=LR;
    node [shape = circle];
    A -> B;
}
```

(a)

```
digraph a_graph
{
    graph [rankdir=LR];
    node [label="\N", shape=circle];
    graph [bb="0,0,112,38"];
    A [pos="19,19", width="0.53", height="0.53"];
    B [pos="93,19", width="0.53", height="0.53"];
    A -> B [pos="e,74,19 38,19 46,19 55,19 64,19"];
}
```

(b)

Figure 6.14: **(a)** *DOT* file without layout information. **(b)** *DOT* file with layout information.

be taken when a certain sequence is recognized should be specified. These actions create a abstract syntax tree of **DCFormula** (1) or a **NamesSet** object (2).

6.3.5 Layout Engine

For the implementation of the GUI CA-Editor, we use a graph visualization library that has no built-in layout engine. Therefore, we use Graphviz as the external layout engine. Graphviz is open source graph visualization software, which has several graph layout programs. We make use of the graph layout program *dot*, which makes layouts for directed graphs. *dot* takes description of a graph in a simple text language, called the *DOT* language, as input. Subsequently, *dot* generates the layout for this graph by reproducing the input along with layout information (figure 6.14).
Grappa is a Java graph drawing package, which can be seen as a port of a subset of Graphviz to Java. Grappa is able to read graphs described in the *DOT* language.

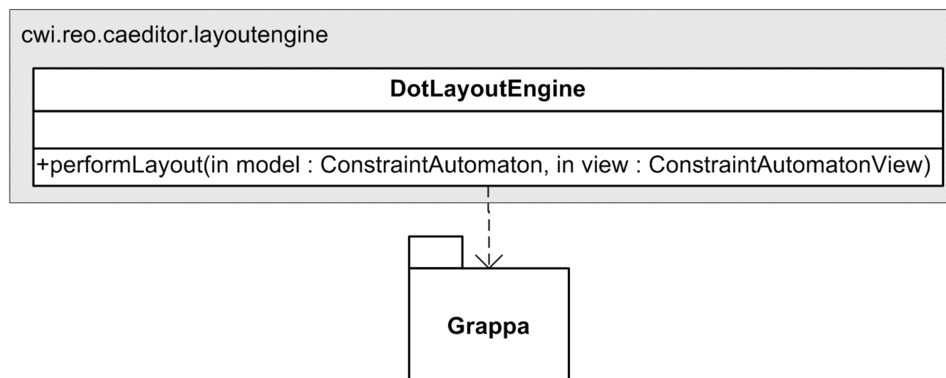

Figure 6.15: The class diagram of the Layout Engine.

Figure 6.15 shows the class diagram of the Layout Engine. The class **DotLayoutEngine**, located in the package **cwi.reo.caeditor.layoutengine**, is the implemented Layout Engine that makes use of the *dot* layout program of Graphviz. Figure 6.16 shows how the **DotLayoutEngine** globally works:

1. translate the CAM data structures to a *DOT* file,
2. call the *dot* layout program which adds layout information to the *DOT* file,
3. read the *DOT* file using Grappa,
4. extract the layout information from the Grappa data structures and apply it to the layout of the states and transitions of the constraint automaton in the GUI CA-Editor.

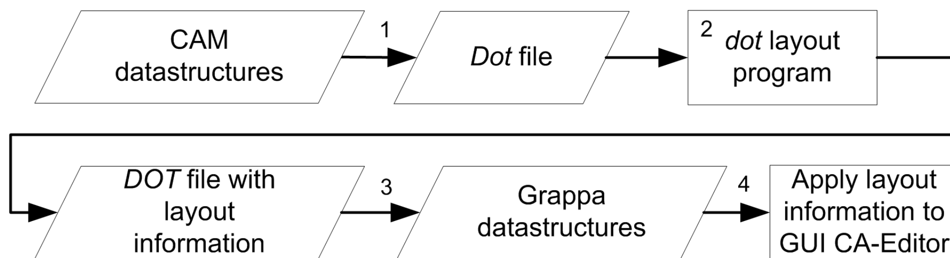

Figure 6.16: The layout process of the **DotLayoutEngine**

6.4 IT-GUI

For the implementation of the Integrated Tool GUI we use the Java MDI Application Framework. This framework provides a skeleton for MDI applications, therefore reducing the development time of a MDI application.
The use of the Java MDI Application Framework is based on the Factory Method design pattern [6]. In this design pattern a framework defines abstract classes and also maintains the relationships between objects. To create an application-specific implementation, one just has to subclass the abstract classes of the framework.
Figure 6.17 shows how the subclassing is done for our tool. These classes are located in the **cwi.reo.itgui** package. The description of the classes is given below:

- **CAEditorMain**, the main class of the tool,
- **CAEditorMainWindow**, the main window of the tool,
- **CAEditorCommands**, handles all commands which can be triggered by users,

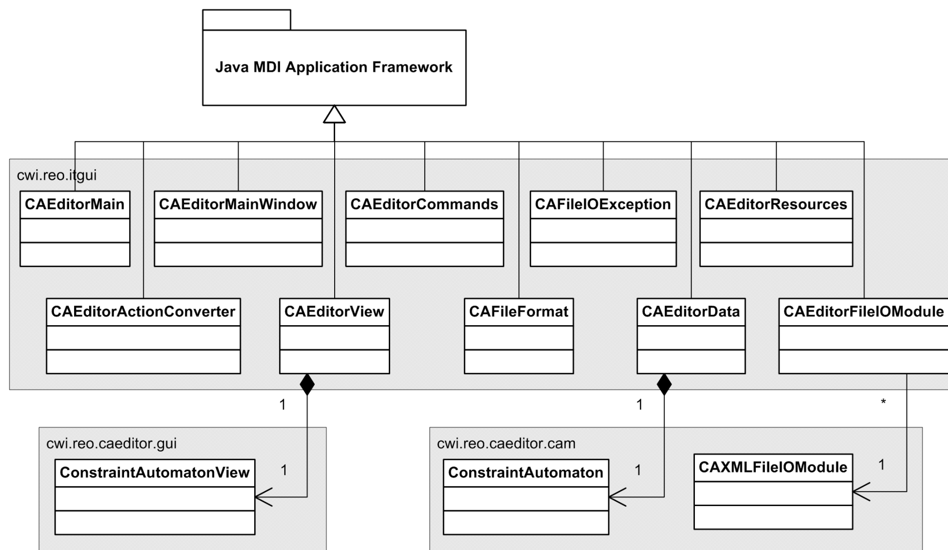

Figure 6.17: The class diagram of the Integrated Tool GUI.

- **CAEditorData**, contains the data of a document, thus a constraint automaton (therefore it contains the **ConstraintAutomaton** class),
- **CAEditorView**, responsible for viewing the data of a document (therefore it contains the **ConstraintAutomatonView** class),
- **CAEditorActionConverter**, responsible for handling the undo, redo, copy, cut and paste command,
- **CAEditorResources**, manages all the resources of the tool (e.g. icons, property file),
- **CAEditorFileIOModule**, responsible for loading and saving (therefore it depends on the **CAXMLFileIOModule**),
- **CAFileFormat**, represents the file format,
- **CAFileIOException**, the exceptions which can occur during loading or saving.

Figure 6.18 illustrates how the IT-GUI looks like and shows which classes the components of the IT-GUI correspond to.

6.5 CA-Engine

In this section we discuss the implementation of the CA-Engine. The first two requirements of the CA-Engine as described in section 5.10 are quite trivial, easy

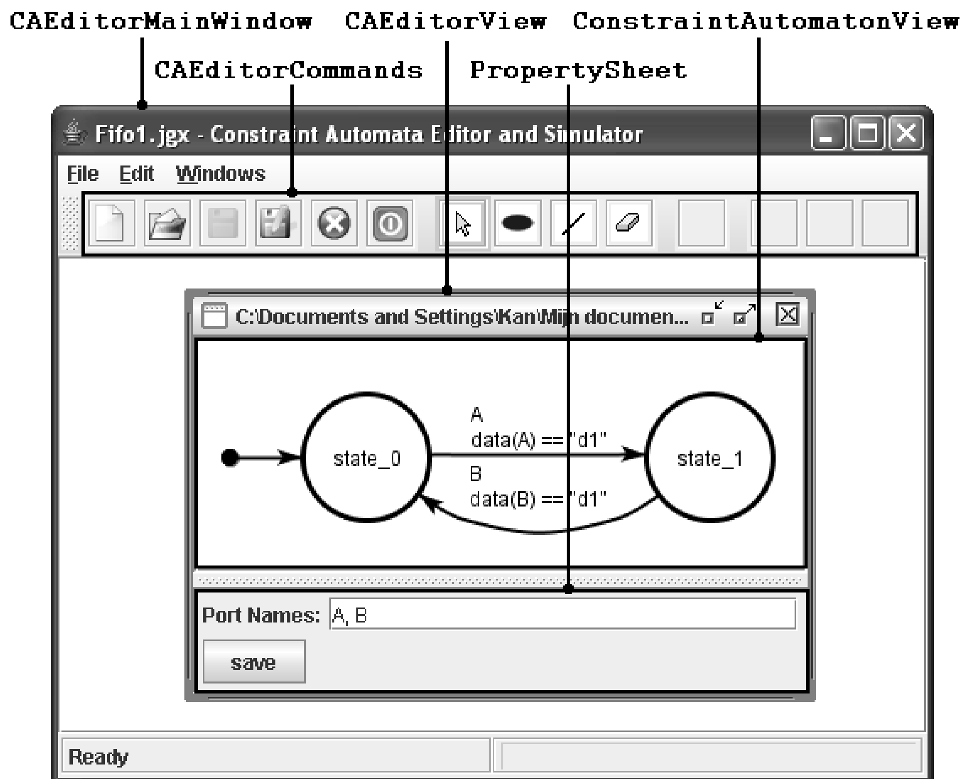

Figure 6.18: The screenshot of the IT-GUI showing the correspondence with the classes.

to realize. However, the third requirement, evaluating data constraints in a Prolog like manner, is hard to implement from scratch.
To implement the third requirement we use Prolog Cafe, a Prolog to Java translator system. It is able to convert Prolog source code to Java source code. Hence, we create a Prolog program that evaluates data constraints against a set of name-data-assignments, while binding free variables to values when possible. With Prolog Cafe we translate this Prolog program to a Java program, which we can use in the tool.
The classes, interfaces and subpackages of the CA-Engine are located in the **cwi.reo.caengine** package (figure 6.19):

- **CAEngine**, implements the requirements of CA-Engine as defined in section 5.10,
- **NameDataAssignment**, represents a name-data-assignment,
- **NDASet**, represents a set of name-data-assignments,
- **Data**, the interface for data objects used in **NameDataAssignment**,
- **DataWrapper**, which implements the **Data** interface, can be used to wrap an object such that the object can be used in **NameDataAssignment**,
- **PrologDCChecker**, checks data constraints using the to-Java-translated Prolog program,
- **prologdcchecker** package, contains all the to-Java-translated source code of the Prolog program.

6.6 TDSLAS

In this section we discuss the design and implementation of the TDSLAS. Using the requirements of the TDSLAS described in section 5.7, the TDSLAS can be decomposed into the following:

- a timed data stream editor part (TDS-Editor), where the user is able to edit, save and load timed data stream,
- a simulator part, where the user simulates a constraint automaton with timed data streams.

In the next subsections the TDS-Editor and the simulator part are discussed separately.

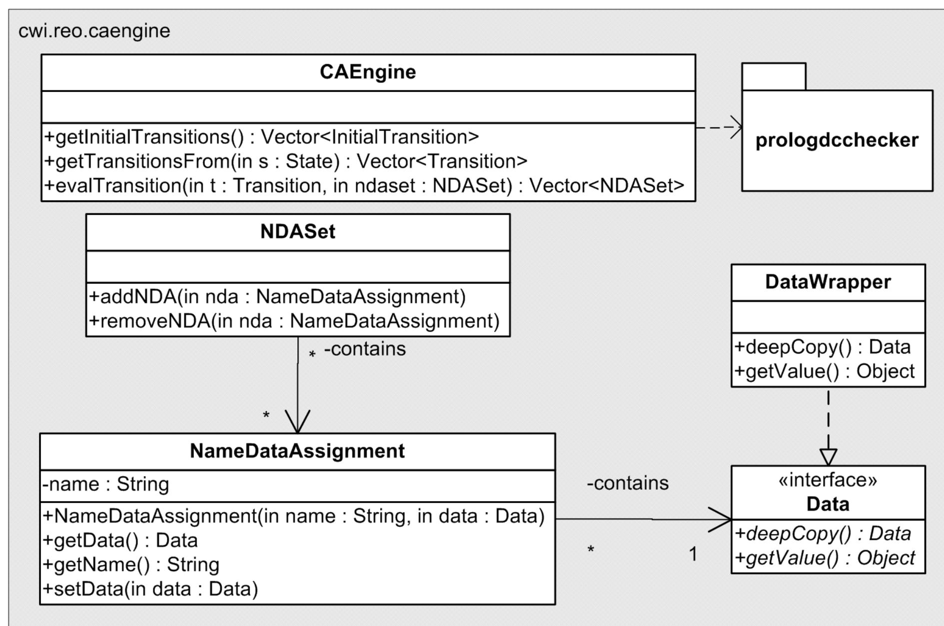

Figure 6.19: The class diagram of the CA Engine.

6.6.1 Model-View-Controller Design Pattern

For the design and implementation of the GUI we often apply the Model-View-Controller (MVC) design pattern[6]. The MVC design pattern decomposes an application in the following three objects:

- the model, which contains the data of the application,
- the view, responsible for displaying the model on the screen,
- the controller, which handles the interactions the user performs with the application and changes the data of the model accordingly.

The MVC decouples the model and view by applying the Observer design pattern[6], where the view is the observer of the model. Whenever the data of the model changes, it notifies its observers. Figure 6.20 illustrates the basic Model-View-Controller relationships.

6.6.2 TDSM and LSTDS

Since the TDSLAS acts as a language acceptor simulator, it only needs to simulate a constraint automaton with timed data streams where the data consists of strings, *timed stringdata streams*. The data structure representing such a timed stringdata stream is the **TimedStringDataStream** class. The **TSDSXMLFileIOMod-**

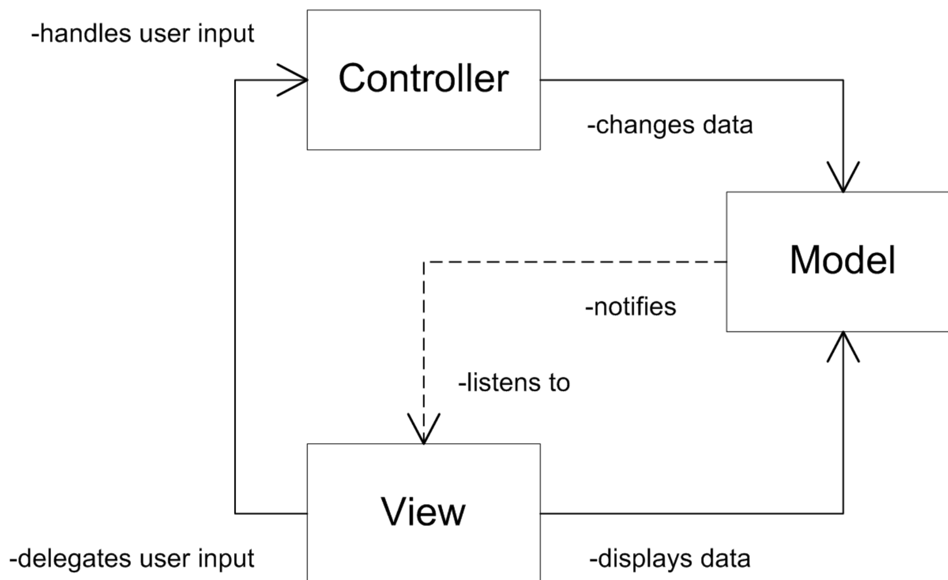

Figure 6.20: The basic Model-View-Controller relationships.

ule class handles the loading and saving of a timed stringdata stream from and to a TSDS-XML file. The class diagram is shown in figure 6.21.

6.6.3 GUI TDS-Editor

Through the GUI TDS-Editor the user is able to edit, save and load timed data streams. In the GUI TDS-Editor a table is shown where each column of the table represents a timed stringdata stream associated with a constraint automaton port. A screenshot of the GUI TDS-Editor is shown in Figure 6.22.
For the GUI TDS-Editor we apply the MVC design pattern[6]. Figure 6.23 shows the class diagram of the TDS-Editor and which part of the MVC design pattern the classes correspond to.

- **PortsTimedStringDataStreams**, associates timed stringdata streams with ports.
- **TimedStringDataStreamsEditorModel**, the model of the TDS-Editor.
- **TimedStringDataStreamsEditorModelListener**, the listener interface for **TimedStringDataStreamsEditorModel**.
- **PortsTimedStringDataStreamsTableModel**, the adapter class between **PortsTimedStringDataStreams** and the table which shows the ports and the associated timed string data streams.
- **TimedStringDataStreamsEditorView**, the view of the TDS-Editor.

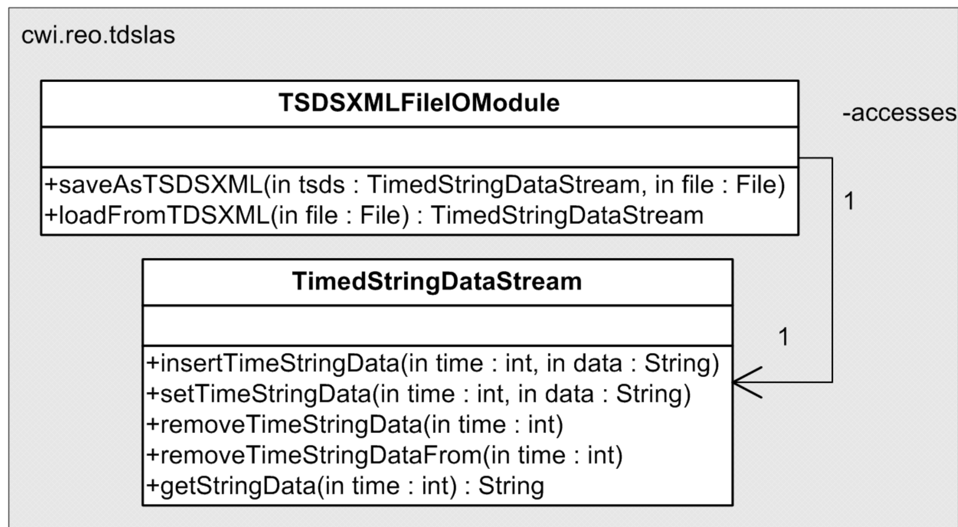

Figure 6.21: The class diagram of **TimedStringDataStream** and **TSDSXML-FileIOModule**.

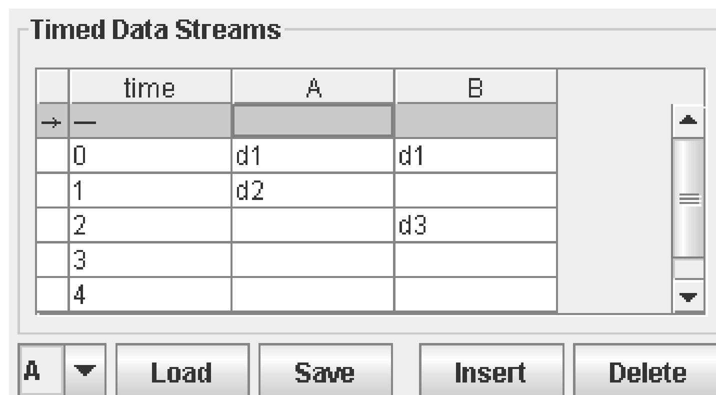

Figure 6.22: Screenshot of the GUI TDS-Editor.

- **TimedStringDataStreamsEditorControl**, the controller of the TDS-Editor.

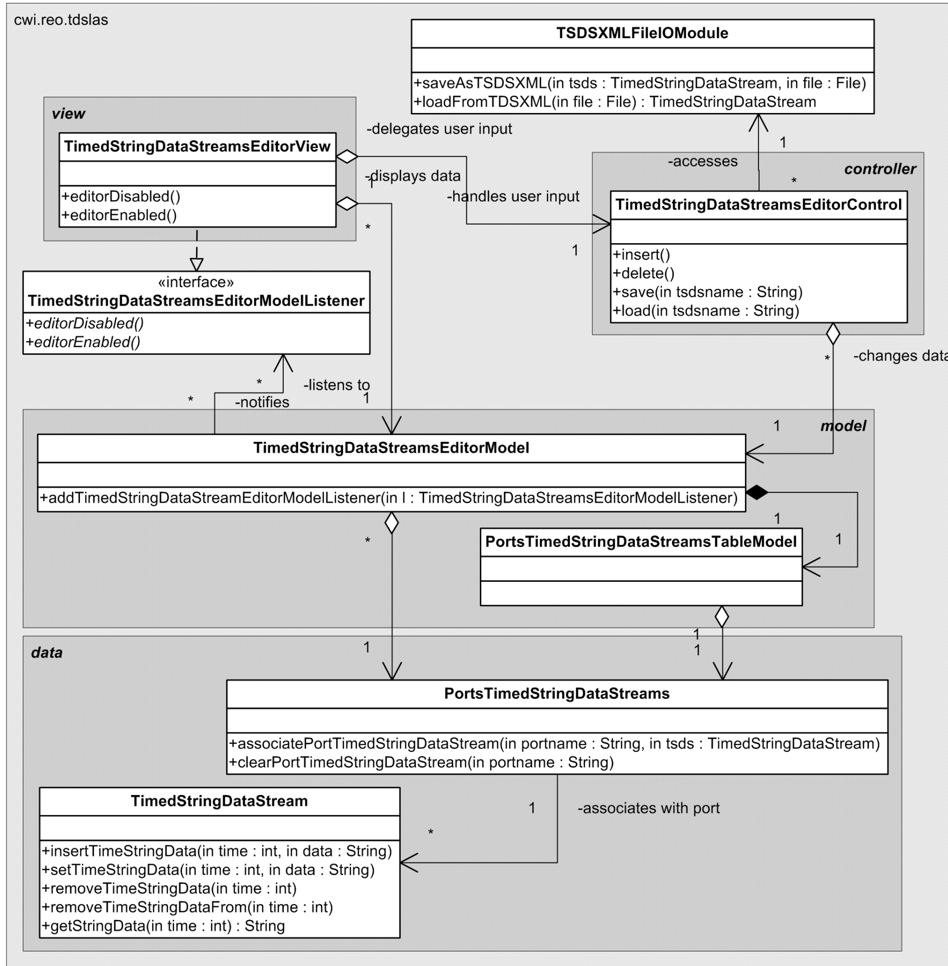

Figure 6.23: The class diagram of the TDS-Editor of the TDSLAS.

The table shown in the GUI TDS-Editor, implemented by a **JTable** (from the Java Swing library), shows the data of **PortsTimedStringDataStreams**. However, the interface of **PortsTimedStringDataStreams** is not the interface **JTable** expects. By applying the Adapter design pattern[6], the class **PortsTimedStringDataStreamsTableModel** converts the **PortsTimedStringDataStreams** interface into an interface **JTable** expects. **PortsTimedStringDataStreamsTableModel** acts as a model where **JTable** gets its data from, but the actual data comes from **PortsTimedStringDataStreams**. Changes in **PortsTimedStringDataStreams** are reflected in **JTable** and vice versa.

6.6.4 TDS-Editor Control

The **TimedStringDataStreamsEditorControl** is the controller object, which handles the following interactions with the user:

- insert timed stringdata tuples
- delete timed stringdata tuples
- save timed stringdata streams
- load timed stringdata streams

The **TimedStringDataStreamsEditorControl** performs these operations directly on **TimedStringDataStream** and **PortsTimedStringDataStreams**. Because of the adapter class **PortsTimedStringDataStreamsTableModel** these changes are reflected in the timed stringdata streams table in the GUI TDS-Editor.

6.6.5 GUI TDSLAS

In the simulator part of the TDSLAS the user simulates a constraint automaton by 'stepping' through the constraint automaton, going from one state to another dependent on the timed data streams. At each step the simulator lists the enabled transitions (the possible transitions), which depends on the current constraint automaton state and timed data streams, from which the user chooses one for the next step. A trace log is available, showing the history of the steps. The following classes are defined:

- **EnabledTransitions**, represents the enabled transitions,
- **Trace**, represents the trace history.

For the GUI of the simulator part we apply the MVC design pattern[6]. Figure 6.24 shows the class diagram and which part of the MVC design pattern the classes correspond to.

- **TraceModel**, the model for **Trace** containing the **TraceTableModel**.
- **TraceTableModel**, the adapter class between **TraceModel** and the table which shows the trace history,.
- **EnabledTransitionsModel**, the model of **EnabledTransitions** containing the **EnabledTransitionsTableModel**.
- **EnabledTransitionsTableModel**, the adapter class between **EnabledTransitions** and the table which shows the enabled transitions.
- **TDSLASModel**, the model of the simulator part.
- **TDSLASModelListener**, the listener interface for **TDSLASModel**.

- **TDSLASView**, the view of the simulator part.
- **TDSLASControl**, the controller of the simulator part.
- **CAEngineModel**, contains the CAEngine and keeps track of the current state, selected transition and last transition made during the simulation.

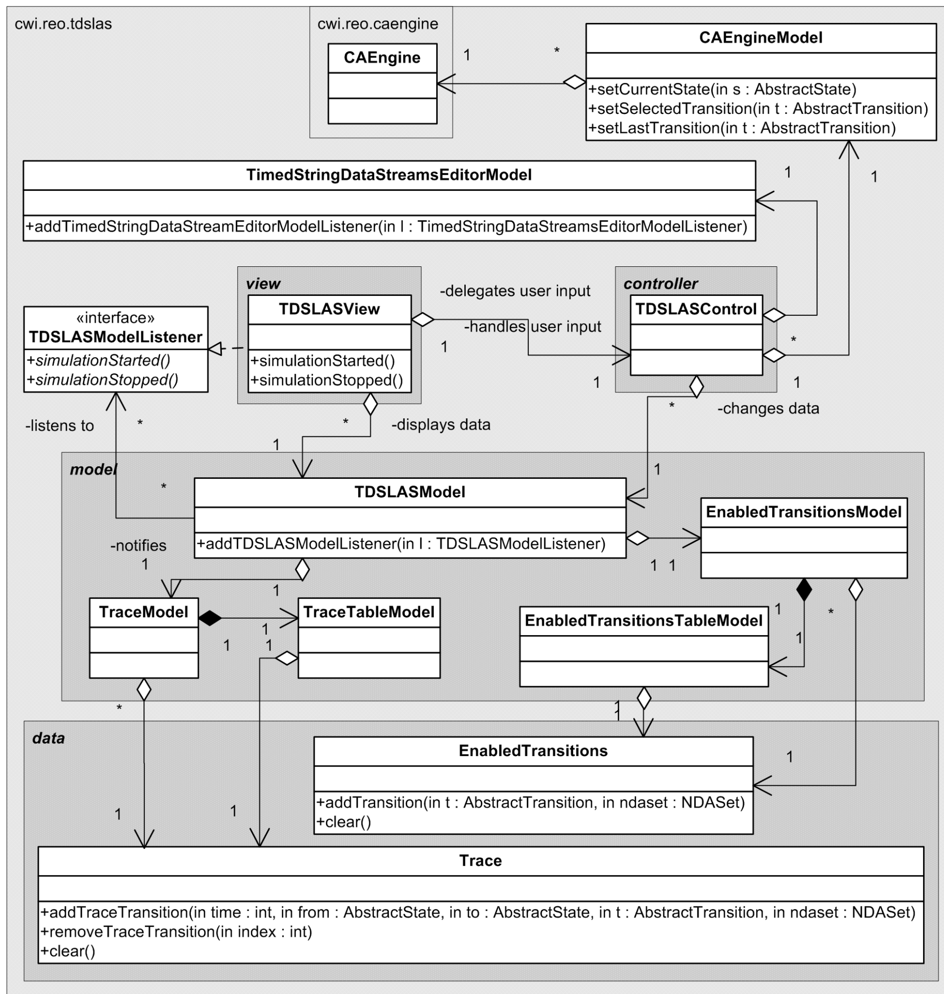

Figure 6.24: The class diagram of the simulator part of the TDSLAS.

For the overall GUI TDSLAS we integrate the GUI TDS-Editor and the GUI of the simulator part and show them together in one window. Figure 6.25 shows a screenshot of the GUI TDSLAS. The class diagram depicted in figure 6.26 shows the implementation.

- **TDSLASFrame**, the window which contains the GUI of the TDS-Editor and the simulator.

- **TDSLASMain**, the main class which starts and initializes the TDSLAS.

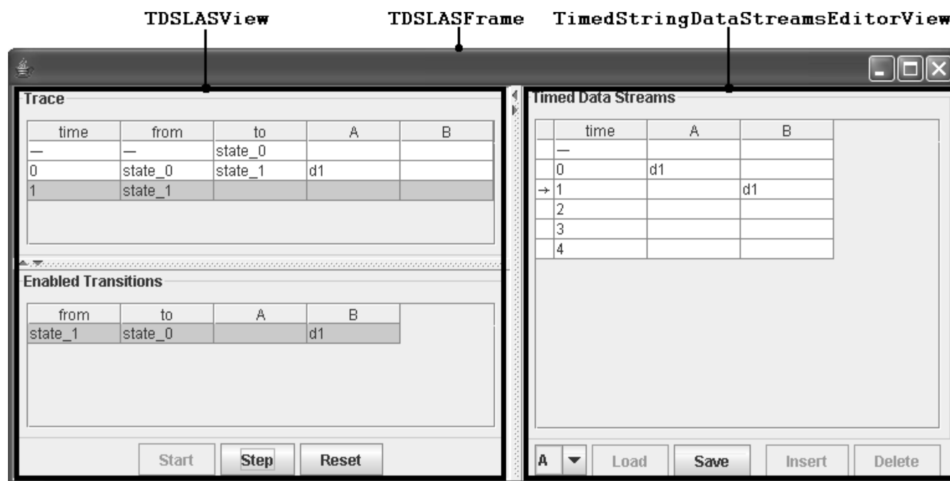

Figure 6.25: The screenshot of the GUI TDSLAS showing the correspondence with the classes.

6.6.6 TDSLAS Control

The TDSLAS Control is implemented by **TDSLASMain** and **TDSLASControl**. **TDSLASMain** is the main class that starts and initializes the TDSLAS. The user is able to access **TDSLASMain** through the IT-GUI.
The **TDSLASControl** is the controller object, which handles the interactions the user performs with the simulator part. Since it is also responsible for changing the data of the model object (the trace log, the enabled transitions table), the **TDSLAS-Control** is the one which actually performs the simulation.
The **TDSLASControl** interacts with the user and performs the simulation as follows:

1. The user starts the simulation.
2. The **TDSLASControl** fills the enabled transitions table with the initial transitions.
3. The user selects one of the initial transition and makes a step.
4. The **TDSLASControl** steps to the state indicated by the selected initial transition (this state becomes the ‘current state’).
5. The **TDSLASControl** clears and fills the enabled transitions table as follows:

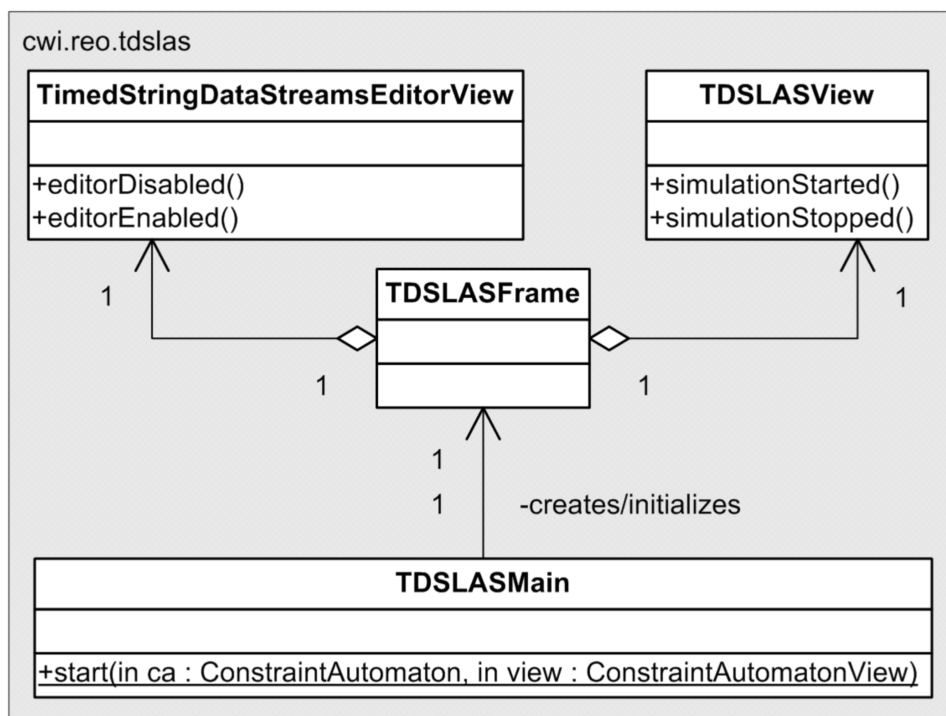

Figure 6.26: The class diagram of the integration of the TD-Editor and simulator.

(a) use the CA-Engine to retrieve the outgoing transitions from the current state
(b) get the timed stringdata streams from TDS-Editor
(c) create a set of name-data-assignments with the timed stringdata streams for time point t
(d) use the CA-Engine to evaluate each transition with this set of name-data-assignments
(e) each transition which is possible (according to the evaluation), is added to the enabled transitions table

6. The user selects a transition of the enabled transitions table and makes a step.
7. The **TDSLASControl** updates its current state with the state indicated by the selected transition.
8. Steps 5, 6 and 7 are repeated until the enabled transitions table becomes empty such that no transition can be selected by the user at step 6 (the constraint automaton rejects the timed stringdata streams).

6.6.7 Simulation Coloring

During the simulation the TDSLAS goes from one constraint automaton state to another state dependent on the timed data streams and the transitions the user chooses. The simulation can be made visible in the GUI CA-Editor, for example, by indicating the current state using colors. To realize this we apply the Observer design pattern[6] on **CAEngineModel** and introduce the following classes:

- **CAEngineModelListener**, the listener interface for **CAEngineModel**,
- **CAColorControl**, which implements the **CAEngineModelListener** interface and changes the colors in **ConstraintAutomatonView**.

The class diagram is shown in figure 6.27.

6.7 RCSwTDS

Looking at the requirements of RCSwTDS described in section 5.8, we decompose the functionality of RCSwTDS as follows:

- a timed data stream editor part (TDS-Editor), where the user is able to edit, save and load timed data stream,
- a simulator part, where the user can simulate a constraint automaton with (incomplete) timed data streams.

For the TDS-editor we reuse the TDS-Editor of the TDSLAS, because they are equivalent.

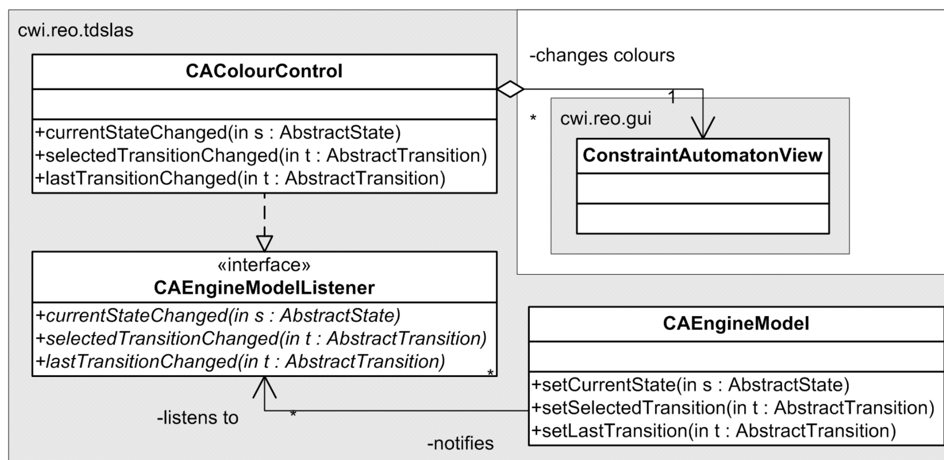

Figure 6.27: The class diagram of **CAEngineModelListener** and **CAColorControl**.

6.7.1 GUI RCSwTDS

The simulation part of the RCSwTDS is very similar to the TDSLAS. Like with the TDSLAS, the user simulates a constraint automaton by stepping through the constraint automaton. At each step the RCSwTDS shows the possible transitions from which the user chooses one for the next step. The RCSwTDS also has a trace log showing the history of the previous steps. Thus, for the simulation part many classes of the TDSLAS can be reused.
In addition to the TDSLAS, the RCSwTDS needs to show the 'observed time stringdata tuple' of each time step. For this we use the TDS-Editor, but we disable the editing of timed stringdata streams. By reusing the TDS-Editor, it is not necessary to implement another table. Figure 6.28 shows a screenshot of the GUI RCSwTDS.
For the GUI RCSwTDS we apply the MVC design pattern[6]. The class diagram is depicted in figure 6.29.

- **RCSwTDSModel**, the model of the simulator part.
- **RCSwTDSModelListener**, the listener interface for **RCSwTDSModel**.
- **RCSwTDSView**, the view of the simulator part.
- **RCSwTDSControl**, the controller of the simulator part.
- **RCSwTDSFrame**, the window which contains the GUI of the TDS-Editor and the simulator.
- **RCSwTDSMain**, responsible for starting and initializing the RCSwTDS.

RCSwTDSView RCSwTDSFrame TimedStringDataStreamsEditorView

Figure 6.28: The screenshot of the GUI RCSwTDS showing the correspondence with the classes.

Figure 6.29: The class diagram of RCSwTDS

6.7.2 RCSwTDS Control

The RCSwTDS Control is implemented by **RCSwTDSMain** and **RCSwTDSControl**. The class **RCSwTDSMain** is the main class that starts and initializes the RCSwTDS. The user is able to access **RCSwTDSMain** through the IT-GUI.
The **RCSwTDSControl** is the controller object, which handles the interactions the user performs with the simulator part. Since it also responsible for changing the data of the model object (the trace log, the observed timed stringdata streams, the enabled transitions table), the **RCSwTDSControl** is the one which actually does the simulation.
The **RCSwTDSControl** interacts with the user and performs the simulation as follows:

1. The user starts the simulation.
2. The **RCSwTDSControl** fills the enabled transitions table with the initial transitions.
3. The user selects one of the initial transition and makes a step.
4. The **RCSwTDSControl** steps to the state indicated by the selected initial transition (this state becomes the 'current state').
5. The **RCSwTDSControl** clears and fills the enabled transitions table as follows:
   (a) use the CA-Engine to retrieve the outgoing transitions from the current state
   (b) get the timed stringdata streams from TDS-Editor
   (c) create a set of name-data-assignments with the timed stringdata streams for time point *t*
   (d) combine this set of name-data-assignments with the delayed name-data-assignments of the previous step
   (e) add the combined set of name-data-assignments to the table showing the observed timed stringdata streams in the GUI RCSwTDS
   (f) use the CA-Engine to evaluate each transition with the combined set of name-data-assignments
   (g) each transition which is possible (according to the evaluation), is added to the enabled transitions table
6. The user selects a transition of the enabled transitions table and makes a step.
7. The **RCSwTDSControl** updates its current state with the state indicated by the selected transition.
8. Steps 5, 6 and 7 are repeated.

6.7.3 Simulation Coloring

Like the TDSLAS the simulation of a constraint automaton in the RCSwTDS can be visualized by coloring the states and transitions in the GUI CA-Editor. Since we reuse the class **CAEngineModel** of TDSLAS, the other classes responsible for the coloring can also be reused (see subsection 6.6.7).

6.8 RCSwC

In this section we discuss the design and implementation of the RCSwC.

6.8.1 GUI RCSwC

Through the GUI RCSwC the user is able to:

- load Reo components into the application,
- load constraint automata into the application as Reo connectors,
- connect Reo component ports with Reo connector ports,
- start/stop the simulation.

After a component has been loaded into the application, the ports of this component are shown in a table in the GUI RCSwC. The same applies for constraint automata. Next, the user can select a component port and a connector port and connect them. Figure 6.30 shows a screenshot of the GUI RCSwC.

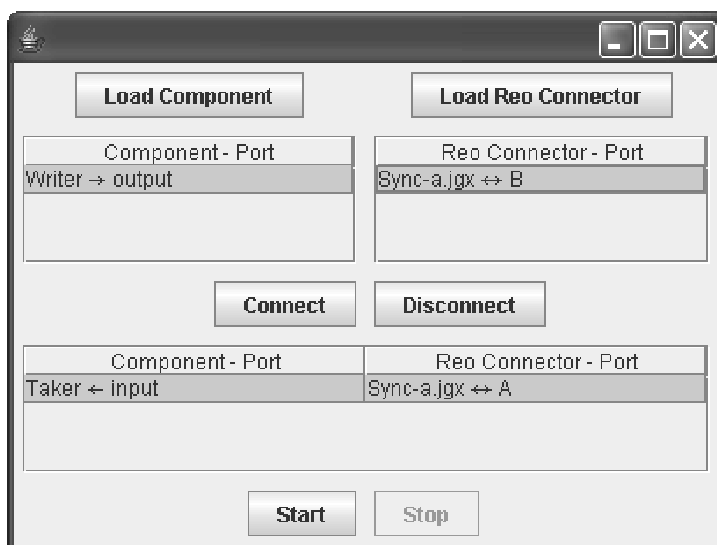

Figure 6.30: Screenshot of the GUI RCSwC.

The MVC design pattern[6] is applied for the GUI RCSwC. The class diagram is depicted in figure 6.31.

- **ComponentPort**, represents a component port.
- **ReoConnectorPort**, represents a Reo connector port.
- **RCSwCModel**, the model of the RCSwC, which contains the data such as the loaded components ports, loaded Reo connector ports and the connections made (between components ports and Reo connector ports).
- **ComponentPortsTableModel**, the adapter class between **RCSwCModel** and the table which shows the component ports.
- **ReoConnectorPortsTableModel**, the adapter class between **RCSwCModel** and the table which shows the Reo connector ports.
- **MappingTableModel**, the adapter class between **RCSwCModel** and the table which shows the connections between component ports and Reo connector ports.
- **RCSwCView**, the view of the RCSwC.
- **RCSwCControl**, the controller of the RCSwC.
- **RCSwCMain**, the main class which starts and initializes the RCSwC.

6.8.2 Load Component and Component

The class responsible for loading a Reo component into the application is the class **ComponentXMLLoadModule**. The **ComponentXMLLoadModule** reads a Component-XML file where the binary name or the location of a class is specified. The Component-XML file can also contain extra information that can be used for initialization of the component. The class is loaded into the memory by the **MultiClassLoader**. If the loaded class implements the **Component** interface, then an object is instantiated from this class, which acts as a component instance. The **Component** interface allows the following:

- pass the Component-XML file to the component instance for initialization,
- get/set the name of the component instance,
- retrieve names of the output ports and input ports,
- connect (disconnect) **Sink** and **Source** objects to (from) input portname (output portnames).

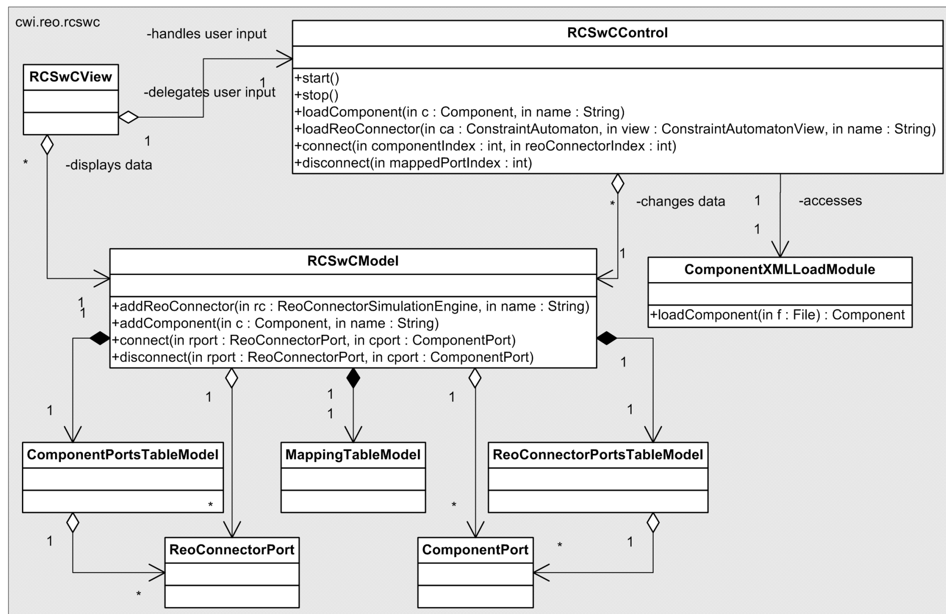

Figure 6.31: The class diagram of GUI RCSwC.

Through the **Sink** and **Source** objects the component instance is able to perform take and write operations with its environment. However, all the objects which pass through the **Sink** and **Source** have to implement the **Data** interface. Figure 6.32 shows the class diagram of the classes that are located in the packages **cwi.reo.rcswc.component** and **cwi.reo.rcswc.core**.
Some standard components have already been implemented. These components are located in the package **cwi.reo.rcswc.component.std**:

- **Taker**, a component which allows the user to perform take operations on a port,
- **Writer**, a component which allows the user to perform write operations on port,
- **FIFO1**, a component which acts as a FIFO1 buffer,
- **PythonComponent**, a component which enables the support for the scripting language Python.

The implementation of the Jython component is discussed in subsection 6.8.3.

6.8.3 Python Component

The Python component is implemented by the class **PythonComponent**, which acts as wrapper around a Python interpreter. For the Python interpreter we use Jython, which is a pure Java implementation of the scripting language Python.

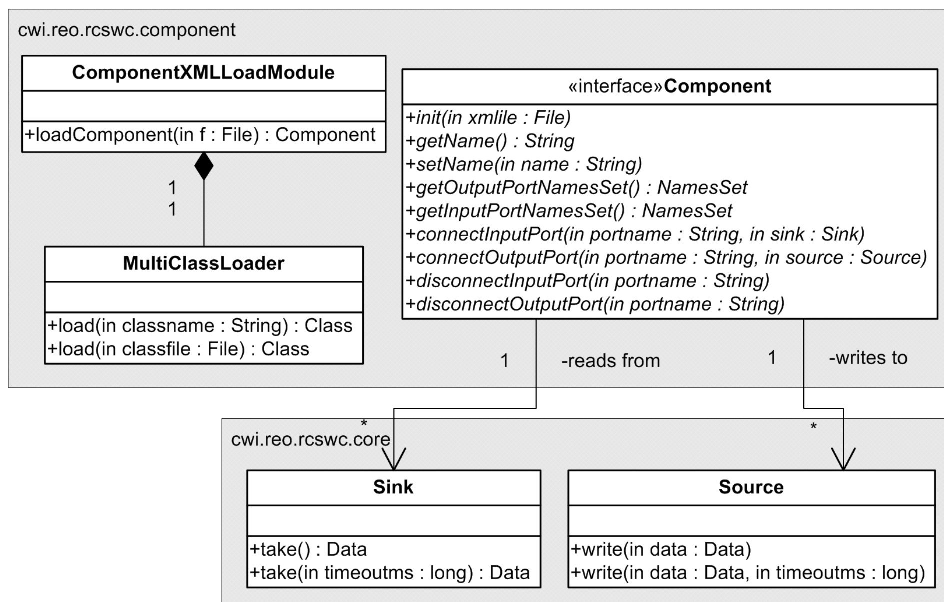

Figure 6.32: The class diagram of Load Component and Component.

When a Python component is instantiated, it creates a Python interpreter, loads the source code of the Python program into the interpreter and executes the source code. The source code is contained in the Component-XML file, which is given to the component instance during initialization.
To allow the Python program to perform write and take operations, a proxy is loaded into the Python interpreter environment, the **PythonComponentProxy**. Through this proxy a Python program is able to retrieve **Sink** and **Source** objects by input and output portnames. However, these objects cannot be used directly within the interpreter environment, therefore the proxy puts a wrapper around them with **PythonSinkWrapper** and **PythonSourceWrapper** before it returns them to the Python program (figure 6.33).
Through the **PythonSinkWrapper** and **PythonSourceWrapper** the Python program is able to perform write and take operations. However, Python data objects cannot just leave the interpreter environment. The **PythonSourceWrapper** wraps the Python data objects with the **PythonDataWrapper** before they leave the interpreter environment.

6.8.4 RCSwC Control

The RCSwTDS Control is implemented by **RCSwCMain** and **RCSwCControl**. The class **RCSwCMain** is the main class that starts and initializes the RCSwC. The user is able to access **RCSwCMain** through the IT-GUI.
The **RCSwCControl** handles the following interactions the user performs with the

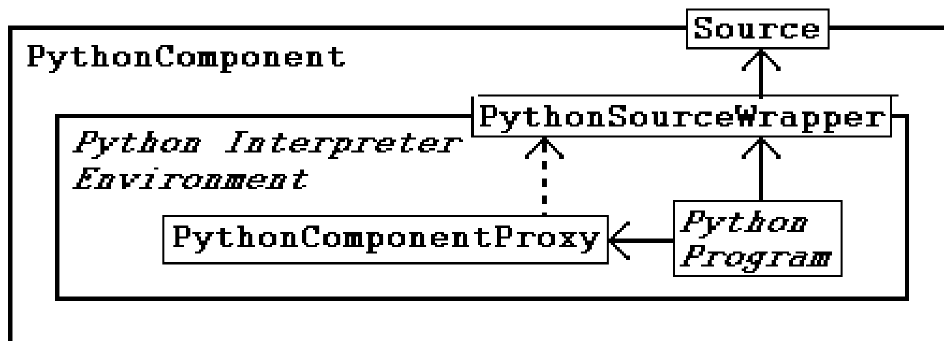

Figure 6.33: The Python component.

application:

- load Reo components into the application,
- load constraint automata into the application as Reo connectors,
- connect/disconnect component ports with Reo connector ports,
- start/stop the simulation.

Load Reo Components

The loading of Reo components is delegated to Load Component, which is discussed in subsection 6.8.2. After the loading, the ports of the component are shown in a table in the GUI RCSwC.

Load Constraint Automaton as Reo Connector

When the user loads a constraint automaton into the RCSwC, a **ReoConnector-SimulationEngine** object is created for this constraint automaton. The **ReoConnectorSimulationEngine** is the class that is responsible for simulating the constraint automaton as a Reo connector. After the loading the ports of the constraint automaton are shown in a table in the GUI RCSwC.

Connect Component Ports and Connector Ports

A component port is not directly connected to a connector port, but this is done with a **Sink** or **Source** object that acts as an intermediary. A **Sink** object is used if the connector port is an input port and a **Source** object in case of an output port. Hence, a component has no knowledge to which connector it is connected and vice versa.

The simulation of a connector is based on a constraint automaton. Since a constraint automaton does not distinguish between input and output ports, the simulated Reo connector, the **ReoConnectorSimulationEngine**, also does not distinguish between these. To accommodate this the **Sink** and **Source** are generalized by the abstract class **ChannelEnd**. Figure 6.34 shows the class diagram.

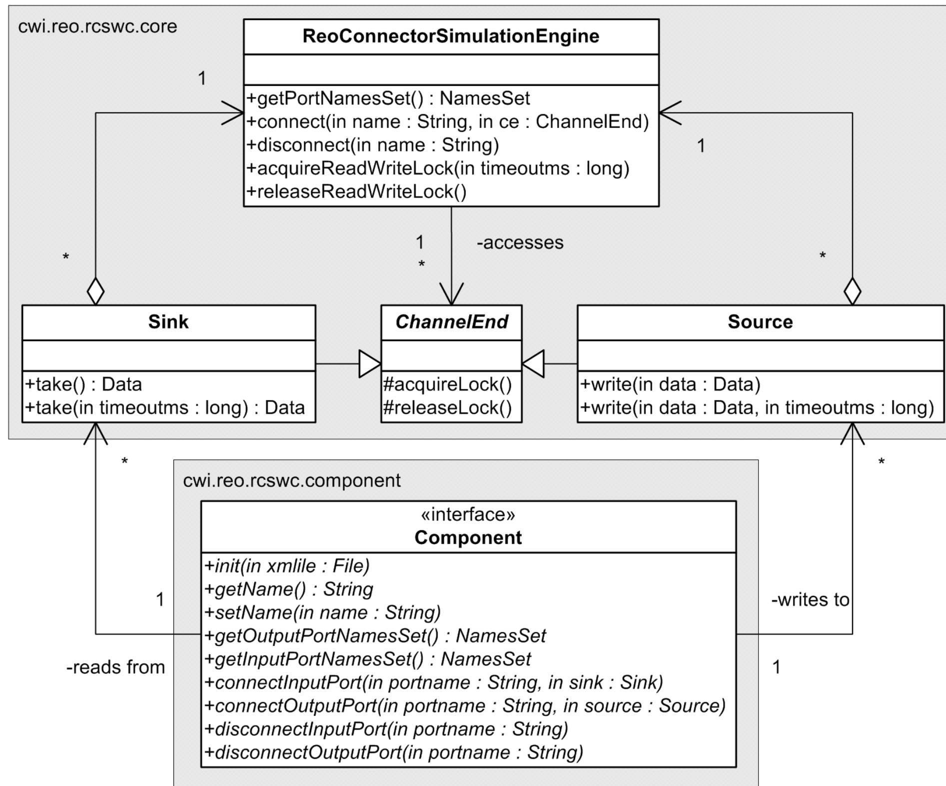

Figure 6.34: The class diagram showing the connection between components and connectors.

Simulation

When the user starts the simulation, all the loaded components and loaded Reo connector simulators are executed, each by a separate thread. During the execution the components perform write and take operations to the connected **Source** and **Sink** objects. The **ReoConnectorSimulationEngine** immediately goes to sleep and waits until an operation is being performed at one of its connected **ChannelEnd**.

The **Source**, **Sink** and **ReoConnectorSimulationEngine** have mutex locks, which are used to synchronize the threads and prevent corrupt situations. A description

of how the components, **Sink**, **Source**, and **ReoConnectorSimulationEngine** interact with each other is given below:

1. A component performs a write (take) operation to its Source (Sink) object.
2. The **Source** (**Sink**) tries to acquire the read-write-lock of the **ReoConnector-SimulationEngine**. The read-write-lock prevents that multiple **Source** and **Sink** objects try to wake the **ReoConnectorSimulationEngine** (see the next step).
3. The **Source** (**Sink**) wakes the **ReoConnectorSimulationEngine** and goes to sleep.
4. The awakened **ReoConnectorSimulationEngine** acquires all the locks of the **ChannelEnd** objects it is connected to. This prevents that a **Channel-End** continues its work when it wakes up while the **ReoConnectorSimulationEngine** is still busy (see step 8).
5. The **ReoConnectorSimulationEngine** evaluates the data of its **Channel-End** objects by checking whether a transition is possible in the constraint automaton.
6. The **ReoConnectorSimulationEngine** changes the data of each **Channel-End** if the data of this **ChannelEnd** was involved in a transition.
7. The **ReoConnectorSimulationEngine** releases all the locks of its **Channel-End** objects and wakes them.
8. The awakened **Source** (**Sink**) tries to acquire its own lock. This prevents that a **Source** (**Sink**) continues its work when it wakes up while the **ReoConnectorSimulationEngine** is still busy (see step 4).
9. The **Source** (**Sink**) releases its own lock and returns (the data) to the component (where the operation originated from).

6.8.5 Simulation Coloring

Each connector that is being simulated in the RCSwC is based on a constraint automaton that is being viewed in the CA-Editor. Hence, the simulation of a connector can be made visible by coloring the states in the constraint automaton in the GUI-CA-Editor, just like the TDSLAS and RCSwTDS. To accommodate this we apply the Observer design pattern[6] on the **ReoConnectorSimulationEngine**:

- **ReoConnectorSimulationEngineListener**, the listener interface for **ReoConnectorSimulationEngine**,
- **CAColorControl**, which implements the **ReoConnectorSimulationEngineListener** interface and changes the colors in **ConstraintAutomatonView**.

The class diagram is depicted in figure 6.35.

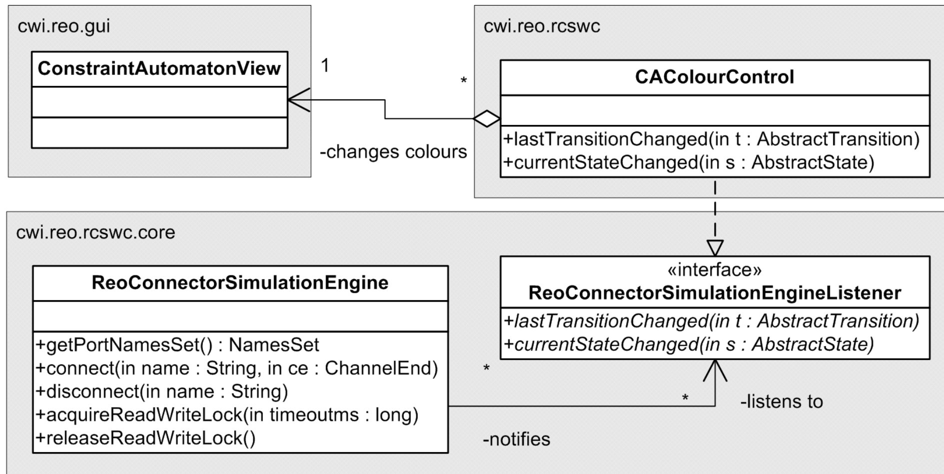

Figure 6.35: The class diagram of **ReoConnectorSimulationEngineListener** and **CAColorControl**.

Chapter 7

Conclusions

In this chapter we summarize our contributions and the MSc project. We briefly give some ideas how our work could be extended or could benefit other projects. Finally, the author reflects on the personal experiences.

7.1 Contributions

The main contribution of this MSc project is a tool for constraint automata, consisting of an editor and three simulators. The constraint automata editor allows the user to visually construct and modify constraint automata. The editor is capable of saving and loading constraint automata to and from files, which allows the user to continue previous work, parses input, which prevents the user from creating corrupt constraint automata, and contains a layout engine for the visual representation of the states and transitions.
The first simulator is the "TDS-Language Acceptor Simulator", which is able to check the acceptance of a TDS-language for a constraint automaton. The second simulator, the "Reo Connector Simulator with TDS", is able to simulate a constraint automaton as a Reo connector where the input is given as timed data streams. This simulator allows one to study the behavior of Reo connectors in non-real-time. The third simulator is the "Reo Connector Simulator with Components", which is able to simulate a constraint automaton as a Reo connector with components attached to it. An API has been provided for implementing arbitrary software components that can be attached to the simulator, including support for scripting components in Python. We believe that our tool is the first that contains a complete visual editor and simulators for constraint automata.
Since constraint automata do not distinguish between input and output, we could not directly determine values for output from given input. This functionality is required for simulating constraint automata as Reo connectors. We illustrated how take operations in Reo can be interpreted as the completion of incomplete timed data streams in constraint automata. Hereby we encountered an issue with the data constraints: they are propositional formulae, not assignments. We solved

this issue by using Prolog-style backward-chaining during evaluation of constraint automaton transitions to bind proper values to the variables representing output ports in the constraint automatons data constraints.
The tool has a modular design, such that many parts can be reused, replaced or improved in the future. The design of our tool can easily be extended towards parameterized constraint automata.

7.2 Summary

Before we started with the MSc project, we performed a literature study to acquire the background knowledge on Reo and constraint automata and become familiar with the concepts and terminology.
The actual execution of the MSc project started with the requirements and analysis of the assignment. The requirements of the assignment were analyzed and decomposed into several smaller subrequirements. Each subrequirement was analyzed again and, where necessary, decomposed into even smaller subrequirements.
Next, we explored which options were available for the implementation each subrequirement. For example, the GUI of the constraint automata editor can be implemented by making use of a graph visualization library, such as JGraph or JHotDraw, but another more sophisticated option would be Grace, a generator for graph editors.
In the design and implementation phase all the options of each subrequirement were examined and the most suitable option for this subrequirement (and for the overall project) was chosen. For example, from JGraph, JHotDraw and Grace, we selected Grace, because the generated editor is not only able to visualize constraint automata, but also able to keep the consistency between the visual representation of a constraint automaton and its data representation.
The tool has a modular design, such that in the future parts of the tool can easily be replaced or reused.
To conclude, all the requirements of the assignment were realized and the complete tool has been delivered, consisting of:

- a constraint automata editor,
- a timed data stream language acceptor simulator,
- a Reo connector simulator with timed data streams,
- a Reo connector simulator with components.

7.3 Future Work

The tool Swiss Watch is a visual editor for constraint automata, but it is still in an early development stage. Besides the editor functionality, Swiss Watch is able

to perform the join and hide operation on constraint automata. Our constraint automata editor is more mature than Swiss Watch, but it does not have the join and hide features, because these functionalities did not concern this assignment. Since both are written in Java, our tool could relatively easily be improved with these features of Swiss Watch by integrating them together.
A feature that can be added to our tool is the support for parameterized constraint automata. This is useful, because parameterized constraint automata allow us to describe (more) complex behavior in a compact way (see section 4.4).
Another direction would be extending the current tool towards timed constraint automata[2], an extended version of constraint automata where time constraints are added on transitions. Timed constraint automata can be used to describe Reo channels whose behavior involves temporal constraints. For instance, a FIFO1 channel that automatically loses a data item from its buffer when the data item has stayed in the buffer longer than t units of time.
Currently the simulators block the constraint automata editor when they are running, preventing the user from modifying the constraint automaton while it is being simulated. An interesting extension to our tool would be allowing the user to pause the simulation, modify the constraint automaton through a set of special operations and then continue the simulation. This would facilitate the research in dynamic reconfiguration of Reo connectors[5], which is about reconfiguring a connector at run time while components are still connected to it. This subject is of importance, because dynamically reconfiguring a connector could lead to a corrupt situation, where the coordinating connector exposes undesirable behavior.
Since our tool is designed in a modularized way, many parts can be reused. For example, the constraint automata engine, which is able to evaluate data constraints in a Prolog like manner, could be used in other constraint automata projects, such as model checking tools. The same applies for the names parser and the data constraint parser.

7.4 Personal Experience

For the last ten months I worked on this MSc project, which included a literature study, analysis of requirements, design and implementation of the tool and writing of the thesis. The project was conducted at the CWI in Amsterdam. Although the traveling from The Hague to Amsterdam and back was hard, I enjoyed working there, because of its international environment with researchers from all kind of nationalities. For the most part I worked autonomously and independently on the project, but I had two supervisors, Nikolay Diakov of CWI and Kees Pronk from TU Delft, who guided me through the overall process.
During the literature study I experienced reading papers about a new subject and how to acquire the knowledge needed for a project. Next, I analysed the requirements and decomposed them into subrequirements. For the implementation of each subrequirement I searched and evaluated which technique was the most appropi-

ate. During the analysis of requirements I tried to foresee the potential areas that could cause problems later on and look for the best way to deal with them. The analysis of requirements phase was personally the most interesting phase, because it required me to think out of the box in order to come up with an elegant solution for each problem.

During the design and implementation of the tool I did not use UML often nor did I document a lot. Keeping UML diagrams and documentation consistent with the actual progress of the project requires a lot of time. Since the project was not that big and consisted only of one person, I did not find that having UML diagrams and documentation benefit against the extra time it requires to create them. The only documentation I kept was a list of the work I did and the problems I encountered during the project, which served as input for my written report, this thesis. I used UML only in complex situations where a lot of classes were involved. This was especially the case during the design and implementation of the graphical user interfaces. However, I found out that I did not have enough experience with UML modeling certain situations. Nikolay Diakov helped me to improve on this aspect.

After the design and implementation I did create the UML diagrams and documentation and included them in my thesis, because they were necessary for the developers who would work on the tool after me. The writing of the thesis was difficult for me, not only because of the lack of experience with writing in English, but also with scientific writing in general. Luckily, during the writing process I got the support from both my supervisors. However, I do think that the TU Delft should pay more attention to this during student projects in general, if they want to prepare future researchers, because writing academic papers is an important and time-consuming part of the job. Another difficulty I experienced was choosing the right level of detail for the thesis. Should the thesis be a high level document where only the concepts are clarified or a low level document where even the implementation is explained?

During the project I dealt with two supervisors. Nikolay Diakov, who was my daily supervisor and therefore more involved in the project, was able to support me throughout the project and gave constructive critisism where necessary. Kees Pronk was less involved in the beginning, but became more involved during the writing of the thesis. Since he was less involved in the project, he could easily spot the gaps and holes in the thesis that might confuse the general reader.

I found this project very interesting, because it required me to use the knowledge acquired at TU Delft, but the project also gave me opportunity to improve my knowledge, to work in an international environment at a highly valued research institute and to gain experience with scientific writing. Finally, I want to conclude that I am very glad how the project proceeded and very content with the end result.

Bibliography

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

[2] Farhad Arbab, Christel Baier, Frank de Boer, and Jan Rutten. Models and temporal logics for timed component connectors. *Second International Conference on Software Engineering and Formal Methods, 2004*, pages 198–207, 2004.

[3] Farhad Arbab, Christel Baier, Jan J.M.M. Rutten, and Marjan Sirjani. Modeling component connectors in reo by constraint automata (extended abstract). *ENTCS*, 97:25–46, 2004. For the full version see http://web.informatik.uni-bonn.de/I/baier/publikationen.html.

[4] Farhad Arbab and Farhad Mavaddat. Coordination through channel composition. *LNCS*, 2315:22–39, 2002.

[5] Dave Clarke. Reasoning about connector reconfiguration I: Equivalence of constructions. Technical Report SEN-R0506 ISSN 1386-369X, CWI, Amsterdam, The Netherlands, February 2004. http://homepages.cwi.nl/ dave/writing/.

[6] Erich Gamma, John Vlissides, Ralph Johnson, and Richard Helm. *Design Patterns CD: Elements of Reusable Object-Oriented Software, (CD-ROM)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1998.

[7] Gerwin Klein. Generating graphical editors for graph-like data structures. MSc thesis, Technische Universität München, November 1999. http://www.doclsf.de/grace/.

[8] Michael Sipser. *Introduction to the Theory of Computation*. International Thomson Publishing, 1996.

[9] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Longman Publishing Co., Inc., second edition edition, 1998.

The user can find links to online resources embedded as footnotes throughout the text of the individual chapters.

Appendix A

User's Manual

This tutorial gives an introduction to the tool, which consists of:

- the Constraint Automata Editor,
- the Timed Data Stream Language Acceptor Simulator (TDSLAS),
- the Reo Connector Simulator with Timed Data Streams (RCSwTDS),
- the Reo Connector Simulator with Components (RCSwC).

First, the installation of the tool is explained and how to run it. Then we continue to the constraint automata editor where a small constraint automaton will be created as an example. Next, we proceed to the different simulators and explain how constraint automata are being simulated using the example.

A.1 Installation

A.1.1 System Requirements

The tool requires an operating system with Java Runtime Environment (JRE) version 5 or above. The JRE is supported by several operating systems, e.g. Windows, Linux and Solaris. It can be downloaded from `http://www.java.com/`.
The graph layout program *dot* must be installed and reachable via the PATH environment variable (reachable from any directory). *dot* is part of Graphviz, which is open source graph visualization software. Graphviz has several graph layout programs and is supported by Windows, Linux and Apple. It can be downloaded from `http://www.graphviz.org/`.

A.1.2 Installing and Running

The tool's binaries are shipped as compressed zip or tar archive. When uncompressed you get the following files and directory structure:

.\Constraint_Automata_Editor_and_Simulators.jar	the executable file
.\lib	contains the libraries the tool depends on

The tool can be started from the command line as follows:
java -jar Constraint_Automata_Editor_and_Simulators.jar

A.2 Constraint Automata Editor

When you start the tool, you see a window with a workspace area and on top the usual menu bar and tool bar. In the workspace area constraint automaton documents can be viewed, each in a separate document window. At the bottom of a document window there is a properties pane where you can see and modify the properties of a constraint automaton. The menu bar and tool bar provide the controls you need to work with documents. Some of the controls are only accessible if a document is opened.

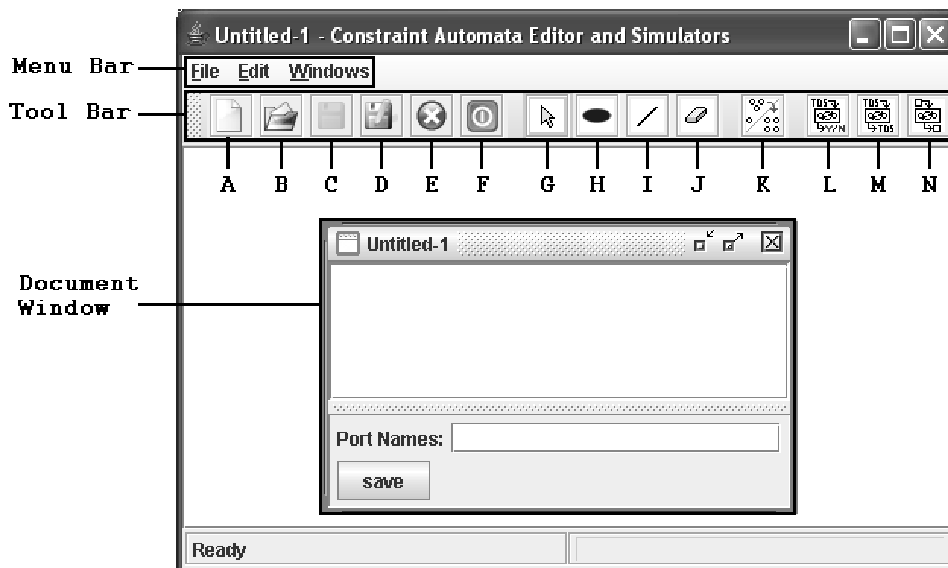

Figure A.1: The workspace area.

Menu Bar

In the menu bar you can perform the following actions:

- File

  New: Creates a new empty constraint automaton document.

  Open: Opens a constraint automaton document from a CA-XML file.

Open Recent: Shows the 10 last documents which have been opened.

Save: Saves a constraint automaton document to a CA-XML file.

Save as: Saves a constraint automaton document to a CA-XML file under a specific name.

Close: Closes the current.

Quit: Shuts down the application.

- Edit

  Edit State/Transition/CA: Puts the application in the edit-mode. In this mode the layout and properties of states, transitions and the constraint automaton can be modified.

  Add State: Puts the application in the 'add state'-mode. In this mode a state can be added.

  Add Transition: Puts the application in the 'add transition'-mode. In this mode a transition can be added.

  Remove State/Transition: Puts the application in the remove-mode. In this mode a state or transition can be removed.

  Perform Layout: rearranges the layout of states and transitions of a constraint automaton.

- Windows

  Tile: tiles the document windows.

  Cascade: cascades the document windows.

Tool Bar

The tool bar provides the following controls (figure A.1):

- **A**: Creates a new empty constraint automaton document.
- **B**: Opens a constraint automaton document from a CA-XML file.
- **C**: Saves a constraint automaton document to a CA-XML file.
- **D**: Saves a constraint automaton document to a CA-XML file under a specific name.
- **E**: Closes the current document window.
- **F**: Shuts down the application.
- **G**: Puts the application in the edit-mode. In this mode the layout and properties of states, transitions and the constraint automaton can be modified.

- **H**: Puts the application in the 'add state'-mode. In this mode a state can be added.
- **I**: Puts the application in the 'add transition'-mode. In this mode a transition can be added.
- **J**: Puts the application in the remove-mode. In this mode a state or transition can be removed.
- **K**: Rearranges the layout of the states and transitions of the constraint automaton.
- **L**: Opens the TDSLAS
- **M**: Opens the RCSwTDS.
- **N**: Opens the RCSwC.

A.2.1 Add State

To add a state to the constraint automaton perform the following steps:

1. Put the application in the 'add state'-mode.
2. Left mouse click on a empty place inside the document window. At this spot the new state will created.
3. The application automatically returns to the edit-mode.

A state becomes an initial state when an initial transition points towards it. See subsection A.2.3 how to create an initial transition

A.2.2 Modify State Properties

The state name is the only property of a state. To modify the state name perform the following steps:

1. Put the application in the edit-mode.
2. Left mouse click on the state of which the name needs to be changed. The properties pane will show the state name.
3. Modify the state name in the properties pane.
4. Left mouse click on the Save button in the properties pane to save the changes.

A state name may only contain letters, numbers, hyphens and underscores.

A.2.3 Add Transition

Transition

To add a transition between two states perform the following steps (figure A.2):

1. Put the application in the ‘add transition’-mode

2. Choose a state, the source state, by pressing down on it and holding the left mouse click.

3. Without releasing move the mouse to a state, the target state.

4. Release the mouse on the target state. This creates a transition from the source state to the target state. If the source state and target state are the same, then a transition will be created from a state to itself.

5. The application automatically returns to the edit-mode.

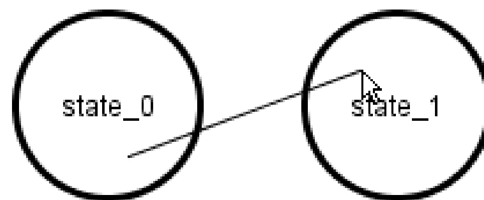

Figure A.2: Create a transition between two states.

Initial Transition

To create an initial transition perform the following steps:

1. Put the application in the ‘add transition’-mode.

2. Press down on a empty place inside the document window and hold the left mouse click.

3. Without releasing move the mouse to a state.

4. Release the mouse on this state. This creates an initial transition towards the state. The state becomes an initial state.

5. The application automatically returns to the edit-mode.

<table>
<tr><th>Precedence</th><th>Operator</th><th></th></tr>
<tr><td>1</td><td>( )</td><td>parenthesis</td></tr>
<tr><td>2</td><td>!</td><td>unary not</td></tr>
<tr><td>3</td><td>==
!=</td><td>equal
not equal</td></tr>
<tr><td>4</td><td>&&</td><td>Boolean AND</td></tr>
<tr><td>5</td><td>||</td><td>Boolean OR</td></tr>
</table>

Table A.1: The operators that can be used in data constraints and their precedence.

Term	
`data(`*<port name>*`)`	represents the data which is assigned to *<port name>*
`"`*<string>*`"`	represents a stringdata constant

Table A.2: The terms that can be used in data constraints.

A.2.4 Modify Transition Properties

The properties of a transition are the port names and the data constraints. To modify properties of a transition perform the following steps:

1. Put the application in the edit-mode.
2. Left mouse click on the transition of which the properties need to be changed. The properties pane will show the transition properties.
3. Modify the transition properties in the properties pane.
4. Left mouse click on the Save button in the properties pane to save the changes.

A port name may only contain letters, numbers, hyphens and underscores. To enter multiple port names separate them by commas.
A data constraint is a propositional formula. Multiple data constraints can be entered by separating them by new lines. Table A.1 lists the operators (and their precedence) that can be used in a data constraint. Table A.2 shows the terms which are allowed in a data constraint. Here are some examples of a data constraints:

```
data(A)==data(B)
data(A)=="d1"
!(data(A)==data(B) && data(A)=="d1")
```

A.2.5 Modify Constraint Automaton Properties

The constraint automaton has one property, the port names. To modify the port names of the constraint automaton perform the following steps:

1. Put the application in the edit-mode.

2. Left mouse click on empty place inside the document window. The properties pane will show the port names of the constraint automaton.
3. Modify the port names in the properties pane.
4. Left mouse click on the Save button in the properties pane to save the changes.

A port name may only contain letters, numbers, hyphens and underscores. To enter multiple port names separate them by commas.

A.2.6 Perform Layout

The layout of the states and transitions can be changed.

Move State

To move the states perform the following steps:

1. Put the application in the edit-mode.
2. Choose a state by pressing down on it and holding the left mouse click.
3. Without releasing move the mouse to another location. The state will follow the mouse movements.
4. Release the mouse on the desired location.

Move Transition

In the constraint automaton editor a transition is drawn as an arrow. This arrow can consist of one or more segments. To divide a segment into two segments perform the following steps (figure A.3):

1. Put the application in the edit-mode.
2. Left mouse click on the transition which needs to be divided.
3. Right mouse click on the segment of the arrow which needs to be divided. A popup menu will be shown.
4. Select 'Subdivide Curve Segment' in the popup menu. The segment will be divided in two.

After a segment is divided into two segments a point will be shown which connects the two segments. This point can also be moved as follows (figure A.4):

1. Put the application in the edit-mode.
2. Left mouse click on the transition which needs to be divided.

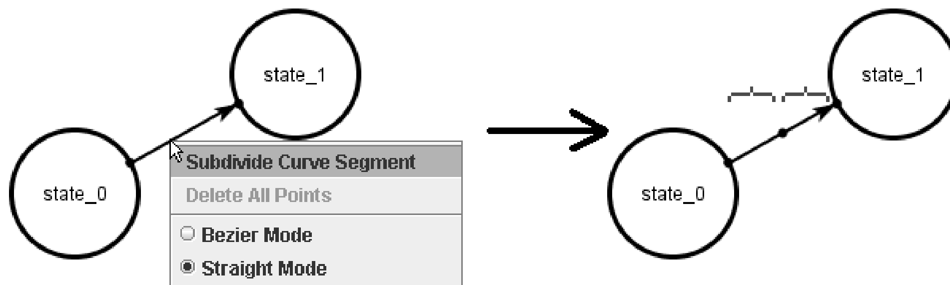

Figure A.3: Subdivide a segment of the arrow representing a transition.

3. Press down on the point which needs to be moved and hold the left mouse click.

4. Without releasing move the mouse to another location. The point will follow the mouse movements.

5. Release the mouse on the desired location.

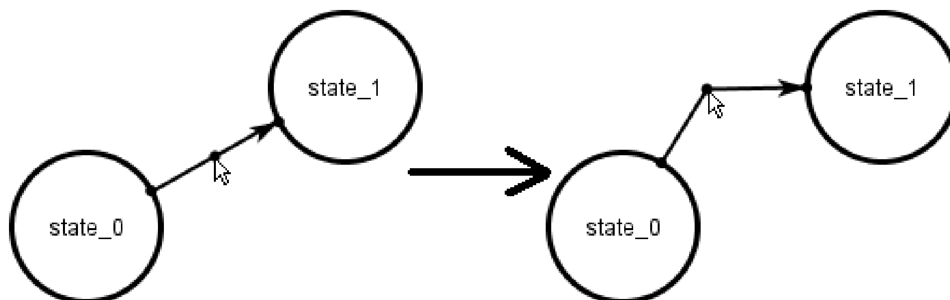

Figure A.4: Move a point connecting two segments to another location.

Such a point can also be deleted, which causes the two segments to become one segment again. To do this follow these steps (figure A.5):

1. Put the application in the edit-mode.

2. Left mouse click on the transition which needs to be divided.

3. Right mouse click on the point which needs to be deleted. A popup menu will be shown.

4. (a) Select 'Delete Point' in the popup menu. The two segments connected by this point will become one segment.

(b) Select ‘Delete All Points’ in the popup menu. All the points of the segments of the arrow are deleted which causes the arrow to consist of one segment again.

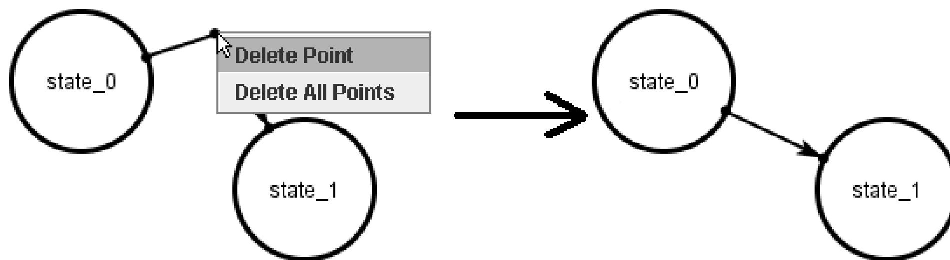

Figure A.5: Delete a point which causes the two segments to become one again.

Each segment of the arrow can be in two modes: a Bezier mode and a straight mode. To select the mode for a segment do as follows:

1. Put the application in the edit-mode.
2. Left mouse click on the transition which needs to be divided.
3. Right mouse click on the segment of which the mode needs to be changed. A popup menu will be shown.
4. (a) Select ’Bezier Mode’ in the popup menu. The segment will become a Bezier curve.

   (b) Select ‘Straight Mode’ in the popup menu. The segment will become a straight line.

When a segment is in the Bezier mode two controls are shown as little black squares. By moving these controls the shape of Bezier curve can be controlled. Perform the follow steps to move these controls (figure A.6):

1. Put the application in the edit-mode.
2. Left mouse click on the transition of which the shape needs to be changed. When a segment is in Bezier mode, the two controls will become visible.
3. Press down on the control which needs to be moved and hold the left mouse click.
4. Without releasing move the mouse to another location. The control will follow the mouse movements.
5. Release the mouse on the desired location.

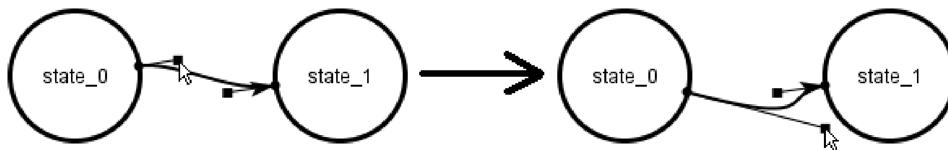

Figure A.6: Move a control which causes the segments to change shape.

Automatic Layout

To perform automatic layout of the states and transitions of the constraint automaton click the ‘Perform Layout’ button on the tool bar (button **K**, see figure A.1).

A.2.7 Exercise

In this subsection we create a small constraint automaton as an example. The constraint automaton will be one describing the behavior of a Sync channel (see subsection 4.1.1). This example will be used again in the next sections explaining the simulators.
To create the Sync channel constraint automaton perform the following steps:

1. Create a new empty constraint automaton document.
2. Change the port names of the constraint automaton to: `A, B`.
3. Add a state.
4. Create a transition from the state to itself.
5. Change the port names of the transition to: `A, B`.
6. Change the data constraint of the transition to: `data(A)==data(B)`.
7. Save the constraint automaton document to a CA-XML file.

Figure A.7 shows how the result should look like after performing these steps.

A.3 Timed Data Stream Language Acceptor Simulator

To start the TDSLAS select the document window of the constraint automaton (in the constraint automaton editor) that needs to be simulated and click on the TDSLAS button at the toolbar (button **L**, see figure A.1).
When you start the TDSLAS a window will be shown which consists of two parts (figure A.8):

- the simulator part (left),
- the timed data stream editor part (right).

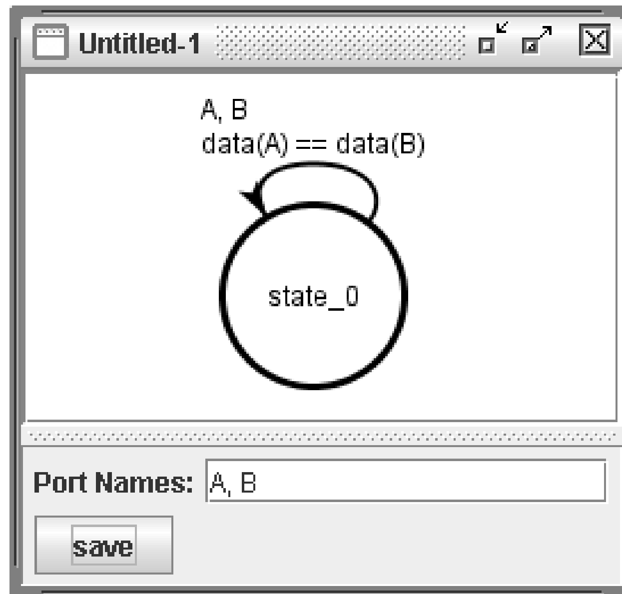

Figure A.7: A constraint automaton describing the behavior of a Sync channel.

Figure A.8: A screenshot of the TDSLAS.

A.3.1 Timed Data Stream Editor

The timed data stream editor consists mainly of a table, which shows the timed data streams. The second column of the table shows the time. In the first column there is an arrow which will indicate the current time step during the simulation. The other columns represents timed data stream, each associated with a port of the constraint automaton. The timed data streams can directly be edited through the table.
The four buttons at the bottom of the timed data stream editor do the following:

- Load: Loads a timed data stream into the editor from a TDS-XML file and associates it with the constraint automaton port indicated by the pulldown menu.
- Save: Saves the timed data stream indicated by the pulldown menu to a TDS-XML file.
- Insert: Inserts a row into the timed data stream table *after* the selected row.
- Delete: Deletes the row which is selected in the timed data stream table.

A.3.2 Simulator

The simulator part has two tables:

- the trace table, showing the history of the steps made during the simulation.
- the enabled transitions table, showing the transitions which are possible for a step.

The three buttons at the bottom of the simulator part do the following:

- Start: Starts the simulation with the timed data streams in the timed data stream editor.
- Step: Performs a simulation step if possible.
- Reset: Stop the simulation and resets the trace table and the enabled transitions table.

A.3.3 Exercise

In this subsection we simulate the constraint automaton of the example created in subsection A.2.7 in the TDSLAS.

1. Load the constraint automaton of the example into the constraint automaton editor.
2. Start the TDSLAS.

3. Create timed data streams such as shown in figure A.8.

4. Start the simulation. The enabled transitions table shows which transitions can be made.

5. Perform a simulation step. The step selected in the enabled transitions table will be made and added to the trace table. The enabled transitions table is again refilled with the possible transitions for the next step.

Step 5 will be repeated until no next step can be made, because the enabled transitions table is empty, which means that no transitions are possible.
By selecting a step in the trace table you can perform this step again. You can even select another transition for this step from the enabled transitions table.
The simulation is also visualized using colors in the constraint automaton that is viewed in a document window of the constraint automaton editor. The selected transition of the enabled transitions table is colored orange. The current state is colored green. The last transition made is colored yellow.

A.4 Reo Connector Simulator with Timed Data Streams

To start the RCSwTDS select the document window of the constraint automaton (in the constraint automaton editor) which needs to be simulated and click on the RCSwTDS button at the toolbar (button **M**, see figure A.1).
When you start the RCSwTDS a window will be shown which consists of two parts (figure A.9):

- the simulator part (left and bottom right),
- the timed data stream editor part (upper right).

The timed data stream editor is equivalent with the timed stream editor of the TDSLAS (see subsection A.3.1).

A.4.1 Simulator

The simulator part is almost equivalent with the simulator part of the TDSLAS. The first difference is the table shown in the bottom right of the window. This table shows the observed timed data streams during the simulation.
Second, the RCSwTDS tries to complete incomplete timed data streams. An incomplete entry in a timed data stream can be created with the string "`\read`" (see figure A.9).

A.4.2 Exercise

In this subsection we simulate the constraint automaton of the example created in subsection A.2.7 in the RCSwTDS.

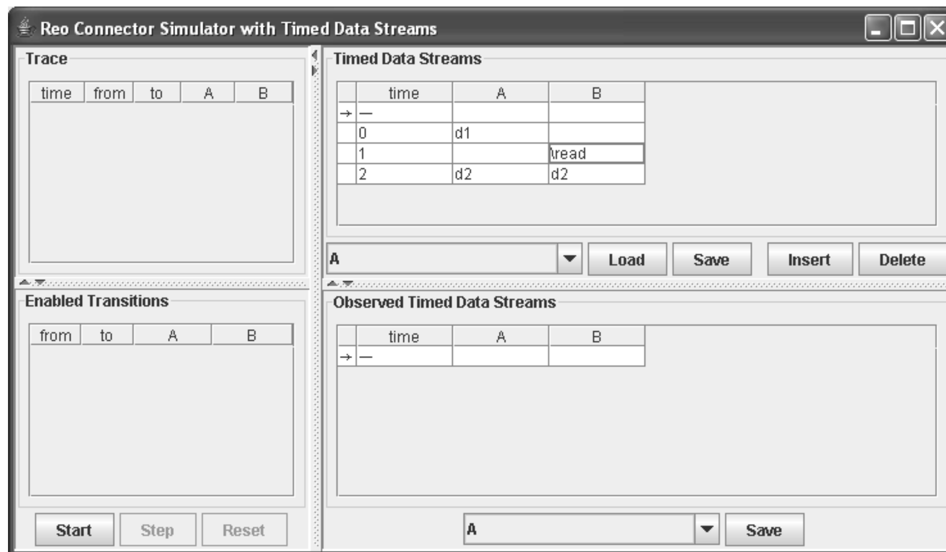

Figure A.9: A screenshot of the RCSwTDS.

1. Load the constraint automaton of the example into the constraint automaton editor.
2. Start the RCSwTDS.
3. Create timed data streams such as shown in figure A.9.
4. Start the simulation. The enabled transitions table shows which transitions can be made. It also shows the completed timed data tuple.
5. Perform a simulation step. The step selected in the enabled transitions table will be made and added to the trace table. The enabled transitions table is again refilled with the possible transitions for the next step. The observed timed data streams table is updated for the current observed data.

By selecting a step in the trace table you can perform this step again. You can even select another transition for this step from the enabled transitions table.
The simulation is also visualized using colors in the constraint automaton that is viewed in a document window of the constraint automaton editor. The selected transition of the enabled transitions table is colored orange. The current state is colored green. The last transition made is colored yellow.

A.5 Reo Connector Simulator with Components

To start the RCSwC click on the RCSwC button at the toolbar (button **N**, see figure A.1). This button is only enabled when constraint automaton documents are

opened in the constraint automaton editor.
When you start the RCSwC a window will be shown which consists of mainly three tables(figure A.10):

- a component ports table, showing the loaded components and the ports of these components,
- a connector ports table, showing the loaded constraint automaton (which will act as connectors) and the ports of these constraint automata,
- a connections table, showing the connections made between component ports and connector ports.

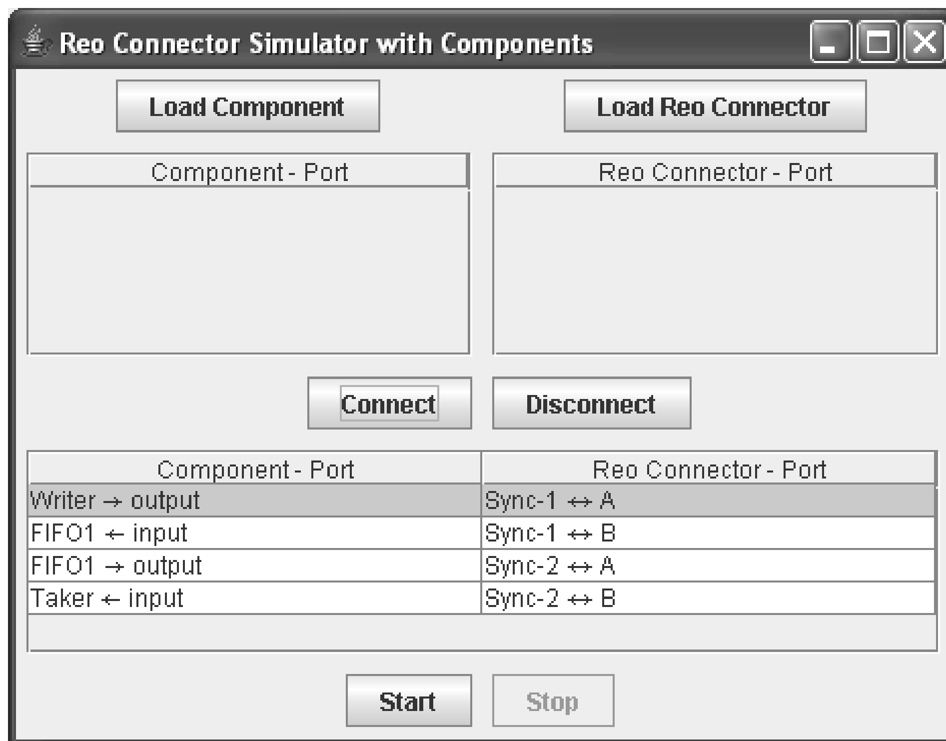

Figure A.10: A screenshot of the RCSwC.

A description of what the buttons do is given below.

- Load Component: Loads a component into the RCSwC from a Component-XML file.
- Load Reo Connector: Loads a constraint automaton as a connector into the RCSwC. Only the constraint automata which are opened in the constraint automaton editor can be loaded.

- Connect: Connects a component port with a connector port.
- Disconnect: Disconnects a component port from a connector port.
- Start: Starts the simulation.
- Stop: Stops the simulation.

A.5.1 Implementing Components

To implement a component you need to create a Java class that implements the **Component** interface from the **cwi.reo.rcswc.component** package. Through this interface the RCSwC is able to communicate with the component and pass the **Source** and **Sink** objects through which the component can perform write and take operations.

A component is loaded into the RCSwC through a Component-XML file, which contains the location of the component. The location can be given by the path to the class file. This path can be absolute or relative to the Component-XML file, for example:

```
<?xml version="1.0"encoding="TF-8">
<component>
<classfile>Taker.class</classfile>
</component>
```

Instead of specifying the path, the binary name of the class can be given. However, in this case the class needs to be reachable from the classpath. An example is given below.

```
<?xml version="1.0"encoding="TF-8">
<component>
<classname>
cwi.reo.rcswc.component.std.Taker
</classname>
</component>
```

The following standard components have already been implemented:

- Taker: A component which can be used to perform take operations. The binary name is **cwi.reo.rcswc.component.std.Taker**.
- Writer: A component which can be used to perform write operations. The binary name is **cwi.reo.rcswc.component.std.Writer**.
- FIFO1: A component which acts a FIFO1 channel. The binary name is **cwi.reo.rcswc.component.std.FIFO1**.

- Python component: This component will be explained in subsection A.5.2. The binary name is **cwi.reo.rcswc.component.std.PythonComponent**.

A.5.2 Implementing Python Components

The Python component can be used to load components written in the scripting language Python. The source code of the Python component needs to be specified in the Component-XML file and also the input and output port names that are going to be used.
The Python component is able to perform take and write operations through sinks and sources. These sinks and sources can be requested by the **componentProxy** as follows:

- `componentProxy.getInputPort("X")`: Returns the sink connected to port "X" (if not connected it returns null).

- `componentProxy.getOutputPort("Y")`: Returns the source connected to port "Y" (if not connected it returns null).

An example of a Component-XML file for a Python component is given below.

```
<?xml version="1.0" encoding="UTF-8"?>
<component>
<classname>
cwi.reo.rcswc.component.std.PythonComponent
</classname>
<inputports><name>X</name></inputports>
<outputports><name>Y</name></outputports>
<pythoncode>portX = componentProxy.getInputPort("X")
portY = componentProxy.getOutputPort("Y")
print portX.take()
portY.write("foo")</pythoncode>
</component>
```

A.5.3 Exercise

In this subsection we simulate the Sync channel constraint automaton of the example created in subsection A.2.7 in the RCSwC.

1. Create two CA-XML files of the constraint automaton of the example.

2. Load both constraint automata into the constraint automaton editor.

3. Start the RCSwC.

4. Create a Component-XML file for the following components (see subsection A.5.1):

(a) Taker
(b) Writer
(c) FIFO1

5. Load a Taker component in to the RCSwC and give it the name “Taker”.
6. Load a Writer component in to the RCSwC and give it the name “Writer”.
7. Load a FIFO1 component in to the RCSwC and give it the name “FIFO1”.
8. Load both constraint automata into the RCSwC and give them the names “Sync-1”, “Sync-2”. The RCSwC will simulate these constraint automata as Sync channels.
9. Connect the component ports and the connector ports as shown in figure A.10
10. Start the simulation. The RCSwC simulates the configuration of the components and connectors as shown in figure A.11.

Figure A.11: The configuration of the components and connectors.

Appendix B

Developer's Manual

The programming language used for the tool is Java 1.5, which can be downloaded from `http://java.sun.com/`. The tool is developed using NetBeans. This is an open source Java IDE from Sun, which can be downloaded from `http://www.netbeans.org/`. If one is interested in the source code of the tool, it is recommended to use NetBeans. However, this is not necessary.
The global structure of the directory containing the source code is as follows:

.\images	some images
.\lib	the external libraries
.\lib\Grace	Grace, the graph editor generator
.\lib\Grappa	Grappa, used for reading dot-files
.\lib\JavaCC	JavaCC, the parser generator
.\lib\JMDIFrameWork	Java MDI Application Framework, used for making Multiple Document Interface applications
.\lib\Jython	Jython, a pure java implementation of the programming language Python
.\lib\PrologCafe	PrologCafe, a translator system which translates prolog-source-files to java-source-files
.\nbproject	the NetBeans project files
.\src	the Java source files
.\src spec files	the specification files from which java-files are generated
.\src spec files\DataConstraintChecker	the prolog program used to check the dataconstraints of transitions
.\src spec files\DCParser	the specification files of the DataConstraint Parser
.\src spec files\Grace	the specification files of the graph editor
.\src spec files\NameSetParser	the specification files of the NameSet Parser