Universiteit Leiden
The Netherlands

Opleiding Informatica

A browser-based graphical editor for Reo networks

Maarten Smeyers

Supervisors:
Prof. dr. Farhad Arbab & Kasper Dokter

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

11-08-2018

Abstract

In this thesis we will present a new graphical editor for Reo. The main feature of our editor is that it will be ready to use without any installation process. This is achieved by running the program in a web browser, using the HTML5 canvas element and a JavaScript animation library called Fabric.js. This enables us to define, draw and change objects in the browser and provide user interaction. By composing several objects we have created channels and nodes that can be used to draw Reo connectors. Additionally, we provide tools to modify the conector as desired. The result can be exported in several formats and it is also possible to import textual Reo from other sources. We believe that our implementation provides all the core functionality required for a graphical Reo editor and also provide some suggestions for future improvements.

Contents

Chapter 1

Introduction

Reo [AM02] is an exogenous coordination model based on an atomic set of channels. These channel types are not provided by Reo, it expects them to be provided by "users". Reo can then be used for building more complex connectors by composing these channels.

One of the goals of Reo is to provide a sort of glue language that can be applied to concurrent components that have been programmed in conventional programming languages. In most of these systems, the components exchange messages in order to synchronize operations or lock resources. However, this makes them dependent on each other; if one program is modified, all the others will need to be updated as well. Additionally, the concurrency model used may be difficult to interpret by someone not involved in the original implementation. Reo seeks to solve both problems: by attaching programs to Reo connectors the coordination is done from outside the program. Additionally, Reo connectors have a graphical representation that makes it much easier to understand the underlying concurrency model.

A natural way for users to build a Reo connector would be to provide a graphical user interface where channels can be drawn and connected to form a network. Currently, this interface is provided as a part of the Extensible Coordination Tools [ECT]. However, the current version of the Extensible Coordination Tools require Java, Flash and Eclipse to be able to function. This means that a user should first download, install and configure these programs before they are able to draw a connector. We believe this process is too complex and therefore it is our goal to create a new graphical editor to overcome these limitations.

1.1 Goal

Our goal is to create a browser-based graphical editor for Reo. We will use modern web technologies supported natively by browsers to enable users to operate the editor out of the box, without any need for plugins or downloads. The code for our editor will be hosted on a publicly available repository, so that anyone who is interested in Reo can use it.

1.2 Thesis Overview

In this thesis we will take a look at the following subjects:
This chapter contains the introduction; In chapter 2 we will explain the basics of Reo; Chapter 3 discusses related work; in chapter 4 we will elaborate on the techniques we used for our implementation. Chapter 5 evaluates the features of our editor; And finally we will discuss our conclusions in chapter 6.

Chapter 2

Basics of Reo

In this chapter we will introduce the basics of Reo. We will introduce some basic elements and demonstrate how they can be used to create a Reo connector. Readers that are already familiar with Reo can skip this chapter and continue with chapter 3.

2.1 Channels

A channel is a medium of communication that consists of two ends and a constraint on the dataflows observed at those ends [Arb11]. A channel end can be one of two types: *source* and *sink*. A source channel end accepts data into its channel, and a sink channel end dispenses data out of its channel. A channel can have two ends of the same type. For example: if a channel has two source ends, then all data accepted by this channel is lost. This type of channel can however be very useful in the composition of more complex connectors.

The constraints that are placed on the dataflows are important to Reo, but our graphical editor has no need for them. As we are only working with a visual representation of channels instead of actual channels it is sufficient for our editor to know what a channel is called and what it should look like. This allows our code to be as general as possible and makes it easier to edit or expand the available set of channels.

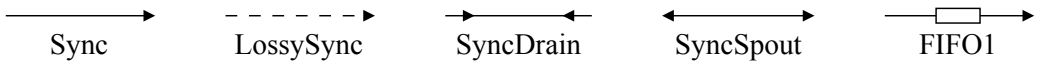

Figure 2.1: A set of Reo channels.

Channels represent an atomic unit of coordination in Reo. Each of the channels shown in figure 2.1 has a specified way of coordinating actions that are performed on its channel ends. Below we will describe the behaviour of these channels:

Sync A Sync channel transmits data from its source end to its sink end synchronously. If data is written to its source end, the operation will only succeed if there is also a take operation on the sink end. Similarly, the success of a take operation on its sink end depends on the presence of a write operation on its source end.

LossySync A LossySync works quite like a Sync, but a write operation on its source end will always succeed. If there is no one to take the data on its sink end, the data item will simply be lost. If there is a pending take operation on the sink end, then the data will be transmitted exactly like in a Sync.

SyncDrain A SyncDrain is a synchronous channel that does not dispense data. All data items that are written to a SyncDrain will simply be lost, as long as two simultaneous write operations are performed on both channel ends.

SyncSpout The SyncSpout works exactly like the SyncDrain, but this channel generates data instead of losing it.

FIFO1 A FIFO1 is the only channel in this list that can store a data item. A FIFO1 will be empty until a data item is written to its source end. After a data item has been written to this channel, it will be stored and the FIFO1 will be full. Any subsequent take operation on the sink end will always succeed and empty the FIFO1 again. If a write operation is performed while the FIFO1 is full, it will not succeed until a take operation has emptied its storage. A FIFO1 is asynchronous; a write operation and a take operation cannot be performed simultaneously.

2.2 Nodes

Complex connectors can be constructed by joining channels together in nodes. A node consists of a set of channel ends and can have one of three node types, depending on the types of channel ends in its set. A node with only source ends in its set is a *source node*, and a node with only sink ends is a *sink node*. If both types are present, the node is called a *mixed node*. Figure 2.2 shows the three node types.

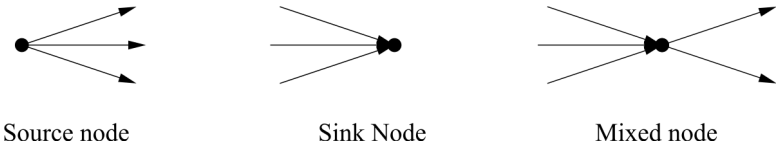

Figure 2.2: Reo nodes [Arb11].

If a component is connected to a source node, it can write a data item to the node. This will however only succeed if all of the connected source channel ends accept the data. If this is the case, the data item is synchronously written to every channel end.

If a component is connected to a sink node, it can try to take a data item from the node. For this to work, at least one connected sink channel end must offer a data item. If multiple channel ends offer a data item, one of them is selected nondeterministically and this data item is taken by the component.

A mixed node combines the behaviour of a source node and a sink node: it nondeterministically takes a data item from a source channel end and replicates it to all connected sink channel ends.

2.3 Connectors

According to [AM02], "a connector is a set of channel ends and their connecting channels organized as a graph of nodes and edges such that:

- Every channel end coincides on exactly one node.
- Zero or more channel ends coincide on every node.
- There is an edge between two (not necessarily distinct) nodes if and only if there is a channel whose ends coincide on those nodes."

Connectors are created by composing channels to create advanced coordination models. By just combining a few channels, completely new models can be created. For example, figure 2.3 shows an exclusive router.

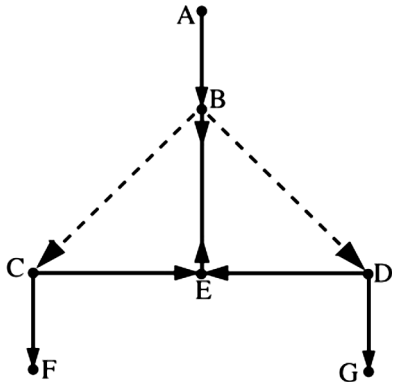

Figure 2.3: An exclusive router connector [CCA07].

It is composed from just three types of channel: Sync, LossySync and a SyncDrain. Data is written to node A and then taken from node F or node G. If a take operation is pending on both F and G, one of them is selected nondeterministically, ensuring that only one take operation succeeds. Hence the name exclusive router: the success of one take operation excludes the success of the other. Figure 2.4 shows the flow of data through the exclusive router. It clearly shows the two possible paths a data item can take.

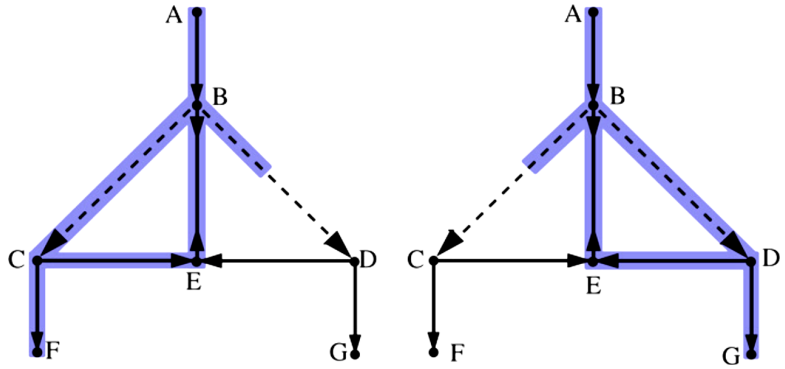

Figure 2.4: Possible data flow behaviour of the exclusive router. The thick solid line marks the part of the connector where data flows synchronously. In unmarked parts, no data flows. [CCA07].

Chapter 3

Related Work

Since the introduction of Reo [AM02], several papers have been pubished about Reo itself and projects that are related to Reo.

A Coinductive Calculus of Component Connectors [AR02] discusses a semantic model for Reo based on timed data streams. This semantic model is applied to the channels introduced in chapter 2 and is then used for the composition of channels into more complex connectors.

Connector colouring I: Synchronisation and context dependency [CCA07] proposes a model of connector colouring for determining the behaviour of a Reo connector. This provides a means to determine the routing alternatives for data flow. Although the current build of our Reo graphical editor does not include connector colouring, it could be included in a future version to give a user insight into the behaviour of a Reo connector as they are designing it.

Treo: Textual Syntax for Reo Connectors [DA18] introduces a textual syntax for the specification of Reo connectors. This syntax can be compiled into executable code that can compete with or even beat the performance of programs written in languages such as C or Java using conventional concurrency constructs. Our editor includes a function to convert a graphical Reo connector to Treo and vice versa. This function was developed by Ali Mirlou and is out of the scope of this thesis.

Chapter 4

Implementation

In this chapter we will explain the choices we made for the implementation of our editor. We will take a look at the technologies that are required to reach our goal and discuss their limitations.

4.1 Choice of technology and library

In the past, making a drawing in a web browser was quite difficult. It usually required a plugin to run in the browser, and then you could create the drawing in the plugin. This changed in HTML5 with the introduction of the canvas element [Can]. This element creates a blank canvas in the web page and enables us to use JavaScript to create a drawing on this canvas. However, drawing on a canvas requires the use of low-level commands. Additionally, commands modify the entire canvas bitmap without enabling us to easily modify something without redrawing the entire canvas. This makes it inefficient for our use case: ideally, we can move a channel by just redrawing it in a new location and leaving all other elements in place. In order to do this, we will use a JavaScript library on top of canvas.

Fabric.js [Fab] is a JavaScript HTML5 canvas library that provides an interactive object model on top of the canvas element. This model allows us to work with objects directly, rather than modifying the entire canvas. The use of objects instead of simply drawing shapes on the canvas enables us to include references to other objects and use this method to build a data structure of linked objects. If one object is then modified, we can ensure that all connected objects are updated as well. This is shown in the Stickman demo [Stc], which is shown in figure 4.1. The first build of our graphical Reo editor was based on this demo. By default, Fabric.js draws a set of borders and controls around each object when it is selected. Using these controls an object can be moved, scaled or rotated. The controls can easily be hidden using an object parameter so that custom controls can be defined. Additionally, selection of specific objects can be disabled altogether.

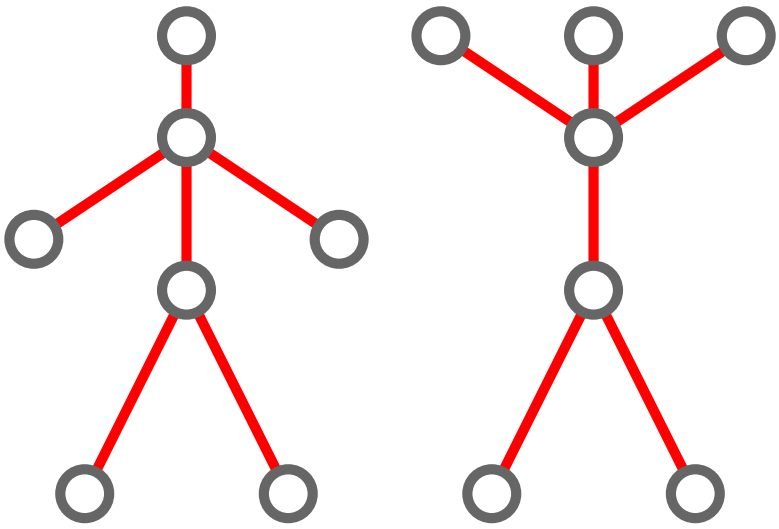

Figure 4.1: The Fabric.js stickman demo.

4.2 Library limitations

Fabric.js objects can be grouped and can then be manipulated together. We have tried to use this feature to create a group with all objects that are needed to draw a channel and then use a single scaling operation to change the size of the channel. However, objects that are part of a group can only be scaled together or not at all. This means that if we enable scaling, the entire channel will look stretched. This is shown in figure 4.2

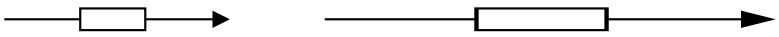

Figure 4.2: Scaling of a Fabric.js group.

In order to solve this issue, we have created a custom channel object. This objects has references to all the individual objects that are needed to draw a specific channel. They can then be updated individually instead of all together. Each object has a parameter that indicates if it should be scaled. Another parameter is used to enable or disable rotation. The resulting datastructure is displayed in figure 4.3. Channels cannot be edited directly, but are updated when the position of the node is changed. The node then calls an update function for each connected channel, which in turn update the position of all their objects.

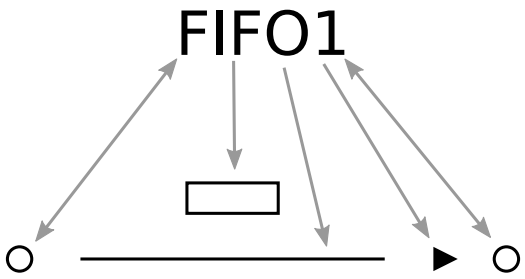

Figure 4.3: Datastructure of a FIFO1 channel.

4.3 Custom channels

As specified before, Reo does not provide a standard list of channel types. Therefore, these should be loaded separately and be editable by the user. In order to achieve this, we have chosen to load the channel types from a separate file. This file contains an array of all channel types in JSON format. For each channel, we use a few parameters to define the name of the channel and the channel end types. Finally, the channel contains a list of objects that need to be drawn to display the channel. These objects also contain positional information to redraw the object when the channel is scaled, rotated, or moved.

```
{
  "class":"channel",
  "name":"sync",
  "end1":"source",
  "end2":"sink",
  "parts":
  [
    {
      "type":"triangle",
      "left":192,
      "top":150,
      "width":8,
      "height":8,
      "fill":"#000",
      "baseAngle":90,
      "referenceAngle":270,
      "referenceDistance":8,
      "referencePoint":"node2",
      "rotate":true,
      "scale":false,
      "evented": false
    }
  ]
}
```

Figure 4.4: (Partial) implementation of a Sync channel.

Figure 4.4 shows a partial implementation of the Sync channel. This example was chosen because it shows how the positioning of individual channel parts is achieved. The triangle, which is a part of the Sync channel, has three parameters to determine its positioning: `referencePoint`, `referenceDistance` and `referenceAngle`. From these parameters, the editor knows that it should position the triangle relative to the position of node 2 of the editor. It should be drawn on the left side of the node, at a distance of 8 pixels. A fourth parameter, `baseAngle` is used to determine the default rotation of the triangle itself. In this example, the angles are shown in degrees, the other parameters are displayed in pixels.
If a channel is moved or rotated, the position of the triangle is recalculated from the position of node 2 and the angle of the channel itself. Additionally, the `scale` and `rotate` parameters will determine if the triangle itself should be scaled and rotated as well.

Chapter 5

Features

Figure 5.1 shows the main interface of our graphical editor. In the center the main canvas is shown, where the user can immediately start drawing a connector. On the right is a list of controls to determine what should be drawn on the canvas, as well as buttons to download the canvas contents either in image or text format. On the left side is a text editor that shows the textual representation of the canvas contents. Thanks to an integration built by Ali Mirlou, this textual representation can also be updated and then redrawn on the main canvas.

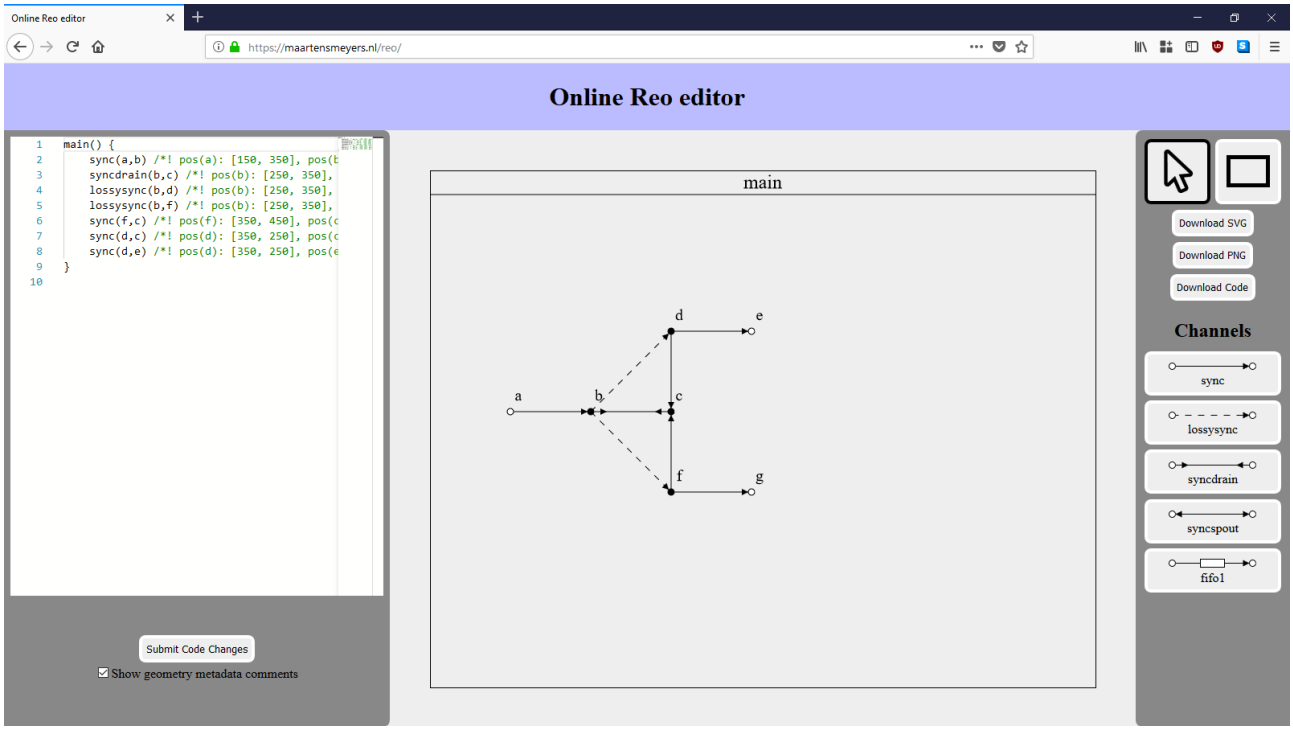

Figure 5.1: Our graphical editor.

Our editor can be user either by downloading the files to a local computer or accessing the files on a webserver. In both cases the interface will be loaded in a web browser and then be available to the user.

5.1 Window manager

Reo can be used for more than just single connectors. It is possible to create multiple components of a system and have them interact with each other. Our editor provides support for this by enabling users to draw components and connect them to each other. This was implemented with a window manager, which allows components to be drawn over each other without affecting the connections between them. This way, it is possible to create one component, then design a second one, and finally link them together. Figure 5.2 shows two overlapping components that are connected being designed in the editor.

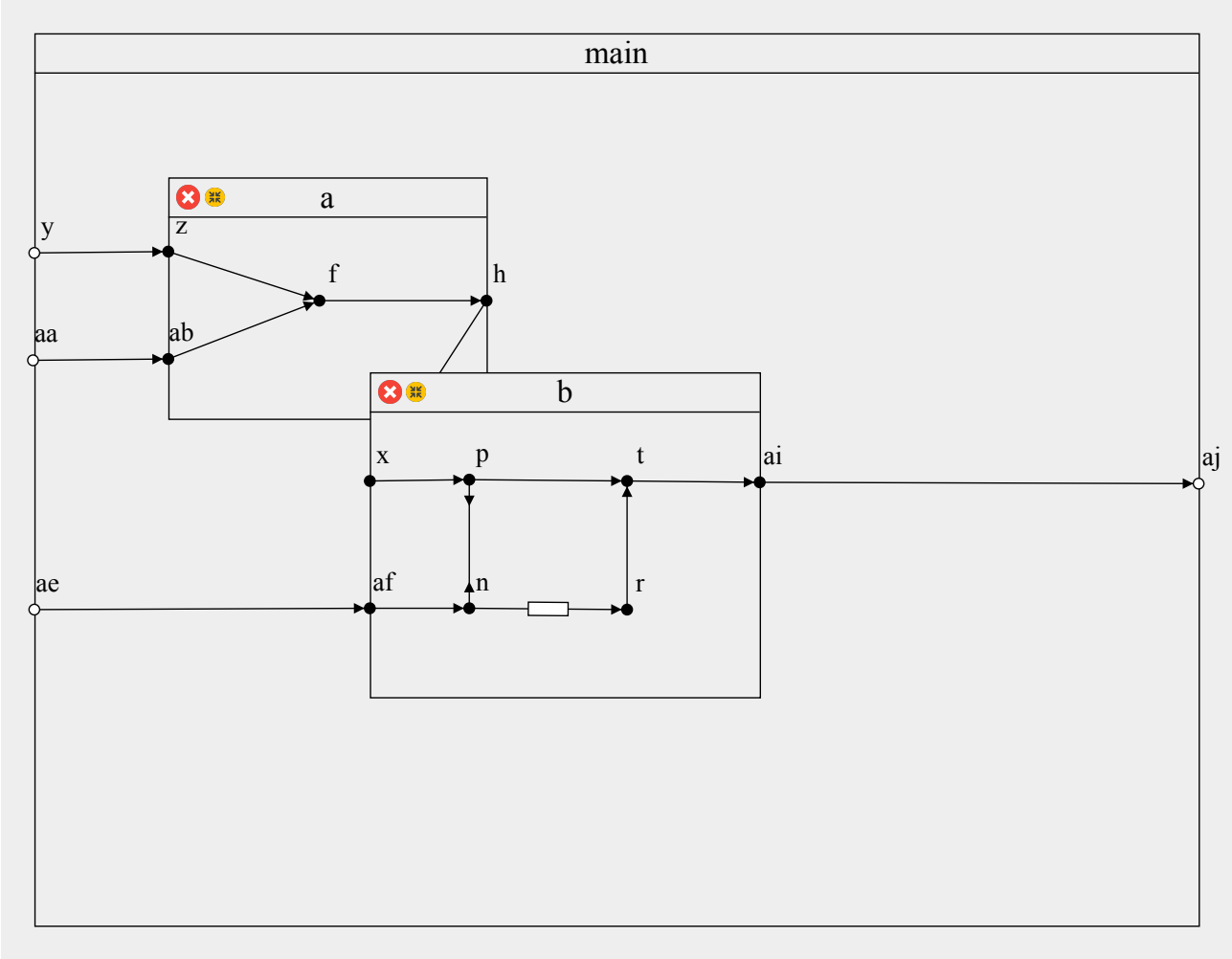

Figure 5.2: Design of two overlapping components.

The Sync channel between nodes `h` and `x` is only partially shown because of the overlap, but for the data connection this does not matter. If we want to make the channel more easily visible, we are free to move the components around until they no longer overlap. We can also select component `a` to bring it to the front.

5.2 Circular channels

As we saw in the definition of a connector in section 2.3, the nodes that are connected by a channel are not necessarily distinct. Reo allows for a channel to connect both of its ends to the same node and thus create a circular channel, as shown in figure 5.3. Our editor automatically checks if both channel ends refer to the same node and draws a circular channel if this is the case.

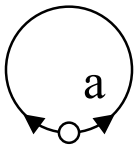

Figure 5.3: A circular SyncDrain channel.

5.3 Merge, split and delete

In order to create connectors, it should be possible to compose channels together. In our editor, it is possible to create a new channel by starting at an existing node. This way, you can quickly draw an entire connector in one go. It is also possible to merge existing channels by dragging a node from one channel end over a different node. As soon as the nodes are close enough together, the editor will show the suggestion to merge them by moving the active node to the position of the other one. As soon as the mouse button is released, the nodes are merged and all connected elements will be updated.

While editing connectors is may be necessary to move existing channels to a different nodes. To enable this, we have created a split mode of operation in our editor. If the split mode is selected, you can select one or multiple channels and then move the shared node to a new position. This way, a connector can be edited without the need to delete and redraw the channels.

However, if channels, nodes or entire components are no longer needed, they can be removed from the connector. This can simply be done by selecting the unwanted object and pressing the Delete button. Alternatively, the delete symbol in the editor can be used.

Chapter 6

Conclusions and future work

We believe that the browser-based graphical editor we have presented in this thesis will enable users to easily create and edit Reo connectors. The current build of the editor provides all the core functionalities required to enable beginning or experienced users of the Reo coordination model to work on connectors without the need for any installation process.

The conversion of graphics to text and text to graphics that has been integrated in our editor enables users to store and load connectors and also provides an integration with the other tools that are available for Reo networks. We think that this integration could further be improved by implementing our graphical editor and the Reo animation engine together in one tool. As a result it would be possible to draw and run Reo networks without the need to move the code from one tool to another.

The way we have chosen to implement our custom channels enables users to define their own channels to expand or replace the existing set. This can be done by modifying the file that stores the channels and adding the desired objects according to the grammar specified in the Fabric.js documentation [Doc]. We believe however that this process can be further optimized so that it does not require end users to edit any code themselves. We suggest the implementation of a custom channel editor, where the user is presented with two nodes. They can then add the desired objects and freestyle drawings to this view until it looks as desired and then save it as a channel to be used in the Reo editor.

Finally, while our editor provides a mechanism to add and delete channels, nodes and components, there are no means to undo a change. If a component is deleted by accident, it has to be redrawn from scratch. Therefore, we think that the usability of our editor could benefit from an undo-redo functionality, so that users can quickly turn back unwanted modifications.

Bibliography

[AM02] Farhad Arbab and Farhad Mavaddat. Coordination through channel composition. In *Proceedings of Coordination Languages and Models*, volume 2315 of *Lecture Notes in Computer Science*, pages 22–39. Springer, 2002.

[AR02] Farhad Arbab and Jan J. M. M. Rutten. A coinductive calculus of component connectors. In *Recent Trends in Algebraic Development Techniques*, volume 2755 of *Lecture Notes in Computer Science*, pages 34–55. Springer, 2002.

[Arb11] Farhad Arbab. Puff, the magic protocol. In *Formal Modeling: Actors, Open Systems, Biological Systems*, volume 7000 of *Lecture Notes in Computer Science*, pages 169–206. Springer, 2011.

[Can] HTML Standard - the canvas element. `https://html.spec.whatwg.org/multipage/canvas.html#the-canvas-element`.

[CCA07] Dave Clarke, David Costa, and Farhad Arbab. Connector colouring I: synchronisation and context dependency. *Sci. Comput. Program.*, 66(3):205–225, 2007.

[DA18] Kasper Dokter and Farhad Arbab. Treo: Textual syntax for reo connectors. *CoRR*, abs/1806.09852, 2018.

[Doc] Fabric.js documentation. `http://fabricjs.com/docs/`.

[ECT] Extensible Coordination Tools home page. `http://reo.project.cwi.nl/reo/wiki/Tools`.

[Fab] Fabric.js Javascript Canvas Library. `http://fabricjs.com/`.

[Stc] Stickman | Fabric.js Demos. `http://fabricjs.com/stickman`.