Internal Report 2010–10

August 2010

Universiteit Leiden
Opleiding Informatica

Physical Reo

Roald de Vries

MASTER'S THESIS

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Contents

1 Introduction **2**

2 Graphs and Circuits **4**

2.1 Graphs 4

2.2 Electric systems 6

3 Reo **12**

3.1 Connectors as Multi-Graphs 14

3.2 Semantic Models 16

3.2.1 Common Problems 16

3.2.2 Connector Coloring 18

3.2.3 Constraint Automata 20

3.2.4 Constraint Satisfaction 21

4 Physical Reo **22**

4.1 Simple Connectors 23

4.1.1 Voltage Based Approach 25

4.1.2 Current Based Approach 26

4.2 Complex Connectors 28

4.2.1 Voltage Based Approach 29

4.2.2 Current Based Approach 30

4.3 A Practical Application 31

4.4 Evaluation 39

5 Conclusion **44**

Chapter 1

Introduction

Concurrency is one of the hot topics in nowadays computing. The interest in it surprisingly comes from two distinct directions. Firstly, the omnipresence of the Internet and the emergence of web services allow programs to be distributed over different computers connected through a direct, real-time connection. Secondly, the increasing number of processors (or cores) in single computers and the cause of this increase, the diminishing enhancement of single processor (or core) performance, raise the need to exploit this increasingly parallel computing power.

One way of implementing concurrency is through coordination of (sequential) components. The Reo coordination language takes such an approach. It "glues" existing components together into new components by connecting their ports to nodes, and by interconnecting nodes through channels. These channels are simple two-sided components through which data can flow. Together with nodes they form the building blocks of Reo, and they decide where data does and does not flow.

Reo [1] is a language under development. Its desired behavior is formulated, but not all implications of that behavior are fully understood, and no implementation exists that is both correct and satisfyingly efficient. To fill this gap, different semantic models have been constructed; models of Reo that give a more formal meaning to the language or some aspect of the language. Often these models choose to narrow their domain of application in favor of providing more efficient or conceptually simpler implementations.

This thesis introduces a new semantic model, Physical Reo. It models Reo as a continuous flow system, instead of the discrete flow system that it is. It takes physical flow systems, primarily electric current systems, as

inspiration, and derives its name from this.

The purpose of Physical Reo is to describe Reo as faithfully as possible as such a physical system, and to benefit from the advantages that physical systems inherently have. These include that the influence that channels and components exercise on each other — possibly through many intermediate channels and components — can be described by local constraints, and that such systems under the influence of these local constraints can optimize themselves to valid interactions. These properties can in turn be used to develop more efficient (distributed) methods for use with Reo.

Although this thesis does not present a definitive implementation of Physical Reo, what it does present are two "semantic models" of or approaches to implementing Physical Reo. The first approach is "voltage based", in the sense that its focus is on the analogy with voltages in electric circuits. The second approach is "current based", focusing on the analogy with electric current. Moreover, an efficient algorithm derived from the voltage based approach is also presented. Apart from the algorithms usefulness because of its low complexity, it is a good example of how Physical Reo can be the inspiration for an efficient implementation of Reo.

The rest of this thesis is organized as follows. Chapter 2 will discuss mathematical (physical) structures underlying electric circuits. Chapter 3 describes Reo and its most important semantic models. Chapter 4 introduces Physical Reo, shows how it can be used for practical applications and compares it to existing semantic models. Finally, chapter 5 will draw the conclusions and suggest future work.

Chapter 2

Graphs and Circuits

2.1 Graphs

This section introduces the graph terminology and related matrix representations and operations as used throughout this thesis.

A graph is normally described as an ordered pair of sets $G = (N, E)$, where N contains nodes (or vertices), and E contains edges. To disambiguate, N_G and E_G may be used. Directed graphs, or digraphs, have directed edges, identified by a pair of nodes $E \subseteq N \times N$. Undirected graphs identify their edges by a set of nodes $E \subseteq \binom{N}{1} \cup \binom{N}{2}$, where $\binom{N}{i}$ is a binomial coefficient that represents the set of all combinations of i elements of N, and $\binom{N}{1}$ is the set of edges with both sides connected to the same node.

Some important graph terms are an edge's endpoints, a directed edge's tail and head, a nodes leaving and entering edges, and source and sink nodes. They are defined in equations (2.1) to (2.4). An edge's endpoints are the nodes to which it is connected; its tail is the first node of its node pair, its head is the second. An edge is said to enter a node if that node is its head, and to leave a node if that node is its tail. A source node is a node without entering edges, and a sink node is one without leaving edges.

$$\texttt{ends}(e) = \{n, n'\} \Leftrightarrow e = (n, n') \vee e = (n', n) \vee e = \{n, n'\} \tag{2.1}$$

$$e = (n, n') \Leftrightarrow \texttt{tail}(e) = n \wedge \texttt{head}(e) = n' \tag{2.2}$$

$$e\,\texttt{leaves}\,n \Leftrightarrow \texttt{tail}(e) = n \qquad e\,\texttt{enters}\,n \Leftrightarrow \texttt{head}(e) = n \tag{2.3}$$

$$\texttt{is}_\texttt{s}\texttt{rc}(n) \Leftrightarrow \nexists e \in E : e\,\texttt{enters}\,n \qquad \texttt{is}_\texttt{s}\texttt{nk}(n) \Leftrightarrow \nexists e \in E : e\,\texttt{leaves}\,n \tag{2.4}$$

Multi-graphs allow multiple edges between the same vertices. These are expressed by splitting the sets of edges from their incidences on vertices. This results in the 'incidence structure'-like notation $G = (N, E, I)$ where E is an abstract set of edges, and (for digraphs) $I : E \to N \times N$ determines its incidence.

A path π in a directed multi-graph G is a sequence of edges, where the head of every edge equals the tail of its successor if it has a successor (2.5). A cycle is a path in which the head of its last edge equals the tail of its first edge (2.6).

$$\texttt{paths}_G = \{(e_0, \ldots, e_n) \in E_G^* | \, \texttt{head}(e_i) = \texttt{tail}(e_{i+1} \text{ for } 0 \leq i < n\} \tag{2.5}$$

$$\texttt{cycles}_G = \{(e_0, \ldots, e_n) \in \texttt{paths}_G \, | \, \texttt{head}(e_n) = \texttt{tail}(e_0)\} \tag{2.6}$$

Graphs can be united to create a new graph. In this thesis, only non-disjoint unions, as defined in (2.7), are used. The other way around, graphs can be decomposed into subgraphs of which the union is the original graph. The ultimate decomposition is into graphs consisting of one edge, which is the graph induced by that edge (2.9).

$$G \cup G' = (N_G \cup N_{G'}, E_G \cup E_{G'}, I_{G \cup G'}) \tag{2.7}$$

$$I_{G \cup G'} : e \mapsto \begin{cases} I_G(e) & \text{if } e \in E_G \\ I_{G'}(e) & \text{if } e \in E_{G'} \end{cases} \tag{2.8}$$

$$\texttt{ind}_G(e) = (\texttt{ends}(e), \{e\}, I_G) \tag{2.9}$$

A convenient representation of a directed multi-graph is its incidence matrix A, where its edges form the row indexes and its nodes the column indexes. The incidence matrix of a graph is the subtraction of its head matrix A_{head} from its tail matrix A_{tail} as described in equation 2.10.

$$A = A_{\text{tail}} - A_{\text{head}} \tag{2.10}$$

$$A_{\text{tail}} = \begin{pmatrix} A_{\text{tail},e_0,n_0} & \dots & A_{\text{tail},e_0,n_m} \\ \vdots & \ddots & \\ A_{\text{tail},e_n,n_0} & & A_{\text{tail},e_n,n_m} \end{pmatrix}$$

$$A_{\text{tail},e_i,n_j} = \begin{cases} 1 & \text{if } \exists n' \in N : I(e) = (n, n') \\ 0 & \text{otherwise} \end{cases} \tag{2.11}$$

$$A_{\text{head},i,j} = \begin{pmatrix} A_{\text{head},e_0,n_0} & \dots & A_{\text{head},e_0,n_j} \\ \vdots & \ddots & \\ A_{\text{head},e_i,n_0} & & A_{\text{head},e_i,n_j} \end{pmatrix}$$

$$A_{\text{head},e_i,n_j} = \begin{cases} 1 & \text{if } \exists n' \in N : I(e) = (n', n) \\ 0 & \text{otherwise} \end{cases} \tag{2.12}$$

In matrix manipulations of such incidence matrices, it can be useful to set some rows or columns to zeros. This can easily be accomplished by a multiplication by a matrix such as in equation (2.14), where σ can specify a subset of N or E in any (intuitively) clear way, for example as a subset or a predicate.

$$1_\sigma : \quad 1_{\sigma,n,n'} = \begin{cases} 1 & \text{if } n = n' \wedge n \in N_\sigma \\ 0 & \text{otherwise} \end{cases} \tag{2.13}$$

$$1_{\sigma,e,e'} = \begin{cases} 1 & \text{if } e = e' \wedge e \in E_\sigma \\ 0 & \text{otherwise} \end{cases} \tag{2.14}$$

2.2 Electric systems

This section will discuss electric current systems with two-sided components as (directed) multi-graphs. After the general case, linear analog systems will be discussed more in depth. Nodes, edges and the incidence function of a graph G will be written hereafter as N_G, E_G and I_G, to disambiguate from physical quantities such as electric current I.

An electric circuit C is described as an edge-labeled directed multi-graph. Every edge is labeled with an electrical component definition by $\texttt{comp} : E_G \to \mathcal{C}_\text{e}$, where the set of all electrical components $\mathcal{C}_\text{e}$ consists of, among others,

voltage sources described by $\mathcal{V} = \{x \cdot V | x \in \mathbb{R}\}$, constant resistors described by $\mathcal{R} = \{x \cdot \Omega | x \in \mathbb{R}, x > 0\}$, or resistors that depend on the potential difference over them, described by $\mathcal{R}^{\mathcal{V}}$. From here on, only voltage sources and constant resistances will be considered, though. Electric systems of this sort are linear systems.

Different graphs and component labelings can describe the same circuit by inverting edges and components accordingly, which is expressed by the electric circuit equivalence relation in (2.15).

$$
\begin{aligned}
C \sim C' \Leftrightarrow N_{G_C} = N_{G'_C} \wedge E_{G_C} = E_{G'_C} \wedge \\
\forall e \in E_{G_C} : \mathtt{ends}_{G_C}(e) = \mathtt{ends}_{G'_C}(e) \wedge \\
\forall e \in E_{G_C} : \begin{cases} \mathtt{comp}_C(e) = -\,\mathtt{comp}_{C'}(e) & \text{if } I_{G_C}(e) \neq I_{G'_C}(e) \wedge \mathtt{comp}_C(e) \in \mathcal{V} \\ \mathtt{comp}_C(e) = \mathtt{comp}_{C'}(e) & \text{otherwise} \end{cases}
\end{aligned}
\tag{2.15}
$$

To describe the state S that an electric circuit is in, a few other labelings are used. These consist of the potential difference $\Delta V : E_G \to \mathcal{V}$, the electric current $I : E_G \to \mathcal{I}$ with $\mathcal{I} = \{x \cdot A | x \in \mathbb{R}\}$, and node voltage $V : N_G \to \mathcal{V}$. The partial resistance labeling $R : E_G \to \mathcal{R}$ is only defined on resistor components, and gives the resistance specified by $\mathtt{comp}(e)$. Electric circuit states can also be described by equivalent labeled graphs (2.16).

$$
\begin{aligned}
S \sim S' \Leftrightarrow C_S \sim C_{S'} \wedge \forall n \in N_{G_S} : V_S(n) = V_{S'}(n) \wedge \\
\forall e \in E_{G_C} : \begin{cases} \Delta V_S(e) = -\Delta V_{C'}(e) \wedge I_S(e) = -I_{S'}(e) & \text{if } I_{G_C}(e) \neq I_{G'_C}(e') \wedge \mathtt{comp}_C(e) \in \mathcal{R} \\ \Delta V_S(e) = \Delta V_{C'}(e) \quad \wedge I_S(e) = I_{S'}(e) & \text{otherwise} \end{cases}
\end{aligned}
\tag{2.16}
$$

Ohm's law (2.17) can now be expressed in this terminology for all resistor edges (2.18).

$$V = I \cdot R \tag{2.17}$$

$$\forall e \in E_G : \Delta V(e) = \begin{cases} I(e) \cdot R(e) & \text{if } \mathtt{comp}(e) \in \mathcal{R} \\ \mathtt{comp}(e) & \text{if } \mathtt{comp}(e) \in \mathcal{V} \end{cases} \tag{2.18}$$

Kirchhoff's laws (2.19, 2.20) describe (the stable state of) an electric system.

$$\sum \Delta V = 0 \tag{2.19}$$

$$\sum I = 0 \tag{2.20}$$

Kirchhoff's Voltage Law describes that the voltage drop through every directed cyclic path is zero. In graph terminology, this is expressed by (2.21). Note that this holds in all equivalent graphs of G as well.

$$\forall \pi \in \texttt{cycles}_G : \sum_{e \text{ on cycle } \pi} \Delta V(e) = 0 \tag{2.21}$$

Kirchhoff's Current Law describes conservation of current; the total current going in to a node equals the total current going out of that node. In graph terminology it can be expressed as (2.21).

$$\forall n \in N_G : \sum_{\{e \in E_G | n \in \texttt{ends}(e)\}} I(e) = 0 \tag{2.22}$$

Linear systems are 'solvable' in the sense that a stable state can be computed for them, which means that ΔV and I can be calculated for resistor edges with Kirchhoff's and Ohm's laws. Kirchhoff's Voltage Law can be interpreted as stating that the potential difference between two vertices is independent of the path between them. This makes it possible to assign an electric potential to every node, which is its potential difference with respect to a chosen ground node.

Kirchhoff's and Ohm's laws can be nicely expressed with linear algebra. Node voltages can be put in a vector $\mathbf{V}_{N_G}$, and multiplication by incidence matrix A_G gives the vector of potential differences $\mathbf{\Delta V}_{E_G}$. This is the matrix form of Kirchhoff's Voltage Law (2.23).

$$\mathbf{\Delta V}_{E_G} = A_G \cdot \mathbf{V}_{N_G} \tag{2.23}$$

Ohm's law in matrix form (2.24) uses the diagonal matrix R_G^{-1}, which is the diagonal matrix containing $R(e)^{-1}$ for all edges. The result $\mathbf{I}_{E_G}$ is the vector of currents through the edges of G.

$$\mathbf{I}_{E_G} = R_G^{-1} \cdot \mathbf{\Delta V}_{E_G} \tag{2.24}$$

$$(R_G^{-1})_{e_i,e_j} = (R_{G,e_i,e_j})^{-1} \qquad R_{G,e_i,e_j} = \begin{cases} R(e_i) & \text{if } e_i = e_j \\ 0 & \text{otherwise} \end{cases} \tag{2.25}$$

Kirchhoff's Current Law (2.26) again makes use of the incidence graph, but now in transposed form. Vector $\mathbf{I_0}_{N_G}$ describes the current coming into or going out of the circuit at every node of G. This vector has 0 on all entries, except for the nodes directly connected to a voltage source.

$$\mathbf{I_0}_{N_G} = A_G^T \cdot \mathbf{I}_{E_G} \tag{2.26}$$

The complete linear equation that needs to be solved to find the stable state of a system is given in (2.27), and can be solved with the constraint given in (2.28) and by choosing one node as ground node, having a per definition voltage of zero (2.29).

$$\mathbf{I_0}_{N_G} = A_G^T \cdot R_G^{-1} \cdot A_G \cdot \mathbf{V}_{N_G} \tag{2.27}$$

$$\mathbf{I_0}_{N_G,n} = 0 \text{ if } \forall e \in E_G : n \in \texttt{ends}(e) \Rightarrow C(e) \in \mathcal{R} \tag{2.28}$$

$$V(n) = 0 \text{ for ground node } n \tag{2.29}$$

Stabilization

The formulas above describe the stable state of a circuit. This section describes the process of stabilization; that is, how a circuit stabilizes from any state to a state that obeys Ohm's and Kirchhoff's laws.

The voltage at the nodes depends on the time a circuit has been stabilizing and its initial state. In this derivation of the stabilization process, first a discrete time approximation will be considered (2.30). Note that $\Delta\mathbf{V}_{N_G}$ is the change in voltage of the nodes over time Δt, and different from the potential differences over the edges $\Delta\mathbf{V}_{E_G}$.

$$t_{i+1} = t_i + \Delta t \qquad \mathbf{V}_{N_G}(t_{i+1}) = \mathbf{V}_{N_G}(t_i) + \Delta\mathbf{V}_{N_G}(t_i) \tag{2.30}$$

The current, the quantity of electrons, coming in from or going out into an edge, is a node's "observation" of the voltage drop per ohm through that edge. Every edge connected to a node has an influence on that node's next voltage level. This can, for any node n that is not connected to a voltage source and for some function f, be described by (2.31).

$$\Delta V(n, t_i) = \sum_{n \in \texttt{ends}(e)} I(e, t_i) \cdot f(\Delta t) = \sum_{n \in \texttt{ends}(e)} \frac{\Delta V(e, t_i)}{R(e)} \cdot f(\Delta t) \quad (2.31)$$

First, it will be considered how $\Delta V(e)$ would change over time if the nodes connected to e were not connected to any other edge, with $\Delta\Delta V(e, t_i)$ being the change in potential difference over e over time Δt (2.32).

$$\Delta V(e, t_{i+1}) = \Delta V(e, t_i) + \Delta\Delta V(e, t_i) \quad (2.32)$$

$\Delta\Delta V(e, t_i)$ is (directly) proportional to $\Delta V(e, t_i)$ and $I(e, t_i)$, and therefore inversely proportional to $R(e)$. Moreover, for a very small Δt, we may assume that $\Delta V(e, t_{i+1})$ approaches $\Delta V(e, t_i)$, or put differently, $\Delta V(e, t_i)$ locally is proportional to Δt, leading, for some constant K, to (2.33), and after some rearranging to (2.34), and for $\Delta t \to 0$ to (2.35).

$$\Delta\Delta V(e, t_i) = K \cdot \frac{\Delta V(e, t_i)}{R(e)} \cdot \Delta t \quad (2.33)$$

$$\frac{\Delta\Delta V(e, t_i)}{\Delta t} = \frac{K}{R(e)} \cdot \Delta V(e, t_i) \quad (2.34)$$

$$\frac{d}{dt}\Delta V(e, t) = \frac{K}{R(e)} \cdot \Delta V(e, t) \quad (2.35)$$

Symmetry requires that both nodes connected to e contribute equally to the change in potential difference over e, as in (2.36) where $\frac{d}{dt}V(n, e, t)$ denotes edge e's contribution to the voltage change in time of node n at time t. The combined contribution of all edges connected to n is given in (2.37), and can again be nicely described as a matrix multiplication (2.38).

$$\frac{d}{dt}V(n, e, t) = \frac{1}{2} \cdot \frac{d}{dt}\Delta V(e, t) \quad (2.36)$$

$$\frac{d}{dt}V(n,t) \;=\; \sum_{n \in \texttt{ends}(e)} \frac{K}{2 \cdot R(e)} \cdot \Delta V(e,t) \;=\; \sum_{n \in \texttt{ends}(e)} \frac{K}{2} \cdot I(e,t) \quad (2.37)$$

$$\frac{d}{dt}\mathbf{V}_{N_G}(t) = \frac{K}{2} \cdot A_G^T \cdot R_G^{-1} \cdot A_G \cdot \mathbf{V}_{N_G}(t) \quad (2.38)$$

Chapter 3

Reo

Using Reo, [1] coordination of (sequential) components/processes is done by coordinating data that is exchanged between them. Components communicate through ports, which Reo requires to be either source or sink ports. Source ports are ports that may or may not accept (try to read) data depending on the state of the component, but never offer (try to write) data. Sink ports are their duals that may or may not offer data, but never accept data. Reo coordinates the components by either (consumptive) reading or (possibly temporarily) blocking data offered on sink ports, and by either writing or (possibly temporarily) not writing data to accepting source ports. The components can adjust their behavior to the state — read/write (not) pending — of their ports.

Reo is a language for creating and modifying circuits that determine which data is read from and written to the ports of components that they are connected to, as well as for the actual communication between these ports. These circuits consist of nodes and different types of channels. Every channel has two ports, which are normally called channel ends. As such, channels can be seen as components. A channel end is either a source end or a sink end. Every channel end is connected to a node. A circuit constructed or constructible by Reo will be called a Reo circuit or connector.

This chapter leaves out the construction part, and focuses on the behavior of constructed connectors.

Nodes try to read from one of their connected sink ends and write the read data to all of their connected source ends. If they can't do all of this, which is also the case if there are no source ends attached to a node, they will not read and write at all. Channels come in different flavours. The

most important ones, from which most other channels can be constructed in a fairly easy way, are described in table 3.1.

notation	drawn edge	description
`sync`	•——▸•	Tries to read data on its source end and write that same data on its sink end; if this fails it blocks
`lossy`	•- - - -▸•	Accepts data on its source end and tries to write that same data on its sink end; if this fails, data is lost inside the edge
`sync_drain`	•▸——◂•	Tries to read (uncorrelated) data on both of its source ends; if this fails it blocks
`async_drain`	•▸‖◂•	Tries to read (uncorrelated) data on both of its source ends; if this fails it blocks
`sync_spout`	•◂——▸•	Tries to read (uncorrelated) data on both of its source ends; if this fails it blocks
`async_spout`	•◂‖▸•	Tries to read (uncorrelated) data on both of its source ends; if this fails it blocks
`cell`	•—□—▸•	The state consists of either a data item (full cell) or nothing (empty cell); its behavior depends on that
`cell`($\top$)	•—▣—▸•	Blocks on its source end, offers its contained data item on its sink end. If the data is read, the cell becomes empty
`cell`($\bot$)	•—□—▸•	Accepts on its source end, does not offer on its sink end. If data is written, the cell becomes full

Table 3.1: Description of some Reo channels

Whenever the state of one of the stateful channels of a connector changes, the state of the connector changes. Such a connector state change is a transition. Every channel state change is induced by a read/write of data, and a connector transition can be described by all reads/writes in it. Note that in this way a connector is a non-deterministic state machine; it can allow different successor states. Note also that different transitions can lead to the same successor state.

The description above is expressed by seemingly local constraints, in the sense that for every individual channel and node, it would be possible to check based on the properties of their connected ports whether they are valid or not. It is not obvious, though, what 'trying to read/write' should mean, and how the trying of one node or channel affects the others.

Reo generally takes an approach in which trying means nothing more than allowing. No action can happen just as well as the tried action. That this looks like channels or nodes are trying to read or write, pushing or pulling data, comes from the fact that states do not change if no action happens. To 'try' to read or write data, expresses that the possibility of reading or writing remains until the read or write actually happens.

The rest of this chapter is organised as follows. Section 3.1 describes Reo circuits more formally as labeled multi-graphs. Section 3.2 looks at different semantic models for Reo, and their reason for existence.

3.1 Connectors as Multi-Graphs

A Reo circuit C is a labeled multi-graph, where every edge is labeled with a channel type by $\mathtt{chan} : E_G \to \mathcal{C}_\mathrm{r}$, where $\mathcal{C}_\mathrm{r}$ is the set of Reo channels. The channels will in this thesis be restricted to those mentioned in table 3.1.

Like between electric circuits, there exists an equivalence relation between Reo circuits. This equivalence is given in (3.1)

$$\begin{aligned} &C \sim C' \Leftrightarrow N_{G_C} = N_{G'_C} \wedge E_{G_C} = E_{G'_C} \wedge \\ &\quad \forall e \in E_{G_C} : \begin{cases} I_C(e) = I_{C'}(e) & \text{if } \mathtt{chan}(e) \in \{\mathtt{sync}, \mathtt{lossy}, \mathtt{cell}\} \\ \mathtt{ends}_C(e) = \mathtt{ends}_{C'}(e) & \text{otherwise} \end{cases} \end{aligned} \tag{3.1}$$

The state S of such a circuit is defined by the states of its stateful channels, in this case only its cell channels. The state of a cell channel consists of its content. This can be either a data item in some data set D, in which case it's a full cell, or no data item at all, denoted as $\perp$, in which case it's an empty cell. The edge labeling $\mathtt{state}$ in (3.2) describes the circuit state at some point in time completely.

$\texttt{state} : E_{\texttt{cell}} \to D_\perp$

$$E_{\texttt{cell}} = \{e \in E | \, \texttt{chan}(e) = \texttt{cell}\} \qquad D_\perp = D + \{\perp\} \quad (3.2)$$

A transition $T : (D_\perp)^{E_{\texttt{cell}}} \to (D_\perp)^{E_{\texttt{cell}}}$ maps a state to a new state. In Reo, transitions are constrained by data flow; a cell channel transits from full to empty if its data flows out of it, and from empty to full if data flows into it. Before this can be formalized, some work has to be done.

A labeling of nodes $\texttt{inp} : N \to E \cup \{\perp\}$ is possible that indicates which of its entering edges writes to it. Such a labeling is valid if it satisfies the constraints of all nodes and channels. These channel constraints are summed up in table 3.2, where $\texttt{r}_{\texttt{side}}, \texttt{w}_{\texttt{side}} : E_G \to D_\perp$ and $\texttt{r}_{\texttt{tail}}(e), \texttt{r}_{\texttt{head}}(e), \texttt{w}_{\texttt{tail}}(e), \texttt{w}_{\texttt{head}}(e)$ refers to the data item that e reads from or writes to its tail or head node respectively. The node constraints are described by equation 3.3.

$$\forall n \in N_G : \forall e \in E_G : \texttt{tail}(e) = n \Rightarrow \texttt{w}_{\texttt{head}}(\texttt{inp}(n)) = \texttt{r}_{\texttt{tail}}(e) \qquad (3.3)$$

Such an $\texttt{inp}$ labeling defines a transition completely (modulo actual data). Data flows out of a full cell channel e exactly if it's chosen by its head node n to write to it — that is, if $\texttt{inp}(n) = e$; Data flows into an empty cell channel e exactly if its tail node n chooses any edge to write to it — that is $\texttt{inp}(n) \neq \perp$.

channel	constraint
sync	$\texttt{r}_{\texttt{tail}}(e) = \texttt{w}_{\texttt{head}}(e)$
lossy	$\begin{cases} \texttt{r}_{\texttt{tail}}(e) = \texttt{w}_{\texttt{head}}(e) & \text{if possible} \\ \texttt{w}_{\texttt{head}}(e) = \perp & \text{otherwise} \end{cases}$
sync_drain	$\texttt{r}_{\texttt{tail}}(e) = \texttt{r}_{\texttt{head}}(e)$
async_drain	$\texttt{r}_{\texttt{tail}}(e) = \perp \vee \texttt{r}_{\texttt{head}}(e) = \perp$
sync_spout	$\texttt{w}_{\texttt{tail}}(e) = \texttt{w}_{\texttt{head}}(e)$
async_spout	$\texttt{w}_{\texttt{tail}}(e) = \perp \vee \texttt{w}_{\texttt{head}}(e) = \perp$
cell(d)	$\texttt{r}_{\texttt{tail}}(e) = \perp \wedge \texttt{w}_{\texttt{head}}(e) \in \{d, \perp\}$
cell($\perp$)	$\texttt{w}_{\texttt{head}}(e) = \perp$

Table 3.2: Constraints on Reo nodes and channels

3.2 Semantic Models

Semantic models are formal descriptions of the behavior of Reo circuits. Their purpose can be to give a valid implementation for it or an efficient one, or even just to give a better understanding of its behavior. As indicated by the word 'model', a semantic model does not necessarily have to describe Reo circuit behavior perfectly and completely.

Often, semantic models focus on an aspect of Reo's behavior, neglecting some other aspects. They can for example describe whether data flows through a node or not, neglecting what the actual data is. A semantic model can also be useful if it provides an efficient calculation of circuit behavior, even if it describes Reo less accurately than some other known model.

Section 3.2.1 takes a closer look at some typical difficulties that semantic models have to deal with. The sections 3.2.2, 3.2.3 and 3.2.4 briefly describe the most important semantic models, and how they deal with the difficulties from section 3.2.1.

3.2.1 Common Problems

This section discusses some common problems that semantic models tend to run into. The semantic models that are discussed in the following sections are evaluated in large part by their ability to solve these problems.

Context Sensitivity

Some components behave differently in different contexts. The lossy synchronous channel, for example, is not allowed to actually lose data if that data is accepted on its sink end; but whether that is the case depends on factors outside the channel. The ability to adapt behavior to whether data is offered or accepted by the context is context sensitivity.

It appears to be difficult to describe context sensitivity well in semantic models. Those that do not, replace the lossy synchronous channel with a channel that may lose data regardless of its context — this is known as the non-deterministic lossy synchronous channel. On the other hand, those that do can get other problems, as will appear in the next paragraphs.

A way to add context sensitivity to a semantic model, is to somehow require that data must flow whenever it can. This can solve the problem of not having context sensitivity, but is still incompatible with the definition of

Reo. Consider, for example, a circuit consisting of two full cell channels, each connected to an empty cell channel (figure 3.1). The given solution would require immediate, and thus simultaneous, flow from both full cells to the empty cells, whereas in Reo, it can just as well happen that flow from the one full cell precedes flow from the other full cell.

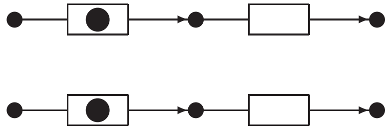

Figure 3.1: A circuit with two unrelated subcircuits

Causality

Causality loops, in which effects are their own cause, form another problem. In Reo, a data item has to come from some data source. This tends to not fit directly in most semantic models. Consider a synchronous channel with its head and tail connected to the same node, and an empty cell channel leaving that same node (figure 3.2). If the circular channel would write data on its sink end, the data would be replicated, and one copy would go into the source end of the same channel. The copy would become its own original on its way to the sink node, and the empty cell would be filled. Though locally this transition looks valid, the fact that a result (the data coming into the circular channel) causes its own cause (the data being offered on the sink end) makes it invalid.

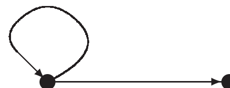

Figure 3.2: A circular circuit

Data Sensitivity

Reo is data sensitive, which means that a connector can behave differently for different data items. For example, a filter channel exists, which behaves as a synchronous channel for a specified subset of data set D, and accepts but destructs the other elements of D. This specification of the subset is done through a pattern to which a data item can conform or not.

Some semantic models focus on whether there is flow or not, and focusing on that neglect actual data. In most cases, it is possible to reconstruct which data item flows through a node or channel by tracing its way back to where it came from. In most cases where this is not possible, it does not matter what the data item is. If it can not be reconstructed which data item flows into a channel, but the channel, for example a filter channel, depends on it, these models have to work around this problem.

A way to work around not having data sensitivity in a semantic model is encoding every data item (and 'no flow') in a sequence of bits. Every data item representation now flows through data buses consisting of synchronised channels and nodes. The disadvantage of this solution is that it makes a circuit more specific to the data sets to which it applies. For example, a pattern "all elements less than 10" can apply to any data set consisting of numbers of some sort, whereas such a general pattern can not be described by the suggested data bus solution.

3.2.2 Connector Coloring

Connector coloring [3] labels all channel ends with a "color". Every valid labeling of a connector determines a transition. A connector labeling is valid if it satisfies all the local constraints that are imposed on it.

In connector coloring, nodes are represented by (combinations of) the three-way merger and replicator channels. With this representation, channel ends are directly connected to at most one other channel end.

The local constraints that rule out invalid colorings consist of the constraints that a channel puts on its ports, and those that are put on unconnected ports and two ports that are connected to each other. Connector coloring comes in two flavors, and the constraints on connected ports are determined by this flavor. The constraints that a channel puts on its ports are channel *and* flavor specific.

The two flavors of connector coloring are discussed separately below.

Two Color Semantics

In two color semantics, every channel end can have one of two colors: flow (drawn as a solid line) or no flow (drawn as a dashed line). The allowed colorings of the most important channels are listed in table 3.3. The constraint on unconnected ports is that they should have no flow. Two connected ports should have the same color.

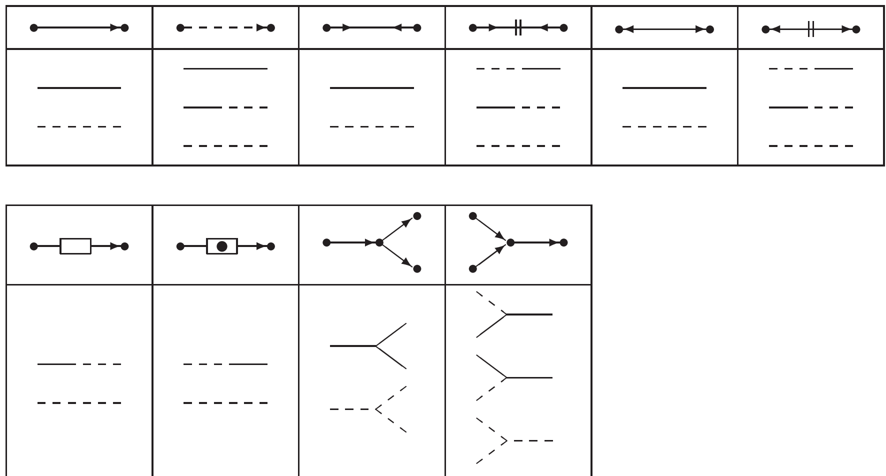

Table 3.3: Colorings of channels with two colors

Two color semantics gives a method for enumerating all transitions of a circuit in a certain state. It does so without context sensitivity, and thus puts too few constraints on transitions. On the other hand, it nicely reflects the fact that data does not have to flow just because it can. The relational nature of the model makes it such that causality loops are not detected, which possibly allows invalid transitions. Finally, two color semantics is data insensitive.

Three Color Semantics

Three color semantics substitutes the color “no flow” from two color semantics with two colors: “needing a reason not to flow,” which is drawn as a dashed line with a closed inward arrow, and “giving a reason not to flow,” which is drawn as a dashed line with a closed outward arrow; both obviously indicate that there is no flow. “Needing a reason” means that the reason needs to be given by the connected port. This implies that a constraint

on two connected ports is that they can not both be “needing a reason”. As a matter of fact, in addition to the constraints of two color semantics, this forms the complete set of constraints on two connected ports. For an unconnected port, no other constraint holds than that there cannot be flow.

Table 3.4 describes the new colorings of the most important channels. Note that, for every coloring of a channel, substituting “giving a reason not to flow” with “needing a reason not to flow” gives a valid coloring too.

Table 3.4: Colorings of channels with three colors

Like two color semantics, three color semantics gives a method to enumerate possible transitions for a circuit state. Its improvement with respect to two color semantics is that it adds context sensitivity. On the other hand, it introduces the Reo incompatibility mentioned in 3.2.1. As far as causality and data sensitivity are concerned, there is no change with respect to two color semantics.

3.2.3 Constraint Automata

A constraint automaton [2] is a description of a Reo circuit as automaton with guarded transitions. Such an automaton is a tuple $(Q, N, \longrightarrow, Q_0)$,

where Q is the set of states, N is a set of names representing the set of all channel ends, $\longrightarrow$ is the transition relation, and Q_0 is the set of initial states. The transition $\longrightarrow$ relation is a subset of $Q \times 2^N \times \mathrm{DC} \times Q$ where DC is the set of data constraints that guard a transition.

A transition can be viewed as a data assignment to a connector's channel ends. 2^N, the powerset of N, is the set of all combinations of ports. Its element in a transition describes which ports participate in it, in the sense that data flows through them. A guard, or data constraint, describes which data assignments to these ports are valid. Data constraints are described by propositional formulae with atoms of the form $d_A = d$ for any d in data set D, where d_A refers to the data in any port $A \in N$. Note that formulae like $d_A = d_B$ can also be constructed if D is finite, namely by $\bigvee_{d \in D} d_A = d \wedge d_B = d$.

Where coloring semantics is most naturally interpreted as a generator of possible Reo circuit transitions, constraint automata can be considered to be acceptors of transitions. Constraint automata are not context sensitive, but correctly always allow the transition with "no flow" (in parts of the circuit). They do not detect causality loops. Data sensitivity is handled correctly.

3.2.4 Constraint Satisfaction

Constraint satisfaction as presented in [4] is more like a semantic meta-model of Reo. It describes valid transitions by constraints, using different sets of constraints for different aspects of the system. Whether there is flow in a port or not is determined by a dedicated set of constraints, and what data is allowed in the ports where there is data is determined by another. Data sensitivity is easily added with yet another set of constraints.

On the one hand, none of the problems presented in 3.2.1 seem to be inherent to constraint satisfaction. On the other hand, constraint satisfaction does not designate the actual sets of constraints used to describe (their corresponding aspects of) valid transitions, and as such is not an actual semantic model. Given a transition candidate, it *can* (easily) be checked whether it is a valid transition or not by checking the constraints. There is no straightforward way, though, to find one or all transitions that are valid for a Reo circuit in a given state.

Chapter 4

Physical Reo

The very name 'Reo' is a transcription of the Greek word for "flow". In Reo circuits, data items flow. In electric circuits, electrons flow. In Reo, it seems like data wants to flow from where it is present to where it is accepted, as is actually the case with electrons in electric circuits. This is a very strong analogy, which is made even more appealing by the fact that although the laws of electric circuits are local, electric circuits are context sensitive; that is, the behavior of a point in an electric circuit can depend on the state of another point in a different location.

There are, however, significant differences. In the context of Reo, "flow" refers to the flow of one data item, rather than to a current of many particles, as in electric circuits. This flow of data can only happen in one direction. And although Reo connectors themselves are often agnostic of actual data, different flowing data items are not interchangeable, as electrons are. Moreover, there is the subtle difference that in traditional interpretations of Reo, data items can be replicated or annihilated (lost) in a circuit. Electrons can enter or leave an electric circuit, but replication and annihilation in the circuit are prohibited by Kirchhoff's Current Law.

Physical Reo is a semantic model for Reo that tries to come as close as possible to a physical model, and specifically Ohm's and Kirchhoff's laws. Physical models, in this respect, are models that are local and continuous. Physical Reo focuses on describing context sensitivity and causality loops, but possibly neglects data sensitivity.

This chapter presents two approaches to Physical Reo, neither of which are satisfactory as a definition of it. One of the presented approaches is correct on context sensitivity and causality loops, but implicitly assigns a

priority to every data item, which sometimes makes it block even if a transition is possible. The other approach correctly gives the choices back to the nodes, but is nondeterministic and leaves its choice context insensitive. This is the reason that it cannot describe the deterministic lossy synchronous channel, and for some circuits never finds a transition.

After presenting the two approaches to Physical Reo, an application of the voltage based approach is given. It leads to a very efficient algorithm for computing a Reo transition, with a complexity linear (given a voltage initialisation) to the number of edges in the circuit. To overcome the possibility that no transition is found for a Reo circuit for which one *does* exist, a hybrid setup with a fall-back method is suggested.

Section 4.1 will look at a simpler subset of Physical Reo circuits, and explain how they provide a semantic model for a subset of Reo circuits. Section 4.2 will extended this subset to describe all data insensitive Reo circuits. Section 4.3 describes an efficient implementation of the voltage based approach. Section 4.4 evaluates the given approaches to Physical Reo, and compares them to existing semantic models.

4.1 Simple Connectors

This section introduces Physical Reo. Physical Reo circuits are more like electric circuits than Reo circuits are, and are more like Reo circuits than electric circuits are. Two approaches to how Physical Reo can find transitions for Reo circuits are given. This section is limited to a small subset of Reo circuits, though many of its statements will apply to any circuit.

A Physical Reo circuit C (4.1) is defined as a directed multi-graph $G = (N_G, E_G, I_G)$ with node labeling $\texttt{nodetype} : N_G \to \{\texttt{reo}, \texttt{oer}\}$ and the (partial) labelings $R : N_G \cup E_G \to \mathcal{R}$, $\texttt{const} : N_G \cup E_G \to \{\texttt{const}, \texttt{var}\}$. In the rest of this section, nodes of type `oer` will be left out of consideration; they will be discussed in section 4.2.

$$C = (G, \texttt{nodetype}, R, \texttt{const}) \tag{4.1}$$

A Physical Reo circuit state S_C (4.2) is defined by the (partial) labeling $V : N_G \cup E_G \to \mathcal{V}$. This labeling can be interpreted as a voltage labeling. The labeling R can be interpreted as a resistance labeling, and `const` as whether a voltage is constant because it is a voltage source. The labeling $I : N_G \cup E_G \to \mathcal{I}$, which is fully determined by V and R, can be interpreted as

(electric) current. Like electric circuits, Physical Reo circuits are differential systems in which voltage, current, and resistance determine the state change.

$$S = (C, V) \tag{4.2}$$

Every Reo circuit C consisting of synchronous channels, cell channels, synchronous spouts and asynchronous drains has an underlying Physical Reo circuit, $\mathtt{und}(C)$ (4.4), that describes the behavior of C. The underlying circuit is obtained by substituting every Reo channel with a certain set of nodes and edges. Which channel is substituted by which set of nodes and edges is indicated by table 4.1. Note that the nodes of a Reo circuit are shared by its underlying circuit.

To formalize the substitution of Reo channels, a decomposition of Reo circuits is needed. Every circuit can be decomposed into atomic subcircuits consisting of one channel and its source and sink nodes (4.3); their union reconstructs the original circuit. How the underlying circuit of these atomic subcircuits is created is given by the equations in table 4.1. The underlying Physical Reo circuit of the original Reo circuit is obtained by taking the union of these underlying circuits (4.4).

$$\mathtt{dec}(C) = \{(\mathtt{ind}_{G_C}(e), \mathtt{chan}_C) | e \in E_{G(C)}\} \tag{4.3}$$

$$\mathtt{und}(C) = \bigcup_{C' \in \mathtt{dec}(C)} \mathtt{und}(C') \tag{4.4}$$

A Reo transition can be derived from a stable state of its underlying circuit. For this derivation, a correspondence $\mathtt{corr} : E_C \to E_{\mathtt{und}(C)}$ is set up between Reo channel ends, and ports in the underlying circuit. These can be found in the last column of table 4.1.

Whether a state is stable is determined by the differential equations that relate voltage, current, resistance and the change in time of these. For such a system of equations, there are different options, with different ways to translate them back to Reo transitions. The following sections will present two such options, one focusing on analogy with voltage, the other on analogy with current in electric circuits.

4.1.1 Voltage Based Approach

The voltage based approach to Physical Reo considers current to be nothing more than the medium through which potential difference is communicated between nodes. This leaves Kirchhoff's Voltage Law as well as Ohm's law untouched.

$$\mathbf{\Delta V} = A \cdot \mathbf{V} \tag{4.5}$$
$$\mathbf{I} = R^{-1} \cdot \mathbf{\Delta V} \tag{4.6}$$

Kirchhoff's Current law, on the other hand, needs revision. A node's voltage will change as if it were connected to only two of its connected edges. These two are selected based on their current; the ones with the biggest entering current contribute to the node's voltage change. This entering current can be the current through an entering edge, or the inverse of the current through a leaving edge. If there are different edges with equal incoming currents, either one can play the role of contributor to the node's voltage change. For a node connected to only one edge, of course, only that one influences its voltage.

A node's selection $\texttt{sel} \subseteq E_G$ of edges can be described as an element of $\binom{E_G}{2}^{N_G}$, the set of all functions that map a node to a set of two edges. A formalization of the constraints as given above on such a selection, are expressed in (4.8).

$$\texttt{sel} \in \binom{E_G}{2}^{N_G} \tag{4.7}$$

$$\forall n \in N_G, e \in \texttt{sel}(n), e' \in E_G \setminus \texttt{sel}(n) : \\ n \in \texttt{nodes}(e') \Rightarrow (n \in \texttt{nodes}(e) \wedge I(e) \geq I(e')) \tag{4.8}$$

$$I_n(e) = \begin{cases} I(e) & \text{if } e \text{ enters } n \\ -I(e) & \text{if } e \text{ leaves } n \end{cases} \tag{4.9}$$

The described substitute for Kirchhoff's Current Law can be expressed as a matrix multiplication, as described in (4.10), where matrix B can be

interpreted as the matrix that expresses a selection `sel` (4.12 - 4.14). Note that, though B is not fully determined by the circuit state, $B^T \cdot \mathbf{I}$ is.

$$\mathbf{I}_{\mathbf{0}N_G} = B^T \cdot \mathbf{I} \tag{4.10}$$

$$\frac{d\mathbf{V}}{dt} = K \cdot 1_{\texttt{var}} \cdot B^T \cdot \mathbf{I} \tag{4.11}$$

$$B = B_{\text{tail}} - B_{\text{head}} \tag{4.12}$$

$$B_{\text{tail},e,n} = \begin{cases} A_{\text{tail},e,n} & \text{if } e \in \texttt{sel}(n) \\ 0 & \text{otherwise} \end{cases} \tag{4.13}$$

$$B_{\text{head},e,n} = \begin{cases} A_{\text{head},e,n} & \text{if } e \in \texttt{sel}(n) \\ 0 & \text{otherwise} \end{cases} \tag{4.14}$$

Such a circuit stabilizes to a state where $\mathbf{I}_{\mathbf{0}N_G}$ is the current entering the circuit through the nodes, which is zero for `var` nodes, resulting in $\frac{d\mathbf{V}}{dt} = \mathbf{0}$. In this stable state, the second biggest current entering a node is the inverse of the biggest current entering it. As a consequence, there is at most one positive entering current, and none only if there are two or more connected edges without current. Before the stabilization process can start, nodes will need to be initialized with voltages.

For the stable state to be translated to a valid Reo transition, there are some requirements on the (initial) voltages of the `const` nodes. Every Physical Reo voltage source representing a data source in Reo should have a higher voltage than any voltage source representing a data sink. Moreover, no two `const` nodes can have the same voltage.

A Reo transition can be derived from the stable state of its underlying circuit in the following way; a data item flows from a data source to a data sink if and only if there is a (directed) path in its underlying circuit from the data source to the data sink with monotonously decreasing voltages.

4.1.2 Current Based Approach

The primary assumption of current based Physical Reo is that current is replicated in nodes. For this, it is most convenient to have the label I apply to nodes, and, as a consequence, V to edges and R to nodes.

Every node nondeterministically selects one of its entering edges that has a higher voltage than any of its leaving edges. The potential difference over

it is then defined as the difference between the voltage of its chosen edge and the maximum voltage among its leaving edges. Through sink or source nodes, of course, there can be no current. The valid selections of entering and leaving edges are formalized in (4.15) to (4.18).

$$\texttt{sel}_{\text{tail}} \in E_G^{N_G} \qquad \texttt{sel}_{\text{head}} \in (E_G \cup \{\perp\})^{N_G} \tag{4.15}$$

$$\begin{aligned}&\forall n, n' \in N_G, e \in E_G : I_G(e) = (n, n') \Rightarrow \\ &\qquad \exists n'' \in N_G : I_G(\texttt{sel}_{\text{tail}}(n)) = (n, n'') \wedge V(\texttt{sel}_{\text{tail}}(n)) \geq V(e)\end{aligned} \tag{4.16}$$

$$\begin{aligned}&\forall n \in N_G : \texttt{opt}_{\text{head}}(n) = \\ &\qquad \{e \in E_G | \exists n' \in N_G : V(e) > V(\texttt{sel}_{\text{tail}}(n)) \wedge I_G(e) = (n', n)\}\end{aligned} \tag{4.17}$$

$$\forall n \in N_G : \begin{cases} \texttt{sel}_{\text{head}}(n) \in \texttt{opt}_{\text{head}}(n) & \text{if } \texttt{opt}_{\text{head}}(n) \neq \emptyset \\ \texttt{sel}_{\text{head}}(n) = \perp & \text{otherwise} \end{cases} \tag{4.18}$$

Every valid selection can be expressed as a matrix multiplication, as in (4.19) to (4.22). Note that not only matrix C is not fully determined by the circuit state, but also the vector $\mathbf{\Delta V}$ (4.19). The current through a node is described by Ohm's law (4.23).

$$\mathbf{\Delta V} = C^T \cdot \mathbf{V} \tag{4.19}$$

$$C = C_{\text{tail}} - C_{\text{head}} \tag{4.20}$$

$$C_{\text{tail},n,e} = \begin{cases} A_{\text{tail},n,e} & \text{if } e = \texttt{sel}_{\text{tail}}(n) \\ 0 & \text{otherwise} \end{cases} \tag{4.21}$$

$$C_{\text{head},n,e} = \begin{cases} A_{\text{head},n,e} & \text{if } e = \texttt{sel}_{\text{head}}(n) \\ 0 & \text{otherwise} \end{cases} \tag{4.22}$$

$$\mathbf{I} = R^{-1} \cdot \mathbf{\Delta V} \tag{4.23}$$

Current goes out of an edge only if the edge is chosen by its sink. On its source end, on the other hand, the current through its connected node

always enters. The matrix expressing this can be described as a combination of A and C (4.24).

$$\frac{d\mathbf{V}}{dt} = K \cdot (A_{\text{tail}} - C_{\text{head}}) \cdot \mathbf{I} \tag{4.24}$$

A node is fixed with its choice until the current through it approaches zero. Then it will again nondeterministically choose another one of its entering edges that has a higher voltage than any of its leaving edges, if possible. If this is not possible, no edge will be chosen at all. Note that the previous choice of a node, the edge from which current has most recently entered, should formally be considered part of its state, and as such part of the circuit state.

The nondeterminism of nodes can cause a circuit not to stabilize or to loop through states indefinitely before finally stabilizing. But even if the voltages at the edges stabilize, it can be that the node's choice is not stable. These instable cases are not translated back to a Reo transition. If a circuit *does* stabilize, the stable state is one in which the current going into an edge equals the current going out of it.

Of course, edge voltages have to be initialized. There is no constraint on a voltage initialization to be valid, but to have data sources 'try' to write their data, their corresponding voltage sources should have higher voltages than those of the voltage sources corresponding to data sinks.

The Reo transition induced by this stable state is the one with flow exactly in those channel ends of which the corresponding port has nonzero current.

4.2 Complex Connectors

The previous section introduced Physical Reo, but left `oer` nodes out of consideration. As a result, not all Reo channels could be described. This section describes the behavior of reverse nodes, and how they are used to describe more Reo channels.

Reverse nodes are the reverse of normal Reo nodes in the sense that the same way that Reo nodes either replicate the data from exactly one connected sink end to all connected source ends or block, reverse nodes either read data on all connected sink ends and write one of them to exactly one source end or block.

With reverse nodes, Physical Reo is able to also describe synchronous drains, asynchronous spouts and lossy synchronous channels. Table 4.2 shows the underlying circuits and correspondence relations for each of these. Reverse nodes are indicated by unfilled circles, normal nodes by filled ones. How reverse nodes affect the stabilization process of the voltage based approach and the current based approach will be explained in the following subsections.

4.2.1 Voltage Based Approach

As with normal nodes, Ohms law and Kirchhoff's Voltage Law are unchanged for the voltage based approach, but Kirchhoff's Current Law is adapted. A reverse node selects its two connected edges with the *lowest* entering current to influence its voltage. For normal Reo nodes, equation 4.8 still holds; for reverse nodes, however, the "greater than" signs have to be reversed. This is written out in equation 4.25. The rest of the derivation presented in this section is identical to that in section 4.1.1.

With the constraints on the voltage initialization of a circuit as given in section 4.1.1, the voltage based approach to Physical Reo describes the lossy synchronous channel as a nondeterministic one. Adding the constraint that all internal voltage sources of lossy synchronous channels must have higher voltages than other voltage sources that represent data sinks is sufficient to make lossy synchronous channels behave deterministically.

$$
\begin{aligned}
&\forall n \in N_G, e \in \texttt{sel}(n), e' \in E_G \setminus \texttt{sel}(n) : n \in \texttt{nodes}(e') \Rightarrow \\
&\qquad\qquad \texttt{nodetype}(n) = \texttt{reo} \Rightarrow (n \in \texttt{nodes}(e) \wedge I(e) \geq I(e')) \\
&\qquad\qquad\quad \texttt{nodetype}(n) = \texttt{oer} \Rightarrow (n \in \texttt{nodes}(e) \wedge I(e) \leq I(e'))
\end{aligned}
\tag{4.25}
$$

With the introduction of reverse nodes, the derivation of a Reo transition from a stable state of its underlying circuit gets more complex. It is no longer a sufficient condition for a data item to flow that there is a path with monotonously decreasing voltages from its data source to a data sink. It is now also necessary that for every edge entering a reverse node on that path, there is a path with (recursively) the same property entering the specified reverse node through the specified entering edge, as well as it is necessary that for every edge leaving a Reo node on that path, there is a directed path

with (recursively) the same property leaving the specified Reo node through the specified leaving edge.

4.2.2 Current Based Approach

As with the voltage based approach, in the current based approach reverse nodes are influenced by low instead of high voltages. This is expressed in the selection of its entering edge with the lowest voltage, and a leaving edge with a lower voltage than that if possible. Equations 4.28 to 4.30 are the equations that formalize `sel` for `reo` nodes, and substitute 4.16 to 4.18. Equations 4.31 to 4.33 extend `sel` for `oer` nodes.

$$\texttt{sel}_{\text{tail}} \in (E_G \cup \{\bot\})^{N_G} \qquad \texttt{sel}_{\text{head}} \in (E_G \cup \{\bot\})^{N_G} \tag{4.26}$$

$$\begin{aligned} &N_{G,\texttt{reo}} = \{n \in N_G | \, \texttt{nodetype}(n) = \texttt{reo}\} \\ &\qquad N_{G,\texttt{oer}} = \{n \in N_G | \, \texttt{nodetype}(n) = \texttt{oer}\} \end{aligned} \tag{4.27}$$

$$\begin{aligned} &\forall n \in N_{G,\texttt{reo}}, n' \in N_G, e \in E_G : I_G(e) = (n, n') \Rightarrow \\ &\qquad \exists n'' \in N_G : I_G(\texttt{sel}_{\text{tail}}(n)) = (n, n'') \wedge V(\texttt{sel}_{\text{tail}}(n)) \geq V(e) \end{aligned} \tag{4.28}$$

$$\begin{aligned} &\forall n \in N_{G,\texttt{reo}} : \texttt{opt}_{\text{head}}(n) = \\ &\qquad \{e \in E_G | \exists n' \in N_G : V(e) > V(\texttt{sel}_{\text{tail}}(n)) \wedge I_G(e) = (n', n)\} \end{aligned} \tag{4.29}$$

$$\forall n \in N_{G,\texttt{reo}} : \begin{cases} \texttt{sel}_{\text{head}}(n) \in \texttt{opt}_{\text{head}}(n) & \text{if } \texttt{opt}_{\text{head}}(n) \neq \emptyset \\ \texttt{sel}_{\text{head}}(n) = \bot & \text{otherwise} \end{cases} \tag{4.30}$$

$$\begin{aligned} &\forall n \in N_{G,\texttt{oer}}, n' \in N_G, e \in E_G : \texttt{nodetype}(n) = \texttt{oer} \wedge I_G(e) = (n', n) \Rightarrow \\ &\qquad \exists n'' \in N_G : I_G(\texttt{sel}_{\text{head}}(n)) = (n'', n) \wedge V(\texttt{sel}_{\text{head}}(n)) \leq V(e) \end{aligned} \tag{4.31}$$

$$\begin{aligned} &\forall n \in N_{G,\texttt{oer}} : \texttt{opt}_{\text{tail}}(n) = \\ &\qquad \{e \in E_G | \exists n' \in N_G : V(\texttt{sel}_{\text{head}}(n)) > V(e) \wedge I_G(e) = (n, n')\} \end{aligned} \tag{4.32}$$

$$\forall n \in N_{G,\texttt{oer}} : \begin{cases} \texttt{sel}_{\text{tail}}(n) \in \texttt{opt}_{\text{tail}}(n) & \text{if } \texttt{opt}_{\text{tail}}(n) \neq \emptyset \\ \texttt{sel}_{\text{tail}}(n) = \bot & \text{otherwise} \end{cases} \tag{4.33}$$

With this definition of $\texttt{sel}_{\text{tail}}$ and $\texttt{sel}_{\text{head}}$, no further changes have to be made to the derivation of matrix C. The equation describing the voltage change of the edges, on the other hand (4.24), needs to be replaced by (4.34).

$$\frac{d\mathbf{V}}{dt} = K \cdot (1_{\texttt{reo}} \cdot (A_{\text{tail}} - C_{\text{head}}) + 1_{\texttt{oer}} \cdot (C_{\text{tail}} - A_{\text{head}})) \cdot \mathbf{I} \tag{4.34}$$

4.3 A Practical Application

This section presents a linear time algorithm for finding a Reo state transition using the voltage based approach and given a voltage initialization. First, an outline of the algorithm is presented. Then the actual algorithm is discussed, illustrated by code snippets written in Python. Finally, complexity and the use of the algorithm is discussed.

The algorithm abstracts away from actual voltages, and considers the *ordering* of adjacent nodes based on their voltages in the stable state of the circuit. It is implemented as a depth-first walk through a Physical Reo circuit in the direction of lower or equal voltages, starting in the highest voltage node. This strategy guarantees that all unvisited nodes have voltages that are lower than or equal to the voltages of visited nodes. During the walk, the status of a node is conditionally copied to the subsequent node.

The possible node statuses are presented in listing 4.1. The node status `offer` indicates that there is a (directed) path from a data source to the node it applies to, with only non-increasing voltages along its edges; `block` and `noffer` both indicate that there is an edge with non-decreasing voltage along it in every path from a data source to the node; `accept` implies the status `offer`, and moreover indicates that there is a path from it to a data sink with only decreasing voltages. The status `revoke` is used for cleaning up, and will be discussed later.

```
offer  = Signal()
block  = Signal()
noffer = Signal()
accept = Signal()
revoke = Signal()
```

Listing 4.1: Possible node statuses

A Physical Reo circuit is described as a list of nodes and a list of edges, where the edges contain references to their tail and head nodes. As an example, the exclusive router circuit is given in listings 4.2 and 4.3. The nodes have booleans `is_reverse` which makes a node normal or reverse, and `is_data_sink` and `is_data_source`. A node is a data sink if and only if `is_data_sink` is true. The meaning of `is_data_source` is a bit more subtle; if a node is a data sink, and `is_data_source` is true, then the node is an internal data sink of a lossy synchronous channel, and thus has a higher voltage than other data sinks. If a node is not a data sink, it is a data source if and only if `is_data_source` is true.

```
nodes = [
    Node(is_data_source=True),
    Node(),

    # top lossy
    Node(is_reverse=True),
    Node(is_data_source=True, is_data_sink=True),
    Node(),

    # synchronous drain
    Node(is_reverse=True),
    Node(is_data_sink=True),

    # bottom lossy
    Node(is_reverse=True),
    Node(is_data_source=True, is_data_sink=True),
    Node(),

    Node(is_data_sink=True),

    Node(),

    Node(is_data_sink=True),
]
```

Listing 4.2: Exclusive router nodes

```
edges = [
    Edge(nodes[0], nodes[1]),

    # top lossy
    Edge(nodes[1], nodes[2]),
    Edge(nodes[2], nodes[3]),
    Edge(nodes[2], nodes[4]),

    # synchronous drain
    Edge(nodes[1], nodes[5]),
    Edge(nodes[5], nodes[6]),

    # bottom lossy
    Edge(nodes[1], nodes[7]),
```

```
    Edge(nodes[7],  nodes[8]),
    Edge(nodes[7],  nodes[9]),

    Edge(nodes[4],  nodes[10]),
    Edge(nodes[4],  nodes[11]),
    Edge(nodes[9],  nodes[11]),
    Edge(nodes[9],  nodes[12]),
    Edge(nodes[11], nodes[5]),
]
```

Listing 4.3: Exclusive router edges

Except for these properties, which fully determine the Physical Reo circuit state, nodes are initialized with a list of all entering edges and a list of all leaving edges. Moreover, lists of data sources, data losers (data sinks internal to lossy synchronous channels), and other data sinks are constructed once. These initializations are shown in listing 4.4. Like the other mentioned properties, these are unchanged during the algorithm.

```
data_sources   = []
data_losers    = []
data_sinks     = []

for node in nodes:
    node.leaving_edges  = []
    node.entering_edges = []
    if node.is_data_sink and not node.is_data_source:
        data_sinks.append(node)
    elif node.is_data_sink and node.is_data_source:
        data_losers.append(node)
    elif not node.is_data_sink and node.is_data_source:
        data_sources.append(node)

for edge in edges:
    edge.tail.leaving_edges.append(edge)
    edge.head.entering_edges.append(edge)
```

Listing 4.4: Circuit initialization

Before starting the walk through the circuit, the node properties that change during the algorithm have to be initialized, as shown in listing 4.5. Their meaning will become clear when their use is explained.

```
for node in nodes:
    node.status = None
    node.choice = None
    node.counter = 0

for edge in edges:
    edge.tail.counter += 1
    if edge.head.is_reverse:
        edge.head.counter += 1
```

Listing 4.5: Algorithm initialization

The walk starts at any of the data sources. Obviously, data is offered there, so the status `offer` will be propagated through the circuit from there on. If the propagation from this data source is completed, `offer` will be propagated from the other data sources one by one (listing 4.6).

```
def propagate(signal, edge_or_node):
    edge, node = (
            (None, edge_or_node) if isinstance(edge_or_node, Node)
            else (edge_or_node, edge_or_node.head)
            if signal == offer or signal == noffer
            else (edge_or_node, edge_or_node.tail)
            )
    ...

for node in data_sources:
    propagate(offer, node)
```

Listing 4.6: Propagation starting in data sources

First, propagation of `offer`, `block` and `noffer` through normal Reo nodes is explained (listing 4.7). The only difference between `block` and `noffer` is that `noffer` is propagated forward — in the direction of the edge — and `block` is propagated backward.

If a Reo node is visited for the first time, its status is set to the signaled status. It means that (nonnegative) current is entering the node from the edge through which it is visited, and implies that (nonnegative) current will flow out through all other edges. Current from an edge head to an edge tail (or no current at all) always implies that no data flows through that edge, so `block` will be signaled from the visited node to all edges entering it. Current from an edge tail to an edge head means that the status of the edge tail can be propagated to the edge head; that is: if `offer` was signaled to a node, `offer` is signaled through its leaving edges, but if either `block` or `noffer` was signaled, `noffer` is signaled through its leaving edges.

```
def propagate(signal, edge_or_node):
    ...
    if not node.is_reverse:
        if signal in (offer, noffer, block) and node.status == None:
            node.status = signal
            node.choice = edge

            for next in node.entering_edges:
                if next != edge:
                    propagate(block, next)
```

```
        for next in node.leaving_edges:
            if next != edge:
                next_signal = offer if signal == offer else noffer
                propagate(next_signal, next)

        if signal == offer and node.is_data_sink:
            node.status = accept
            propagate(accept, edge)
```

Listing 4.7: Propagation of offer, block and noffer through Reo nodes

If data is offered to a data sink, the data is accepted, and the `accept` status comes into play. Acceptance is propagated backwards through nodes with status `offer` (listing 4.8). Initially, a Reo node's counter is set to the number of leaving edges. Every edge through which `accept` is signaled subtracts one from its tail's counter. If the counter reaches zero, which indicates that the offered data was accepted on all leaving edges, the offered data is also accepted on the current node. Next, its status is adapted accordingly, and propagated back to its choice — the edge through which the offer entered.

```
def propagate(signal, edge_or_node):
    ...
    if not node.is_reverse:
        ...
        if signal == accept:
            node.counter -= 1
            if node.counter == 0:
                node.status = accept

                if node.is_data_sink or not node.is_data_source:
                    propagate(accept, node.choice)
```

Listing 4.8: Propagation of accept through Reo nodes

Where normal Reo nodes have at most one edge through which current enters, reverse nodes have a most one node through which current leaves. This means that a reverse node has to be visited from all but one edges before it can propagate a status through the one edge through which it has not been visited yet. The complete code for propagation of `offer`, `block` and `noffer` through reverse nodes is given by listing 4.9.

As soon as a `noffer` is signaled to a reverse node, it is clear that the node will never be able to offer data anymore, so its status is set. Note, however, that the propagation of `block`s and `noffer`s is delayed until only one edge is left to propagate through, so that the signal does not prematurely block the propagation of data coming from lower voltage nodes.

Just as normal Reo nodes use their counter to count down the unvisited leaving edges, reverse nodes use their counter to count down all unvisited edges. A status that is signaled to a reverse node stops propagating there, unless only one edge is left through which it can be propagated further. In that case the node's `choice` is set to this last edge. The signal to propagate is `offer` if all entering edges have signaled `offer` or otherwise `block` or `noffer`.

```
def propagate(signal, edge_or_node):
    ...
    if edge:
        if node == edge.head: edge.visited_tail = True
        if node == edge.tail: edge.visited_head = True
    ...
    if node.is_reverse:
        if signal in (offer, noffer, block):
            if signal == noffer:
                node.status = noffer

            node.counter -= 1
            if node.counter == 1:
                node.counter = 0

                node.choice = reduce(
                        lambda c, e: c or (e if not e.visited_head else None),
                        node.entering_edges, None) or reduce(
                        lambda c, e: c or (e if not e.visited_tail else None),
                        node.leaving_edges, None)

                next_signal = (block if node.choice.head == node
                        else noffer if node.status == noffer
                        else offer)
                node.status = offer if next_signal == offer else block

                propagate(next_signal, node.choice)

                if node.status == offer and node.is_data_sink:
                    node.status = accept
                    for edge in node.entering_edges:
                        propagate(accept, edge)
```

Listing 4.9: Propagation of offer, block and noffer through reverse nodes

If all entering edges of a reverse data sink have signaled `offer`, its status is set to `accept` and propagated back through all entering edges. No book-keeping has to be done by reverse nodes receiving the `accept` signal, and the signal is immediately propagated back to all of its entering edges, which all have nodes with status `offer` on their other side (listing 4.10).

```
def propagate(signal, edge_or_node):
    ...
    if node.is_reverse:
        ...
        if signal == accept:
```

```
        node.status = accept

        for next in node.entering_edges:
            propagate(accept, next)
```

Listing 4.10: Propagation of accept through reverse nodes

After the propagation of `offer` starting from the data sources has finished, the data sinks — being voltage sources too — start propagating `block` signals if their statuses are not set yet (listing 4.11). Because data losers are required to have higher voltages than other data sinks, they start propagating first.

```
for node in data_losers:
    propagate(block, node)

for node in data_sinks:
    propagate(block, node)
```

Listing 4.11: Propagation starting in data sinks

After this walk through the circuit, if acceptance is propagated back to a data source, data is in principle able to flow. This is in principle because it is possible that another offered data item has been accepted by a data sink, and that this acceptance has propagated back to this data source, but not to all offered data items. In that case, the "unjustifiedly" accepted data item has to be revoked. This is done by propagation of the `revoke` status from every data source that might be a cause of such an unjustified acceptance (listing 4.12). Revocation sets the status of all nodes that depend on the acceptance of the current node to `revoke` (listing 4.13).

```
for node in data_sources:
    if node.status == offer:
        propagate(revoke, node)
```

Listing 4.12: Revocation starting in data sources

```
def propagate(signal, edge_or_node):
    ...
    if node.is_reverse:
        ...
        if signal == revoke and node.status in (offer, accept):
            node.status = revoke

            for next in node.entering_edges:
                propagate(revoke, next.tail)
            if node.choice:
                propagate(revoke, node.choice.head)
        ...
```

```
    if not node.is_reverse:
        ...
        if signal == revoke and node.status in (offer, accept):
            node.status = revoke

            for next in node.leaving_edges:
                propagate(revoke, next.head)
            if node.choice:
                propagate(revoke, node.choice.tail)
```

Listing 4.13: Propagation of revoke through nodes

After revocation is propagated where needed, the nodes with status `accept` — and only those — are the ones where data flows in the next Reo transition.

As mentioned at the outset of this section, the complexity of the presented algorithm is (at worst) linear with the number of edges in the circuit. That is, given a voltage initialization, computing the Reo transition according to the voltage based approach. It can be, though, that one voltage initialization does not lead to a Reo transition, whereas another one does. This means that all voltage initializations — in this context meaning all combinations of all permutations of data sources, data losers and non-losing data sinks — might have to be tried before a transition is found, which gives a complexity of $\mathcal{O}(\#\text{data sources}! \cdot \#\text{data losers}! \cdot (\#\text{data sinks} - \#\text{data losers})!)$.

As mentioned in the previous sections, it is also possible that the voltage based approach does not find a transition for any voltage initialization. In that case, any other method can be used as a fall-back. As a matter of fact, it is more efficient not to compute the transitions for *all* voltage initializations before using the fall-back. As an illustration, a rough (worst case) estimation of the complexity of three color semantics is $\mathcal{O}(4^{(\#\text{edges})})$, whereas the complexity of the linear time algorithm executed for all voltage initializations is roughly approximated by $\mathcal{O}(\#\text{edges}^{(\#\text{edges})})$.

The explanation for this bad performance of the voltage based approach applied to all voltage initializations is that the differences between resulting walks through the circuit on average differ very little per voltage initialization. Finding the optimal set of permutations to which to apply the voltage based approach before using the fall-back method requires additional research, but suggestions could be 1) a singleton set, 2) a set of two permutations, the second one consisting of the reverse order of both data sources, losers and sinks with respect to the first one, and 3) a set of eight permutations, all having either the same or the reverse order of data sources, losers and sinks with respect to each other.

4.4 Evaluation

Physical Reo intuitively is the analog counterpart of discrete Reo. In Reo, write attempts are propagated through a circuit, and acceptance or blocking is propagated back. In Physical Reo, a measure of how much a data item wants to flow continuously propagates back and forth through the system. The forward propagation of this measure is the analog counterpart of write attempts; the backward propagation that of acceptance or blocking.

One of the uses of semantic models for Reo is to make it easier to reason about circuits. As far as simplifying reasoning about circuits is concerned, Physical Reo makes a few contributions.

Firstly, the substitution of channels by (reverse) nodes and simple directed edges makes it possible to describe circuits as directed graphs, and represent them as incidence matrices. Not only circuits, but also transitions — determined by which connected sink end is selected by every Reo node — can be represented as a matrix. This substitution of channels is not limited to Physical Reo only; it might find applications in other contexts too.

Secondly, Physical Reo makes the context of Reo channels explicit. In the voltage based approach, context information is passed between nodes in the form of current, and current only. It is used to determine the correct behavior of all channels, including the lossy synchronous channel. In the current based approach, *voltages* are the context information. No method has been presented to use this information to describe the lossy synchronous channel correctly, though.

Physical Reo's most notable practical contribution to Reo is the algorithm presented in section 4.3. It is a method for finding a valid Reo transition with a reasonable chance, and it does so in linear time. Inherent to using the voltage based approach, the found transition is valid, including causality loop behavior and context sensitivity.

Concerning practical applications, the concept of Physical Reo has two interesting characteristics. Firstly, it is inherently distributed. This is a consequence of the requirement that it is local and continuous, as in physical systems. Every node and every edge can base its state change only on the current state of itself and its direct neighbors (the local part). Moreover, for small changes in time, these state changes are small, and the order in which nodes and edges change their states is irrelevant (the continuous part).

Secondly, Physical Reo is (piecewise) continuous and piecewise differentiable. The process of stabilization can be seen as the optimization of a

circuit's stability. This stability, by some measure, is a function of the states of the nodes and edges of a circuit. For approaches where nodes and edges have no (discrete) memory, this stability function is continuous and piecewise differentiable. If nodes or edges do have discrete memory, this function becomes piecewise continuous. Such properties make Physical Reo more suitable for analytical methods to find transitions. It has to be said, though, that the discontinuities (of the derivative) correspond to the choices that nodes can make.

In the presented voltage based approach, both nodes and edges have no discrete internal state. It is therefore continuous and piecewise differentiable. In the presented current based approach, nodes do have an internal discrete state, which is in fact the reason that some circuits do not stabilize.

Comparison With Other Semantic Models

Coloring semantics, whether using two or three colors, is good at summing up Reo transitions, even though these might be invalid. In the voltage based approach to Physical Reo, the transition matrices B (4.12) sum up all choices of all nodes (including no-flows). Each of these gives a linear equation that is solvable, and for which it can be checked if the nodes' choices really form a stable state.

Two color semantics is best compared to the current based approach to Physical Reo. Both are not context sensitive and not data sensitive. For the cases where the current based approach does stabilize, it does guarantee that no causality loops occur. Three color semantics is best compared to the voltage based approach to Physical Reo. Both handle context sensitivity well, and are data insensitive. Here too, an advantage of Physical Reo is that it permits no causality loop. A disadvantage is that not all transitions are found by it.

Constraint automata are, like two color semantics, best compared to the current based approach. They are both not case sensitive, but constraint automata have the added value of data sensitivity. Again, if a transition is found by Physical Reo, it has the advantage of not having causality loops.

Since constraint satisfaction is actually a meta-model, and Physical Reo is a semantic model, it is better to discuss whether Physical Reo is a useful model to use with constraint satisfaction than to compare the two. This is probably not the case, since Physical Reo is a method to determine a valid Reo transition, and not a set of constraints that determine whether a

transition is valid.

channel	underlying circuit	`corr`
$N_G = \{n_1, n_2\}$ $E_G = \{e_1\}$ $I_G : e_1 \mapsto (n_1, n_2)$	$N_G = \{n_1, n_2\}$, $E_G = \{e_{e_1}\}$ $I_G : e_{e_1} \mapsto (n_1, n_2)$ $\texttt{const} = \emptyset$	$\texttt{corr}$: $(n_1, e) \mapsto (n_1, e_{e_1})$ $(n_2, e) \mapsto (n_2, e_{e_1})$
$N_G = \{n_1, n_2\}$ $E_G = \{e_1\}$ $I_G : e_1 \mapsto (n_1, n_2)$	$N_G = \{n_1, n_{e_1}, n'_{e_1}, n_2\}$, $E_G = \{e_{e_1}, e'_{e_1}\}$ $I_G : e_{e_1} \mapsto (n_1, n_{e_1}), e'_{e_1} \mapsto (n'_{e_1}, n_2)$ $\texttt{const} = \{e_{e_1}, n_{e_1}\}$	$\texttt{corr}$: $(n_1, e) \mapsto (n_1, e_{e_1})$ $(n_2, e) \mapsto (n_2, e'_{e_1})$
$N_G = \{n_1, n_2\}$ $E_G = \{e_1\}$ $I_G : e_1 \mapsto (n_1, n_2)$	$N_G = \{n_1, n_{e_1}, n'_{e_1}, n_2\}$, $E_G = \{e_{e_1}, e'_{e_1}\}$ $I_G : e_{e_1} \mapsto (n_1, n_{e_1}), e'_{e_1} \mapsto (n'_{e_1}, n_2)$ $\texttt{const} = \{n'_{e_1}, e'_{e_1}\}$	$\texttt{corr}$: $(n_1, e) \mapsto (n_1, e_{e_1})$ $(n_2, e) \mapsto (n_2, e'_{e_1})$
$N_G = \{n_1, n_2\}$ $E_G = \{e_1\}$ $I_G : e_1 \mapsto \{n_1, n_2\}$	$N_G = \{n_1, n_{e_1}, n'_{e_1}, n_2\}$, $E_G = \{e_{e_1}, e'_{e_1}, e''_{e_1}\}$ $I_G : e_{e_1} \mapsto (n_{e_1}, n_1), e'_{e_1} \mapsto (n_{e_1}, n_2), e''_{e_1} \mapsto (n'_{e_1}, n_{e_1})$ $\texttt{const} = \{n'_{e_1}, e''_{e_1}\}$	$\texttt{corr}$: $(n_1, e) \mapsto (n_1, e_{e_1})$ $(n_2, e) \mapsto (n_2, e'_{e_1})$
$N_G = \{n_1, n_2\}$ $E_G = \{e_1\}$ $I_G : e_1 \mapsto \{n_1, n_2\}$	$N_G = \{n_1, n_{e_1}, n'_{e_1}, n_2\}$, $E_G = \{e_{e_1}, e'_{e_1}, e''_{e_1}\}$ $I_G : e_{e_1} \mapsto (n_1, n_{e_1}), e'_{e_1} \mapsto (n_2, n_{e_1}), e''_{e_1} \mapsto (n_{e_1}, n'_{e_1})$ $\texttt{const} = \{e''_{e_1}, n'_{e_1}\}$	$\texttt{corr}$: $(n_1, e) \mapsto (n_1, e_{e_1})$ $(n_2, e) \mapsto (n_2, e'_{e_1})$

Table 4.1: Definition of `und` for some simple channels

channel	underlying circuit	`corr`
$N_G = \{n_1, n_2\}$ $E_G = \{e_1\}$ $I_G : e_1 \mapsto (n_1, n_2)$	$N_G = \{n_1, n_{e_1}, n'_{e_1}, n_2\}$, $E_G = \{e_{e_1}, e'_{e_1}, e''_{e_1}\}$ $I_G : e_{e_1} \mapsto (n_1, n_{e_1}), e'_{e_1} \mapsto (n_{e_1}, n_2), e''_{e_1} \mapsto (n_{e_1}, n'_{e_1})$ $\texttt{const} = \{e''_{e_1}, n'_{e_1}\}$	$\texttt{corr}$: $(n_1, e) \mapsto (n_1, e_{e_1})$ $(n_2, e) \mapsto (n_2, e_{e_1})$
$N_G = \{n_1, n_2\}$ $E_G = \{e_1\}$ $I_G : e_1 \mapsto \{n_1, n_2\}$	$N_G = \{n_1, n_{e_1}, n'_{e_1}, n_2\}$, $E_G = \{e_{e_1}, e'_{e_1}, e''_{e_1}\}$ $I_G : e_{e_1} \mapsto (n_1, n_{e_1}), e'_{e_1} \mapsto (n_2, n_{e_1}), e''_{e_1} \mapsto (n_{e_1}, n'_{e_1})$ $\texttt{const} = \{e''_{e_1}, n'_{e_1}\}$	$\texttt{corr}$: $(n_1, e) \mapsto (n_1, e_{e_1})$ $(n_2, e) \mapsto (n_2, e'_{e_1})$
$N_G = \{n_1, n_2\}$ $E_G = \{e_1\}$ $I_G : e_1 \mapsto \{n_1, n_2\}$	$N_G = \{n_1, n_{e_1}, n'_{e_1}, n_2\}$, $E_G = \{e_{e_1}, e'_{e_1}, e''_{e_1}\}$ $I_G : e_{e_1} \mapsto (n_{e_1}, n_1), e'_{e_1} \mapsto (n_{e_1}, n_2), e''_{e_1} \mapsto (n'_{e_1}, n_{e_1})$ $\texttt{const} = \{e''_{e_1}, n'_{e_1}\}$	$\texttt{corr}$: $(n_1, e) \mapsto (n_1, e_{e_1})$ $(n_2, e) \mapsto (n_2, e'_{e_1})$

Table 4.2: Definition of `und` for some channels with `oer` nodes

Chapter 5

Conclusion

In this thesis, Physical Reo, a new semantic model for the Reo coordination language, has been presented. Physical Reo is a computational semantic model, in contrast to constraint semantic models like connector coloring and constraint automata. It has been presented in the form of two approaches: a voltage based approach and a current based approach.

To support the introduction of Physical Reo, an underlying, more orthogonal description of connectors has been introduced. With the introduction of nodes that have a behavior that is dual to that of the known Reo nodes, most connectors can be described with synchronous channels as their only used channel type.

The two developed approaches to Physical Reo must be noted not to implement it perfectly. The voltage based approach, due to its implicit assignment of a relative priority to every data item, may not allow data flow for all voltage initializations, even if the circuit it describes does have a solution *with* flow. This problem seems to be inherent to the voltage based approach.

The current based approach as presented is not guaranteed to stabilize for all circuit states. This makes the approach unsuitable for implementation. Moreover, it is not clear if and how data sources can determine locally whether a Physical Reo circuit has stabilized, which is needed before data can flow. The current based approach, on the other hand, *does* have advantages over the voltage based approach. These are, firstly, that decisions about which data flows where are made by the nodes, as in Reo, instead of by the data items flowing through them, and secondly, that these decisions are not deterministic. The latter means that one circuit state can lead to different transitions, depending on the nondeterministic choices that nodes

make. It is for these reasons that the current based approach seems to be more interesting for further research.

Notwithstanding the limitations of the voltage based approach, the presented algorithm based on it has an added value because of its efficiency. Future work needs to be done, though, to determine how big the chance is that a voltage initialization leads to a non-blocking Reo transition, and which combination of voltage initializations gives the best performance for finding a transition.

Apart from being an implementation of Physical Reo, the presented algorithm may also instigate more research on similar walk based implementations on Reo. The abstractions it makes with respect to the voltage based approach may open doors for improvements that are not *necessarily* compatible with Physical Reo, but still work well as adaptations of the Physical Reo based walk through through connectors.

Bibliography

[1] Farhad Arbab. Reo: a channel-based coordination model for component composition. *Mathematical Structures in Computer Science*, 14(3):329–366, 2004.

[2] Christel Baier, Marjan Sirjani, Farhad Arbab, and Jan Rutten. Modeling component connectors in reo by constraint automata, 2004.

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

[4] Dave Clarke, Jose Proenca, Alexander Lazovik, and Farhad Arbab. Channel-based coordination via constraint satisfaction, 2010.