Universiteit Leiden

Opleiding Informatica

Reducing copying and network traffic in Reo circuits

Name: Lieuwe Vinkhuijzen
Date: 29/08/2014

1st supervisor: Farhad Arbab
2nd supervisor: Marcello Bonsangue

BACHELOR THESIS

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

Abstract

The Reo language allows software developers to implement concurrency protocols by drawing Reo circuits. The Reo compiler translates Reo circuits into concurrent `C` code. Optimizing the compiler to produce faster code is a major part of making Reo a competetive alternative to traditional synchronization methods.

The compiler currently overestimates the amount of copies necessary to satisfy all components in three ways: (1) Components like the `LossySync` and the `Drain` channels do not always need data, (2) `Reader`s which do not modify a data object may share one copy of that object and (3) too many copies are sent over the network in distributed systems.

Shortcomings (1) and (2) are addressed with a colouring scheme. Shortcoming (3) is addressed with a set of semantics-preserving circuit transformations and a smart distribution and caching protocol.

Contents

Chapter 1

Introduction

The advent of multicore computers and multi-node computer networks gives programmers new challenges: those of designing modular but fast concurrent software. Whereas it is now commonplace to design modular non-parallel software, this is not the case for the concurrent parts of the software. In particular, it is commonplace to use many different low-level synchronization primitives such as locks, mutexes, semaphores and monitors, dispersed over large amounts of code. Sequential programs can be modular because their code can be found in one place. Concurrent protocols, on the other hand, are not defined as a protocol. They emerge from the handful of synchronization primitives used in many different threads.

Reo [1] bridges this gap by allowing developers to define concurrency protocols in single units: Reo circuits. These circuits can be composed in a modular fashion to form larger, more complex circuits which define arbitrarily complex behaviour.

While it is now possible to generate concurrent code from Reo circuits which can replace traditional concurrent code without loss of functionality, it has been difficult to generate code which can compete with those methods in terms of execution speed.

As part of the effort to improve the Reo engine, in Chapter 3 I expose three ways in which the current Reo compiler overestimates the amount of copies necessary to satisfy all elements of the circuit. In Chapters 4, 5, 6 and 7, I propose solutions to these shortcomings by extending the colouring scheme proposed by Clarke et. al [2], by introducing a set of semantics-preserving circuit transformations and by outlining a distribution and caching protocol. These solutions constitute my contribution to Reo. Finally, in Chapter 9 I outline benchmark estimates of the implementation of my solutions.

Chapter 2

Related works

2.1 Colouring a Reo circuit

A channel in Reo represents a constraint on the data flow between the nodes on either end of the channel. Composing channels into a circuit amounts to composing these constraints, and firing a Reo circuit amounts to solving these constraints. Solving the composed constraints imposed by a circuit is no trivial task. If memory permits, therefore, it would be faster to enumerate the possibilities beforehand.

Of the thirty semantic formalisms proposed for Reo [3], the colouring scheme proposed by Clarke et al. [2] is one that explicitly enumerates all alternative ways to fire a Reo circuit and it is the scheme I will build upon to solve two of the compiler's shortcomings outlined in the Introduction.

Clarke et al. propose a scheme in which every channel and every node is assigned one of a small set of colours. In this work, we deviate from this convention. Instead of assigning each node a colour, we assign each end of a channel a colour.

A scheme with three colours is sufficient to describe the behaviour of any Reo circuit. For the purposes of illustration, consider for now a scheme with only two colours: *flow*, denoted F, and *no flow*, denoted N. The following tables list the possible ways to *colour* the two ends of several Reo primitives using a two-colour colouring scheme.

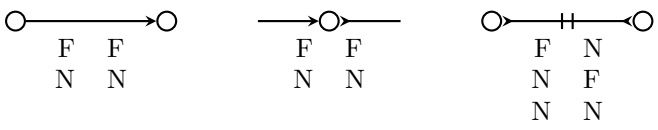


Figure 2.1: The two-colour tables for the `Sync` channel, the binary `Replicator` node and the `AsyncDrain` channels.

Each entry, or colouring, in the colouring table represents a possible flow configuration for a particular component or node. By composing the tables of the components in a circuit, we obtain the colouring table for that circuit. This composition process is very simple. A channel or node is chosen to be the starting point, and its colouring table is taken as the starting colouring table. Next, another channel or node is chosen. Every colouring in the original colouring table is checked pairwise with every colouring in the new primitive's table. Every colouring in which the colours of all channel ends common to both subcircuits match are valid colourings, and are taken up in the composed colouring table. The resulting colouring table is in turn composed with another component in the circuit, and so on, until all components have been incorperated into a colouring table.

2.2 Example

To illustrate the process of composing a circuit's colouring table, the example used by Clarke is reproduced here. Consider the circuit below, in which an `AsyncDrain` receives input from two `Sync` channels.

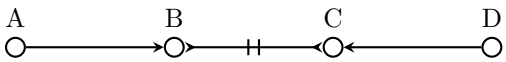


Figure 2.2: A simple circuit.

To compose the colouring table of this circuit, we compose the colouring tables of the two `Sync`s, the `AsyncDrain` and nodes B and C. The order in which this is done is irrelevant for the final colouring table, so we will work from left to right. First, we start with the colouring table of the leftmost `Sync`.

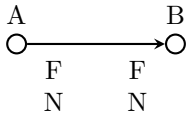


Figure 2.3: The colouring table of a Sync channel.

Next, we compose this colouring table with the colouring table of the `Replicator` node. The subcircuits of the `Sync` channel and the `Replicator` node overlap on the `Sink` end at node B, so the colours on that channel end are compared for equality pairwise.

Next, in Figure 2.5 the resulting colouring table is composed with the colouring table of an `AsyncDrain`. The circuit of the resulting colouring table and the `AsyncDrain` overlap on the `Sink` node at node C, so that is where the pairwise comparison of colours happens.

Composing this colouring table with that of `Replicator` node C and then with that of the `Sync` between D and C yields the final colouring table in Figure

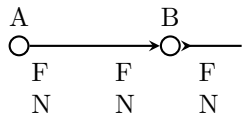


Figure 2.4: The colouring table of a Sync channel, composed with that of Replicator node B.

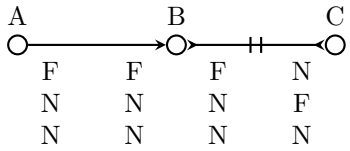


Figure 2.5: The colouring table of Figure 2.4, composed with that of the `AsyncDrain`.

2.6.

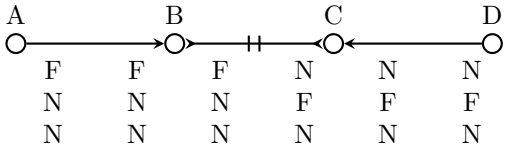


Figure 2.6: The colouring table of Figure 2.5, composed with that of a SyncDrain.

2.3 Distinguishing between different sets of `put` and `get` calls

`Writer`s with a pending `put` may be coloured with a different set of colours than writers which perform no `put`. In general, components have a separate colouring table for each state they can be in. FIFOs, for example, can be in one of two states, other channels can be in only one state and, as we will see later, `Reader`s and `Writer`s can be in one of four states.

Whenever the states of components are independent, the circuit as a whole can be in any combination of these states. In generating the compositional lookup table, therefore, one colouring table is generated for each combination of states. For example, a circuit with one `Writer` and one `Reader` will have 16 colouring tables. The states of components are not always independent. For example, the `FIFO`s of an n-sequencer may be in only one of n states at any given time, rather than in one of 2^n states.

Chapter 3

Shortcomings of the Reo compiler

At compile-time, a Reo circuit is translated into executable code by the Reo compiler. In doing so, the Reo compiler overestimates the amount of copies necessary to satisfy all `Reader`s, `Filter`s, `Transformer`s and `FIFO`s in the circuit, because it does not take into account the fact that not all components always *need* a copy; sometimes, distributing a shared reference will suffice, sometimes distributing only a signal will suffice, and sometimes only one copy need be transmitted over a network, so that the receiving component can make the remaining copies locally. I propose three modifications to the engine to make better estimates of the minimum amount of copies.

First, the engine naively passes data to all elements in the circuit. However, not all channels always need data. Making new copies to feed to these channels can be avoided if, instead of *data*, a *signal* were sent to these elements. The elephant in the room is the class of Drain channels. These channels do not pass their data to other components, nor do they inspect the data themselves. Indeed, data is destroyed as soon as it enters a `Drain`.

Second, each `Reader` currently receives its own, personal copy of data received from a `Writer`. If the `put` and `get` functions were modified to include information about how the caller intends to use the data, then copies need only be made for those who intend to modify the data, and all who intend only to read the data but not modify it can share a common reference to the data object.

Third, in distributed systems, several physical processing units may implement different parts of a Reo circuit. A data item may therefore flow back and forth between these units through several channels, even within one run of a synchronous region. A set of protocols that limit the amount of copies transmitted is described in section 7.

To illustrate the added value of these three modifications, consider the code that may be generated by a naive Reo compiler for the circuit in Figure 3.1.

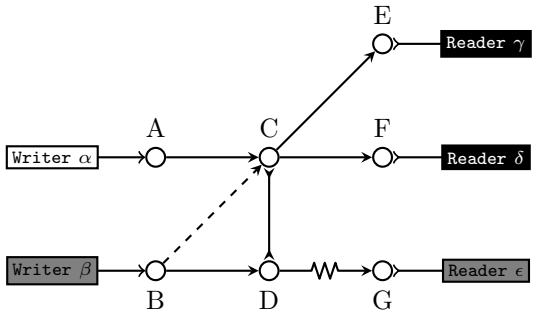


Figure 3.1: A Reo circuit which allows `Writer`s α and β to send packets of data to `Reader`s γ, δ and ϵ. One site hosts α, one site hosts γ and δ and one site hosts β and ϵ.

This circuit non-deterministically propagates data from α and β to γ, δ and ϵ. The colour of a component represents the site on which it is hosted. This circuit is hosted on three different sites (say, in Leiden, Amsterdam and Utrecht), two of which host multiple components.

Code generated by a naive Reo compiler might send data into the two LossySyncs, even though only one of them will ever propagate data. It might send data produced by α over the network twice: once to γ and once for δ. This is not necessary: when one copy is sent over the network, the black site can locally copy the data without more network traffic. Even that may not always be necessary. If γ and δ only intend to inspect the data but not modify it, they can share a reference to a mutually accessible chunk of memory. A similar situation arises when β produces data for ϵ. If neither intends to modify the data after β has published it, the data need not be copied, and ϵ need only receive a reference to the data object. When β produces data that does not pass the Filter's pattern, it should find out before sending it to ϵ's site.

This suboptimal compiler is not the compiler used at the time of writing. In fact, the Reo compiler already produces very fast code. Comparisons to a naive compiler, therefore, only serve an explanatory purpose. Even if most suggestions go unimplemented, this dissertations study of colouring tables and circuit transformations yield insights which are valuable in their own right and on which we will reflect in the conclusion.

Chapter 4

Passing signals instead of data

4.1 Introduction

As we saw in chapter 3, a compiler needs to take into account many facts about the nature of certain channels, the intentions of components and their distribution among different host sites lest it unnecessarily makes or transmits copies of data. We will call a compiler which does none of these things but does preserve the semantics of Reo a *naive* compiler. By code generated by a naive engine, we mean code which behaves exogeneous: nodes have no knowledge of the types of the channels they send their data to. When a node propagates data, each source end adjacent to the node will receive its own copy of that data, regardless of whether data is needed there.

It is already clear that this is a suboptimal implementation of Reo, because we know that many components do not need to be provided with data. Most notably, `Drain` channels immediately delete data they receive, and the `LossySync` channel may lose its data before a component reads it. If these channels were provided with a signal rather than with data, some of the overhead of copying data would be avoided.

4.2 Extention to the *trois couleurs* scheme

To implement this improvement, we use two modifications to our previous colouring scheme: In the case of data flow, we distinguish between *flow of data*, denoted D, and *flow of a signal*, denoted S. In the case of no flow, we distinguish between *no flow because nothing was offered*, denoted O, and *no flow because nothing was requested*, denoted R. The well-read reader will notice that the distinction between data and signal is the only extention to the *trois couleurs* colouring scheme proposed by Clarke et. al. [2].

Interpreting the no-flow colours can be tricky, because the interpretation of the O and R colours depend on which end of a channel is coloured. Table 4.1 offers a field guide to these two colours and translates them to the symbols used by Clarke et. al. Note that subsequent literature swapped the meanings of $\triangleleft$ and $\triangleright$. The animations in the Reo ECT [4] uses the new notation.

Colour	Source end	Sink end
O	node offered no data $\triangleright$	channel offered no data $\triangleleft$
R	channel made no request $\triangleleft$	node made no request $\triangleright$

Figure 4.1: Interpretation of the no-flow colours, and their translation to the notation used in earlier literature [2]. Prepend each reason with *there is no flow because the...* .

The rules for composing two tables remain the same, save the fact that we will use a larger set of colours and a different set of colouring tables both for channels and for nodes.

For a colouring to be useful, channels coloured D should indicate that the data they propagate will at one point enter a data-sensitive component such as a `Reader` or Filter. In particular, a channel end should *only* be coloured D when the data whose flow it represents enters a data-sensitive component later in the circuit, and should be coloured S otherwise. For the source end of a channel, this means it may be coloured data if the channel itself is data-sensitive. While it it useful as a first principle and as a validity check to require this property of any colouring we design, it is of no use in designing colouring tables for primitives because the requirement is indogenous in nature whereas all primitives are exogenous by design.

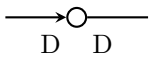
Instead, it turns out that colouring tables which satisfy this requirement can be designed by employing the *no promotion or demotion principle* (NPD Principle). This principle will be useful in the design of colouring tables both in the four-colour scheme under consideration here and the five-colour scheme we consider in the next chapter.

NPD principle No data-insensitive primitive may promote a signal to data, and no primitive may demote data to a signal from its `Source` to its `Sink` end.

The principle applies to directed channels and to Merger and Replicator nodes. To illustrate the principle, the colouring tables of the `Sync` and the unary `Replicator` node are shown in Figure 4.2.

The NPD principle speaks of promotion because we impose a hierarchy on the flow colours, in which D stands above S. For a Replicator with two output channels, this means the input channel must be coloured D whenever at least one

`Replicator` node



D	D
S	S
O	O
R	R
O	R

`Sync` channel

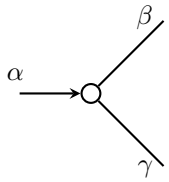
D	D
S	S
O	O
R	R

Figure 4.2: The colouring tables in five-colour semantics for the `Sync` and the binary Replicator node. Neither promotes signals or demotes data.

of the output channels is coloured data. Only when both outgoing channels are coloured S is the input channel coloured S. The binary `Replicator`'s colouring table is shown in Figure 4.3a. Larger Replicators can be constructed using the composition rules.

The colouring table for a binary Merger in Figure 4.3b enumerates all colourings in which there is either no flow, or the Merger requests as input whatever is requested as output. If these two tables seem rather large, it is because Clarke's *flip-rule* has not been implemented. In Appendix A, we introduce a new kind of flip rule, dubbed the NPD composition rule, which curbs the size of the flow-part of the `Replicator`'s colouring table.

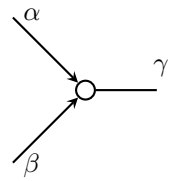
Binary `Replicator` node



α	β	γ
D	D	D
D	S	D
D	D	S
S	S	S
R	R	R
O	O	O
O	O	R
O	R	O
O	R	R

(a) The colouring table of a binary `Replicator`

Binary `Merger`



α	β	γ
D	D	O
D	D	R
D	O	D
D	R	D
S	S	O
S	S	R
S	O	S
S	R	S
O	O	O
O	O	R
O	R	O
O	R	R
R	R	R

(b) Colouring table of a binary `Merger`

The tables for `Reader`s and `Writer`s are copied from those proposed by Clarke et. al., save the fact that *flow* has been replaced by *flow of data*. Their tables are outlined in Figure 4.5. [1]

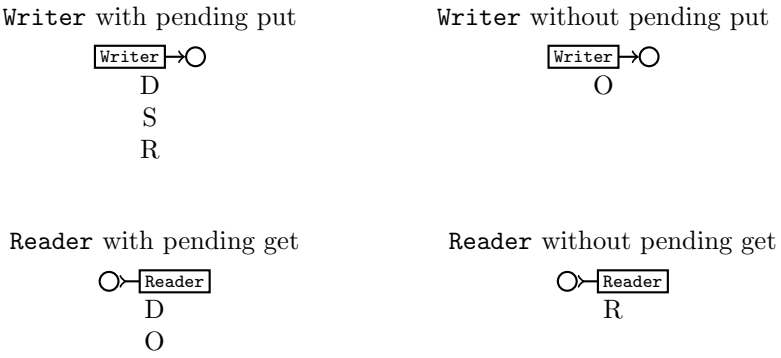


Figure 4.5: Colouring tables for `Reader`s and `Writer`s in four-colour semantics.

4.3 Data-sensitive primitives and the NPD principle

The no promotion or demotion principle applies to data-insensitive channels. Consider a scenario where a Filter feeds directly into a `SyncDrain`. The Filter must demote an incoming data item that satisfies the Filter's predicate. On the other hand, a hypothetical `Constructor` channel which autonomously builds new data whenever it receives an input may promote a signal to data.

The following addendum is appended to the NPD principle: data-sensitive channels should request from their `Source` end the lowest item in the hierarchy to satisfy a request. They must be able to satisfy every kind of request on their `Sink` end. To illustrate, Figure 4.6 shows the colouring tables for the Filter and the Constructor channels.

Figure 4.6: The four colour tables of the `Filter` and the `Constructor` channels.

[1] It was mentioned earlier that some `Reader`s are sometimes contend with being sent a signal. We choose to postpone that discussion to the next chapter, which deals specifically with the way components intend to inspect and modify their data.

Chapter 5

Using references instead of copies

5.1 Introduction

Not all `Reader`s always intend to modify the data they receive. In some cases, they are contend simply inspecting the data. When there are multiple such `Reader`s, copying workload can be reduced by providing these readers with a common reference to a data object instead of providing each `Reader` with a personal copy. `Writer`s, similarly, do not always intend to modify or even inspect the data after publication. The common reference may therefore be shared by `Reader`s and `Writer`s alike. To implement this improvement, the Reo engine described in the last section is modified in two ways.

First, we modify the set of colours used in the last section. In the case of *flow of data*, we distinguish between *flow of copy*, denoyed &, and *flow of reference*, denoted $*$. These symbols are chosen to denote their respective data flows because they represent similar concepts in the `C++` language.

Second, the put and get methods, which components use to interface with Reo, are modified to include a parameter indicating how a component intends to use the data. Currently, the `put` and `get` method signatures are as follows:

```
procedure put(p, v)          function get(p)
```

The `usage` parameter is added to both methods:

```
procedure put(p, v, usage)        function get(p, usage)
```

where `usage` $\in \{\&, *, \phi\}$ and where & indicates that the component intends to modify its own copy, $*$ indicates that the component intends only to read

the data and ϕ indicates that the component intends never even to inspect the data.

It follows that a `Writer` is always in one of four states: Either is has done one of three `put` versions, or it has no pending put. It is instructive to see what colouring tables a writer may have, depending on its state.

put(p, v, &)	put(p, v, *)	put(p, v, ϕ)	(no put)
&	*	*	
R	R	R	O

The tables for the `Reader` are similar, save the fact that O and R switch places.

get(p, &)	get(p, *)	get(p, ϕ)	(no get)
&	*	*	
O	O	O	R

5.2 The no promotion principle with five colours

In the four colour scheme of the last chapter, the colouring tables of the `Replicator` node guaranteed certain properties of coloured Reo circuits: channel ends coloured D were guaranteed to propagate their data to a data-sensitive component at some point, and channel ends coloured S were guaranteed not to. The five colour scheme of this chapter has similar guarantees for copies and references.

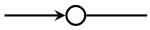
A reference passing through a channel is, as was previously the case, guaranteed to enter a data-sensitive component at some point. Moreover, it is guaranteed to enter a component that requested a reference. The same is true for copies.

To implement this guarantee in the colouring tables, `Replicator` nodes may not promote a reference to a copy or demote a copy to a reference, just as the `Replicator` nodes of the last chapter could not promote a signal to data or demote data to a signal. Tables 5.5b and 5.5a show the colouring tables of the binary and ternary Replicators.

5.3 Modifying and non-modifying Transformers

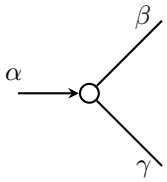
The Transformer deserves special attention because it is the only synchronous channel that can modify data that Reo defines by default[1]. According to its definition, a data object x is taken as input, and another data element, $f(x)$, is produced as output. In practice, there are two kinds of Transformers, corresponding to the two ways the output can be generated.

[1]Developers are free to create new, arbitrarily complex channels on their own.



&	&
*	*
S	S
O	O
R	R

(a) The `Replicator` node's table with copy and reference modifications, with the flip rule implemented.



α	β	γ
&	&	&
&	&	*
&	&	S
&	*	&
&	S	&
*	*	*
*	*	S
*	S	*
S	S	S
R	R	R
O	O	O

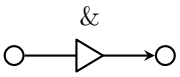
(b) The colouring table of a binary `Replicator`, with the flip rule implemented.

Figure 5.5: The `Replicator` never promotes or demotes data.

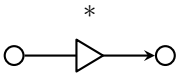
First, a *modifying Transformer* modifies the fields of x, if x is an Object, and outputs the modified x. This Transformer needs a copy of x as input.

Second, a *creating Transformer* takes x as an input and allocates new memory in which to produce its output $f(x)$. None of the fields of x are modified and none are referenced in the output. Consequently, this Transformer may be provided with a reference to x, which it may inspect but not modify.

These two Transformers can be distinguished at compile-time and have different corresponding colouring tables. They can be distinguished by drawing a & or a $*$ above them. The $(*, O)$ tuple in the tables in Figure 5.6 represents the fact that Transformers may reject data, just like Filters.



&	*
*	O
O	O
R	R



*	*
*	O
O	O
R	R

Figure 5.6: Two kinds of `Transformer`s consitute two distinct channel types, with two different colouring tables.

Chapter 6

Circuit transformations

6.1 Introduction

Running Reo on multiple processors inside a cluster, or on several computers, all in different cities, presents challenges not encountered in single-processor multi-threaded applications. In such cases, a Reo circuit coordinates the distribution of data between several sites connected by a network. The Reo compiler must be extended with a protocol geared specifically towards multiple-site Reo networks to avoid sending redundant copies over the network. I propose circuit transformations as this extention.

Of the many conceivable ways to implement Reo on multiple sites, we consider here only an implementation in which each site hosts a set of `Reader`s, a set of `Writer`s, a set of nodes in the circuit, and the set of channels ends adjacent to its nodes. During runtime, sites have no knowledge of the topology of the subcircuits implemented by other sites. During compile-time, however, all such knowledge is available in order to optimize the circuit. To see why optimization is sometimes necessary, consider the following Reo circuit, implemented by computers in Leiden and Amsterdam.

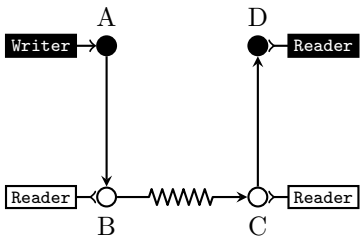


Figure 6.1: A Reo circuit implemented jointly by computers in Leiden (black nodes) and Amsterdam (white nodes).

In diagrams of Reo circuits implemented on multiple computers, we represent the nodes implemented by different sites by colouring the nodes they implement. In Figure 6.1, data may flow from Leiden to Amsterdam and back to Leiden again.

The implementation under consideration does not include a smart distribution protocol which prevents Amsterdam from sending data back to Leiden. Therefore, where the circuit suggests that data is sent from node C to node D, it is interpreted as actually sending data over the network. The circuit in the example is suboptimal: Leiden already has the data; it would be better off copying the data that A produced than receiving it over the network.

By moving channels around, we can construct a different circuit while preserving semantics. We will call such operations circuit transformations. Figure 6.2 shows one such optimized cicuit.

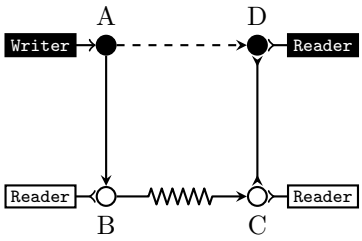


Figure 6.2: The circuit from Figure 6.1, adjusted to avoid redundant network traffic.

6.2 Circuit deployment

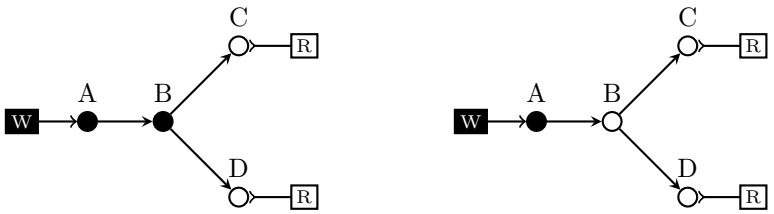
Even when data is not sent 'back' to the original `Writer`'s site, the nodes in the circuit need to be allocated to sites in an intelligent way. Consider the replicator in Figure 6.3. We interpret the circuit in Figure 6.3a as sending data to Amsterdam twice, and the circuit in Figure 6.3b as sending it only once.

In Reo, *deployment* refers to the allocation of `Reader`s and `Writer`s to host computers. *Circuit deployment* extends this concept and refers to the allocation of nodes to host computers.

This example illustrates that there are optimal and suboptimal alternative circuit deployments to choose from.

6.3 Circuit optimality

We are now ready to define what we have loosely referred to as *circuit optimality* in terms of colourings. A circuit is suboptimal whenever either there is a colouring in which:



(a) A Reo circuit implemented jointly by computers in Leiden (black nodes) and Amsterdam (white nodes). Its circuit deployment is suboptimal.

(b) An optimal version of the circuit in Figure 6.3a.

Figure 6.3: Not all circuit deployments are optimal.

1. data passes through a site which does not host a `Reader` which reads that data, or
2. the same data is sent to a site twice, or,
3. data is sent back to the site which hosts the `Writer` or Transformer which created it.

A circuit is said to be optimal whenever none of these conditions hold. The word *optimality* is only used for the sake of brevity. Actually, it refers to *optimality with respect to the amount of copies transmitted over the network in the absence of a smart distribution protocol*. While circuit transformations can optimize a circuit with this definition, it is far from clear that this is a good definition of optimality, a fact on which we will reflect later.

The circuit in Figure 6.2 introduced circuit transformations, which we define as follows:

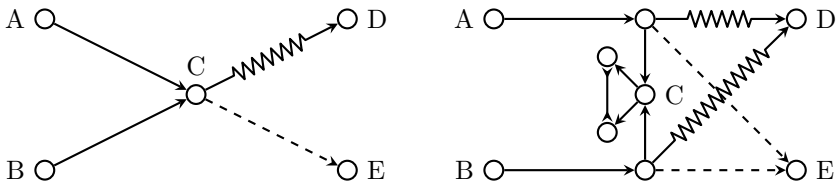
A circuit transformation which operates on a set of nodes N_t preserves all nodes outside N_t and all channels between nodes outside N_t. It may add a set of nodes N_{new}, it may add channels between nodes in $N_t \cup N_{new}$ and it may remove channels between nodes in N_t.

6.4 Transforming a circuit into an optimal one

Two circuit transformations are employed to optimize Reo circuits. The first *postpones* Merges, the second, dubbed *eliminating multiple entry* (EME), makes sure data is not sent to the same site twice.

6.4.1 Postponing a Merger

The first transformation operates on a Merger node and all nodes to which it is directly connected through directed channels, as illustrated in figure 6.4.



(a) Data is merged in node C.

(b) The merge of the data is *postponed* to nodes D and E.

Figure 6.4: The postponing a Merger transformation operates on nodes A, B, C, D and E. The `Sync`, `Filter` and `LossySync` channels represent arbitrary channels. Node C's output nodes are drawn for the sake of visualisation but are not strictly required.

Applying the transformation of Figure 6.4 repeatedly yields a circuit in which all `Writer`s' outgoing directed channels form a tree up to the `Reader`'s `Merger` nodes. Specifically, data from a `Writer` will encounter only replicator nodes, and Merger nodes whose only outgoing channel feeds the data to a `Reader`. This transformation is only applicable when the channels adjacent to the Merger node can be in only one state. [1]

6.4.2 Eliminating multiple entry

A circuit is not guaranteed to be optimal after all of its Mergers have been postponed. The left circuit in Figure 6.5 shows a `Writer` that conditionally sends data to two `Reader`s on the same site. To see that there is no optimal circuit deployment, consider to which site we might allocate the Replicator node. If the Replicator node is allocated to the white site, then data enters the black site through two different channels. There would be multiple entry points from the white site to the black site. If it is allocated to the black site, data will be unnecessarily sent when the data passes neither of the two Filters.

The right circuit in Figure 6.5 shows the circuit after applying the *eliminating multiple entry* transformation (EME), with the nodes optimally allocated to the site. The transformation introduces a new `LossySync` between nodes A and B, intended as the only channel through which data can flow between the two sites. The node labeled OR serves the purpose of an OR gate: it only fires when at least one of its inputs fires. Node B, therefore, only fires when the data packets made it through at least one Filter. The three `SyncDrain`s between the two sites guarantee this coordination. Specificallly, they guarantee that data only flows from node A to node B when it is requested by a `Reader`, offered by the

[1] Only when the synchronous directed channels of a subcircuit are a DAG does postponing Mergers produce a tree. Rare circuits containing a cycle of synchronous directed channels are discussed in section 6.5.

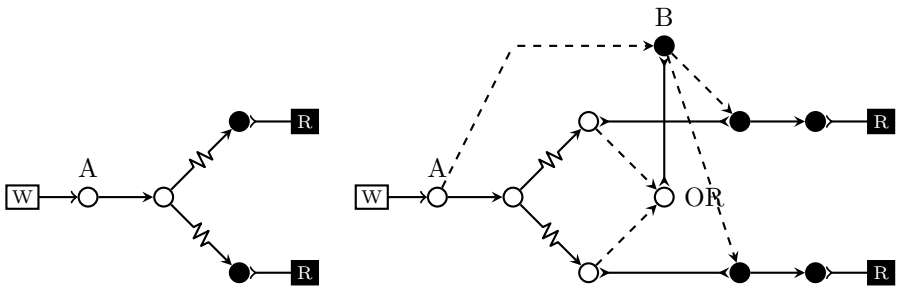


Figure 6.5: The *eliminating multiple entry* transformation, applied to a simple `Replicator`. Data may only flow between the white and black sites through the channel between nodes A and B.

`Writer` and the Filters, and when it is guaranteed to arrive at a `Reader`.

The EME transformation can be generalized to circuits with many `Writer`s and many `Reader`s on the same site. The `Reader`s in Figure 6.6a may receive data from either of two `Writer`s. Data may reach a `Reader` via one of two paths. Applying the EME transformation to this circuit yields the circuit in Figure 6.6b, where the nodes have been coloured according to the optimal circuit deployment.

The idea behind the EME construction here is the same as the one behind the simpler one in Figure 6.5. The `LossySync`s from the `Writer`s to nodes A and B are intended to be the only paths to the black site. As such, they propagate data to all black `Reader`s. The nodes labeled OR_A and OR_B serve as OR gates, guaranteeing, as in the simple case, that when A or B fires, at least one of their respective data packets made it through. Here, the EME transformation was applied to both `Writer`s at once, but it can be applied to a single $(Writer, Reader\ site)$ pair.

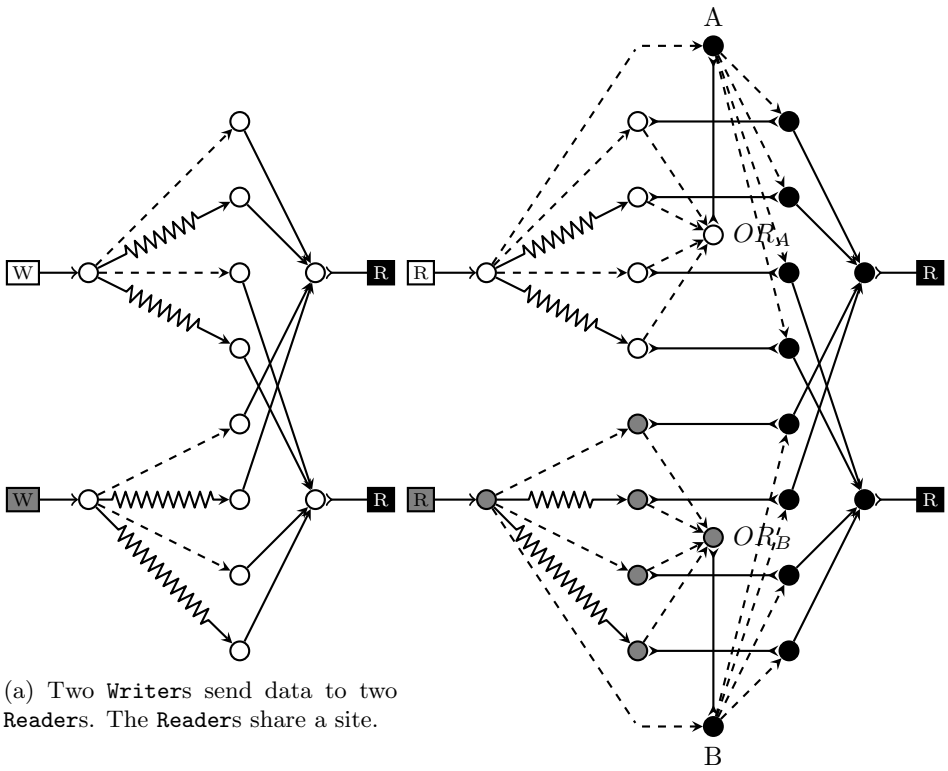
To optimize a circuit in which `Reader`s are distributed over more than one site, first postpone each Merger, and all Mergers that result from that transformation, until no Merger ever sends data to a Replicator. The outgoing channels from the `Writer`s will now form a tree. Afterward, apply the EME transformation consecutively to each $(Writer, Reader\ site)$ pair.

While the transformations presented here yield an optimal circuit, a handful of questions remains. They are addressed in the following sections.

6.5 Circuits regions with a synchronous cycle

When a circuit has a cycle of directed synchronous channels, the transformation described in section 6.4 fails to yield an optimal circuit. The Merger postpone transformations have not eliminated the cycle in the circuit, and so the `Writer`s' outgoing channels do not form a tree.

Figure 6.7a shows that the Merger postponing transformations on nodes A, B and C have preserved the cycle in nodes α, β and γ. The `Writer`s' outgoing



(a) Two `Writer`s send data to two `Reader`s. The `Reader`s share a site.

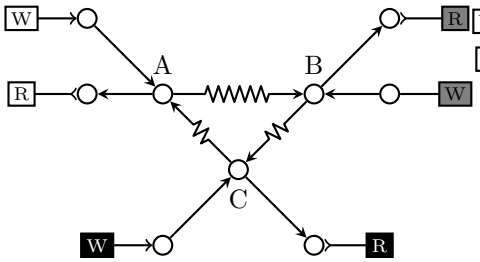
(b) The nodes' colours indicate the optimal circuit deployment after the EME transformation.

channels do not form a tree, and so we cannot arrive at an optimal circuit. Note however, that α will never receive data from γ, because γ receives all its data from α. Since optimal circuit deployment is defined not in terms of the topology of the circuit, but in terms of its colouring table, we can ignore the channel between α and γ. Without it, the circuit becomes acyclic, and circuit transformations will yield an optimal circuit.

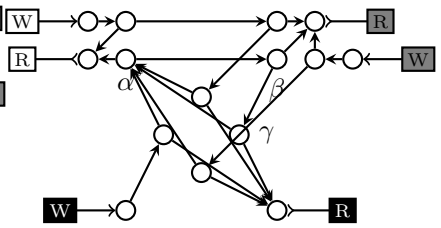
6.6 EME is only necessary between sites

Applying the EME transformation aims to eliminate the problem where `Writer`s send data to another site through two different channels simultaneously. When a `Writer` shares a site with two `Reader`s, therefore, there is no *multiple entry* to speak of, because the data never leaves the site. In these cases, applying EME is not necessary.

Similarly, when a `Writer` sends data to only one of a site's `Reader`s, there is no *multiple entry* either. Applying EME is not necessary.



(a) A synchronous cycle before transformations, distributed over three sites.

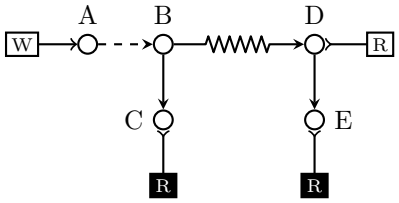


(b) The circuit after postponing Mergers A, B and C, in that order, where we abstract away from the channel types.

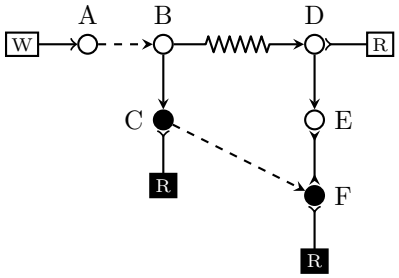
Figure 6.7: Circuit transformations preserve cycles. There is no optimal circuit deployment either before or after postponing the `Merger`s.

6.7 Tiny optimal circuits

Sometimes, only very small changes to a circuit are necessary to make it optimal. Consider the circuit in Figure 6.8a, and an equivalent optimal circuit in Figure 6.8b. We have prevented the elaborate circuit transformations proposed earlier by making a simple observation: node E receives data only when node C receives data. Specifically, if C_E is the set of colourings in which data passes through node E, and C_C is the set of colourings in which data passes through B, then $C_E \subseteq C_C$. Whenever this is the case, and E always receives the same data as C (i.e. from the same `Writer` or Transformer), a new node F may be synchronized with node E through a `SyncDrain` and receive data from C through a `LossySync`, as in Figure 6.8.



(a) A circuit which is very easy to optimize.



(b) F receives data whenever E does, and there is an optimal circuit deployment.

Figure 6.8: Circuits in which some nodes which only receive data whenever other nodes do are easy to optimize.

This trick generalizes to sets of nodes. If $C_A \subseteq C_X \cup C_Y$, then a new node B may be synchronized with A and receive data only through `LossySync`s from

X and Y. Moreover, when $C_A \cup C_B \subseteq C_X$, then the EME transformation can be applied to A, B and X, interpreting X as the `Writer` and A and B as two `Reader`s.

6.8 Data-sensitive drain channels

The set of transformations used to optimize a circuit described in section 6.4 does not take into account data-sensitive drain channels. Indeed, the interpretation of multi-site circuits described in section 6.1 does not specify which site should host channels which straddle two sites, and Reo by default defines no data-sensitive drain channels.

For the purposes of optimizing a circuit, then, a user-defined data-sensitive drain channel should be allocated to some site and be interpreted as two distinct `Reader`s both hosted by that site - one `Reader` for either end of the Drain channel. This interpretation of such drain channels allows the transformations to achieve optimality in the usual way.

Similarly, the interpretation does not define which site should host spouts. They, too, should be allocated to some site. The algorithm will achieve optimality without interpreting these channels as `Writer`s.

Chapter 7

Smart distribution protocol

7.1 Introduction

In distributed Reo circuits, we may choose to employ an alternative method to circuit transformations to solve the problem of sending duplicate data across the network. If each site were to employ a *smart distribution protocol*, then, after establishing which of the circuit's colourings is chosen for this firing of the circuit, a site could communicate to other sites which of its `Writer`s are going to send data, and other sites could indicate that they will receive a particular `Writer`'s data. This allows a site to build a set of recipients of each of its `Writer`s, and so prevent sending duplicate data.

There are two reasons we might want to employ a smart distribution protocol instead of or in tandem with circuit transformations. First, the circuit transformations defined here solve the problem using a particular definition of circuit optimality. Distributed Reo systems are still in their early stages of development, and it may turn out that this definition is not relevant. Second, when there are `FIFO`s in the circuit, data is stored between consecutive runs of the circuit. Data may enter a site multiple times over the course of several runs of the circuit, either by being stored in one of the `FIFO`s that sites host, or by entering its `Reader`s. Moreover, `Writer`s may publish the same data multiple times. Circuit transformations are not helpful in the context of multiple runs of a circuit.

7.2 Smart distribution protocol

The outline of a protocol we propose is as follows.

First, the sites that implement a single synchronous region must establish which of the circuit's colourings they will use for this run. When a data item is produced, a unique bitstring is associated with it. Upon sending data to another computer, the sender checks whether it knows that the receiver already has a copy of that data item. If the receiver has a copy, only the bitstring identifier

is sent to inform the receiver which data item it should fetch from its cache. Otherwise, it asks the receiver whether it has a copy. If not, then a copy of the data item is sent, and the receiver is added to the sender's local copy of the set of computers which have received a copy of the data. Upon receiving a copy of a data item, the receiver remembers that the sender has a copy of that data item.

A pseudocode implementation of this protocol is shown in Figure 7.1.

```
// Within each computer, save the computers that have received a certain data item
map<bitstring data identifier, set of sites> received;

onProduce(dataItem) {
  dataItem.identifier := generateIdentifier();
}

send(dataItem, receiver) {
  if (received[dataItem.identifier].contains(receiver)) {
    sendIdentifier(dataItem.identifier, receiver);
  }
  else {
    if (ask(receiver,dataItem.identifier)) {
      sendIdentifier(dataItem.identifier, receiver);
    }
    else {
      sendCopy(dataItem, receiver);
      received[dataItem.identifier].add(receiver);
    }
  }
}

onReceive(dataItem, sender) {
  received[dataItem.identifer].add(sender).
}
```

Figure 7.1: Pseudocode implementation of the smart distribution algorithm

It may not be desirable to cache all data items ever received. For example, in the context of a simple producer and consumer, the consumer may need each item just once. In addition to a smart distribution protocol, therefore, we need a smart caching protocol.

We should note that, while the protocol described here does reduce the amount of copies sent over the network, it introduces new complexity and new network traffic. When transmitting small packets of data over a fast connection, therefore, it may be faster to simply send the data than to apply the protocol.

7.3 Dynamic deployment of buffers

This protocol spawns the question of which site should host a `FIFO`. Ideally, to limit network traffic, a `FIFO` is hosted by a site with components that will eventually read the data, or the site that produced the data. However, it is not always possible to predict where in a circuit a data item will flow, and multiple sites may take turns to fill a `FIFO`, so neither suggestion is a silver bullet.

One way to solve the buffer problem is to let sites take turns to host a `FIFO`. For example, we may choose to allocate a `FIFO` to a site (and away from the site that last hosted it) whenever a `Writer` on that site deposits an item into the `FIFO`. That way, data is only sent between sites when it arrives at a `Reader`. Alternatively, if, for some state of the circuit, it is possible to predict to what site a `FIFO` will transmit its data item when it next fires, the `FIFO` may be allocated to that site.

This protocol would perform optimally with respect to the amount of copies sent over the network, but to implement it, we would have to step away from static circuit deployment and implement dynamic circuit deployment. The sites implementing the synchronous region adjacent to the buffer would have to allocate the buffer during runtime and communicate where the `FIFO` is hosted.

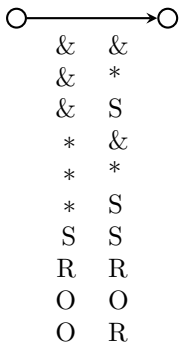
Chapter 8

Colouring tables implemented by the engine

We have proposed a colouring system with five colours: S, $*$, &, R and O. For technical reasons, the Reo engine would generate much faster code if a system with four colours were implemented than if it were to use five colours.

When a component requests a signal, it indicates that it does not intend to inspect any data it receives. These components would be happy to accept a reference instead of a signal. Since references need not be much bigger than signals, we can replace signals by references in all our tables. The colouring tables to be implemented by the Reo engine are the tables produced by this operation, with duplicate entries removed. The compositional rules do not change.

For example, the colouring table of the replicator node will change as depicted in Figure 8.1.



&	&
&	$*$
&	S
$*$	&
$*$	$*$
$*$	S
S	S
R	R
O	O
O	R

(a) `Replicator` node colouring table as described by copy-reference

&	&
&	$*$
$*$	&
$*$	$*$
R	R
O	O
O	R

(b) Replicator node colouring table after changing S to $*$

Figure 8.1: The process of modifying the `Replicator` node serves as an example for modifying the rest of the tables.

Chapter 9

Expected speedup

The speedup to be expected from implementing the proposed modifications is measured as the amount of times making or transmitting a copy was prevented. It can be measured separately for each of the three modifications.

Consider again the example from chapter 3, reproduced here in Figure 9.1.

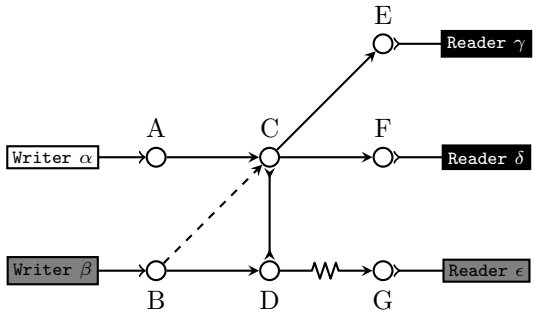


Figure 9.1: A Reo circuit which allows `Writer`s α and β to send packets of data to `Reader`s γ, δ and ϵ. One site hosts α, one site hosts γ and δ and one site hosts β and ϵ.

When both α and β send data, the data-signal modification prevents a copy being sent into the LossySync between B and C. It will also prevent two copies being made for the SyncDrain between nodes C and D. Instead, the SyncDrain will be provided with two signals.

In general, for each component which requires a signal and not data, a copy is saved in comparison to a naive implementation of Reo.

If β produces an item received by γ, δ and ϵ, and all four components need only inspect the data but not modify it after publication, then no copies need be made. The copy-reference modification will have prevented the creation of three copies. In general, the amount of copies necessary in total to satisfy both

`Writer` action	Are there `Reader`s which do put(*)?	
	Yes	No
put(&)	$R_c + 2$	$R_c + 1$
put(*)	$R_c + 1$	$R_c + 1$
put(ϕ)	$R_c + 1$	R_c

Figure 9.2: The amount of duplicates of a data item necessary to satisfy both `Reader`s and `Writer`s

`Reader`s and Writers can be deduced from the following table, where R_c is the amount of readers that request a copy.

The table can be summarized by the following rule of thumb: The amount of duplicates needed is the amount of components which need a copy, plus one more if there are components which require only a reference. Note that the amount of copies that need to be made is one less than the amount reported in table 9.2, because the first duplicate is the original, produced by the Writer.

Circuit transformations do not always solve all redundant network traffic. The Merger in Figure 9.3 cannot be postponed because the `FIFO1` is a multi-state channel. Which site should host the `FIFO1`? If the white site hosts it, the black site will invariably send its data to that site, even when that data ends up flowing only to the black `Reader`. Moreover, when data does flow back from the `FIFO1` to the black site, the black site may already have the data in cache, because it was the one that produced it.

Suppose the black `Writer` and `Reader` perform a put and a get, respectively. If the `FIFO1` is allocated to the white site, data published by the black Writer is transmitted unnecessarily. When, in a next firing of the Reo circuit, the data flows to the black `Reader`, it flows there unnecessarily, because the black site can cache the data when its Writer publishes it. A smart distribution protocol will prevent the last unnecessary transmission, but not the first.

We should remark that the amount of times copying a data item is prevented may not be a good measure of the speedup of the system. The time spent copying is saved, but it is only worthwhile if having multiple threads share memory is efficient. For example, in an architecture with many small processors in a cluster with no notion of shared memory, in which each thread runs on its own processor, it may be faster to have each processor store a copy in its own cache than to have each processor communicate with a central hub for every data request.

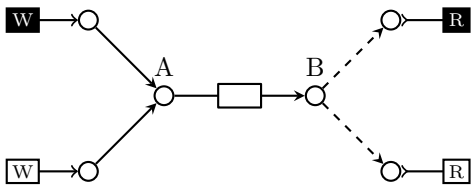


Figure 9.3: A `FIFO1` in the circuit makes it unoptimizeable with circuit transformations.

Chapter 10

Conclusions

We have proposed three modifications to the existing Reo engine:

- Distinguish between flow of data and flow of signal, so that components which do not inspect data do not receive a copy of that data.
- Distinguish between components which require their own copy of data and components which may share a common reference, so that components which do not modify data do not receive their own, personal copy.
- In distributed computing clusters, use circuit transformations and a smart distribution protocol to avoid redundant network traffic.

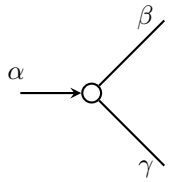
Each of these modifications can be implemented separately. We have already demonstrated that each modification reduces the amount of copies to be made of published data when compared to a naive implementation of Reo. In reality, the current Reo engine is not a naive implementation, but nevertheless each modification still improves the engine.

The last recommendation involves circuit transformations. For now, it is unclear whether the circuit transformations proposed here are useful, because the circuits they produce are large, and because it is not obvious that the definition of circuit optimality discussed here is relevant to the implementation of Reo in distributed systems. Nevertheless, circuit transformations represent two contributions to Reo. First, while it was already clear that there exist classes of semantically equivalent Reo circuits [7], circuit transformations are a tool with which to go about finding semantically equivalent circuits in a structured fashion. Second, given any Reo circuit and two nodes within that circuit between which data may flow, the transformations presented here give a method to construct a semantically equivalent circuit in which there is exactly one path that data can take between those two nodes.

Appendix A

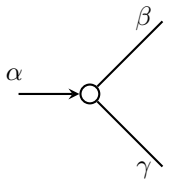
Applying the flip rule to data colours

Clarke et al. [2] introduced the *flip rule* for no-flow colours to curb the size of colouring tables. The flip rule uses the observation that, for the purposes of coordinating data flow, it is sufficient to know that a channel end has no flow, without knowing the reason for that flow. In this work, we have expanded the colour *flow* into *flow of data* (D) and *flow of signal* (S). A rule very much like the flip rule can be used to curb the size of intermediate colouring tables.



α	β	γ
D	D	D
D	S	D
D	D	S
S	S	S
O	O	O
R	R	R

(a) The `Replicator`'s colouring table in the four-colour scheme.



α	β	γ
D	D	D
S	S	S
O	O	O
R	R	R

(b) The `Replicator`'s colouring table, with the new flip rule implemented.

The `Replicator`'s colouring table, reproduced here in Figure A.1a, obeys the no promotion or demotion (NPD) principle by enumerating all the possible ways that data or signals may flow without violating the NPD principle. An alternative way to arrive at a composed colouring table is to simplify the `Replicator`'s colouring table to the one in Figure A.1b, and to make the com-

position algorithm more complex.

In composing two colourings, the composition algorithm will no longer demand that a channel end be coloured the same colour in both tables. Instead, when comparing colours of a `Source` end, it should compose the two colourings only if the NPD principle is not violated.

A.1 Example application

Consider how the two alternative approaches would compose the colouring table for the circuit in Figure A.2, supposing that the Writer performs a `put()`, `Reader 1` performs a `get(Data)` and `Reader 2` performs a `get(Signal)`.

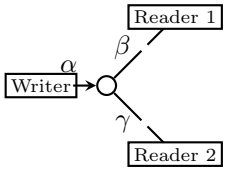


Figure A.2: A circuit with three channel ends, composed of four subcircuits: the three boundary components and the Replicator node.

To compose the circuit's colouring table, we must compose the colouring table of the `Replicator` node (containing channel ends α, β and γ), the `Writer`'s colouring table (containing channel end α) and the two `Readers`' colouring tables (containing, respectively, β and γ). In this example, we will compose the colouring table of the `Replicator` with that of `Reader 1`, then with the table of the `Writer` and lastly with the table of `Reader 2`. Figure A.3 shows the first step of these three steps in the composition process.

`Replicator`

α	β	γ
D	D	D
S	S	S
O	O	O
R	R	R

`Reader 1`

β
D
O

Intermediate table A

α	β	γ
D	D	D
O	O	O

Figure A.3: The first step of the composition process results in Intermediate table A.

In the second step, demonstrated in Figure A.4, we compose the resulting table with that of the `Writer`.

In the third step, intermediate table B is composed with the colouring table of `Reader 2`, which can only receive a signal. Intermediate table B contains no colouring that colours channel end γ S. By employing the new flip rule, we can match these colourings by observing that if the colourings were composed, the NPD principle would not be violated.

Intermediate table A

α	β	γ
D	D	D
O	O	O

`Writer`

α
D
S
R

Intermediate table B

α	β	γ
D	D	D

Figure A.4: The second step of the composition process starts from Intermediate table A, from Figure A.3, and results in Intermediate table B.

Intermediate table B

α	β	γ
D	D	D

`Writer`

α
S
O

Completed table

α	β	γ
D	D	S

Figure A.5: γ in Intermediate B can be matched with γ in the colouring table of `Reader 2`.

In total, 16 pairwise comparisons of colourings were necessary to compose the circuit's colouring table. By contrast, 24 pairwise comparisons of colourings are necessary when using the colouring table in Figure A.1a with a simple composition algorithm.

It is useful to phrase the NPD composition rule in the same language in which Clarke phrases the flip rule:

NPD composition rule The colour of a `Source` end coloured D may be replaced by the colour S, so long as the NPD principle is not violated.

Recall that a `Replicator` does not violate the NPD principle so long as either it propagates a signal, or it takes a data item on one of its `Sink` ends and outputs a data item on at least one of its `Source` ends.

The NPD composition rule generalizes to other colouring schemes, because the NPD principle extends to other schemes. In the five-colour scheme, a `Replicator` obeys the NPD principle when at least one of its outputs is of the same level in the colour hierarchy as its input, and none are higher, where the hierarchy of colours is $\& > * > S$. In the colouring scheme implemented by the engine, the hierarchy is $\& > *$.

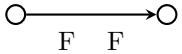
Appendix B

List of colouring tables

Following is an overview of the colouring tables of all channels Reo defines by default, in two colour, four colour, five colour and the four colour scheme adjusted for the Reo engine.

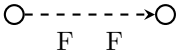
B.1 Two colour scheme

Sync



F	F
N	N

LossySync

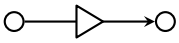


F	F
F	N
N	N

Filter

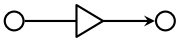
F	F
F	N
N	N

Modifying Transformer



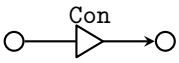
F	F
N	N

Non-modifying Transformer



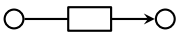
F	F
N	N

Constructor



F	F
N	N

Empty FIFO1

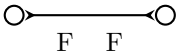


F	N
N	N

Full FIFO1

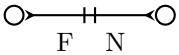
N	F
N	N

SyncDrain



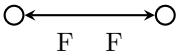
F	F
N	N

AsyncDrain



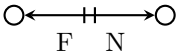
F	N
N	F

SyncSpout



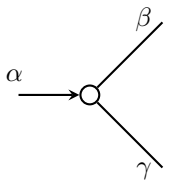
F	F
N	N

AsyncSpout

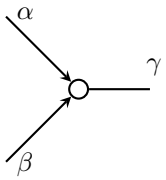


F	N
N	F

F	F
N	N



α	β	γ
F	F	F
N	N	N



α	β	γ
F	N	F
N	F	F
N	N	N

B.2 Four colour scheme

Following is a list of colouring tables that implement the data-signal modification introduced in chapter 4.

Sync

D	D
S	S
O	O
R	R

LossySync

D	D
S	S
S	R
O	O
O	R

Filter

D	D
D	S
D	O
O	O
O	R

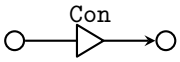
Modifying Transformer

D	D
D	S
D	O
O	O
R	R

Non-modifying Transformer

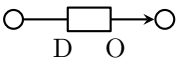
D	D
D	S
D	O
O	O
R	R

Constructor



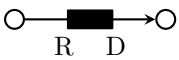
S	D
S	S
O	O
R	O

Empty FIFO1



D	O
O	O

Full FIFO1

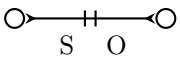


R	D
R	S
R	R

SyncDrain

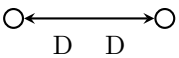
S	S
O	R
R	O

AsyncDrain



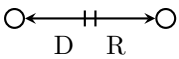
S	O
O	S
O	O

SyncSpout

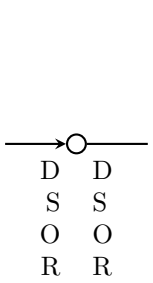


D	D
S	D
D	S
S	S
R	O
O	R

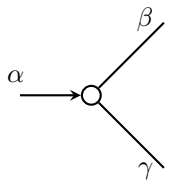
AsyncSpout



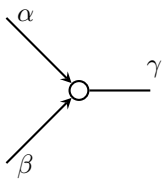
D	R
S	R
R	D
R	S
R	R



D	D
S	S
O	O
R	R



α	β	γ
D	D	D
D	D	S
D	S	D
S	S	S
R	R	R
O	O	O

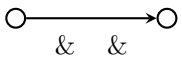


α	β	γ
D	R	D
S	R	S
R	D	D
R	S	S
R	R	R
O	O	O

B.3 Five colour scheme

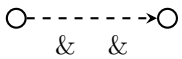
Following is a list of colouring tables that implement the data-signal modification introduced in chapter 4 and the copy-reference modification introduced in chapter 5.

Sync



&	&
*	*
S	S
O	O
R	R

LossySync

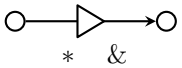


&	&
*	*
S	S
S	R
O	R

Filter

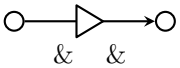
&	&
*	*
*	S
*	O
O	O
R	R

Modifying Transformer



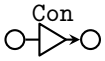
*	&
*	*
*	S
*	O
O	O
R	R

Non-modifying Transformer



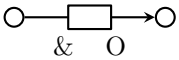
&	&
*	*
*	S
*	O
O	O
R	R

Constructor



S	&
S	*
S	S
O	O
R	R

Empty FIFO1

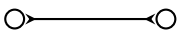


&	O
O	O

Full FIFO1

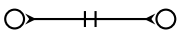
R	&
R	*
R	S
R	R

SyncDrain



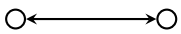
S	S
O	R
R	O

AsyncDrain



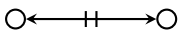
S	O
O	S
O	O

SyncSpout

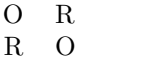


&	&
∗	&
S	&
&	∗
∗	∗
S	∗
S	S
R	O
O	R

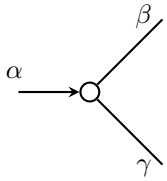
AsyncSpout



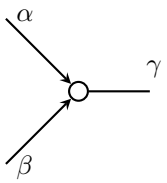
&	R
∗	R
S	R
R	D
R	S
R	R



&	&
∗	∗
O	O
R	R



α	β	γ
&	&	&
&	&	∗
&	∗	&
∗	∗	∗
R	R	R
O	O	O



α	β	γ
&	R	&
∗	R	∗
S	R	S
R	&	&
R	∗	∗
R	S	S
R	R	R
O	O	O

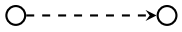
B.4 Implemented tables

In chapter 9, we learned that the Reo engine works faster if there were only four colours, rather than five. The following tables are arrived at by taking the five colour tables, replacing each S with ∗, and then removing duplicate entries.

Sync

&	&
∗	∗
O	O
R	R

LossySync

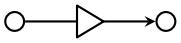


&	&
∗	∗
∗	R
O	R

Filter

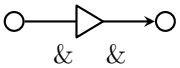
&	&
∗	∗
∗	O
O	O
R	R

Modifying Transformer



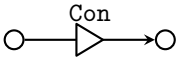
∗	&
∗	∗
∗	O
O	O
R	R

Non-modifying Transformer



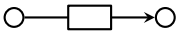
&	&
*	*
*	O
O	O
R	R

Constructor



*	&
*	*
O	O
R	R

Empty FIFO1



&	O
O	O

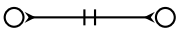
Full FIFO1

R	&
R	*
R	R

SyncDrain

S	S
O	R
R	O

AsyncDrain

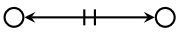


S	O
O	S
O	O

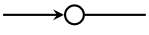
SyncSpout

&	&
*	&
&	*
*	*
R	O
O	R

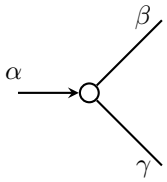
AsyncSpout



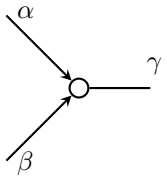
&	R
*	R
R	D
R	S
R	R



&	&
*	*
O	O
R	R



α	β	γ
&	&	&
&	&	*
&	*	&
*	*	*
R	R	R
O	O	O



α	β	γ
&	R	&
*	R	*
R	&	&
R	*	*
R	R	R
O	O	O

Bibliography

[1] Arbab, F.: Puff, the Magic Protocol (2011)

[2] Clarke, D., Costa, D., Arbab, F.: Connector Colouring I, Synchronization and Context Dependency, Proceedings of FOCLASA 2005

[3] Jongmans, S.-S.T.Q., Arbab, F.: Overview of thirty Semantic Formalisms for Reo, Scientific Annals of Computer Science, 2012

[4] Extensible Coordination Tools, *Tools - Reo Coordination Language*, `http://reo.project.cwi.nl/reo/wiki/Tools` (August 2014)

[5] Proen'ca, J.: Synchronous Coordination of Distributed Components. PhD thesis, Leiden University, 2011. `https://openaccess.leidenuniv.nl/handle/1887/17624`

[6] Proen'ca, J., Clarke, D.,de Vink, E.P., Arbab, F.: Decoupled execution of synchronous coordination models via behavioural automata. In: Mousavi, M.R., Ravara, A. (eds.) FOCLASA. EPTCS, vol. 58, pp. 65-79 (2011)

[7] Blechmann, T., Baier, C.: Checking Equivalence for Reo Networks, in ScienceDirect (2008)