Universiteit Leiden
Opleiding Informatica

Developing Efficient Concurrent C Application Programs Using Reo

Name:	Mathijs van de Nes
Date:	October, 2015
1st supervisor:	Prof. Dr. F. Arbab
Assisted by:	S.-S.T.Q. Jongmans
2nd supervisor:	Dr. M.M. Bonsangue

MASTER'S THESIS

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

Abstract

Reo is a language along with an Eclipse-based tool-set that allow a programmer to externally describe protocols, which can then be converted to a working computer program. Previously, only the code for the protocol itself could be generated, which then had to be manually connected to the rest of the program by a programmer. In this thesis an extension to the Eclipse tools is presented that allows the user to import his/her computation code and link it graphically. If done naively however, the program would not be as fast as someone would like it to be. One of the problems lies in the semantics of Reo, which dictates that a value must be copied before being passed to a component. However, in many situations it is in fact safe to simply pass a reference to some data item instead of copying its value. Both the Reo circuit and the computational code of components must be analysed to determine when it is safe to pass a reference to a data item among concurrent threads instead of making a copy of that data item for each.

We propose two analyses, one which operates on the protocol and one which analyses the components, to prevent some of these copies if it can be determined that they are not necessary. One attempt for the analysis of protocol code works by computing extended colouring tables of Reo circuits. Unfortunately this method is too slow and the overhead of using it at runtime is prohibitive. Another algorithm, which is fast enough, works by finding paths in the graph of a Reo circuit. The algorithm for analysis of components inspects the code of external programs. With the help of LLVM, it is able to determine when a data item is only read from and never written to. To combine these methods, we propose a simple scheme based on reference counting that involves virtually no extra run-time overhead and except for a few rather rare situations, is able to avoid unnecessary copies that a run-time colouring-table-based analysis can identify. Individually, but especially combined, these optimization show a good result in reducing the time spent on copying data.

Contents

Chapter 1

Introduction

Reo [Arb04, Arb11, CWI15] is graphical language to create protocols for the coordination and communication in concurrency. Rather than deep inside the program, it is designed to create an external protocol that can easily be altered or replaced, while still being tightly integrated. Next to its intuitive notation it also has a strong mathematical formalism as its semantics [JA12] which helps the compiler and user reason about the created protocol.

Creating a Reo circuit can be done in different ways. One way is to draw a circuit by hand. For this process an Integrated Development Environment (IDE) is available in the form of the Extensible Coordination Tools (ECT) [AKM+08]. There is also the possibility to convert from other languages, such as UML. Once a circuit is created, it can be used as a component in another circuit. This makes the creation of a circuit very flexible.

One of the main goals of Reo is to make it as easy as possible for a programmer to create a multi-threaded program. An area where this goal is visible is the data passing semantics of Reo. It states that every data item is exchanged through the protocol by value. This means that nothing is shared and the programmer is allowed to read and write any data without having to worry about aliasing, where modifications in one thread might affect another. No-sharing semantics implies that the programmer has no need for explicit concurrency constructs such as locks and semaphores to protect shared data. While the semantics are those of value passing, that does not mean that all data must actually be passed by value internally. The current C runtime for example uses reference counting [Boe04] to pass along all values. This has an obvious performance gain, but some care does need to be taken at the circuit-user boundaries to preserve the value-passing semantics of Reo on top of this reference-passing implementation, which is explained in more detail in Section 2.7.2.

Any Reo protocol can also be converted to actual program code. The current ECT is able to generate Java and C code. This creates an object which can be connected to other components of a program. By having the code for the protocol generated rather than written manually, a lot of potential hard-to-debug problems are avoided. It also allows for easy replacement of the protocol in the program. One click to generate it and one copy/paste action to replace the old code.

Although the ECT currently can create all ingredients for a concurrent program, it is not yet able to connect them. In Chapter 2 we describe an extension to the ECT to be able to automatically link externally written component code to Reo connectors. This entails parsing external code and generating code that interfaces with it. We have also added facilities for a user to experiment with the protocol. Instead of connecting a component to the connector, a console can be attached such that the programmer can interact with the protocol circuit directly.

Connecting custom components to a circuit can be very inefficient when done naively. The main culprit here is the value passing semantics. Without any extra knowledge, a copy of the data has to be made for at least every put and get call. To optimize the program we propose two types of analysis which help the runtime to avoid unnecessary copies. The first, described in Chapter 3, is done on the Reo diagram. For each port we check whether data items are used, or could be ignored.

Chapter 4 is about the other analysis, which takes place on the get/put call sites. Even without analysis, we can allow the programmer to specify the intended destiny of every data item exchanged through every get/put operation. For example, when a program only reads from the data, we can just provide a reference to an immutable copy of that data to the user instead of a full-blown copy. But manual methods are unreliable and error-prone. To help the programmer, and to ensure he/she does not introduce any errors, we provide an analysis framework to check the intended destiny of every data item. Our tool goes over the code and determines whether a data item might be read from or written to after it is exchanged, and takes action accordingly.

To validate the new methods, we run some experiments in Chapter 5. We first show the ease of use of the new ECT extensions by demonstrating its ability to create and alter protocols easily. The second experiment demonstrates how the various optimizations affect the execution time of the generated program. Lastly, we demonstrate what the second analyser is able to calculate and change in a piece of code.

Chapter 2

Generating a Complete Program

Reo aims to make the writing of multi-threaded applications as easy as it can be for a programmer. Previously though, the IDE for Reo, the ECT, was able to generate code only for the protocol. The other components of a program had to be manually connected to this generated code by the programmer. Some set-up code needs to be invoked and threads need to be started in order to interact with the ports of a protocol. The goal of the ECT extensions we describe in this chapter is to automatically generate all these parts, such that programs can be created, compiled and run without requiring programmers to write any additional connector code.

This chapter introduces an upgrade to the ECT that implements these ideas. This upgrade allows a user to parse and connect functions from an existing code base to a Reo circuit. The ECT then allows the user to generate the protocol and wrapper code such that, after compilation, they produce a fully functioning executable. Section 2.1 shows the high-level steps needed to be taken by the user to construct a program. The subsequent sections contain some details involved in this process. Section 2.4 describes which functions may be connected to a Reo circuit, while Section 2.5 describes how they are actually read and parsed. Section 2.2 shows an example of the output code. The details of the code are explained in Section 2.6.

As described in Section 2.7.3 our ECT upgrade also substitutes a console for a component for any boundary node of the Reo circuit that remains unbound to a port of any component.

2.1 How to Generate C Code Using the ECT

Suppose we want to create a full concurrent application that consists of custom computation code provided as C functions, and a protocol expressed as a Reo circuit, using the ECT. Our program features a producer and a consumer that exchange data, after a signal from a clock has been send. For this, we have created a step-by-step guide in Figure 2.1. In this section we will refer to the red numbers in this figure as *Steps*.

The first step is to draw a Reo connector. The protocol in Step 1 is a good implementation of what we try to accomplish. It consists of two sync channels and one sync-drain. The semantics of this diagram ensures that all ports fire at the same time, at which data is transferred from A to B.

The next step is writing the components in an external language. Currently, the ECT supports only the C language, which we thus use. For the producer, called *sender*, we create a function that accepts three arguments: an output port, plus a `string` that specifies the data item to send, and `max_iters` that specifies how many times the producer produces the data item. The consumer, called *printer*, accepts two arguments: an input port and `timeout`, the time it waits in

milliseconds after the last transmission before it stops accepting further data. Finally, we create a clock called *signaller* with an output port and `max_iters` as parameters. The full implementation of these components is listed in Section A.1 of the appendix.

With the code of the components ready, we may now add them to the diagram. The first action is to right click on the diagram and select *Add Source Code* (Step 2). In Step 3 we select either the header file or the source file of our code. The user then specifies which of the eligible functions to add to the diagram in Step 4. Once selected, a user must select the object file of the code (the file that contains the compiled version of the C function) as depicted in Step 5. Completing this step will generate elements in the diagram for every selected function. These will consist of yellow boxes as explained in Section 2.3.

We are now able to connect the components to the connector. The producer to port A, the consumer to B and the clock to C. This is done by drawing a *link* between some square of a component and some circle of the connector, see Step 6. Besides drawing connections, we also fill in values for the non-port arguments of the components (Step 7). In the example we have filled in all values except the `string` argument of the producer. The `timeout` and `max_iters` will be constants in the program. Leaving the value of the `string` parameter unspecified makes this parameter dynamic: its value and must be specified in a command-line parameter as `./program --sender_string <value>`.

When everything is filled-in and connected, the executable code for the application can be generated. To do this, first we select what we want to generate at Step 8, in our case a C program. The other options are for generating Java code, but currently this option produces only the protocol code and does not connect the components. Next, we choose a destination folder in Step 9. Finally, by clicking on the generate button in Step 10,the ECT generates the full executable code for our application program.

2.2 Example of Generated Code

We run the protocol and components of Figure 2.1 through the ECT and its code generator to produce an actual program. All code is listed in Appendix A. The components are listed in Section A.1. Sections A.2–A.5 list the code that implements the protocol and connects it to the components. The Makefile can then be used to compile the program. Only the components are manually written, the rest of the files are generated by the ECT.

2.3 ECT Interface

Here we describe the new interface elements that are added to the ECT and how to interact with them.

We represent a source code component by a yellow box, as seen in Step 7 of Figure 2.1. By default, each input port is a white square on the left of the box and output ports are represented by gray squares on the right of the component. The user is allowed to move a port to anywhere on the edge of the connector as to enhance the readability of a circuit. Other types of parameters are visible as a list inside the yellow box.

Connections from a component to a connector can be made by drawing a *link* between a port and a node. A link is a gray line such as the one in Step 6. Sink nodes, i.e. nodes having only incoming edges, may be connected to input ports while source nodes, i.e. nodes having only outgoing edges, may be connected to output ports. This ensures that a link always has one side that produces data and one side that accepts data. In fact, a link binds a boundary node of a Reo circuit as an actual parameter to a port parameter of a component. Not all boundary nodes

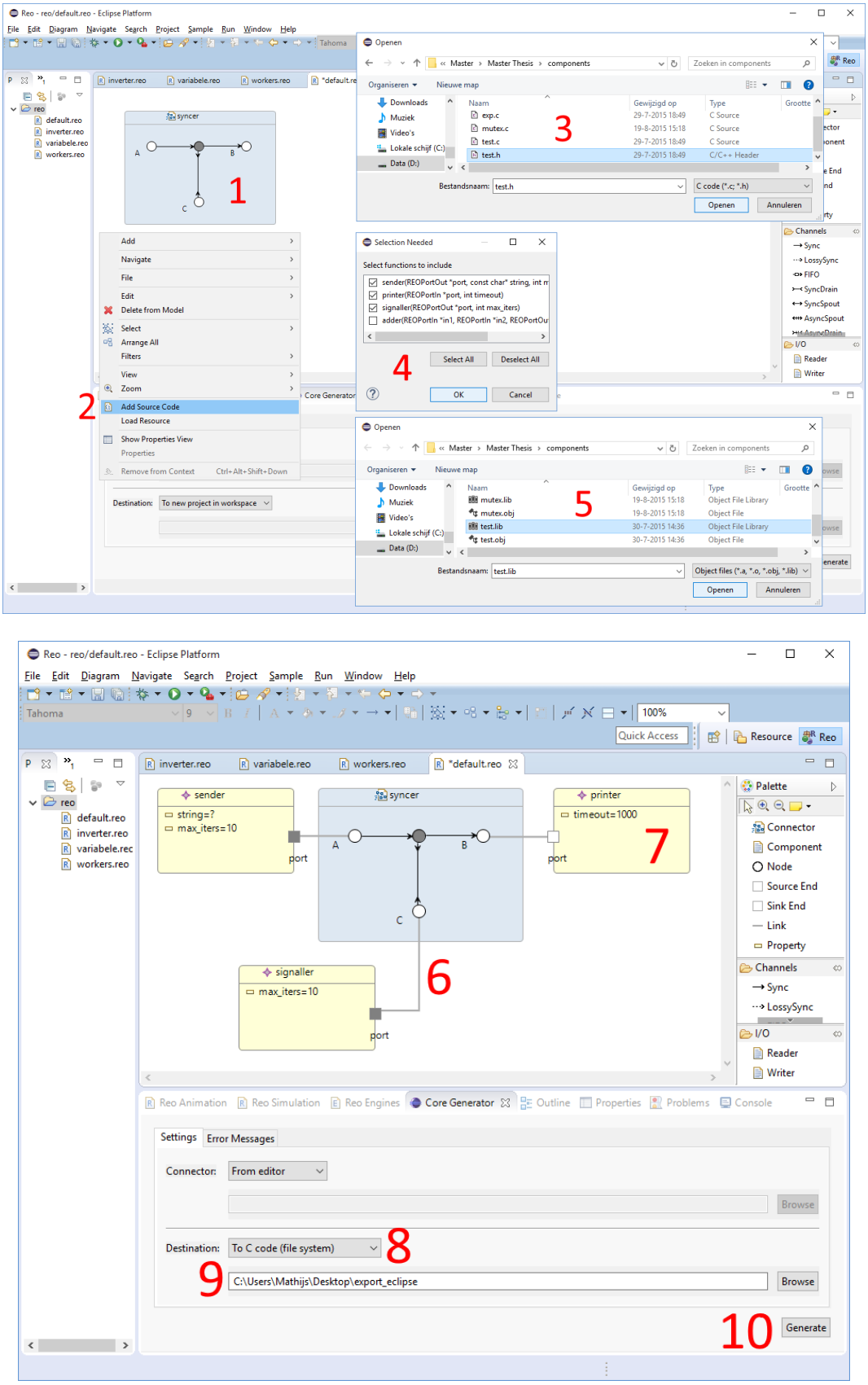


Figure 2.1: Adding Components

in a protocol have to be connected to be able to compile a program. Every boundary node of a Reo circuit that is not linked to a component will then be connected to the console, as specified in Section 2.7.3. In contrast, every port of the components *does* need to be connected to a port on the circuit. This is only a superficial limitation however, as you can always add a small circuit consisting of a single sync-channel and connect the component port to one of the endpoints. The other circuit port then has the console attached, as if it was attached directly to the component.

The non-port parameters are left unset by default, displayed as `parameter=?`. They can be modified in the properties tab in the bottom of the ECT. This tab allows a user to inspect the type of the argument and to set a new value. If set, the argument will be a constant within the program. When left unset, a parameter will be added as a command-line argument for the program. It can then act as a tuning parameter for the program, which must be specified at the start of an execution.

2.4 Function Eligibility

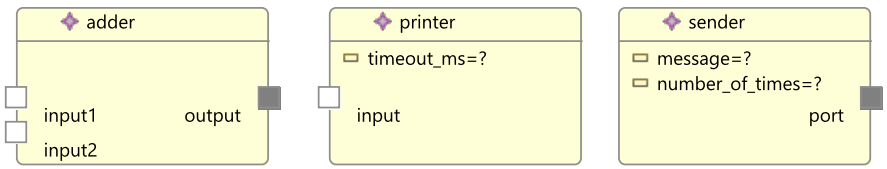
A component may be constructed from a C function declaration or specification.[1] Such a component simply runs that function as the body of an independent thread. However, not every function qualifies as the body of a component. With the help of the signature of a function we can determine its eligibility for use as a Reo component. For a function to be eligible it must meet the following requirements:

1. Have a return type of `void`
2. Must not be a variadic function[2]
3. Have zero or more parameters of the following types:
   - input port: `REOPortIn*`
   - output port: `REOPortOut*`
   - integer: `int`
   - floating point number: `float` or `double`
   - string: `char*`

Example 2.2. The following functions are elligible:

- `void adder(REOPortIn* input1, REOPortIn* input2, REOPortOut* output);`
- `void printer(REOPortIn* input, int timeout_ms);`
- `void sender(REOPortOut* port, char* message, int number_of_times);`

When selected these functions are displayed by the ECT as the following diagrams:



The following functions are *not* elligible:

[1] A declaration consists of the prototype of a function, mostly found in header files. The specification of a function also includes the body.

[2] A function which does not have an upper bound on the number of arguments it accepts, e.g. `printf`

- `int adder(REOPortIn* input1, REOPortIn* input2, REOPortOut* output);`
  This function has a non-void return value
- `void printer(REOPortIn* input, struct my_custom_data data);`
  This function has an unknown data type in one of its arguments
- `void sender(REOPortOut* port, int number_of_messages, ...);`
  This function accepts a variable number of arguments

Although it does not really make sense, it is currently allowed to add a component that has no port argument. Such a component will actually run as a separate thread, but because it has no ports, it cannot participate in the protocol by linking to its Reo circuit.

Requirement 1 ensures the component does not impose a special meaning to the return type of the function. The circuit always discards the return value, so a value of `void` is preferable. Requirement 2 is needed since the interface and the code generator currently are not able to handle a varying number of arguments. The use case for variadic functions is thought to be limited, and adding support would increase the complexity of the tool considerably.

The types in Requirement 3 consist of both port and other types. The port types are used to connect the component to a connector. Any finite number of input and output ports may be specified. The other arguments are used to either tune a components behaviour to be suitable for a specific connector, or to add a command-line option to the resulting program that enables us to alter a parameter without recompilation. The supported primitive types provide a proper base for creating programs. In principle, support for user-defined and complex structured types can be added in future extensions.

2.5 Parsing Source Code for Component Bodies

Parsing a C file or a C header file is done with the help of the Eclipse C/C++ Development Tooling (CDT) package in a manner described by [PJSS12]. This package includes a parser that allows us to identify function declarations. It also gives us detailed information about the types used in the arguments and return values. The CDT is able to parse a correct C source or header file. The result of this parsing is an Abstract Syntax Tree (AST), representing the structure of the code. To extract information from the AST we visit all *names* and *declarators* that it contains.

A *declarator* is a piece of code that declares the existence a variable or function. It does not contain the type information. For each declarator we come across we check if it is a function declarator. If it is, we then extract the raw signature from the declarator and store it. A statement `void foo(int bar);` will return a raw signature of `foo(int bar)` with the name `foo`. This information can be used to declare and bind to the implementation of the function in the component.

A *name* is a semantic object present in the program. This includes virtually all variables and other declarations, although we are only interested in a name belonging to a function declaration and skip over all others. When a name is visited all of its type information becomes available. We extract all information regarding return type and arguments of a function. If the function is eligible (see Section 2.4) we then add it to the set of known functions.

Combining the raw signature and type information gives us an object that we can convert to a component to be used in the ECT.

2.6 Generated Program

The code of a generated program consists of several parts. One part consists of the libraries which are copied without any modification. Currently this is only the *console*, which is described in Section 2.7.3.

The connectors represent another part. A header and source file is generated per connector. A connector x is represented by a structure `Connector_`x which contains the `REOAutomaton`, `REOFifo` and `REOPort` structures that make up the connector. Furthermore, there are two functions provided. The function `build_`x creates and initializes the connector structure and returns a reference to it. This connector struct can then be used in the running of the program. When the program is finished, the reference is passed to the function `clean_`x to dispose the structure.

The file `main.c` contains multiple things. Most notable, it contains the references to the functions that implement the components. It also contains the glue between the components and connectors, and the initialization code. Each component has its raw signature inserted to be able to call the implementation present in the external library. There is also some wrapper code added to be able to call such a function via POSIX threads (pthreads). If everything is connected and set, *main* will simply construct the connectors, start a thread for each component and let them run. If one of the non-port variables was left unset, it will also first check for the options on the command-line. If a boundary node in a connector is not linked to a port of a component, *main* spawns a console and links it to that node. The user, thus, will play the role of the missing component at run time through this console.

2.7 Runtime

2.7.1 Connector Runtime

To enable the generated structures of Section 2.6 to perform their actions, a runtime library is provided called the Reo C Runtime (Reo CRT). This C library was developed in [vdN15]. Structures exist for an *automaton*, which consists of *states* which have *transitions* between them. A transition is taken when an activation and a data constraint are met. The activation constraint is given by a set of ports that must be ready. The data constraint is a relationship between the data items that the ports submit or request. Checking the activation is done by looking at a *context*, which is a bitfield signifying which ports are ready. The data constraint is checked and solved by a constraint solver. This constraint solver is currently implemented by first determining the domain, and then using a Depth-First-Search algorithm to check all assignments.

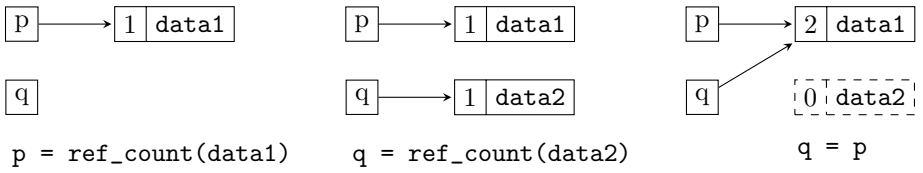
Most of the actions that happen behind the scenes consist of locking and unlocking mutexes, waiting on other components and checking if there is a transition to fire. Each step ensures that no data races or other incorrect behaviour will occur, and thus the automaton will behave as to its specifications.

2.7.2 Reference Counting

One of the optimizations already present in the runtime is the use of reference counting [Boe04]. Instead of passing data objects around by value, the Reo runtime uses references. Every data item is augmented with a counter to keep track of how many pointers reference that value. This is called Reference Counting.

Example 2.3. Consider the figure below. Here we have two pointers, `p` and `q`. We first construct a new object and let `p` reference it. The number on the item is the reference count, showing that it currently has a single reference. We repeat this for `q`. If we then assign the reference of `p` to

q, the reference counts will be updated. Since the count of the second item has become zero, it has been erased from memory.



Using references helps the runtime handle the data exchanged among ports. Generally, primitives in a Reo circuit do not "process" the data items that they transport. Thus, most replications prescribed by a Reo circuit can be avoided by passing references to the same data items, instead of making copies of them. However, with multiple references, it is no longer obvious when an object can be disposed of. This is why the count has been added to the reference. When a count reaches 0, its object has no longer any reference to it and can be freed from the memory.

Reference counting does come at a cost. Each data item will have a count added, which increases its memory usage. The other cost is the added time needed for altering the count variable. Each time a reference-counted item is passed to another function or used in an assignment, the count needs to be altered. Since we use them in a multi-threaded context, this must be an atomic operation as well to avoid data races. Atomic operations are more costly than non-atomic operations on most architectures.

When an object is exposed to the outside, it normally has to be converted to a value. If it is passed in the form of a reference, the runtime thus has to ensure that this reference is unique. Normally a copy will be made of the object to accomplish this. However, when the reference count of an item is 1, the holder of this reference is actually the owner of the object. This means that instead of copying it, it may be passed as-is to the port. This side-effect of reference counting is a powerful thing. If a port releases the ownership of some data to the protocol, it can then be passed on to another port without having to make a copy. Further details on this scheme are in Section 3.4.

2.7.3 Console

It is possible to leave some ports of a connector unconnected. In this case, the *main* function will attach these ports to a console object.

Interface

The console is an additional tool to allow a user inspect the protocols behaviour by performing the get/put actions him/herself. The console mimics multiple terminals, each being connected to one of the unconnected boundary nodes of a circuit.

To provide this functionality, a console object communicates on the background with the actual program. Once started, it will receive the ports that are available, and will launch the windows through which the user can interact with the ports, as in Figure 2.4. Data send or received by each port will be displayed in its own window. Any other messages received will be displayed in the main console.

A program can be started either by specifying its name and arguments as command-line parameters to the wrapper or by selecting the program and filling in the arguments in a dialog upon opening the wrapper.

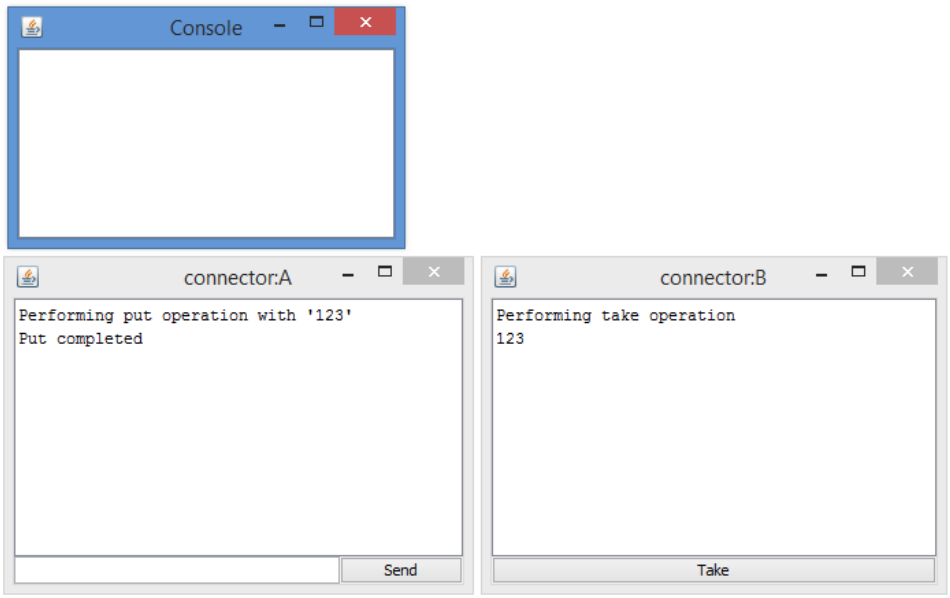


Figure 2.4: Console connected to two nodes

Textual Implementation

In the background, the graphical interface communicates with the actual program via its standard-in and standard-out. The protocol for communication is very simple. First, the program announces which ports are attached in the form of `[CONSOLE]*port <connector name>:<port name> <in|out>`. This is helpful to identify which actions can be taken. The user can than do a get/put operation on a port via `[CONSOLE]:<connector name>:<port name> <data>`. When performing a get operation, the data argument may be omitted. When a port receives some data or a put has been completed, the console prints information about it in the following form: `[CONSOLE]:<connector name>:<port name> <information>`.

Due to the textual nature of the console, it will always send string data to a node. Whilst adding support of other data types should not be a problem, it is currently not desired to add the complexity of handling this. Connectors themselves are data agnostic and user implemented components can convert a string to another data type on their own.

The interface for exchanging data is verbose on purpose. It makes it very easy for the graphical interface to recognize and send commands. It will automatically add or remove the specified prefixes to/from a message, so the right content will be delivered to the correct port or window.

Example 2.5. This is an example of the textual console. The data sent to the program, typed by the user, is prepended with `>` while the data received is displayed as-is.

```
  [CONSOLE]*port connector:A in
  [CONSOLE]*port connector:B out
> [CONSOLE]:connector:A 123
  [CONSOLE]:connector:A Performing put operation with '123'
> [CONSOLE]:connector:B
  [CONSOLE]:connector:B Performing take operation
```

```
[CONSOLE]:connector:B 123
[CONSOLE]:connector:A Put completed
```

This communication is the same one as that shown in Figure 2.4. In this case the user first submitted '123' to `connector:A` and then pressed the *Take* button of `connector:B`.

Chapter 3

Optimization of Data Flow

Constraints in a Reo circuit have both a synchronization and a data part. The semantics of Reo specify that each item is passed by value. This means that any data item passed around should be allowed to be inspected and modified without affecting any other part of the program. For small data items such as integers this is no problem since a copy operation on them is cheap. For larger objects, such as strings or large binary data, a copy operation costs additional memory and execution time for the data allocation and transfer.

Instead of value passing, a channel could also pass a reference to a data item. A reference is a handle to a shared data item that gives access to that data item. If multiple owners have references to the same data item, however, none of them may modify that data item. Any modification by one reference holder would be visible to other reference holders which is not allowed according to the value passing semantics of Reo. The advantage of reference passing is that it is a lot cheaper than making (deep) copies, as the actual transferred item is just a single pointer.

A third possibility is to send only a signal. This means that only the readiness of an action (get or put) on a port will be sent through to the connector, without the actual data. A signal signifies the availability of a data item, rather than the data item or its value. In practice a signal will take the same amount of space as a reference. The advantage over a reference is that a lot of checks can be made simpler and faster, such as checks for equality between data items. Another advantage is that a data item does not need to be copied from its port of origin.

Signals can only be used when a data item is never read from. Some channels which are only used for synchronization can guarantee this. One example is the sync-drain channel. For some Reo circuits there may be ports where any data item put in is destined to only end up in such channels, not being visible to an outside observer.

In the current implementation of the Reo runtime, all data is passed internally as a reference since no channel will internally modify any data. See also Section 2.7.2. This means that in the internal circuit there is no distinction between a value and a reference.

In this chapter we explore a method to determine when it is sufficient to only send a signal in a put or get, or when a data item in the form of a reference is needed. Section 3.1 is an extension to an existing theoretical framework which helps us determine when a signal is sufficient. Its subsections note some difficulties and interesting cases that arose when constructing the theory. How the theory is implemented and used to obtain some interesting results is depicted in Section 3.1.3.

Unfortunately this method does have some problems. Section 3.2 shows a faster method to determine the data flow of a network. We then compare these two methods in Section 3.3.

The reference counting scheme also lends it to do more optimizations based on the intended

usage of a data item. Section 3.4 gives a detailed overview of all the possibilities. Section 3.5 then gives a quick overview of all the methods and lists which combination is the best option to be implemented in the ECT.

3.1 Colouring Tables

Colouring schemes have previously been used to determine whether data flow is possible in a connector [CCA07]. A colouring is an assignment from a set of colours $\mathcal{C}$ to the set of nodes N of a circuit. A colouring table for a circuit has rows for each colouring that is possible in that circuit. Each channel in Reo has a colouring table of its own. To create the colouring table of a circuit, we join the tables of its channels with each other until we have all nodes in the circuit. The joining process is done by calculating the Cartesian product of the two current tables, and then removing the rows in which one node has two different colours.

The three-colour scheme of [CCA07] is the scheme most used. It provides three colours, '——', '–◁–' and '–▷–'. The first signifies that a data item may flow through the channel. The other two are for the case when data flow is not possible. The arrow in this case points in the direction of propagation, away from the reason that inhibits the flow. A possible reason might be that a port does not offer a data item, or that a FIFO is full and is not able to accept any new data. The actual colour depends on the position of the arrow with respect to the node it is connected to. The colour ◯–◁– is the same as –▷–◯, but the opposite of –◁–◯. Combining colours on a node may be done by letting them point towards each other, i.e. –▷–◯–◁–, or in the same direction: –▷–◯–▷– or –◁–◯–◁–.

Colouring tables can become rather large, even for a single channel. To counter this, the *flip-rule* was invented by [CCA07]. It states that if there is a row with –▷–◯, then any other row with the same colouring except for a –◁–◯ on that location is redundant and may be removed. This decision is easy to justify in view of the rules for combining these no-flow colours.

3.1.1 Colouring Extension

Apart from the distinction between flow and no-flow, it makes sense to try to also use them to make a distinction between data flow and signal flow. In fact, Vinkhuijzen proposes several extensions that try to do exactly this. [Vin14]

We propose a colouring scheme inspired by the four-colour scheme of Vinkhuijzen. It uses the same symbols as in the original three-colour scheme, with the addition of a new colour S. This colour represents signal flow, i.e. flow with no data. The colour '——', previously representing all flow, will now mean flow *with data*.

The article of Vinkhuijzen uses an O colour to represent that 'a channel/node offered no data' and an R colour to represent that 'a channel/node refused to accept data'. In the original three-colour scheme the direction of the arrow with respect to a node differentiates the two no-flow colours. Because of the historical use of the –◁– and –▷–, and because we think they are a little more clear in their meaning, we revert back to using that notation.

To combine colouring tables with our new scheme, we propose an additional matching rule. In addition to all the normal cases, we also allow incoming data flow to be matched to outgoing signal flow. Analogous to the original flip-rule, this produces an additional *signal/data flip-rule*. For any row which has a data flow colour on an outgoing edge e, any other row with the same colours except for a signal flow colour on e is redundant and can be removed.

The tables for the new scheme can be seen in Figure 3.1. The entries assume that both flip-rules are implemented, so the actual tables are somewhat larger. The differences between our

scheme and that of Vinkhuijzen is the filter, and the reduced size of the tables caused by the new flip-rule.

While constructing the tables, some issues arose with the filter and the FIFO. Thoughts on these channels are detailed in the following section.

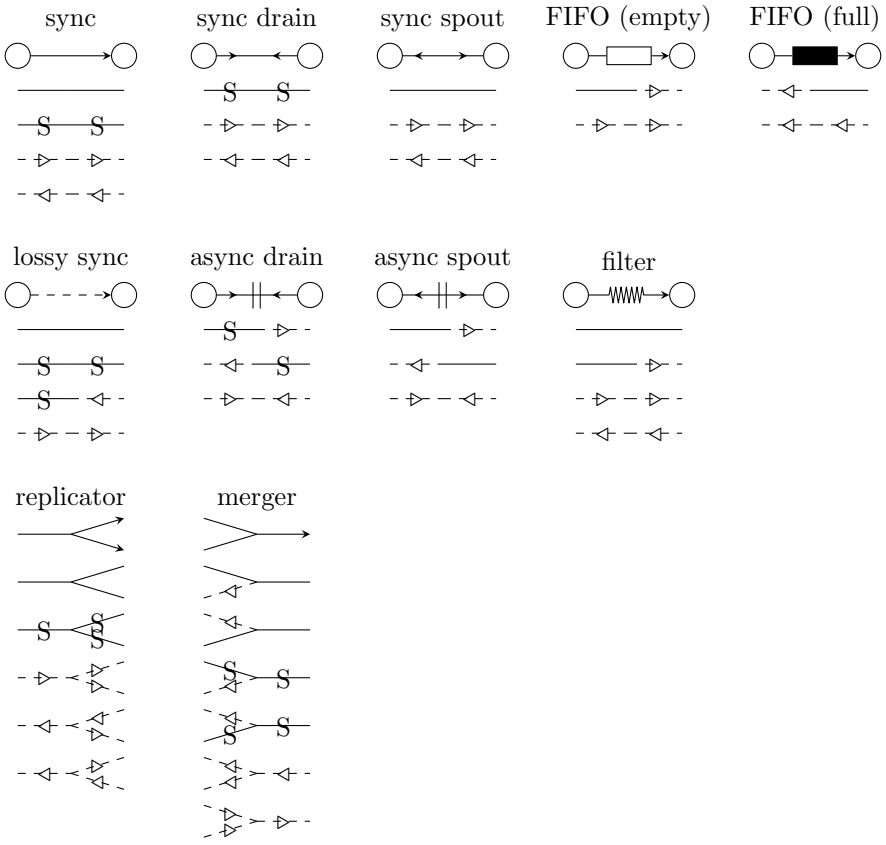


Figure 3.1: New four colour scheme, using the flip-rule

3.1.2 Difficulties

Colouring of a Filter

While the colouring tables for most channels where easily derived from their three-colour counterparts, the filter proved to be not as trivial as initially thought. This is caused by the data-dependent behaviour of this channel.

The filter with predicate P and input data $d(A)$ is defined to act like a synchronous channel when $P(d(A))$ is true. When $P(d(A))$ is false, it accepts the data item but loses it regardless of whether the other end can accept the data item or not.

The filter's –▷—▷– row is the easiest to explain. If there is no data offered, none can be received. If data is received, it can either be passed if the predicate holds, or lost if it does not. The rows —— and ——▷– account for this.

The last row left, –◁—◁–, is the most controversial one. It states that if there is no taker on the sink end of a filter, then the source end refuses to accept data. The case for not adding it is made by the fact that then a data item which does not match the predicate may be blocked even though the channel should accept and lose it. It may be the case however that the data item does match the predicate, and as such the channel not being able to send data should be blocking the receiving side. Because the colouring scheme is inherently data-agnostic, it is not possible to express the necessity for the filter to discriminate between these two behaviour alternatives based on the value of the data item at its source end. The discussion here is a matter of soundness versus completeness. If we add this colouring, we "over-approximate" the actual behaviour of the filter and we may get colourings for a circuit that would not occur, i.e. unsoundness. If we omit this colouring, we "under-approximate" the actual behaviour of the filter and we may lose colourings that otherwise would have occurred, i.e. incompleteness.

A similar problem arose with the two colour scheme and the lossy-sync. Completeness was preferred here, in order to not lose any behaviour. For the same reason, we choose completeness over soundness in the case of filter and include this colouring in its table. Inclusion assures that the answer of the analysis will never cause incorrect behaviour. One method for getting both soundness and completeness can be found in [JA11], but is not implemented since this would mean an entire reimplementation of the colouring system.

FIFO Containing a Signal

The colouring of a FIFO is currently split into two states, one for empty and one for full. The empty state needs a data item, while the full state may send its data item either as data or as a signal. This scheme might not be optimal however.

Consider the Reo circuit in Figure 3.2. In this case, whatever comes out of the FIFO would always be discarded and just used as a signal. In the optimal case this would propagate to the input of the FIFO, meaning that this FIFO would only accept a signal as its input. The first intuition to fix this is to add a state, indicating the FIFO is filled with a signal rather than actual data. A proposed table for FIFO can be seen in Figure 3.3.

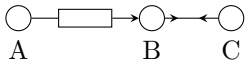


Figure 3.2: FIFO and sync-drain

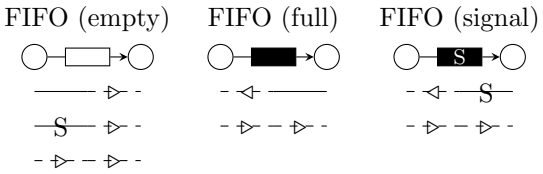


Figure 3.3: Alternative three-state colouring of a FIFO

This proposal seems to fix the problem, but unfortunately it does not work just using these colouring tables.

If combining colouring tables right-to-left, it is obvious that only the signal state can be connected to the sync-drain. The only input entry that corresponds to this has the signal colour, so so far so good. However, we are not prevented from also adding the data flow colour to the

combined table. Emptying of a FIFO is not (and should not be) guaranteed. This means that although the FIFO cannot send a data item according to the table, it will be able to accept one. With this construction we thus are not able to indicate that the FIFO only accepts a signal. Some other mechanism, such as a special 'signal-only' FIFO, is needed.

3.1.3 Using the Colouring Scheme

The goal of the analysis is to determine whether it is the case that a signal is always sufficient, or that a data item might be needed. Using the colouring scheme proposed in the previous section this analysis is fairly straightforward. For each channel we pick the corresponding colouring table. We then apply the flip-rule to get the full table of each channel. The method for obtaining the combined table for two channels goes the same way as in the three colours scheme. Repeating this we end up with a table with rows of possible colourings for each channel and node.

For each state of the circuit a table is computed. After this is done, we inspect its rows as in Algorithm 1. For each table we group the rows together by their set of nodes that have flow. The algorithm then checks these groups for nodes that always need data. These nodes are then added to the list of nodes that may need data.

Algorithm 1: Four Colour analysis

Input: A connector C
Result: The set of boundary nodes that may require data

```
1  Algorithm
2      Let S be the set of colouring tables of C. Each table belongs to a distinct state.
3      Let N ← ∅.
4      foreach Table T of S do
5          Let FS be a collection which maps a set of primitives to another set of primitives.
6          foreach row r of T do
7              Let f be the nodes that are coloured with Data or Signal in r.
8              if f is not in the domain of FS then set FS[f] ← ∅
9              Let s be the nodes that are coloured with Signal in r.
10             FS[f] ← FS[f] ∪ s
11         end
12         foreach f in the domain of FS. do
13             foreach Node n of f do
14                 if n ∉ FS[f] then
15                     N ← N ∪ {n}
16                 end
17             end
18         end
19     end
20     return N
21 end
```

Example 3.4. Consider the Reo circuit in Figure 3.5. In Table 3.6 the colouring tables are drawn for each channel, as well as that for the combined circuit. The last column is the same as the second-to-last one, but with the flip-rule implemented. The bottom three rows of the 'Combined' column can be derived from the other two no-flow colourings.

There are two rows in which there is flow. Both rows have the same set of nodes that experience flow (i.e. all of them). For both nodes A and B, there is a row in which they are *not* marked as Data. Thus we can conclude that both nodes need only to accept a signal, and may discard all data.

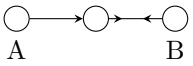


Figure 3.5: A small Reo circuit

sync	sync-drain	combined	combined, flip-rule
—	S S	S S	S S
S S	▷ ▷	S S S S	S S S S
▷ ▷	◁ ◁	▷ ▷ ▷ ▷	▷ ▷ ▷ ▷
◁ ◁	▷ ◁	◁ ◁ ◁ ◁	◁ ◁ ◁ ◁
▷ ◁		▷ ◁ ◁ ◁	
		▷ ▷ ▷ ◁	
		▷ ▷ ◁ ◁	

Table 3.6: Colouring tables when connecting a sync and sync-drain

Example 3.7. Consider the protocol in Figure 3.8. The asynchronous drain between B and C and the synchronous channels from A to B and C cause node A never to accept data. The lossy-sync channel is thus never able to send data and always loses. A losing lossy-sync does not need any data, and as such *in* should not either.

The analysis done by Algorithm 1 will correctly mark *in* as a Signal node as the data will never be visible to the outside.

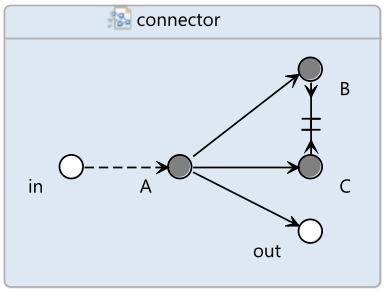


Figure 3.8: Protocol where the lossy-sync will always lose data. *in* only has to send a signal.

3.1.4 Algorithm Characteristics

The algorithm does most of its work in combining colouring tables. Unfortunately, with the current techniques, this is an exponential operation. Combining one table with n rows with one containing m rows may lead to a combined table of $n \times m$ rows. Applying the flip-rules to intermediate results may remove some of these, but in the worst case all entries could be needed for the analysis. In addition, some channels may have a state which further increases the amount of tables. For each FIFO in the network, all combinations of empty and filled may be explored.

This exponential blow-up is not new. It is already a problem when doing other types of analysis which use colouring schemes. The addition of a new colour however does increase the number of table entries. Although there are small protocols for which this algorithm runs fine, it fails to run within reasonable time and memory for large networks.

3.2 Flow Analysis via Path Finding

As an alternative to using a colouring scheme, we can also do a flow analysis by looking for a path between a source node and a data consumer. When such a path exists it may also be taken by a data item. If such a path does not exist, we know for sure that any data item entering this source node will not be inspected. Such nodes can thus be marked as only needing a signal.

The existence of a path between nodes can be done with a variety of search algorithms. For graphs with a relatively small number of nodes, a depth-first search is both simple to implement and efficient. To determine which boundary nodes in a Reo connector require data, we propose Algorithm 2. It starts by initialising $S = \emptyset$ as the list of nodes that the algorithm has visited and $D = \emptyset$ for the nodes that need data. It then calls the AnalyseNode function on each boundary node. This function first checks if we have already processed the node, to avoid infinite loops. It then checks if the node is a sink node. As sink nodes are generally connected to external components, we mark them as needing data. When a node is not a sink node, it will have one or more outgoing channels. The algorithm checks if any of these channels might need data. A channel needs data either when it is data-aware, or when it is connected to another node that needs data. The information about the latter is obtained by doing a recursive call into AnalayseNode.

3.2.1 Algorithm Characteristics

Although the algorithm is recursive, it is easy to see that it will always terminate. It keeps a list of nodes visited, from which no node is ever removed. For a protocol with n nodes, the full body of AnalyseNode will be at most n times executed. In case of a linear protocol of n nodes with $n - 1$ sync-channels, this limit will also be reached. Thus the asymptotic behaviour of this algorithm is $\mathcal{O}(n)$.

3.2.2 Examples

Example 3.9. Consider the Reo circuit of Figure 3.2. The algorithm will analyse this circuit as follows:

- We start at boundary node A
- $S = \emptyset, D = \emptyset, n = A$: Mark A as visited. The only connected channel is a FIFO. The FIFO is not data-aware, but is connected to node B.
- $S = \{A\}, D = \emptyset, n = B$: Mark B as visited. The only outgoing channel is a sync-drain. This channel is not data-aware, and also has no outgoing edges. B does not need data.
- $S = \{A, B\}, D = \emptyset, n = A$: B does not need data, so A does not either.
- Analysis of A is finished. Start analyses of node C.
- $S = \{A, B\}, D = \emptyset, n = C$: Mark C as visited. The only outgoing channel is a sync-drain. This channel is not data-aware, and also has no outgoing edges. C does not need data.
- $S = \{A, B, C\}, D = \emptyset$: Analysis finished. Both boundary nodes A and C do not need data and can accept signals.

The algorithm has correctly determined that this protocol is for signalling only. The actual data put into the circuit is always lost somewhere without being inspected.

Algorithm 2: Path Finding

Input: A connector C
Result: The set of boundary nodes that may require data

```
1  Algorithm
2      Let S ← ∅  // Set of nodes visited
3      Let D ← ∅  // Set of nodes that may need data
4      foreach Boundary node n of C do
5          AnalyseNode(n)
6      end
7      return D
8  end
10
11 Function AnalyseNode(n)
12     if n ∈ S then return true iff n ∈ D
13     S ← S ∪ {n}
14     if n is a sink node then
15         D ← D ∪ {n}
16         return true
17     end
18     foreach outgoing channel c of n do
19         if c is data-aware then
20             D ← D ∪ {n}
21             return true
22         end
           /* The channel has at least one source edge, so also at most one sink edge  */
23         foreach sink edge e of c do
24             Let n' be the node e is connected to.
25             if AnalyseNode(n') returns true then
26                 D ← D ∪ {n}
27                 return true
28             end
29         end
30     end
31     return false
32 end
```

Example 3.10. Consider the protocol in Figure 3.8. This protocol has a path from *in* to *out* via *A*. The analysis will thus determine that node *in* may need data. This is not true however. No data can ever flow to *out*, as already stated in Example 3.7. Although the colouring analysis can determine this, our path finding does not.

It is not a problem that the analysis marks *in* as needing data. Although it will make the protocol slightly less efficient, it does not cause any incorrect behaviour. Situations like this example are also mostly artificial, and will be rare in protocols that will be used in production. As these situations occur only when flow will be always inhibited in a certain part of the circuit, it is often best to remove these parts entirely.

3.3 Comparison of Algorithms

Both algorithms are not perfect for deciding when a port can do with just a signal. Sometimes they give a too pessimistic verdict. The trouble for the colouring scheme lies mostly in the use of a FIFO. For the path-finding algorithm it will often occur in parts where flow is inhibited for the entire lifetime of a circuit. The former occurs a lot more often in real-world circuits than the latter.

The speed of the colouring analysis is good for small circuits, but fails when processing larger ones. It suffers from an exponential state-space explosion. The path-finding analysis does not suffer from this problem. It performs well for all circuits tried in experiments.

In both cases the path-finding algorithm is clearly the winner. Although the colouring scheme extension is interesting for research purposes, it is not very useful in practice. However, if the performance could be improved, it could be worthwhile to do flow analysis using both algorithms. Since the answer is always pessimistic, we can combine the results by taking the 'minimum' to obtain an improved answer.

3.4 Data Flow Analysis at Run Time

Data flow with value-passing semantics can be efficiently implemented on a shared memory platform by passing references to data. This way, only references to data items, rather than data items themselves are copies and passed around. Efficiency of such a scheme can be significantly improved with more information about what a producer or consumer exchanges does with a data item after it exchanges with the Reo circuit. Specifically, subsequent to the put/get operation that produces or acquires a data item, a producer or consumer may (1) modify the data item, (2) use its content without modifying it, or (3) never refer to the data item. We use the following intention values to designate that subsequent to a put/get operation:

W: the producer/consumer modifies the data item.
R: the producer/consumer only reads from the data item.
N: the producer/consumer neither reads from nor modifies the data item.

We assume that every put and get operation specifies the correct intention value for the data item it exchanges. In Chapter 4 we show how the source code of a component can be analysed to automatically derive these intention values in many cases. If an analysis tool or the programmer of a producer or consumer cannot ascertain the intention value for a data item exchanged through a get/put operation, it is always correct and safe to specify W, although this may make the implementation of the data flow less efficient by creating an unnecessary copy of that data item.

The value-passing semantics of Reo implies that any change made to a data item by a producer/consumer of that data item must be invisible to any other party with access to that data item.

This can be ensured if we provide a dedicated copy of a data item to every party that declares a W intention. All parties that declare an R intention for a data item can indeed share references to the same immutable copy of that data item. All consumers that declare an N intention can simply receive a null pointer.

Passing references to immutable data items and keeping track of them using a reference counting scheme, we can avoid many instances of making unnecessary copies of data. We present here a scheme for implementing the run time system of Reo that avoids making unnecessary copies in many cases by taking advantage of the intention values specified by producers and consumers for data items that they exchange through a Reo circuit, the properties of the Reo circuit, and reference counting. We refer to this scheme as the Reo Run-time Reference Counting (R3C) scheme.

The R3C scheme requires that the only items that pass through a Reo circuit at run time are either (1) references to data items, each of which has a reference count field; or (2) null references. A null reference serves as a signal that indicates the availability of a data item, without revealing its content. Non-null references point to data items that reside in shared memory. The ports of producer and consumer components, as well as the elements in a Reo circuit (channels and nodes) are the parties that generate, copy, pass, and eliminate references to implement flows of data. A data item whose reference count is greater than 1 is considered immutable and can be used for read-only access. Data items with reference counts of 1 can be modified by the party that has the reference, without affecting other parties. A data item whose reference count reduces to 0 is deallocated.

Elements in the circuit that simply transfer data, such as sync or FIFO channels, merely pass these references to shared data items. An element in the circuit whose behaviour depends on the content of the data, such as a filter channel, can access this content by dereferencing. Passing a data item through, a filter channel simply passes the reference that it has received. To lose a data item, a filter channel decrements the reference count of the data item that it receives.

An element in a circuit, such as a transformer channel, that needs to modify the content of the data that it transfers, checks the reference count of the data item that it receives. If this count is 1, the channel is the sole owner of the only reference to that data item, which allows it to directly modify the data item and pass its reference through. Otherwise, the channel makes a new copy of the data item and decrements the reference count of the data item it receives; subsequently, the channel modifies the new copy of the data item (whose reference count is 1).

An element in the circuit, such as a drain channel, whose behaviour depends only on the availability of a data item, needs to merely receive a null reference as a signal of its availability. If such a channel receives a non-null reference, it decrements the reference count of the data item that it receives.

When it can be determined that they suffice, passing null references instead of references to immutable data items saves the overhead of unnecessarily incrementing the reference count of a data item to produce such a signal, and then decrementing this count when the signal is received.

In the R3C scheme, a Reo circuit always transports references, either null references representing signals (S) or references to data items with reference counts (D). A producer port always produces a data value, although different put operations on this port may specify different intention values. Therefore, a Reo circuit expects either data (D) from a producer port, or signals (S). The R3C scheme currently over-estimates that a consumer port may always expect a data value, because different get operations on this port may specify different intention values. Therefore, a Reo circuit currently always delivers data (D) to a consumer port.

As a future refinement of the R3C scheme, if every get operation performed on a consumer port always specifies the N intention value, the port can be statically marked as expecting only a signal (S). Such a port, thus becomes analogous to a source end of a drain channel, and the

colouring analysis, described in Chapter 4, can then use this extra information to further avoid unnecessary data copies and deliver only a signal (S) to such consumer ports.

At a producer port, the combination of 3 intention values of put operations (W, R, N) and 2 expectations of a Reo circuit (D, S) results in 6 distinct cases: WD, WS, RD, RS, ND. and NS. Accordingly, the producer port behaves as follows.

WD: The port makes a new copy of the data item (whose reference count is set to 1) and passes the reference to this new copy to the circuit.
WS: The port passes a null pointer to the circuit.
RD: The port increments the reference count of the data item it receives and passes a reference to this same data item to the circuit.
RS: The port passes a null pointer to the circuit.
ND: The port passes a reference to the same data item that it receives to the circuit.
NS: The port decrements the reference count of the data item that it receives and passes a null pointer to the circuit.

At a consumer port, the combination of 3 intention values of get operations (W, R, N) and the expectation of a Reo circuit to deliver data (D) results in 3 distinct cases: DW, DR, and DS. For completeness, we also consider the case of SN as a future extension. (Observe that by definition, SW and SR are not possible: a Reo circuit would not deliver a signal (S) to a consumer port, unless every get operation on this port specifies an N intention.)

DW: If the reference count of the data item that the port receives from the circuit is 1, then the port passes the same reference to the consumer. Otherwise (i.e., if the count is greater than 1), the port makes a new copy of the data item (whose reference count is set to 1) and passes the reference to this new copy to the consumer; and then decrements the reference count of the original data item.
DR: The port passes the same reference that it receives from the circuit to the consumer.
DN: The port decrements the reference count of the data item that it receives and passes a null pointer to the consumer.
SN: The port passes the same null pointer that it receives from the circuit to the consumer.

3.5 Optimizations at Run and Compile Time

This chapter has provided three alternatives for doing flow analysis. For most, we can choose whether we want to run the analysis at compile time, or at run time. Running them at compile time means that any behaviour at run time cannot be accounted for. Take for example a circuit consisting of a single sync channel. Static analysis will then determine that there is a data colouring for both endpoints. At runtime one could imagine a scenario that the receiver will sometimes use the channel only for signalling. In that case it would have been sufficient for the sender to send only a signal as well. A dynamic analysis could have detected this situation.

A method for performing full dynamic analysis could be done by a modified version of the colouring method from Section 3.1. For example by pre-computing all colouring tables at compile time, and then selecting the appropriate row at run time. Although this system would be perfect at determining signal versus data, it has some large drawbacks. One is the extra memory needed for storing all the colouring tables. Another would be the computation overhead for selecting the applicable row. This overhead can become significant for connectors with lots of external ports.

On the other hand, the dynamic analysis of Section 3.4 comes at a very low cost. Although keeping a reference count for an object costs a little bit of execution time and memory, the net

memory usage is decreased as well is the execution time. Whilst not being perfect at preventing all copies, we have seen that it is very good at preventing the bulk of them.

The Path Finding of Section 3.2 can be done best at compile time. Channels and nodes of a protocol normally do not change at runtime. One exception is when using dynamic reconfiguration [KMLA11]. We can mark a node with S or D at compile time, and then let the run time analysis use this information in deciding whether data items need to be copied or can be dropped.

Combining the Path Finding and Run Time analysis, we seem to get the best results. Both have a minimal impact on compilation and execution, and give a boost in run time performance. Especially the Run Time analysis reduces the number of copies that have to be made of data, while the Path Finding brings a further reduction at no run time cost and insignificant compilation overhead.

Chapter 4

Preventing Copies of Read-Only Data

As described already in Section 2.7.2, a Reo circuit semantically passes values around, but our run-time system optimizes this by internally passing references. Generally, we do not know what a user intents to do with a data item. A user may modify the data item after it exchanges it with its environment, which according to the value-passing semantics of Reo must not be visible to other actors. The easiest and safest way to alleviate this problem is to create a copy of the data item before passing it back to the user.

Users might not need a full copy however. A program might do a put or get request and afterwards only read from the data, or might not even inspect the data at all. In both cases the copy was not needed.

To prevent these copies, we add an extra parameter to each put or get request. This parameter will specify the users intent on what it will do with the data item after the request. This is both for data items in a put request as well as for the items received in a get request. If a write is specified, the data will be copied as usual. For a read intent, a reference will be returned. If the programmer intents on doing nothing with the data, no data will be available. In case of a get, the data will be discarded before it reaches the user. When doing a put, the circuit will assume the ownership of the data item, invalidating the reference that the component has. A further explanation on this will be given in Section 4.1.

This solution does put a burden on the programmer. Giving a read intent while actually writing to the data will introduce errors in the execution of the program. On the other hand, choosing write while only reading does not introduce errors, but will slow down the application unnecessarily. This is why we have developed a tool which helps the user determine the correct intent. A static analyser is able to determine the correct parameter and will fill it in. The algorithm for this analysis is described in Section 4.2 and the sections following.

4.1 Intents

To optimize the amount of copies made of a data item we augment the put/get calls with an extra parameter. The signatures in the C runtime are modified to the following:

```
REOPortStatus REOPort_get(REOPort *port,
                          REOContainer **data,
                          REOPortIntent data_intent);
```

```
REOPortStatus REOPort_put(REOPort *port,
                          REOContainer *data,
                          REOPortIntent data_intent);
```

There are three possible intent values:

- *Write:* The safest but most expensive case. Used when a user will be modifying the data item within the container after the get/put call. The port will ensure that the item is unique to the user, possibly by making a copy of it.
- *Read:* To be used when the component will only read from the pointer. The container must be treated as if it is shared among other components, i.e. no writes may be done.
- *Nothing:* When doing a put, the data item will afterwards be invalid. The pointer to the container may no longer be dereferenced. In case of a get, the user requests a signal only. The port will return a `NULL` value.

As said before, it is always correct to fill in *Write* for the intent. The user will then obtain a container with which anything may be done. The other options should be used with more caution.

4.2 Algorithm

To determine in what way a data item is used after a put or get call, a static analysis is proposed in the form of Algorithm 3. The idea behind determining this is simple. By looking at the second parameter of the call, we can determine which variable we must analyse. We then analyse all instructions operating on this value, and values that may be derived from it. For each instruction we determine whether it reads or writes data. We then finally take the maximum over all these values, where $Nothing < Read < Write$.

The algorithm also uses the notion of derived instructions, such as on line 22. Some instructions load a value from a pointer. This new value is said to be a derived value of this pointer. Derived instructions are instructions that operate on such a derived value. Not all computed values are of interest however. Only new pointer values are inspected as they have a side-effect when read or modified.

Instead of the write/read/nothing system, we classify instructions in a more fine-grained way. Our fine-grained scheme uses the following classes:

- *Read:* Instruction reads a new value from the pointer.
- *Write:* Instruction performs a store on a the pointer.
- *Reference:* Used for instructions that create a new reference from an existing pointer. One example is an instruction that calculates the offset of a pointer offset in a struct. While the instruction itself does not read or write any data, the pointer obtained may be used in reading and/or writing. We thus add the value this instruction creates in the set of values to look at.
- *Dereference:* This is an instruction that loads a new pointer from the current one. It entails the combined effects of a *Read* and a *Reference*.
- *Noop:* Instruction does nothing that might affect the data referred to by the pointer in any way.

- *Unknown:* Instructions which cannot be classified. The analyser is forced to assume the worst case, which is a *Write*.

To do the analysis of a variable, we first determine its origin on line 4. Further details on how this is done is in Section 4.3.

4.2.1 Algorithm Characteristics

In this section we analyse the properties and characteristics of Algorithm 3 in combination with the reachability analysis as described in Section 4.4.

Termination

The algorithm is recursive, and as such we need to show that it will terminate after a finite amount of time. We recurse over derived instructions of the current instruction. Since derived instructions do not introduce any loops, the recursion depth is limited by the amount of instructions present in a program. A program always has a finite amount of instructions and the algorithm is thus guaranteed to terminate.

Complexity

The complexity is given in terms of amount of instructions we need to traverse. If a program has n instructions, then in the worst case all of these n are related to the variable we are analysing. For each instruction c of this set, we need to check all instructions between the put/get call f and c. This may be up to n instructions. Thus the full complexity of the algorithm is $\mathcal{O}(n^2)$.

4.3 Origin Analysis

On Line 4 from Algorithm 3 the origin of a variable is to be determined. What we mean by origin is the actual declaration, or data allocation, of that variable. If we look at Listing 4.1 we can see that there may be more than one origin for a given variable. The variables `container1` and `container2` alias, so we consider each of them as an origin.

Alias analysis is a complex subject. Several papers talk about this problem [HBCC99, Hor97, Ruf95]. For our first algorithm we decide to keep this analysis simple. This means that it will sometimes incorrectly assume two values alias, but is a lot simpler and faster to implement.

In the C language, an origin value is a pointer-to-pointer. In the case of a put operation, we start with a plain pointer type. To get the origin value we search for Load operations from which these values originate. With each instruction followed, we check the level of indirection and keep track of the maximum. We then add all pointers that have that are on the maximum level of indirection to the list of origins, as all other variables can be derived from them.

Usually, a variable has a single origin. There are cases however where a variable has multiple origins. The most natural reason occurs due to aliasing. Listing 4.1 shows a small example where this occurs. In this case, any operation on the data of `container1` will affect `container2` and vice-versa. Since the pointers do not have a shared parent, we both consider them as origin. Generally, we detect this by looking at all assignments. If the source operand is a variable we consider as an origin, the destination will be considered an origin as well.

There is also another cause of multiple origins. Some compilers, such as our analysis tool LLVM (see Section 4.5), have their intermediate representation in Single Static Assignment (SSA) form [HH98]. This form requires that variables are assigned to exactly once, and must

Algorithm 3: Intent Analysis

```
Input: A put or get call f within a function
Result: Write/Read/Nothing
Algorithm
    Let c be the data item put or retrieved by f
    Let r ← Nothing
    foreach Instruction o where c could originate from do
        Let s ← AnalyseInstruction(o, f)
        r ← max(r, s)
    end
    return r
end

Function AnalyseInstruction(I, f)
    switch Type of instruction I do
        case Read
            if I is reachable from f then return Read
            else return Nothing
        case Write
            if I is reachable from f then return Write
            else return Nothing
        case Reference
            Let r ← Nothing
            foreach derived instruction d of I do
                Let s ← AnalyseInstruction(d, f)
                r ← max(r, s)
            end
            return r
        case Dereference
            Let r ← Read
            foreach derived instruction d of I do
                Let s ← AnalyseInstruction(d, f)
                r ← max(r, s)
            end
            return r
        case Nothing
            return Nothing
        case Unknown
            return Write
    endsw
end
```

Listing 4.1: Container with complex origin

```
1 REOContainer *container1, *container2;
2 container1 = REOContainer_wrap_integer(42);
3 container2 = container1;
4 REOPort_put(port, container1, RPI_Write);
5 container2->data.i = 32;
```

be defined before they are used. For branching code, this does create a problem. Consider `x = 1; if (cond) { x = x + 1; }; y = x;`. In SSA form, this may become `x1 = 1; if (cond) { x2 = x1 + 1; } y = x?;`. Unfortunately, `x?` now depends on the branch taken. To resolve this, a virtual instruction ϕ is used. This instruction 'chooses' the correct value to assign to `y` as such: `x3 =` ϕ`(x1, x2); y = x3;` . In this example, the origin of variable `y` may be either `x1` or `x2`. Since statically we cannot determine which branch was taken, we add both of them to the list of origins.

4.4 Reachability

There are multiple places in which Algorithm 3 assesses the reachability of an instruction. The reason for doing this is simple. By looking at all derived values, we may encounter instructions that are before a put/get call, or after another instruction that overwrites the received/sent value. By first determining whether these instructions are reachable from the get/put, we may ignore the read or write that the instruction does in case it is not reachable. Otherwise the conclusion may be more pessimistic than it needs to be.

Reachability analysis is done by traversing the instructions after the put/get. If we encounter the current instruction before the value is overwritten, we deem it reachable. Otherwise we do not. This traversal needs to take branches and jumps into account. The easiest way to do this is by looking directly at the basis block structure of a program, like in Figure 4.2. Basic blocks consist of a sequence of instructions that do not contain any branches or labels. A complete program consists of a Directed Acyclic Graph (DAG) of basic blocks, in which the edges represent (un)conditional jumps. In Figure 4.2, the word *spawn* denotes that the value of interest will get overwritten, i.e. a new value is spawned into existence. In this case we see that location 2 and 3 are reachable from the `REOPort_get(..)`, while location 1 is not. The instruction on line 8 overwrites the value obtained in the get before execution may reach location 1. If we remove this spawn, all locations will be reachable.

There are three types of spawns. The first is a `REOPort_get` call. This call will receive a data item and thus overwrite the original value. A call to `REOContainer_wrap_*` will also overwrite any existing value. The final spawn operations is a plain assignment of a value.

An instruction will be reachable if it lies on any control-flow path after the get/put and before a spawn. We propose Algorithm 4 for our reachability analysis. For a put/get call p and instruction f, it checks whether f is reachable from p. It does so by first analysing the first basic block. In the first block, three interesting cases may happen:

1. f comes after p with no spawn in between. f is reachable. (line 14)
2. After p comes a spawn before f is encountered. f is not reachable. (line 11)
3. f comes before p. If there are any spawns before f or after p, then f is not reachable. (line 15)

```
1  for (int i = 0; i < 10; ++i) {
2    // location 1
3    REOPort_get(port, &container, RPI_Write);
4
5    if (container->data.i < 0) break;
6
7    // location 2
8    container = REOContainer_wrap_integer(-1);
9  }
10
11 // location 3
12 container = REOContainer_wrap_integer(2);
```

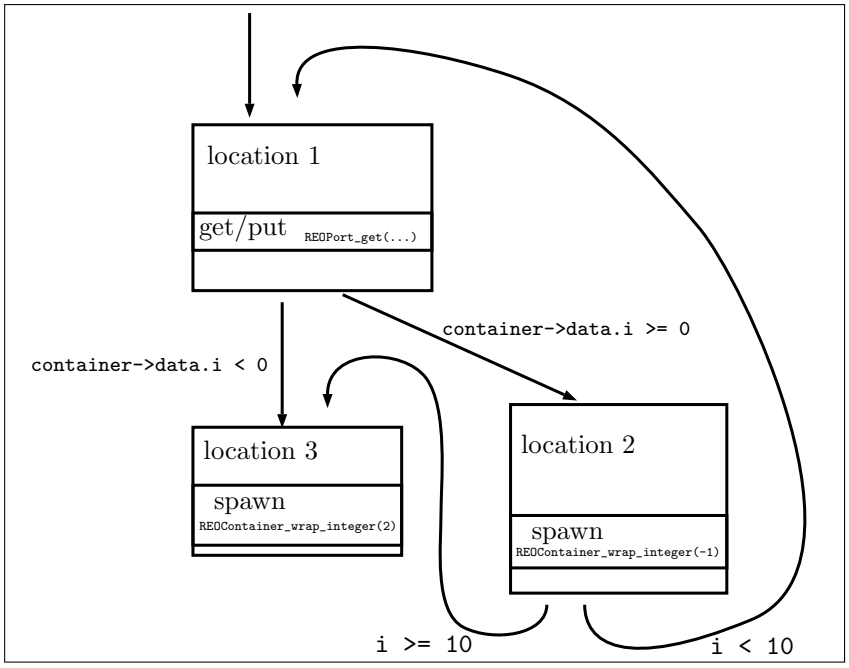


Figure 4.2: Sample code with an illustration of its basic block structure

If none of these rules match, the algorithm moves on to other basic blocks. It does a breadth-first search over the graph of basic blocks, starting with the successors of the first basic block. The algorithm iterates over each instruction contained in the basic block. If it finds a spawn, the search in this block is stopped. If the instruction f is found first however, the algorithm will return true.

When all blocks are processed, and the instruction is not found, it will thus lie after a spawn somewhere. In this case the instruction is not reachable and we should return false.

4.5 LLVM Back End

The analysis is done using a general algorithm, adaptable to most of the languages Reo could target. However, to process and modify the C code, we have implemented our tool using LLVM [LA04].[1] LLVM is a project that consists of a collection of compilers and tools, including the C/C++ compiler Clang.

We first pass the code to Clang and its optimizer to produce a file in the LLVM Intermediate Representation (IR). This representation is a lot easier to process, as most high level constructs are converted to assembly like instructions. This means that any read or write operation is now visible as a load or store operation. Furthermore, by first running the optimizer we also simplify the code such that redundant operations are mostly removed.

Once run through LLVM, we load the bitcode file containing the IR in our own analyser. It begins with a search for all occurrences of the `REOPort_get` and `REOPort_put` functions. For each call, we can then extract the instructions from which the values originated, as described in Section 4.3. Each originating instruction produces a value which can be used multiple times. These uses are then analysed, and if needed the variables they produce are analysed, and so on, as described in Algorithm 3.

Once we have the result of an analysis, we want to modify the function call to insert the correct intent. The LLVM API allows us to modify the IR with ease. We can just re-assign a constant which represents the intent to the third parameter of the put/get call. This proves to be a more robust approach than to alter the original C source code. The modified IR can then be saved to a file. The last step involves invoking another LLVM tool to translate the IR into object code. This object code will be the artefact that will be linked to when generating a component, as described in Chapter 2.

Using LLVM limits the end-user in its freedom of choosing a compiler and platform. Using a Java based compiler that would integrate into Eclipse and the ECT without external dependencies would be a more preferred method from a usability standpoint. We still chose LLVM due to its high-quality API and ease-of-use. Also, the object files our tool creates can also be linked by an other compiler, not only LLVM.

[1] The LLVM compiler infrastructure project: `http://llvm.org/`

Algorithm 4: Reachability Analysis

Input: Instruction f and put/get call p
Result: True if f might be reachable from p, false otherwise

```
1  Algorithm
2      if f = p then return false
3      B ← the basic block p is located in.
4      seenSpawn ← false
5      seenCall ← false
6      foreach Instruction I of B do
7          if I = f then
8              seenCall ← true
9              seenSpawn ← false // Any spawn seen before is no longer relevant
10         else if IsSpawn(I) then
11             if seenCall = true then return false
12             seenSpawn ← true
13         else if I = f then
14             if seenCall = true then return true
15             if seenSpawn = true then return false
16         end
17     end
       /* f is not in the first basic block, we continue with the successors */
18     S ← ∅ // Contains basic blocks of whom we have investigated the successors
19     Q ← empty queue // Queue for a breadth-first search
20     Q.push(B)
21     while Q is not empty do
22         B' ← Q.pop()
23         if B' ∈ S then continue while
24         S ← S ∪ {B'}
25         foreach Successor C of B' do
26             shouldTraverseChildren ← true
27             foreach Instruction I of C do
28                 if IsSpawn(I) then
29                     shouldTraverseChildren ← false
30                     break foreach
31                 else if I = f then
32                     return true
33                 end
34             end
35             if shouldTraverseChildren = false then S ← S ∪ {C}
36             else Q.push(C)
37         end
38     end
39     return false
40 end
```

Chapter 5

Experiments

5.1 Protocol with Components

The modifications made to the Reo tools in this paper are meant to ease the creation of multi-threaded programs and their protocols. In this first experiment we try to create such a program with various protocols and discuss the results.

5.1.1 Set-up

The program will consist of 6 producers and a single consumer, connected to each other via a protocol. Code for the components can be found in Listing 5.1. Producers will run the `sender` function and the consumer will use the `printer` function as the body of their respective threads. The consumer will have a time-out of 100 milliseconds, which is enough to ensure that it waits for data items if they are still available. The producers will all have a string argument of '1' to '6' depending on their position, and a variable number of iterations.

To connect the consumer with the producers we explore varying protocols. They are presented in Figure 5.2 and explained in detail in Sections 5.1.2–5.1.5. The way the protocols are attached can be seen in Figure 5.3, where they are connected to protocol 4.

5.1.2 Protocol 1 – Merger

The first protocol we analyse is a straight forward one. When the consumer is ready it will pick a (ready) producer at random and prints the data item it transmitted. The protocol is implemented as the top-left diagram of Figure 5.2.

The code generation phase of this protocol is almost instantaneous. This is hardly a surprise as it is equivalent to a six-way merger. Running the generated program produces a random stream of the numbers 1 to 6, as expected.

5.1.3 Protocol 2 – Sequencer

This protocol orders all producers. The consumer will first receive an item from producer 1, then from 2, then 3, 4, 5, 6, and then back to number 1. The consumer will block until the producer next in line is ready. The protocol is mainly formed by the FIFOs at the bottom. Together with the sync-drain channels, they provide the synchronisation mechanism that will give an order to the producers.

Listing 5.1: Code for producer and consumer components

```
1  void sender(REOPortOut *port, const char* string, int max_iters) {
2      for (int i = 0; i < max_iters; ++i) {
3          struct REOContainer *container = REOContainer_wrap_string(strdup(string));
4          REOPort_put(port, container, RPI_Nothing);
5      }
6  }
7
8  void printer(REOPortIn *port, int timeout) {
9      REOPort_set_timeout(port, timeout);
10     struct REOContainer *container;
11
12     while (REOPort_get(port, &container, RPI_Read) == RPS_Success) {
13       switch (container->type) {
14             case RCDT_string:
15                 printf("printer: %s\n", container->data.s);
16                 break;
17             case RCDT_integer:
18                 printf("printer: %d\n", container->data.i);
19                 break;
20             case RCDT_double:
21                 printf("printer: %f\n", container->data.d);
22                 break;
23             default:
24                 printf("printer: came accross an unknown data type");
25                 break;
26         }
27         REOContainer_release(container);
28     }
29 }
```

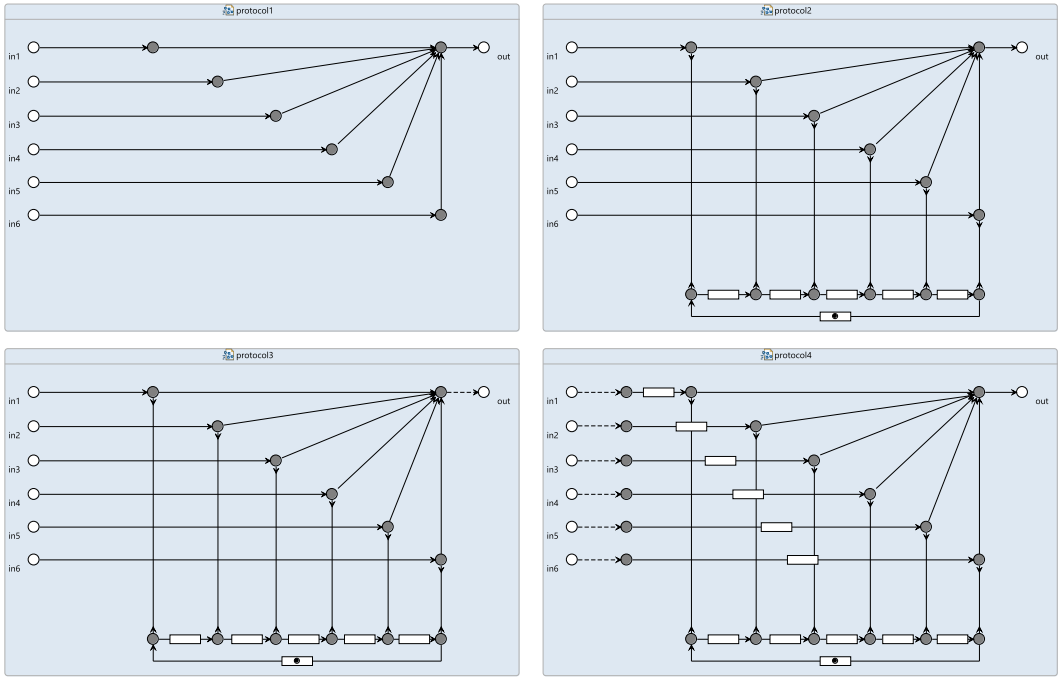


Figure 5.2: Various Reo protocols. Top: protocols 1 and 2, bottom: 3 and 4.

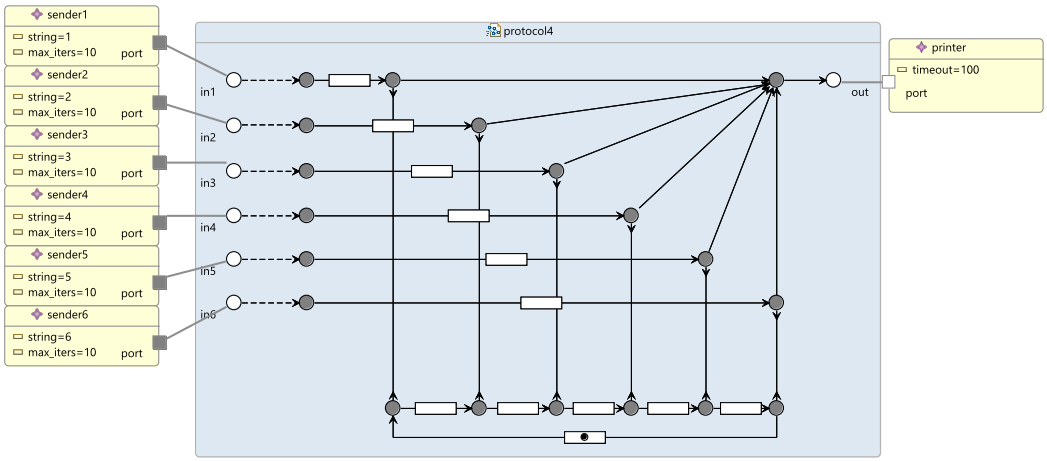


Figure 5.3: Protocol 4 with the producers and consumer attached

Using the ECT to create an animation of this network took about 30 seconds. Creating animations use colouring tables, just like the analysis method in Section 3.1. From this we conclude that that analysis would hinder the compilation process. With the other analysis methods in place, compilation was instantaneous. Running the generated program gives us the expected result: a stream of `1 2 3 4 5 6 1 ...`.

5.1.4 Protocol 3 – Lossy Sequencer

Blocking a producer is sometimes not desired. To prevent this, we allow the producers to lose data when a consumer is not ready. This is accomplished in protocol 3, at the bottom left of Figure 5.2. This protocol is almost the same as number 2, except that the last sync channel before the output node is now replaced with a lossy-sync.

This protcol is a lot like protocol 2 with respect to the compilation process. At runtime, the program now produces the numbers in order, but with some missing.

5.1.5 Protocol 4 – Buffered Lossy Sequencer

Protocol 4 is an adaptation of protocol 3. Instead of always dropping the value, we introduce a FIFO so that the first value created will at least be saved. This protocol is the bottom right image of Figure 5.2.

Creating animations for this protocol is unfortunately problematic. The ECT produces an error message regarding memory usage after about an hour. From this we conclude that the colouring table analysis is not a viable option for this circuit. The path finding analysis fortunately is able to handle this, as it does not suffer from the state-space explosion.

At runtime the program again worked as expected. The program prints the numbers `1 2 3 4 5 6` at least once and in order.

5.1.6 Conclusion

This experiment shows primarily that it is indeed easy to switch protocols in an existing program. When we had drawn the diagram of protocol 1, it was just a matter of adding nodes and replacing channels to create a program with a different protocol. While some of the protocols in this experiment are of questionable usefulness, it does show the ease of creating them. The compilation time was also good, not hindering the end-user.

5.2 Number of Copies

Section 2.7.2 talks about how reference counting, which is an optimization in the runtime already present, that tries to minimize the copies made in a program. The optimizations in Chapters 3 and 4 both also try to achieve this goal even further. In this section we analyse the difference made by applying the new optimizations.

5.2.1 Set-up

For this experiment we create a very simple program. The diagram can be seen in Figure 5.4. It features a producer, a consumer, and 1, 3 or 5 clocks. The producer will send the string ‘test’ a fixed amount of times to the consumer, when all the clocks have sent their signals.

The code for the components can be seen in Listing 5.5. To keep the analysis as clean as possible we have removed all `printf` statements. `printf` statements cause extra time spent in system calls which is hard to quantify.

In this program, two optimizations are possible. The path finding optimization should be able to detect that the clock port (B) only needs a signal and thus the data item can be discarded. The other optimization, regarding the intent of the data item, is already present in the listing. The programmer has specified that it only intends to read the data items after the calls of the producer and consumer, and thus no copies need to be made there.

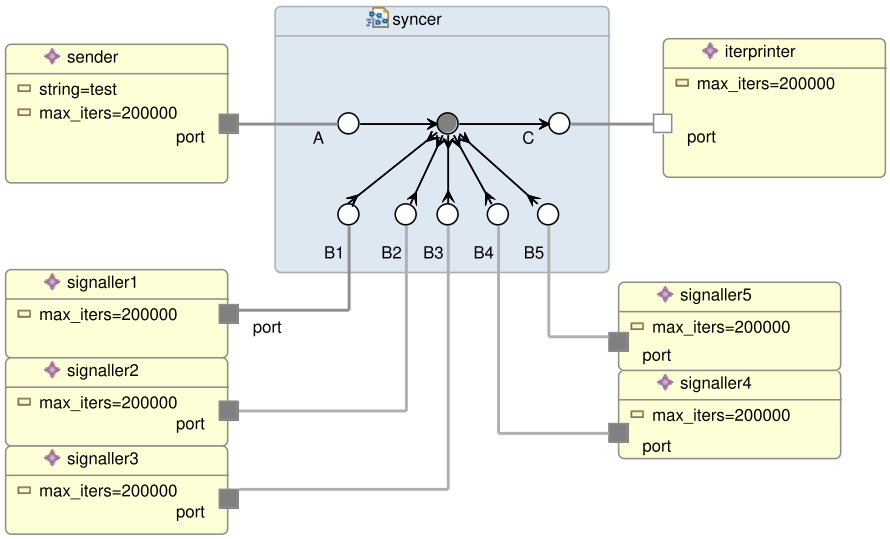


Figure 5.4: Experiment to test number of copies made, featuring 5 clocks

5.2.2 Testing

For this test we want to measure the impact of the optimizations. We generate the program multiple times, each time with a different combination of optimizations enabled. To analyse the impact of the value–signal detection as described in Section 3.2, we selectively turn off this analysis. When it is off, the code will assume the worst case, i.e. that all ports need a data value. To analyse the intent optimization of Chapter 4, we alter the components. By setting the intents on lines 4 and 21 of Listing 5.5 to `RPI_Write`, the runtime has no choice but to revert to the original way of always providing a value.

Turning the two optimizations on and off gives us 4 programs to test. For the test we set all `max_iters` parameters to 200,000 iterations. We then run each program with the tool *callgrind* of the program *valgrind* (`valgrind --tool=callgrind`) to analyse how many times functions are called and how much time that takes.

5.2.3 Results

Instead of measuring the time a program takes to complete, we measure the number of instructions executed. This enables us to better analyse the percentage of execution time the (de-)allocations take. The number of instructions that are executed in a program are rounded

Listing 5.5: Code for the malloc experiment

```
1  void sender(REOPortOut *port, const char* string, int max_iters) {
2      struct REOContainer *container = REOContainer_wrap_string(strdup(string));
3      for (int i = 0; i < max_iters; ++i) {
4          REOPort_put(port, container, RPI_Read);
5      }
6      REOContainer_release(container);
7  }
8
9  void signaller(REOPortOut *port, int max_iters) {
10     struct REOContainer *container = REOContainer_wrap_integer(0);
11     for (int i = 0; i < max_iters; ++i) {
12         REOPort_put(port, container, RPI_Write);
13         container->data.i += 1;
14     }
15     REOContainer_release(container);
16 }
17
18 void iterprinter(REOPortIn *port, int max_iters) {
19     struct REOContainer *container;
20     for (int i = 0; i < max_iters; ++i) {
21         REOPort_get(port, &container, RPI_Read);
22         // printf("printer: %s\n", container->data.s");
23         REOContainer_release(container);
24     }
25 }
```

to the nearest million. The number of times the (de-)allocation functions `malloc` and `free` are called are rounded to the nearest thousand.

The results of this experiment can be seen in Table 5.6. The first column shows which optimizations have been applied. The second column shows the number of instruction fetches performed in the program. This is a nice measure of the execution time of a program, as it is not influenced by other programs running on the system, as opposed to using the wall clock. How many instructions are performed relative to the 'None' row is displayed next. In column four you can see the number of calls that are made to `malloc`. Since any memory that is allocated needs also to be freed, this is (almost) the same as the number of calls made to `free`.[1] However, the number of instructions performed in a `free` call is higher than in a `malloc` call. The number of instructions that the program spends on these functions relative to the total number of instructions are the last two columns.

In Figure 5.7 the speed-up of the various optimizations is presented. The first thing to notice is that both optimizations positively influence the instruction count in the program. We are able to reduce the number of instructions executed by about 25% in all cases. An unoptimized program apparently spends a lot of time dealing with memory operations. After both optimizations, the allocations are reduced to a constant number for initialization plus one for each iteration.

We see that each optimization alone can be very useful by itself. With only one clock, the intent optimization performed the best. With 3 and 5 clocks, the signal gave better results. This correlates with the number of components the optimization works for. In all cases, the intent optimizes 2 components, the producer and consumer. The signal analysis optimizes only the clocks. Combined, they are able to eliminate all allocations apart from those in the set-up code, and those needed when firing a transition.

[1] The number of calls to `free` were always 3 less than the number of calls to `malloc`. This would normally indicate a memory leak, but valgrind running the *memcheck* tool did not indicate any was present.

Optimizations	instructions	relative	calls to malloc	instrs. `malloc`	instrs. `free`
1 clock					
None	588M	100%	836K	8,39%	10,38%
Signal (ch 3)	541M	92%	626K	6,83%	8,45%
Intent (ch 4)	484M	82%	400K	4,87%	6,03%
Both	439M	75%	200K	2,69%	3,32%
3 clocks					
None	874M	100%	1249K	8,44%	10,44%
Signal	736M	84%	648K	5,20%	6,43%
Intent	764M	87%	800K	6,18%	7,64%
Both	626M	72%	200K	1,89%	2,33%
5 clocks					
None	1274M	100%	1647K	7,63%	9,44%
Signal	1039M	82%	641K	3,64%	4,51%
Intent	1164M	91%	1200K	6,08%	7,52%
Both	932M	73%	200K	1,27%	1,57%

Table 5.6: The amount of instructions total and spend in memory (de)allocations with various levels of optimization

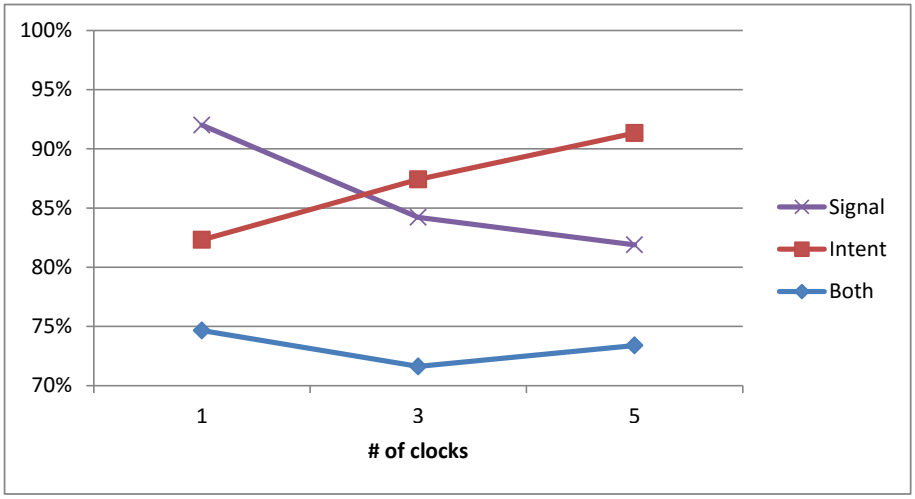


Figure 5.7: Executed instructions as a percentage compared to no optimizations

5.3 Analysing Intents with LLVM

In this section we analyse two C components to check whether our analyser can correctly determine the intent of the data item.

For this analysis we run the analyser on the code in Listing 5.8. This code has get and put calls that are annotated with the correct intent, but specify *Write* for intent argument. When the analyser is run it produces the following results:

```
[Info] Analysing function call at row 27, column 7
[Info] Result: RPI_Read
[Info] Modifying from 0 to 1
[Info] Analysing function call at row 14, column 5
[Info] Result: RPI_Write
[Info] Analysing function call at row 24, column 5
[Info] Result: RPI_Nothing
[Info] Modifying from 0 to 2
[Info] Analysing function call at row 24, column 5
[Info] Result: RPI_Nothing
[Info] Modifying from 0 to 2
[Info] Analysing function call at row 17, column 5
[Info] Result: RPI_Nothing
[Info] Modifying from 0 to 2
```

The modifications talk about the integer representations of the intents, with `RPI_Write` = 0, `RPI_Read` = 1 and `RPI_Nothing` = 2. As can be seen, the algorithm correctly determines all intents on its own. One thing to note is that the call on row number 24 is analysed twice. If a compiler optimizes code it can sometimes decide to duplicate some code in order to speed things up. This is not a problem for the analyser, however, because it works with the intermediate LLVM-code that the C compiler generates. Although they come from the same source line, to the analyzer the two LLVM-level calls look as two independent calls.

Listing 5.8: Code to be analysed for the data intent optimizer

```
9  /* some includes and defines */
10
11 void square_unit(REOPortIn *input, REOPortOut* output, int offset) {
12   for (int i = 0; i < LOOPS; ++i) {
13     struct REOContainer *container;
14     REOPort_get(input, &container, RPI_Write); // correct intent
15     container->data.i *= container->data.i;
16     container->data.i += offset;
17     REOPort_put(output, container, RPI_Write); // should be RPI_Nothing
18   }
19 }
20
21 void print_and_pass(REOPortIn *input, REOPortOut* output, int inputs) {
22   for (int i = 0; i < LOOPS; ++i) {
23     struct REOContainer *container = REOContainer_wrap_integer(i);
24     REOPort_put(output, container, RPI_Write); // should be RPI_Nothing
25     for (int j = 0; j < inputs; ++j) {
26       struct REOContainer *input_container;
27       REOPort_get(input, &input_container, RPI_Write); // should be RPI_Read
28       printf("loop %d, data %d: %d\n", i, j, input_container->data.i);
29       REOContainer_release(input_container);
30     }
31   }
32 }
```

Chapter 6

Conclusions

Creating a Reo protocol and generating code that implements that protocol was already fairly easy in the Extensible Coordination Tools. In this thesis we have made the creation of multi-threaded applications easier by enabling the ECT to link protocols to externally written code. Using a familiar user interface. The programmer can draw a protocol, import his own computation code as components, and link them all together. Currently only the C language is supported, but the possibility for adding new languages is taken in consideration in the design of our extensions to the ECT.

Some effort has also been spend to make generated programs easy to use for research purposes. Arguments to functions can be either fixed at compile time or left unset so that the user can specify them on the command line. It is also possible for a programmer to leave some ports of the protocol unconnected, so that he/she may interact with them directly at run-time through a console. A GUI has been created to mimic multiple terminals, each of them connected to a different port that is left unconnected to a component.

The value passing semantics of Reo make it easy for a programmer to construct a program. It allows a programmer to interact with a data item in any way he/she wants, without restrictions on reading or writing. If implemented literally, this could make a program rather slow. To prevent this, two optimizations have been investigated and implemented.

The first analysis tries to optimize the protocol. It is possible to design a protocol that uses some ports for synchronization only. This means that only the timing of a put action matters, not the data that it provides. One way for determining the behaviour of a port is by using a newly constructed colouring scheme. It uses four colours instead of the usual two or three. After constructing a colouring table for the entire circuit we are able to look at each port and determine if the data put into it is used somewhere else. If it is only used for synchronization, we discard the data item early to prevent it from being unnecessarily copied.

The colouring analysis in its current form works well for small protocols but can cause issues with larger ones. Adding lots of FIFOs for example can cause a state space explosion, causing the tools to exhaust the available memory and crash. Future work may be done on the search for an improved algorithm for the colouring engine.

There is also another, more fundamental, problem in the colouring of a FIFO. If the output of a FIFO is used for signalling only, it would be preferable to also not accept anything but a signal. The current colouring semantics are unfortunately not powerful enough to express this. This situation is similar to a filter channel that drops some data. It would be better to recognize this dropping at an early stage, so that some computations in channels sitting in front of the filter may be omitted. One idea is to have some sort of algebra for Reo. This algebra could then

be used to be able to determine in which way a circuit could be transformed where the semantics of a protocol is preserved. Circuits in which FIFOs and filters come before other complicated channels could help us better in analysing the original circuit.

An alternative to the colouring scheme is to do flow analysis via path finding. The idea is that for each input port we check if there is a path to an output port or data-aware channel. If such a path exists, the data put into the port may be inspected. If such a path does not exist we can conclude that all data items will be lost before being inspected, which means we can drop them early. Path finding can be done very efficiently on Reo protocols. In contrast to the colouring method, this algorithm is able to handle a FIFO correctly. Although there are other protocols in which it produces a sub-optimal result, these are a lot more rare when considering real-world Reo circuits.

The second optimization deals with the components. The value passing semantics of Reo allow a programmer to do anything with a data item after it has been send or received from a port. The runtime must ensure all writes from one component are never visible to any other. Conceptually the runtime accomplishes this by giving each component a unique copy of a data item. In some situations this is too much effort. If a user will never write to its received data, obtaining a shared reference would have sufficed. This would be a lot faster since a memory allocation and deep copy can be omitted. Thus we give the programmer the ability to specify what he/she intends to do with the data item. To prevent issues we also provide a framework which can automatically determine this intent. This framework analyses the component code using a low level representation obtained with LLVM. An object file is created after the analysis, which the programmer can then link to the protocol.

Analysing the use of a data item in C is not an easy task [HBCC99, Hor97, Ruf95]. The owner of a data item and (shared) reference to an item make it difficult to get a clear view of which variables are actually aliases of the same item and which are not. In the past, programming languages have been proposed that have a stronger notion of ownership. This would ease the analysis tremendously, if not even make it obsolete. The compiler for such a language could check whether the user transfers ownership or provides a (read-only) reference. If done correctly, it could handle all of the static analysis mentioned in Chapter 4 for us. Ownership types [CPN98, CÖSW13] provide a mechanism that could do just this. An example of a language with strong ownership semantics is *Rust* [MK14].[1] As a compiled language without a garbage collector, it can run as fast as C. Unlike C, the compiler enforces that each object has a single owner, and that no-one may issue a write on data obtained via a shared reference. A get call on a port in Rust could for example return an object of type `Arc<T>`. This is a pointer which uses (atomic) reference counting that pointing to some (copyable) type `T`. The compiler enforces such a pointer to be read-only. When calling `make_mut` on such a pointer it is converted to a mutable reference, copying only when necessary. Although Rust is not very mature, these features would make any language a prime candidate for users to write their computational component code in, and a suitable target language for Reo.

The C analysis also has some issues when two components want to work independently on a large shared object. For example, when multiple components want to write to a buffer used for displaying an image. It is most likely that in this case two components will use the protocol to send a pointer to each other for writing. The analysis is currently not able to distinguish between a reference that is inherent to the data item and a pointer. Thus any write to the pointer will be seen as a write action on the data item. Future work here can be done on annotating fields to indicate that certain fields may be excluded from further analysis.

The Reo runtime also contains an optimization in the form of the R3C scheme. This scheme, especially combined with the intent and path finding analysis, is able to prevent almost all

[1] `https://www.rust-lang.org/`

unnecessary copies that would otherwise be made by passing data back and forth between ports and the protocol.

One of the most important optimizations still needed in the Reo runtime is the constraint solving. Simple programs spend the majority of their time on the solving of data constraints of the constraint automata. Luckily, [JA15] proposes a new way of constraint solving by converting the constraints into commands that can be used to check a transition more efficiently. It is intended to port the C runtime and code generator to this new mechanism.

Support for other programming languages is also important to work on in the future. Java already has a Reo runtime, but currently lacks a code generator which is also able to handle the new components. Other languages need much more work for them to be supported. To quickly add new languages an indirect approach is also possible. Lots of languages can interface directly with C. It is possible to create a small wrapper function in C which then calls the real function in the target language. This is supported in for example Python, Lua, D and Rust. A native code generator and runtime can then be written when the language has become a popular choice.

Appendix A

Example of Generated Code by the ECT

This appendix shows the code from the example in Section 2.2. Section A.1 lists the code for the components and Sections A.2–A.5 show the code generated by the ECT.

A.1 demo.c

This is the code listing of the three components.

```
1  #include "reocrt.h"
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <string.h>
5
6  #ifdef _MSC_VER
7  #   define strdup(s) _strdup(s)
8  #endif
9
10 typedef struct REOPort REOPortIn;
11 typedef struct REOPort REOPortOut;
12
13 void printer(REOPortIn *port, int timeout) {
14     REOPort_set_timeout(port, timeout);
15     struct REOContainer *container;
16
17     while (REOPort_get(port, &container, RPI_Read) == RPS_Success) {
18       switch (container->type) {
19             case RCDT_string:
20                 printf("printer: %s\n", container->data.s);
21                 break;
22             case RCDT_integer:
23                 printf("printer: %d\n", container->data.i);
24                 break;
25             case RCDT_double:
26                 printf("printer: %f\n", container->data.d);
27                 break;
28             default:
29                 printf("printer: came accross an unknown data type");
30                 break;
31         }
```

```
32          REOContainer_release(container);
33      }
34  }
35
36  void sender(REOPortOut *port, const char* string, int max_iters) {
37      for (int i = 0; i < max_iters; ++i) {
38          struct REOContainer *container = REOContainer_wrap_string(strdup(string));
39          REOPort_put(port, container, RPI_Nothing);
40      }
41  }
42
43  void signaller(REOPortOut *port, int max_iters) {
44      for (int i = 0; i < max_iters; ++i) {
45          REOPort_put(port, NULL, RPI_Nothing);
46          puts("signalled");
47      }
48  }
```

A.2 main.c

```
1   #include "connector/syncer.h"
2   #include "console.h"
3
4   #include <stdlib.h>
5   #include <stdio.h>
6   #include <string.h>
7   #include <pthread.h>
8
9   typedef struct REOPort REOPortIn;
10  typedef struct REOPort REOPortOut;
11
12  // functions as imported from the source file
13  void sender(REOPortOut *port, const char* string, int max_iters);
14  void printer(REOPortIn *port, int timeout);
15  void signaller(REOPortOut *port, int max_iters);
16
17  // struct to wrap the arguments of the sender function
18  struct sender_s {
19      REOPortOut * port;
20      const char * string;
21      int max_iters;
22  };
23
24  // function to start sender as a pthread
25  void* sender_f(struct sender_s *args) {
26      sender(args->port, args->string, args->max_iters);
27      return NULL;
28  }
29
30  // struct to wrap the arguments of the printer function
31  struct printer_s {
32      REOPortIn * port;
33      int timeout;
34  };
35
36  // function to start printer as a pthread
37  void* printer_f(struct printer_s *args) {
38      printer(args->port, args->timeout);
39      return NULL;
```

```
40  }
41
42  // struct to wrap the arguments of the signaller function
43  struct signaller_s {
44      REOPortOut * port;
45      int max_iters;
46  };
47
48  // function to start signaller as a pthread
49  void* signaller_f(struct signaller_s *args) {
50      signaller(args->port, args->max_iters);
51      return NULL;
52  }
53
54  int main(int argc, char *argv[]) {
55      const char* sender_string = NULL;
56
57      // collect all command-line arguments
58      for (int i = 0; i < argc - 1; ++i) {
59          if (strcmp(argv[i], "--sender_string") == 0) {
60              sender_string = argv[i + 1];
61              i++;
62              continue;
63          }
64      }
65
66      // check if all arguments are specified
67      int ok = 1;
68      if (sender_string == NULL) {
69          fprintf(stderr, "Missing parameter --sender_string\n");
70          ok = 0;
71      }
72
73      if (!ok) return 1;
74
75      // construct the protocol
76      struct Connector_syncer *data_syncer = build_syncer();
77
78      // construct sender thread
79      pthread_t thread0;
80      struct sender_s arg0;
81      arg0.port = &data_syncer->A;
82      arg0.string = sender_string;
83      arg0.max_iters = 10;
84
85      // start sender thread
86      pthread_create(&thread0, NULL, (void*(*)(void*))sender_f, &arg0);
87
88      // construct and start printer thread
89      pthread_t thread1;
90      struct printer_s arg1;
91      arg1.port = &data_syncer->B;
92      arg1.timeout = 1000;
93
94      pthread_create(&thread1, NULL, (void*(*)(void*))printer_f, &arg1);
95
96      // construct and start signaller thread
97      pthread_t thread2;
98      struct signaller_s arg2;
99      arg2.port = &data_syncer->C;
100     arg2.max_iters = 10;
101
```

```
102     pthread_create(&thread2, NULL, (void*(*)(void*))signaller_f, &arg2);
103
104     // wait for all components to finish
105     pthread_join(thread0, NULL);
106     pthread_join(thread1, NULL);
107     pthread_join(thread2, NULL);
108
109     // wait for the protocol to finish and release the memory
110     clean_syncer(data_syncer);
111
112     return 0;
113 }
```

A.3 connector/syncer.h

```
1  #ifndef connector_syncer_H
2  #define connector_syncer_H
3
4  #include "reocrt.h"
5
6  // Struct definition which holds all data regarding to the
7  // regions, ports and FIFOs
8  struct Connector_syncer {
9      struct REOAutomaton region1;
10     struct REOPort A;
11     struct REOPort B;
12     struct REOPort C;
13 };
14
15 struct Connector_syncer *build_syncer();
16 void clean_syncer(struct Connector_syncer *data);
17
18 #endif // connector_syncer_H
```

A.4 connector/syncer.c

```
1  #include "syncer.h"
2
3  #include <stdbool.h>
4  #include <stdio.h>
5  #include <stdlib.h>
6  #include <string.h>
7
8  #ifdef _MSC_VER
9  #   define strdup(s) _strdup(s)
10 #endif
11
12 struct Connector_syncer *build_syncer() {
13     struct Connector_syncer *data = calloc(sizeof(struct Connector_syncer), 1);
14
15     /* Automaton definitions */
16     REOAutomaton_construct(&data->region1, 3, 1, 0);
17
18     // Construct ports A, B and C
19     struct REOPort *A = &data->A;
20     REOPort_construct(A, &data->region1, 0, RPDT_Data);
21     struct REOPortHandler *A_handler = &A->handler;
22     A->can_put = true;
23     struct REOPort *B = &data->B;
```

```
24      REOPort_construct(B, &data->region1, 1, RPDT_Data);
25      struct REOPortHandler *B_handler = &B->handler;
26      B->can_get = true;
27      struct REOPort *C = &data->C;
28      REOPort_construct(C, &data->region1, 2, RPDT_Signal);
29      struct REOPortHandler *C_handler = &C->handler;
30      C->can_put = true;
31
32      // Attach the ports to the automaton
33      REOAutomaton_attach_ports(
34          &data->region1,
35          (struct REOPortHandler*[]){A_handler, B_handler, C_handler}
36      );
37
38      // Construct the states and transitions
39      {
40          struct REOTransition *transition;
41          struct REOProblem *problem;
42          struct REOState *state0 = &data->region1.states[0];
43          REOState_construct(
44              state0,
45              1,
46              3,
47              (struct REOPortHandler*[]){A_handler, B_handler, C_handler}
48          );
49
50          transition = &state0->transitions[0];
51          problem = &transition->problem;
52
53          REOTransition_construct(
54              transition,
55              &data->region1,
56              0,
57              3,
58              (unsigned[]){0,1,2}
59          );
60
61          REOProblem_construct(problem, 2, 1);
62
63          REOProblem_set_variable(problem, 0, A_handler, 0);
64          REOProblem_set_variable(problem, 1, B_handler, 1);
65
66          REOProblem_set_constraint(problem, 0, RCT_Equality, 2, (unsigned[]){0,1});
67
68          REOState_bind_transitions_to_ports(state0);
69      }
70
71      return data;
72  }
73
74  void clean_syncer(struct Connector_syncer *data) {
75      REOAutomaton_cleanup(&data->region1);
76
77      REOPort_cleanup(&data->A);
78      REOPort_cleanup(&data->B);
79      REOPort_cleanup(&data->C);
80
81      free(data);
82  }
```

A.5 Makefile

```
1  REOCRTLIBRARY := "/path/to/workspace/Reo-CRT/libreocrt.a"
2  REOCRTINCLUDE := "/path/to/workspace/Reo-CRT/Reo-CRT"
3  CFLAGS := -I$(REOCRTINCLUDE) -std=gnu99
4  LDFLAGS := $(REOCRTLIBRARY) -lpthread -lrt
5  OBJECTFILES := main.o console.o connector/syncer.o
6
7  # Custom libraries as defined by the Module
8  LIB_libdemo := "/path/to/workspace/components/libdemo.a"
9
10 all: program
11
12 program: $(OBJECTFILES)
13   $(CC) -o $@ $^ $(LIB_libdemo)  $(LDFLAGS)
14
15 clean:
16   $(RM) program $(OBJECTFILES)
```

Bibliography

[AKM+08] Farhad Arbab, Christian Koehler, Ziyan Maraikar, Young-Joo Moon, and José Proença. Modeling, testing and executing Reo connectors with the Eclipse Coordination Tools. *Tool demo session at FACS*, 8, 2008.

[Arb04] Farhad Arbab. Reo: a channel-based coordination model for component composition. *Mathematical Structures in Computer Science*, 14(03):329–366, 2004.

[Arb11] Farhad Arbab. Puff, the magic protocol. In *Formal Modeling: Actors, Open Systems, Biological Systems*, pages 169–206. Springer, 2011.

[Boe04] Hans-J. Boehm. The space cost of lazy reference counting. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '04, pages 210–219, New York, NY, USA, 2004. ACM.

[CCA07] Dave Clarke, David Costa, and Farhad Arbab. Connector colouring I: Synchronisation and context dependency. *Science of Computer Programming*, 66(3):205–225, 2007.

[CÖSW13] Dave Clarke, Johan Östlund, Ilya Sergey, and Tobias Wrigstad. Ownership types: A survey. In Dave Clarke, James Noble, and Tobias Wrigstad, editors, *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*, volume 7850 of *Lecture Notes in Computer Science*, pages 15–58. Springer, 2013.

[CPN98] David G. Clarke, John M. Potter, and James Noble. Ownership types for flexible alias protection. In *Proceedings of the 13th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '98, pages 48–64, New York, NY, USA, 1998. ACM.

[CWI15] CWI Formal Methods group. Reo home page. `http://reo.project.cwi.nl/`, 2015.

[HBCC99] Michael Hind, Michael Burke, Paul Carini, and Jong-Deok Choi. Interprocedural pointer alias analysis. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 21(4):848–894, 1999.

[HH98] Rebecca Hasti and Susan Horwitz. Using static single assignment form to improve flow-insensitive pointer analysis. In *ACM SIGPLAN Notices*, volume 33, pages 97–105. ACM, 1998.

[Hor97] Susan Horwitz. Precise flow-insensitive may-alias analysis is NP-hard. *ACM Trans. Program. Lang. Syst.*, 19(1):1–6, January 1997.

[JA11] Sung-Shik TQ Jongmans and Farhad Arbab. Correlating formal semantic models of Reo connectors: Connector coloring and constraint automata. *arXiv preprint arXiv:1108.0468*, 2011.

[JA12] Sung-Shik TQ Jongmans and Farhad Arbab. Overview of thirty semantic formalisms for Reo. *Sci. Ann. Comp. Sci.*, 22(1):201–251, 2012.

[JA15] Sung-Shik TQ Jongmans and Farhad Arbab. Take command of your constraints! In *Coordination Models and Languages*, pages 117–132. Springer, 2015.

[KMLA11] Christian Krause, Ziyan Maraikar, Alexander Lazovik, and Farhad Arbab. Modeling dynamic reconfigurations in Reo using high-level replacement systems. *Science of Computer Programming*, 76(1):23 – 36, 2011. Selected papers from the 6th International Workshop on the Foundations of Coordination Languages and Software Architectures FOCLASA07.

[LA04] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pages 75–86. IEEE, 2004.

[MK14] Nicholas D. Matsakis and Felix S. Klock, II. The Rust language. *Ada Lett.*, 34(3):103–104, October 2014.

[PJSS12] Danila Piatov, Andrea Janes, Alberto Sillitti, and Giancarlo Succi. Using the Eclipse C/C++ development tooling as a robust, fully functional, actively maintained, open source C++ parser. *OSS*, 378:399, 2012.

[Ruf95] Erik Ruf. Context-insensitive alias analysis reconsidered. *SIGPLAN Not.*, 30(6):13–22, June 1995.

[vdN15] Mathijs van de Nes. Implementation of the Reo runtime for the C language. Research project, Leiden Institute of Advanced Computer Science, Febuary 2015.

[Vin14] Lieuwe Vinkhuijzen. Reducing copying and network traffic in Reo circuits. Bachelor thesis, Leiden Institute of Advanced Computer Science, August 2014.