Reo Documentation

Release 1.0

Kasper Dokter

Apr 09, 2020

Contents

Reo is an *exogenous coordination language* designed by prof. dr. F. Arbab in the late 1990's at the Centre for Mathematics and Computer Science (CWI) in Amsterdam. What distinguishes Reo from other contemporary formal models and languages for concurrent systems is that at its core, Reo offers an interaction-centric model of concurrency. In contrast to process- or action-centric models of concurrency, Reo allows a programmer to directly specify protocols that describe interaction among components in a concurrent application as explicit, concrete pieces of composable software.

CHAPTER 1

What are protocols, and why do we care?

A protocol specifies proper sequences of steps that a set of concurrently executing processes must follow to orchestrate their interactions in order for the system to manifest a desirable behavior, such as mutually exclusive access to a shared resource (mutual exclusion) or exchanging information according to a producer/consumer pattern. Unfortunately, most general purpose programming languages do not provide syntax for expressing these protocols as concrete, modular pieces of software. This shortcoming forces programmers to implement their protocols indirectly, by manually adding locks and buffers and ensuring their correct usage. Even if the implementation correctly manifests the intended protocol, the implicit encoding of the protocol makes it hard, if not impossible, to reason about its correctness and efficiency, scale it to engage more processes, or reuse it in some other application.

Reo addresses this problem by providing syntax that enables explicit high-level construction of protocols. When a protocol is specified explicitly, it becomes easier to write correct protocols that are free of dead-locks, live-locks, or data races. Moreover, a compiler is then able to optimize the actual implementation of the protocol.

CHAPTER 2

Reo compiler

This document describes the usage of a Reo compiler that generates a multithreaded application from a set of single-threaded components and a Reo protocol specification.

CHAPTER 3

Contents of this documentation

The contents of the documentation is as follows:

3.1 Installation

1. Install Java SDK 1.6+. You can check if the correct java version is installed by running `java -version`.
2. Download the reo compiler corresponding to your operating system (linux, mac, or windows).
3. Unzip the archive in the directory you want reo to be installed, and run the install script (either `installer.sh` or `installer.bat`).
4. Run `reo` to see if the installation succeeded. Then, follow the README to get your first example running.

3.2 Introduction

3.2.1 A simple concurrent program

Writing correct concurrent programs is far more difficult than writing correct sequential programs. Let us start with a very simple program that repeatedly prints "Hello, " and "world!" in alternating order. We split this program into three different processes: two producers that output strings "Hello, " and "world!", respectively, and a consumer that **alternately** prints the strings that it obtains from these producers, starting of course with "Hello, ".

If you are asked to write a small program that implements the above informal specification, you may come up with the following Java code:

```
import java.util.concurrent.Semaphore;

public class Main {

        private static final Semaphore greenSemaphore = new Semaphore(0);
```

(continues on next page)

(continued from previous page)

```
        private static final Semaphore redSemaphore = new Semaphore(1);
        private static final Semaphore bufferSemaphore = new Semaphore(1);
        private static String buffer = null;

        public static void main(String[] args) {
                Thread redProducer = new Thread("Red Producer") {
                        public void run() {
                                while (true) {
                                        for (int i = 0; i < 30000000; ++i);
                                        String redText = "Hello, ";
                                        try {
                                                redSemaphore.acquire();
                                                bufferSemaphore.acquire();
                                                buffer = redText;
                                                bufferSemaphore.release();
                                                greenSemaphore.release();
                                        } catch (InterruptedException e) { }
                                }
                        }
                };

                Thread greenProducer = new Thread("Green Producer") {
                        public void run() {
                                while (true) {
                                        for (int i = 0; i < 50000000; ++i);
                                        String redText = "world! ";
                                        try {
                                                greenSemaphore.acquire();
                                                bufferSemaphore.acquire();
                                                buffer = redText;
                                                bufferSemaphore.release();
                                                redSemaphore.release();
                                        } catch (InterruptedException e) {
                                                e.printStackTrace();
                                        }
                                }
                        }
                };

                Thread blueConsumer = new Thread("Blue Consumer") {
                        public void run() {
                                int k = 0;
                                while (k < 10) {
                                        for (int i = 0; i < 40000000; ++i);
                                        try {
                                                bufferSemaphore.acquire();
                                                if (buffer != null) {
                                                        System.out.print(buffer);
                                                        buffer = null;
                                                        k++;
                                                }
                                                bufferSemaphore.release();
                                        } catch (InterruptedException e) {
                                                e.printStackTrace();
                                        }
                                }
                        }
```

(continues on next page)

(continued from previous page)

```
                };

                redProducer.start();
                greenProducer.start();
                blueConsumer.start();

                try {
                        blueConsumer.join();
                } catch (InterruptedException e) {
                        e.printStackTrace();
                }
        }
}
```

The main method in the above Java code instantiates three different Java threads, namely a Red and a Green producer and a Blue consumer. These threads communicate with each other via a shared buffer that is protected by a semaphore. If a thread wants to operate on the buffer, it tries acquire the semaphore that protects the buffer. Once acquired, the process can write to the buffer without being disturbed by any other process. Finally, the process releases the semaphore, which allows other processes to acquire it and operate on the buffer undisturbed.

This program uses additional *turn-keeping* semaphores to alternate the writes by producers to the buffer: each producer has its own turn-keeping semaphore. If a producer wants to write to the buffer, it first tries to acquire its turn-keeping semaphore. After writing to the buffer, the producer hands over the *turn to write* to the other producer, by releasing the latter's turn-keeping semaphore.

3.2.2 Analysis

Let us now analyze this Java implementation by answering a few simple questions.

1. Where is the "Hello, " string computed?

On line 15: *String redText = "Hello, ";*.

2. Where is the text printed?

On line 53: *System.out.print(buffer);*.

For the next question, however, it is not possible to point at a single line (or even a contiguous segment) of code:

3. Where is the protocol?

    a. What determines which producer goes first?

    This is determined by the initial values of the semaphores on lines 5 and 6, together with the acquire statements of semaphores on lines 17, and 33.

    b. What determines that the two producers alternate?

    This is determined by the initial values of the semaphores on lines 5 and 6, together with the acquire and release statements of the semaphores on lines 17, 21, 33, and 37. Observe that 4 out of these 6 lines are the same lines involved in the answer to question 3.a; a fact that means if we change one of these common lines of code with the intention of modifying one aspect of the protocol, we must simultaneously consider whether and how this change affects the other aspect of the protocol.

    c. What takes care of buffer protection?

    This is established by the acquire and release statements of the buffer semaphore on lines 18, 20, 34, 36, 51, and 57.

The reason why Question 3 is much more difficult and has a more complicated answer is that unlike the *computation* in this application, its protocol is **implicit**. Modern programming languages provide suitable constructs to explicitly specify computation as concrete, contiguous segments of code; however, the constructs that they provide to program concurrent software allows programmers to specify concurrency protocols only indirectly, as the side-effect of the execution of (concurrency) primitive actions scattered throughout the code of various processes.

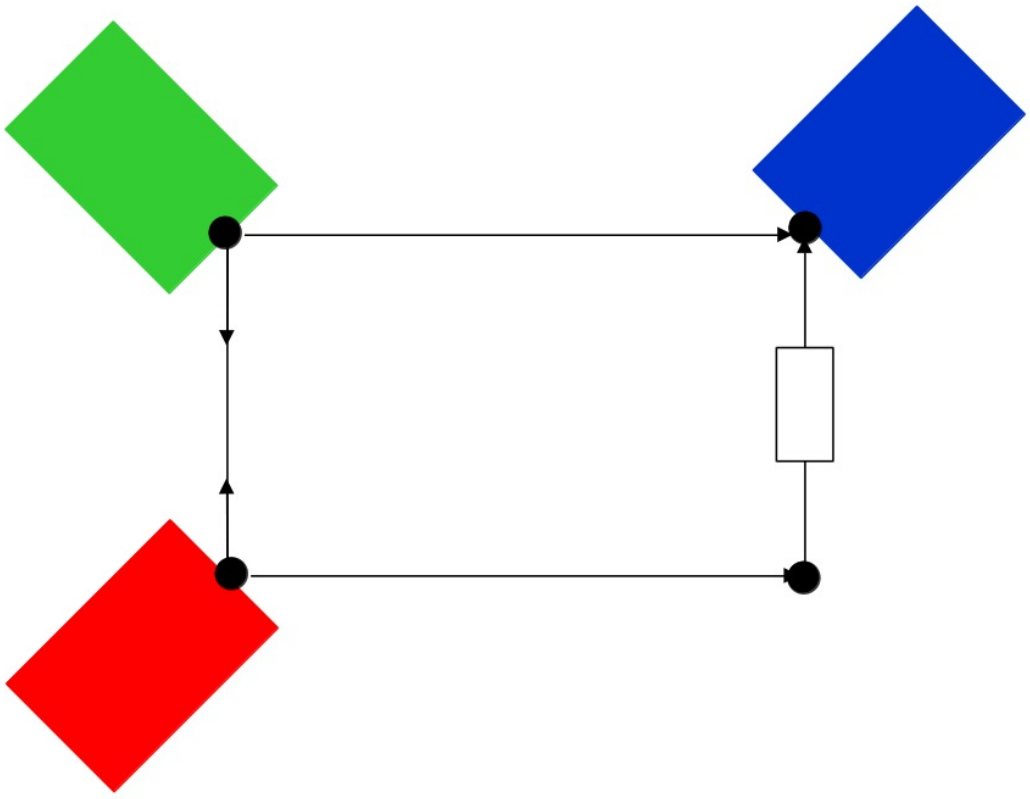
For such a simple program, you may argue that the fact that the protocol is implicit is not big deal. However, if you really think this, then you may be surprised by the output when you run the above application:

```
Hello, world! Hello, Hello, world! Hello, Hello, Hello, Hello, Hello,
```

If you expected to see nothing but "Hello, world!" as output, then you must agree that there is a bug! Can you spot the error?

3.2.3 Reo protocols

The Reo language offers a solution by providing a domain specific language that allow you to specify your protocol explicitly, as a concrete, identifiable piece of software. The following diagram shows an example of such an explicit protocol:



Every process is represented as a box together with a set of **ports** that define the interface of each process. These boxes, called **components**, are connected via a network of **channels** and **nodes**, which constitutes the protocol. The components now interact with each other by offering values to the protocol. The protocol, then, coordinates the exchanges of values amongst components.

The channel between Red and Green is a *syncdrain* channel that repeatedly accepts a data item from each of its input ends in a single atomic action, and loses both data items. The channel between Red and Blue is a *sync* channel that

atomically takes a data item from its input end and passes this data item through its output end. The other incoming channel connected to Blue is a *fifo1* channel that has a buffer (depicted as the box in the middle of the arrow) with the capacity to store a single data item. The buffer of this channel is initially empty. Whenever its buffer is empty, this channel can accept a single data item through its input end, which it then places in its buffer, making it full. With its buffer full, this channel cannot accept input. When its buffer is full, this channel allows a get operation on its output end to succeed by offering it the data item in its buffer, after which its buffer becomes empty. As long as its buffer is empty, a get operation at its output end remains pending, because the channel has no data to offer.

Suppose Red wants to output some data. Then, Red issues a *put request* at its port. As soon as Green has also issued a *put request*, and Blue has issued a *get request*, the protocol synchronously (i.e., atomically) accepts the data produced by Red and Green, offers Green's data to Blue, and stores Red's data in the buffer of the *fifo1* channel. As long as this buffer remains full, the *fifo1* channel cannot accept any more data, which means further put requests by the producers will suspend until this buffer becomes empty. Upon the next get request by Blue, Blue receives the data from the buffer, which empties the buffer and returns the protocol to its initial configuration. Therefore, this protocol implements the informal specification that prescribes alternation.

Although we may think of such a protocol as message passing, the code that is generated by the compiler is (depending on the target) based on shared memory.

3.2.4 Compilation

The first step consist of isolating the computation that is done in each process. To this end, we create a Java class in `Processes.java` that contains the a method for each original process:

```
import nl.cwi.reo.runtime.Input;
import nl.cwi.reo.runtime.Output;

public class Processes {

public static void Red(Output<String> port) {
   while (true) {
      for (int i = 0; i < 30000000; ++i);
      String datum = "Hello, ";
      port.put(datum);
   }
}

public static void Green(Output<String> port) {
   while (true) {
      for (int i = 0; i < 50000000; ++i);
      String datum = "world! ";
      port.put(datum);
   }
}

public static void Blue(Input<String> port) {
   for (int k = 0; k < 10; ++k) {
      for (int i = 0; i < 40000000; ++i);
         String datum = port.get();
         System.out.print(datum);
      }
   }
   System.exit(0);
}
```

Note that the code of each Java method is completely independent of any other method, since no variables are explicitly shared. Synchronization and data transfer is delegated to put and get calls on output ports and input ports, respectively.

This way, we strictly separate computation from interaction, defined by the protocol.

In the next step, we declare the protocol by means of the Reo file called `main.treo`:

```
import reo.syncdrain;
import reo.sync;
import reo.fifo1;

// The main component
main() { green(a) red(b) blue(c) alternator(a,b,c) }

// The atomic components
red(a!String) { #JAVA "Processes.Red" }
green(a!String) { #JAVA "Processes.Green" }
blue(a?String) { #JAVA "Processes.Blue" }

// The alternator protocol
alternator(a,b,c) { syncdrain(a, b) sync(b, x) fifo1(x, c) sync(a, c) }
```

This Reo file defines the main component, which is a set containing one instance of each of Red, Green, and Blue processes, and an instance of the **alternator** protocol. The definitions of Red, Green, and Blue processes just refers to the Java source code from `Processes.java`. The definition of the alternator protocol is expressed using primitive Reo channels, which are imported from the standard library.

Before we can compile this Reo file into Java code, please first follow the instructions in *Installation* to install the Reo compiler. Next, change directory to where `main.treo` and `Processes.java` are located, and execute:

    reo main.treo javac Main.java java Main

These commands respectively

(1) compile Reo code to Java source code (by generating `main.java`),

(2) compile the generated Java source code to executable Java classes, and

(3) execute the complete program.

Since the alternator protocol defined in `main.treo` matches the informal specification, and since the generated code correctly implements the alternator procotol, the output now looks as follows:

```
Hello, world! Hello, world! Hello, world! Hello, world! Hello, world!
```

3.3 Tutorial

3.3.1 Components and interfaces

A Reo program consists of a number of *components* each of which interacts with its environment via a set of **nodes** that form its **interface**. A **node** is a means of *communication by exchange of values* between a component and other components (which collectively, form its environment).

The interface of a component is the only thing that it shares with its environment, and communication by exchange of values means components cannot exchange references or pointers: components do not share variables, data structures, or objects. Moreover, the nodes that form the interface of a component comprise the only means of communication between the component and its environment: a component has no means of communication or concurrency control other than blocking I/O operations that it can perform on its own nodes.

A component is either *atomic* or *composite*. A **composite component** consists of an expression in the language of Reo that specifies a composition of other components. See the section on *Composite components*, below.

An **atomic component** is one whose operational details are given elsewhere in some unknown language, e.g., a piece of hardware, or a computation expressed in some conventional programming language, such as Java. An atomic component may use each node in its interface exclusively either to acquire input, or to dispense output, but not both: the nodes in the interface of an atomic component are *unidirectional*. For clarity, we refer to unidirectional nodes as **ports**.

An atomic component may synchronize with its environment via *put* operations on its own output ports and *get* operations on its own input ports. A put or get operation may specify an optional time-out. A put or get operation blocks until either it succeeds, or its specified time-out expires. A put or get operation with no time-out blocks for ever, or until it succeeds.

Although specification of the internal details of an atomic component (e.g., its executable code) may be given in a language other than Reo, its existence and some information about the *externally observable behavior* of an atomic component must be specified in Reo, if this component participates in a Reo application. On *the outside*, Reo does not distinguish between atomic and composite components; therefore, interface specification of all components share the same Reo syntax. Whereas the body of the specification of a composite component uses Reo to define a composition (see below), the body of the specification of an atomic component uses one of a variety of concrete *sub-languages* to provide some information about the behavior of that atomic component. The syntax of each such sub-language is completely independent of the syntax of Reo or that of other sub-languages.This seperation makes is very easy to extend the compiler to support other sub-languages for atomic component specification.

In this document, we consider only two such sub-languages: the one for components programmed in Java, and a generic one for definition of externally observable behavior.

3.3.2 1. Reference to Java source code

We can define an atomic component by referring to its Java source code, as:

```
// buffer.treo
buffer<init:String>(a?String, b!String) {
   #JAVA "MyClass.myBuffer"
}
```

This code in buffer.treo (a Reo source code file) defines an atomic component called buffer with an input port **a** of type String, an output port **b** of type String, and a parameter `init` of type String. The Java-tag indicates that the body of this component definition uses the syntax of Java-component specification sub-language. This sub-language allows a programmer to provide a link to a piece of Java code that constitutes the implementation of this component. Because the target code is Java, the type tags, String, are optional. Unspecified type tags default to Object. The parameter block `<init:String>` is also optional, as we will see further below.

The Java sub-language interprets the provided reference as a link to a Java class that implements the `buffer` component as the Java method `myBuffer` in class `MyClass`.

```
// MyClass.java
import nl.cwi.reo.runtime.Input;
import nl.cwi.reo.runtime.Output;

public class MyClass {
   public static void myBuffer(String init, Input<String> a, Output<String> b) {
      String content = init;
      while (true) {
         b.put(content);
         content = a.get();
      }
   }
}
```

Note that the order of the parameters and ports in `buffer.treo` is identical to the order of the arguments of the static function `myBuffer`. Furthermore, the type tags of the parameter and ports correspond to the data types of the arguments: `a?String` corresponds to `Input<String> a`, `b!String` corresponds to `Output<String> b`, and `init:String` corresponds to `String init`.

Ensure that the current directory `.` and the Reo runtime `reo-runtime-java-1.0.jar` are added to the Java classpath. Then, we can compile the producer via:

> ls MyClass.java buffer.treo > reo buffer.treo -p "Hello world!" > javac buffer.java > java buffer

The `p` option allows us to speficy an initializing string for the buffer. Using commas in a string splits the string into multiple arguments. The last command runs the generated application and brings up two *port windows* that allow us to put data into the buffer and take it out of the buffer.

The current version of Reo can generate only Java code, and therefore, only Java components can be defined. It is only a matter of time before Reo can generate code for other languages, such as C/C++, and that components defined in these languages can be used in Reo applications as well.

Arguments of the Java function are automatically linked to the ports in the interface of its respective atomic component. Recall that ports are unidirectional nodes. The interface of a component specifies how that component uses each node in its interface using node-type tags. The exclamation mark (`!`) after a node name designates that node as an output port of its component. For instance, `b!` in the above example indicates that the Reo component `buffer` uses the node `b` as an output port. A question mark (`?`) after a node name designates that node as an input node of its component. For instance, `a?` in the above example indicates that 11buffer‘‘ uses `a` as an input port. A colon (`:`) after a node name `x` indicates that its component uses `x` both as input and as output. Because the interface of an atomic component can contain only (unidirectional) ports, colon-tag is not allowed in the interface of atomic components.

Observe that the above definition of `buffer` tells Reo only about this atomic component's *points of contact* with its (Reo) environment: an instance of `buffer` requires a string parameter for its instantiation, and has an input port and an output port through which it exchanges strings with its environment. The `#Java` sub-language in the body of this definition of `buffer` merely provides a link to its Java source code, `MyClass.myBuffer`, which defines the internal workings of this component. A Java programmer can read the source code of `MyClass.myBuffer` in `MyClass.java` and surmise that this component is indeed an asynchronous buffer with capacity to hold at most one string, which is initialized with the value of its parameter `init` on instantiation, whose behavior consists of an infinite loop, in each iteration of which the component attempts to `put` the content of its buffer through its output port `b`, and then obtain a string as its new buffer content by a `get` on its input port `a`. However, the environment of this component, specifically Reo, knows nothing about how `buffer` works: any possible relationship that in fact exists among its initial value, the values that `buffer` produces and consumes through its ports, and the relative order or timing of these actions is simply *hidden* inside the Java code and remains unknown to Reo.

3.3.3 2. Definition via externally observable behavior

The relations among the values that a component produces and consumes and their relative order constitutes the **externally observable behavior** of that component. Reo supports a number of sub-languages that a programmer can use to specify the externally observable behavior of an atomic component in the body of its definition. The *Rule-Based Automata (RBA)* specification language is one such sub-language that we consider in this document. Intuitively, RBA is a formal language for defining the transitions of constraint automata with memory in terms of sets of rules:

```
// buffer.treo
buffer<init:String>(a?String,b!String) {
   #RBA
   $m=init;

   b =*  , a!=* , $m = b, $m' = a
   b!=* , a =* , $m = b, $m' = a
}
```

The buffer atomic component defined here consists of a single memory cell m, whose initial value is given by init. If the buffer is empty ($m = * = a), it can perform an I/O operation on port a and blocks port **b** (indicated by the constraint b=* and a!=*) and assigns the observed value at **a** to $m' which designates the value of memory cell m in the next state of the component. If the buffer is full ($m = b != *), it can perform an I/O operation on port **b**, block port **a** (indicated by the constraint b!=* and a=*), assign the current value in memory cell m to port **b**, and clear the value of m ($m' = *).

We can define a component both as a Java component and in the RBA sub-language simultaniously:

```
// buffer.treo
buffer(a?,b!) {
   #JAVA "MyClass.myBuffer"
   #RBA
   $m=init;
   b =*  , a!=* , $m = b, $m' = a
   b!=* , a =* , $m = b, $m' = a
}
```

In this case, the Reo compiler treats the Java code as the definition of the component, and regards its RBA definition is used only as annotation. Although the current version of the Reo compiler simply ignores the RBA definition, future versions of the compuiler can use the constraint automaton defined by the RBA for analysis or monitoring tools to verify or detect properties like deadlock.

3.3.4 3. Composite components

A composite component consists of a set of components, specified as an intensionally or extensionally defined set of components that may communicate with one another via shred nodes. Consider the following simple example:

```
// buffer2.treo
buffer2(a?,b!) {
   buffer<"*">(a,x)
   buffer<"*">(x,b)
}

buffer<init:String>(a?String,b!String) {
   #RBA
   $m=init;
   b =*  , a!=* , $m = b, $m' = a
   b!=* , a =* , $m = b, $m' = a
}
```

This Reo program defines an atomic component, `buffer`, and a composite component, `buffer2`. Since Reo is declarative, the order of the definitions of `buffer` and `buffer2` is not important.

The body of the composite component `buffer2` consists of a set of component instances, in this case defined in extensional format (without the separating commas) within the pair of curly brackets. The component instances that comprise this set are two instances of another component, `buffer`, namely, `buffer<"*">(a,x)` and `buffer<"*">(x,b)`. Observe that the body of `buffer2` implicitly defines a new Reo node, `x`, which is local to the definition of `buffer2`, i.e., *hidden* from outside, as it is not exposed in the interface of `buffer2`.

The node `x` is shared between the two instances of the atomic component `buffer`, where the formal parameter ports `a` and `b` of `buffer` get substituted, respectively, by the actual parameters `a` and `x` (of `buffer2`) in the first instance, and by `x` and `b`, respectively, in the second instance. The signature of `buffer` shows that the node `x` serves as an output port in instance `buffer(a,x)`, while instance `buffer(x,b)` treats `x` as an input port. This sharing of `x` implies that `x` must be a mixed node.

A mixed node provides a **broadcast** mechanism for communication of the component instances that communicate

through it. a **single** *put/send operation* by one of the components that use a mixed node as an *output port* synchronizes with a *get/receive operation* by **all** components that use that mixed node as an *input port*. In the case of our simple example, here, every *put* operation by `buffer(a,x)` synchronizes with a *get* operation by `buffer(x,b)`.

Note: This broadcast communication mechanism should not be confused with broadcast communication as used by other models of concurrency. Usually a single send operation on a node A (also called a *channel* in the literature) synchronizes with multiple, but **arbitrary** number, receive operations on A.

Predicates

The definition of `buffer2` as a composition of two atomic buffer instances explicitly lists every one of its constituent component instances as in *extensional* specification of a set. More expressively, Reo also allows *intensional* set speficication for definition of composite components, using **predicates**.

The following example shows a an intesional set definition where the predicate `i : <0..1>` simply means that the variable `i` can assume integer values in the (inclusive) range of 0 and 1:

```
{ buffer(a[i],a[i+1]) | i : <0..1> }
```

Although the above definition is merely a declarative definition of a set, it sometimes helps to intuitively think of what it *means* in terms of a for-loop in an imperative programming language. This for loop *unfolds* to the composition expressed above into the set:

```
{ fifo1(a[0],a[1]) fifo1(a[1],a[2]) }
```

Additionally, predicates may contain variables. All variables used in predicates must be grounded during instantiation.

```
{ buffer(a,b) | x=1 }
{ buffer(a,c) | x!=1, x=2 }
{ buffer(a,d) | x!=1, x!=2 }
```

Terms

Besides the ordinary terms in predicates, such as 0, 1, `n` and `<1..n>`, we can also have component definitions as terms. For example,

```
section slides.main;

import reo.fifo1;
import reo.sync;
import reo.lossy;
import slides.variable.variable;
import slides.lossyfifo.lossyfifo1;
import slides.shiftlossyfifo.shiftlossyfifo;

import slides.main.red;
import slides.main.blue;
import slides.sequencer.seqc;

main11()
{
   { red(a[i]) | i : <1..n> }
   blue(b)
```

(continues on next page)

(continued from previous page)

```
   connector11<ileg[1..n], sync>(a[1..n], b)
|
   ileg[1..n] = <sync, lossy, fifo1, variable, shiftlossyfifo, lossyfifo1>
}

connector11<ileg[1..n](?, !), oleg(?, !)>(a[1..n], b)
{
   seqc(x[1..n])
   { ileg[i](a[i], x[i]) sync(x[i], m) | i : <1..n> }
   oleg(m, b)
}
```

3.3.5 Sections and Imports

In large application, it is likely that different component would get the same name. To be able to distinguish between the two components, we put the components in different sections. For example, we can put the `buffer` component defined above in a section called MySection by adding the statement `section mySection;` to the beginning of the file.

```
// buffer.treo
section mySection;

buffer(a?,b!) {
   Java: "MyClass.myBuffer"
   q0 -> q1 : {a}, x' == a
   q1 -> q0 : {b}, b == x
}
```

In other files, we can reuse this buffer by simply importing it as follows:

```
// other.treo
import mySection.buffer;

other() {
        buffer(a,b)             // #1
        mySection.buffer(a,b)   // #2
}
```

Option 1 is the simplest way to use an imported component, as it does not explicitly defines from which section it comes. However, if we imported two buffer components from different sections, then Option 2 allows us to be precise on which buffer we mean.

3.4 Library of components

We list all components written in the "./example/slide/" folder, and shortly explain their behavior.

3.4.1 1. Atomic components

We list a set of atomic components, and use a logical formula as semantic (see tutorial).

Sync

The sync channel atomically get data from input port a, to output port b.

```
// sync.treo
sync(a?String, b!String) {
   #RBA
   a=b
}
```

Fifo1

The fifo1 channel behaves as a buffer of size one, where a is its input port, and b is its ouput port. The internal memory $m is hidden.

```
// fifo1.treo
fifo1(a?String, b!String) {
   #RBA
   $m=*;
   a!=*, $m' = a, b=*, $m=*
   a!=*, $m' = a, b=*, $m=*
}
```

Syncdrain

The syncdrain channel ensures that data flows atomically at port a and b, without any constraint on the data.

```
// syncdrain.treo
syncdrain(a?String, b?String) {
   #RBA
   a!=*, b!=*
   a=*, b=*
}
```

Lossysync

The lossysync channel either get atomically data from the input a, to the output port b; or dropes the data from input port a.

```
// lossysync.treo
lossysync(a?String, b?String) {
   #RBA
   a!=*, b!=*, a=b
   a!=*
   a=*, b=*
}
```

Filter

The fileter channel

```
// filter.treo
filter<P:string>(a?, b!) {
        #RBA
        a!=*, P(a), a=b
        a!=*, b=*, !P(a)
}
```

3.4.2 2. Connector components

Regulatorwwr

The regulator write/write/read (regulatorwwr)

```
import reo.sync;
import reo.syncdrain;

regulatorwwr(a, b, c) {
  sync(a, m) syncdrain(m, b) sync(m, c)
}
```

Barrier

The barrier connector

```
// barrier.treo
import reo.sync;
import slides.regulatorwwr.regulatorwwr;

barrier(a, b, c, d) {
  regulatorwwr(a, m, c) sync(b,m) sync(m, d)
}
```

Alternator

The alternator connector

```
// alternator.treo
import reo.fifo1;
import reo.sync;
import reo.syncdrain;

alternator(a, b, c) {
  syncdrain(a, b) sync(b, x) fifo1(x, c)
  sync(a, c)
}
```

Circulator

The circulator connector

```
// circ1.treo
import slides.regulatorwwr.regulatorwwr;
import reo.syncdrain;
import reo.sync;

circ1(a?, b?, c?, d!) {
 regulatorwwr(a, x, d) sync(b, x) sync(c, x)
}
```

Ovflfifo

The over flow fifo connector

```
import reo.lossy;
import reo.fifo1;

ovflfifo(a,b){
        lossy(a,m) fifo1(m,b)
}
```

Regulatorwrr

The regulatorwrr connector

```
import reo.sync;

regulatorwrr(a, b, c) {
  sync(a, m) sync(m, b) sync(m, c)
}
```

Seqp

The sequencer producer connector

```
import reo.fifo1;
import reo.fifofull;
import reo.sync;

seqp(p[1..n]) {
  {
    fifo1(x[i], x[i+1])
    sync(x[i+1], p[i])
  |
    i : <2..n>
  }
  fifofull<"0">(x[1], x[2])
  sync(x[2],p[1])
  sync(x[n+1], x[1])
}
```

Seqc

The sequencer consumer connector

```
import reo.syncdrain;
import slides.sequencer.seqp;

seqc(p[1..n]) {
  seqp(x[1..n])
  { syncdrain(x[i], p[i]) | i : <1..n> }
}
```

Shiftlossyfifo

The shiftlossyfifo connector

```
import reo.sync;
import reo.fifo1;
import reo.fifofull;
import reo.syncdrain;
import slides.xrouter.xrouter;
import reo.sync;
import reo.syncdrain;
import reo.lossy;

shiftlossyfifo(in, out) {
  sync(in,a) fifo1(a, b) fifo1(b, c)
  xrouter (c,d,e)
  syncdrain(a,g) sync(d,f) sync(e,g) sync(f,out) fifofull<"0">(f,g)
}
```

Variable

The variable connector

```
import reo.sync;
import slides.shiftlossyfifo.shiftlossyfifo;

variable(a, b) {
  sync(a, x) sync(x, y) shiftlossyfifo(y, z)
  sync(z, b) sync(z, t)  shiftlossyfifo(t, y)
  sync(x, t)
}
```

Xrouter

The xrouter connector

```
import reo.sync;
import reo.syncdrain;
import reo.lossy;

xrouter(in, out[1..n]) {
```

(continues on next page)

(continued from previous page)

```
  sync(in, s) syncdrain(s, m)
  {lossy(s, x[i]) sync(x[i], m) sync(x[i], out[i]) | i:<1..n> }
}
```

3.4.3 3. Boundary components

Depending on your target language, you may want to compose your protocol with specific functions in say Java, C, Promela, Maude, ... We list some examples of boundary components for target languages currently supported.

3.1 Java

Producer

A producer component with a single output port, and a Java method as semantic.

```
//producer.treo
producer(a!String)
{
  #JAVA "Producer.produce"
}
```

The producer component is linked with the following class, where the datum ” Hello ” is produced 100 times. For more information on the runtime of ports, see runtime section.

```
import nl.cwi.reo.runtime.Input;
import nl.cwi.reo.runtime.Output;

public class Producer {

        public static void produce(Output<String> port) {
                for (int i = 1; i < 100; i++) {
                        port.put(" Hello ");
                }
                System.out.println("Producer finished.");
        }
}
```

Consumer

A consumer component with a single intput port, and a Java method as semantic.

```
//consumer.treo
consumer(a!String)
{
  #JAVA "Consumer.consume"
}
```

The consumer component is linked with the following class, where the method get() is performed 100 times, and the datume is displayed. For more information on the runtime of ports, see runtime section.

```
import nl.cwi.reo.runtime.Input;
import nl.cwi.reo.runtime.Output;

public class Consumer {

        public static void consume(Input<String> a) {
                for (int i = 1; i < 100; i++) {
                        System.out.println(a.get());
                }
                System.out.println("Consumer finished.");
        }
}
```

EvenFilter

An EvenFilter component instantiates a filter (see atomic filter component) with a boolean java method called Relation.Even .

```
import reo.filter;

filterJava(a?String,b!String){
        filter<"Relation.Even">(a,b)
}
```

The Java class Relation linked to the component contains two boolean methods: Even and Odd.

```
public class Relation {

        public static boolean Even(String datum) {
                if (datum == null) return false;
                return Integer.parseInt(datum) % 2 == 0;
        }

        public static boolean Odd(String datum) {
                if (datum == null) return false;
                return Integer.parseInt(datum) % 2 != 0;
        }

}
```

3.2 Promela

3.5 Contribute

This page describes how to contribute to the Reo compiler.

3.5.1 Clone the repository

Confirm with `git --version` that Git is installed on your system. Clone the git repository, and change directory into the `Reo` folder:

```
git clone https://kasperdokter@bitbucket.org/kasperdokter/Reo.git
cd Reo
```

Note: If you are not familiar with Git version control, please consult a tutorial on Git.

3.5.2 Build the project

This compiler project is build using Apache Maven, a software project management and comprehension tool.

Note: If you are not familiar with the Maven build tool, please consult a tutorial on Maven. Maven automatically takes care of all the dependencies of this project (such as the dependency on ANTLR4 for lexer and parser generation). This tool is therefore essential for smooth development of this single project by multiple developers, each using his/her own operating system.

Confirm with `mvn -v` that Maven is installed on your system. Otherwise (on Ubuntu), run command:

```
sudo apt-get install maven
```

to install the latest Apache Maven.

To build the project, run:

```
mvn install
```

This Maven command creates in every module (i.e., `reo-compiler/`, etc.) a folder called `target` that contains a Java archive with all compiled code of that particular module. Furthermore, it creates a self-contained Java archive `reo-compiler/target/reoc.jar` that contains the Reo compiler.

3.5.3 Documentation by Javadoc

To allow people other than yourself to understand what your code is supposed to do, it is extremely important to document your code. Since the Reo compiler is written in Java, we use Javadoc for documentation. Using Eclipse, you can easily generate a skeleton of the documentation using a plugin like JAutodoc. Once the documentation is in place, you can generate a static Javadoc website for the whole project by running:

```
mvn javadoc:aggregate
```

3.5.4 Setting up Eclipse

If you use Eclipse as development environment, run:

```
mvn eclipse:eclipse
```

to generate eclipse configuration files, such as `.project` and `.classpath`. Manually define the M2_REPO classpath variable in Eclipse IDE as follows:

1. Eclipse IDE, menu bar;
2. Select Window > Preferences;
3. Select Java > Build Path > Classpath Variables;

4. Click on the new button > define a new variable called `M2_REPO` and point it to your local Maven repository (i.e., `~/.m2/repository` in Linux).

3.5.5 Adding new semantics to the compiler

There exist more than thirty different semantics for Reo [JA12]. Although all these different kinds of semantics for Reo have a fully developed theory, not all of them are actually implemented in the Reo compiler. In view of the variety of Reo semantics, the Reo compiler has been build in a generic way that allows for easy extension to new Reo semantics.

This tutorial provides a step-by-step procedure that extends the current Reo compiler with a new type of semantics called `MyReoSemantics` (or `MRS` for short).

1. Implement your semantics as java objects.

   a. In the `reo-semantics` module, add an alternative `MRS` to the enum `nl.cwi.reo.semantics.SemanticsType`. Update the `toString()` method by adding a case

```
...
case MRS: return "mrs";
break;
...
```

Note: The value of the `toString()` method is used by the Reo compiler when it searches component definitions in on the file system.

   b. In the `reo-semantics` module, create a new package called `nl.cwi.reo.myreosemantics`, and add a class called `MyReoSemantics` that implements `nl.cwi.reo.semantics.Semantics` as follows:

```
package nl.cwi.reo.myreosemantics;

import nl.cwi.reo.semantics.Semantics;

public class MyReoSemantics implements Semantics<MyReoSemantics> { }
```

   If `MyReoSemantics` can be viewed as an extension of port automata with a particular type of labels on its transitions, then we can reuse the generic automaton implementation and instantiate it using our own type of labels on the transitions.

      i. Implement the transition label by creating a class `nl.cwi.reo.myreosemantics.MyReoSemanticsLabel` that implements the `nl.cwi.reo.automata.Label` interface. This interface requires you to implement how composition and hiding affects transition labels.

      ii. let the class `MyReoSemantics` extend the class `nl.cwi.reo.automata.Automaton` as follows:

```
package nl.cwi.reo.myreosemantics;

import nl.cwi.reo.automata.Automaton;
import nl.cwi.reo.myreosemantics.MyReoSemanticsLabel;
import nl.cwi.reo.semantics.Semantics;

public class MyReoSemantics extends Automaton<MyReoSemanticsLabel>
                implements Semantics<MyReoSemantics> { }
```

2. Design an ANTLR4 grammar for your semantics. For further details on ANTLR4, we refer to the manual [Parr13].

   a. In the folder `reo-interpreter/src/main/antlr4/nl/cwi/reo/interpret/`, ceate a grammar file `MRS.g4` that contains a rule called `mrs`:

```
grammar MRS;

import Tokens;

mrs : //...// ;
```

   b. Add an alternative `| mrs ;` to the rule of `atom` in the main grammar `Reo.g4` of Reo.

3. Implement an ANTL4 listener that annotates the parse tree with our `MyReoSemantics` classes.

   a. In the `reo-interpreter` module, create a class `nl.cwi.reo.interpret.listeners.ListenerMRS` that extends `nl.cwi.reo.interpret.listeners.Listener` as follows:

```
package nl.cwi.reo.interpret.listeners;

import org.antlr.v4.runtime.tree.ParseTreeProperty;

import nl.cwi.reo.interpret.listeners.Listener;
import nl.cwi.reo.myreosemantics.MyReoSemantics;

public class ListenerMRS extends Listener<MyReoSemantics> {

        private ParseTreeProperty<MyReoSemantics> myReoSemantics =
                        new ParseTreeProperty<MyReoSemantics>();

        public void exitAtom(AtomContext ctx) {
                atoms.put(ctx, automata.get(ctx.pa()));
        }
}
```

   b. In the root directory of this repository, run `mvn clean install` to let ANTLR4 generate a parser and a lexer for your new grammar.

   c. Go to the folder `reo-interpreter/target/generated-sources/antr4/nl/cwi/reo/interpret` that contains all classes generated by ANTLR4, and copy all (empty) methods from class `MRSBaseListener` to our listener class `ListenerMRS`. Replace all occurrences of `MRSParser.<rule>Context` with `<rule>Context` and import `ReoParser.<rule>Context`. For example:

```
package nl.cwi.reo.interpret.listeners;

import org.antlr.v4.runtime.tree.ParseTreeProperty;

import nl.cwi.reo.interpret.listeners.Listener;
import nl.cwi.reo.interpret.ReoParser.MrsContext;
import nl.cwi.reo.myreosemantics.MyReoSemantics;

public class ListenerMRS extends Listener<MyReoSemantics> {

        private ParseTreeProperty<MyReoSemantics> myReoSemantics =
                        new ParseTreeProperty<MyReoSemantics>();

        public void exitAtom(AtomContext ctx) {
```

(continues on next page)

(continued from previous page)

```
                atoms.put(ctx, automata.get(ctx.pa()));
        }

        public void enterMrs(MrsContext ctx) { }

        public void exitMrs(MrsContext ctx) { }

        /**
         * All other rules go here.
         */
}
```

d. Implement all other rules to eventually assign a `MyReoSemantics` object to the parse tree as follows:

```
public void exitMrs(MrsContext ctx) {
        //...
        myReoSemantics.put(ctx, new MyReoSemantics( ... ));
}
```

4. Implement an interpreter for your semantics by creating a class `nl.cwi.reo.interpret.InterpreterMRS` with the following implementation:

```
package nl.cwi.reo.interpret;

import java.util.List;

import nl.cwi.reo.interpret.listeners.ListenerMRS;
import nl.cwi.reo.myreosemantics.MyReoSemantics;
import nl.cwi.reo.semantics.SemanticsType;

public class InterpreterMRS extends Interpreter<MyReoSemantics> {
        /**
         * Constructs a Reo interpreter for MyReoSemantics.
         * @param dirs          list of directories of Reo components
         * @param params        list of parameters passed to the main Reo component
         */
        public InterpreterPA(List<String> dirs, List<String> params) {
                super(SemanticsType.MRS, new ListenerMRS(), dirs, params);
        }
}
```

5. Edit the `run()` method of the compiler by using your new interpreter InterpreterMRS as follows:

```
public void run() {
        ...
        Interpreter<MyReoSemantics> interpreter = new InterpreterMRS(directories,␣
↪params);
        Assembly<MyReoSemantics> program = interpreter.interpret(files);
        ...
}
```

3.5.6 Future work

Since this is a young project, many features are yet to be implemented:

- Animation
- Autocompletion of code
- Dynamic reconfiguration (by graph transformations)
- Exports (to BIP, ect.)
- Graphical editor
- Imports (from BPEL, UML sequence sequence diagrams)
- Model checking (via Vereofy and MCRL2)
- Simulation (of stochastic Reo connectors)
- Syntax highlighting

3.5.7 References

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Bibliography

[Parr13] Terence Parr. 2013. The Definitive ANTLR 4 Reference (2nd ed.). Pragmatic Bookshelf.

[JA12] Sung-Shik T. Q. Jongmans, Farhad Arbab: Overview of Thirty Semantic Formalisms for Reo. Sci. Ann. Comp. Sci. 22(1): 201-251 (2012)