Reo Documentation

Release 1.0

Kasper Dokter

Oct 22, 2019

Contents

1 What are protocols, and why do we care? 3

2 Reo compiler 5

3 Contents of this documentation 7

3.1 Installation 7
3.2 Introduction 7
3.3 Tutorial 12
3.4 Contribute 16

4 Indices and tables 23

Bibliography 25

Reo is an *exogenous coordination language* designed by prof. dr. F. Arbab at the Centre for Mathematics and Computer Science (CWI) in Amsterdam. Reo allows the programmer to specify protocols that describe interaction among components in a concurrent application.

CHAPTER 1

What are protocols, and why do we care?

Protocols, such as mutual exclusion protocol and producer/consumer, orchestrate interaction amongst concurrently executing processes. Unfortunately, most general purpose programming languages do not provide syntax for expressing these protocols. This absence requires programmers to implement their protocols by manually adding locks and buffers and ensuring their correct usage. Even if the implementation correctly resembles the intended protocol, the implicit encoding of the protocol makes it hard, if not impossible, to reason about its correctness and efficiency.

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

CHAPTER 2

Reo compiler

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

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 and run the reo-installer.
3. Go to your installation directory and follow the step written in the README.
4. Test the installation by running `reo`.

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 very simple program that repeatedly prints “Hello, ” and “world!” in alternating order. We split this program into three different processes: two producers that output a string “Hello, ” and “world!”, respectively, and a consumer that **alternately** prints the produced strings “Hello, ” and “world!”, 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:

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

public class Main {

        private static final Semaphore greenSemaphore = new Semaphore(0);
        private static final Semaphore redSemaphore = new Semaphore(1);
```

(continues on next page)

(continued from previous page)

```
        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 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 a token from 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 token, which allows other processes to operate on the buffer.

The same stategy is used to alternate the writes to the buffer. Each producer has its own semaphore. If a producer wants to write to a buffer, it first tries to acquire a token from its semaphore. After writing to the buffer, the producer hands over the token to the other producer.

3.2.2 Analysis

Let us now analyze the 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 of code:

3. Where is the protocol?

    a. What determines which producers goes first?

    This is determined by the initial value 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.

    b. 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 this third question is much more difficult to answer is because the protocol is **implicit**.

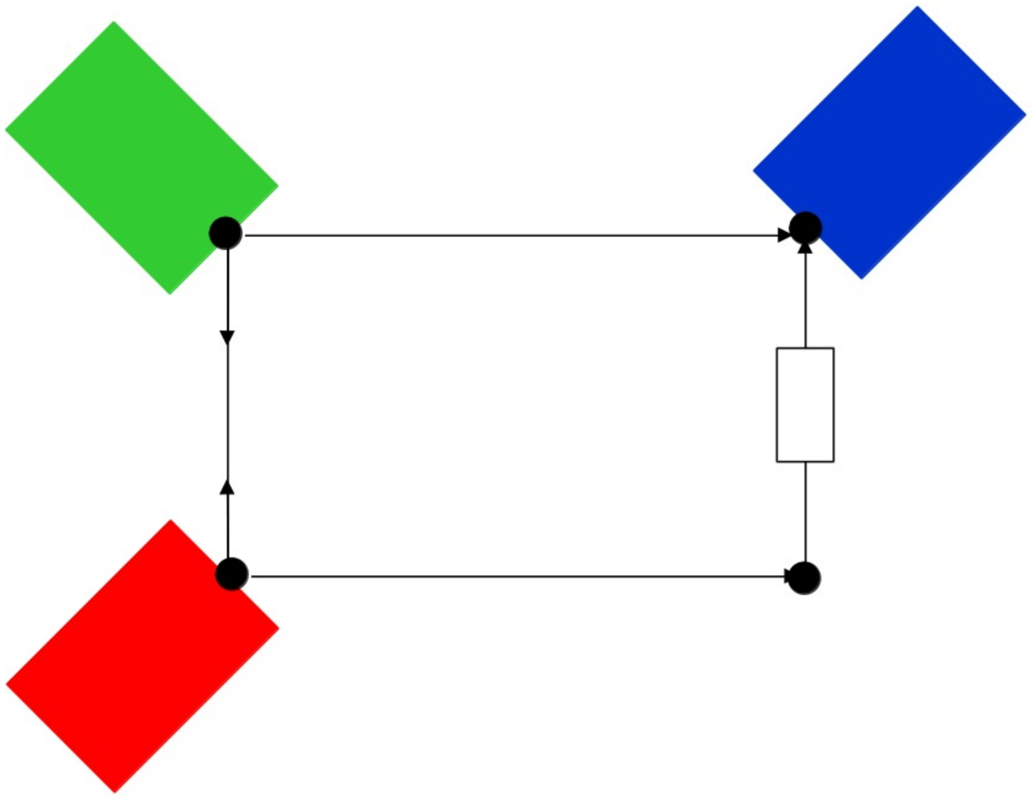
For such a simple program, you may argue that the fact that the protocol is implicit is not big deal. However, if you really think this, then you may be surprised by the output:

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

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 declare your protocol explicitly. 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 messages to the protocol. The protocol, then, synchonizes components and exchanges the messages.

The channel between Red and Green is a *syncdrain* channel that accepts data from both its input ends simultaneously, and then it looses the data. The channel between Red and Blue is a *sync* channel that atomically takes data from its input end and offers this data to its output end. The other incoming channel connected to Blue is a *fifo1* channel that stores a single data item that it receives at its input end. After the buffer became full, it offers this data to its output end. 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 issued a *get request*, the protocol synchronously accepts the data produced by Red and Green, offers Greens data to Blue, and stores Reds data in a buffer. Upon the next get request by Blue, Blue receives the data from the buffer, after which the protocol returned 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:

```java
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 to 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(a,b,c) { 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 an instance of the Red, Green, and Blue process, and an instance of the alternator protocol. The definition 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 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

Reo programs consist of *components* that interact via shared *nodes*. Each component has access to a given set of nodes, called its *interface*. To distinguish them from other nodes, the nodes that comprise the interface of a component are also called its *ports*. An atomic component uses each of its ports either for input or for output, but not both. Ports 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 ports. A component may synchronize with its environment via *put* operations on its own output ports and *get* operations on its own input ports. The put and get operations 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.

To define a component in Reo, we refer to Java source code and/or define its behavior as a constraint automaton, or, define it as a composition of other components.

3.3.2 1. Reference to Java source code

First of all, we can define a component by referring to Java source code.

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

This code in buffer.treo defines an atomic component called buffer with an input port a of type String, an output port b of type String, and an parameter init of type String. The Java-tag indicates that the implementation of this component consists of a piece of Java code. The type tags, String, are optional. On default, type tags are Object. The parameter block `<init:String>` is also optional, as we will see for the second possibility to define a component.

The provided reference links to a Java class that implements the producer component as a Java method myProducer 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 the initial string in 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 until Reo can generate code for other languages, such as C/C++, and that components defined in these languages can be defined.

Arguments of the Java function are automatically linked to the ports in the interface of its respective atomic component. The exclamation mark (`!`) indicates that the Reo component `producer` uses the node `a` as its output port. A question mark (`?`) after a node `a` in an interface indicates that its component uses `a` as an input node. A colon (`:`) after a node `a` indicates that its component uses `a` both as input and as output (this is not allowed for non-atomic components).

3.3.3 2. Definition of formal semantics

Second, we can define a component by defining its behavior in a formal semantics.

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

The buffer consists of a single memory cell m, whose initial value is given by init. If the buffer is empty ($m = null), it can perform an I/O operation on port a and blocks port b (indicated by the synchronization constraint {a, ~b}) and assign the observed value at a to the next value of memory cell $m'. If the buffer is full ($m != null), it can perform an

I/O operation on port b and block port a (indicated by the synchronization constraint {~a, b}) and assign the current value in memory cell m to port b, and clears the value of m ($m' = null).

We can define a component as a Java component and a constraint automaton with memory simultaniously:

```
// buffer.treo
buffer(a?,b!) {
   Java: "MyClass.myBuffer"
   #RBA
   $m=init
   {a, ~b} $m = null, $m' = a
   {~a, b} $m != null, b = $m, $m' = null
}
```

In this case, the Reo compiler treats the Java code as the definition of the component, while the constraint automaton with memory is used only as annotation. Although the current version of Reo simply ignores this annotation, future versions of can use the constraint automaton for tools like deadlock detection.

The syntax for constraint automata is completely independent of the syntax of the rest of the language. This seperation makes is very easy to extend the current language with other types of formal semantics of components.

3.3.4 3. Definition as composition

The most expressive way to define a component in Reo is via composition.

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

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

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

In the composite buffer2 component, we created implicitly a new Reo node x. This new node is local to the definition of buffer2, as it is not exposed in the interface.

This node is shared between two instances of the atomic buffer component, with a and b substituted by respectively a and x in the first instance, and by respectively x and b in the second instance. As seen from the signature of the atomic buffer component, instance buffer(a,x) writes to x, while instance buffer(x,b) reads from x. The two buffer instances communicate via shared node x using the **broadcast** mechanism: a *put/send operation* by a **single** component that uses node x as an *output node* synchronizes with a *get/receive operation* by **all** components that use node x as an *input node*.

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 is explicit in the sense that every subcomponent instance is defined directly. In this case, may can obtain the same construction using only one explicit instantiation using a **predicate**

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

This for loop unfolds to the composition

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

Although predicates are already expressive enough, we add some syntactic sugar for if-then-else and for loops. For example,

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

is equivalent to

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

and

```
if (x=1) { buffer(a,b) }
else (x=2) { buffer(a,c) }
else { buffer(a,d) }
```

is equivalent to

```
{ 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)
   connector11<ileg[1..n], sync>(a[1..n], b)
```

(continues on next page)

(continued from previous page)

```
|
   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 Contribute

This page describes how to contribute to the Reo compiler.

3.4.1 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.

3.4.2 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.

3.4.3 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
```

3.4.4 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).

3.4.5 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);
        ...
}
```

3.4.6 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

3.4.7 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)