Eclipse Coordination Tools User Manual

Christian Koehler Ziyan Maraikar

6 November 2008

Contents

1 Introduction

This article introduces the Eclipse Coordination Tools (ECT), a framework for developing component-based software using the coordination language Reo [1]. The framework consists of a set of integrated tools which are implemented as plug-ins for the Eclipse platform[1].

Currently, ECT provides functionality for the design, verification and execution of component-based applications. The channel-based coordination language Reo is used for defining the *glue code* between the components in a network. In Reo, so-called *connectors* constitute this glue code. Both, reasoning about these connectors as well as deriving executable implementations is based on compositional, formal models.

In this article, we describe how to use the core parts of the ECT. Our running example is a simple instant messaging application. For readability, we introduce the underlying formal models when needed and only on an informal level. The given references should be used as a further source of information.

Organisation. The remainder of this article is organised as follows. Section 2 introduces the graphical Reo editor and briefly describes the set of standard channels, that are support by the tools. Further it explains how to define networks of connectors and components. Section 3 discusses a network animation tool, which generates Flash animations. Section 4 contains an overview of a code generator for deriving centralized coordinator implementations. Finally, Section 5 contains conclusions and future work.

2 Reo editor

For specifying connectors and networks of connectors and components, ECT includes a graphical Reo editor. This editor also serves as a bridge to other tools, including the animation tool and the code generator, which we will discuss in detail in Section 3 and 4.

A screenshot of the graphical editor is given in Figure 1. The network shown in this screenshot will later serve as our running example. The palette on the right-hand side of the editor contains a number of tools for adding

[1] `http://www.eclipse.org`

elements to a Reo diagram. In the remainder of this section, we will explain the structure and the behaviour of these elements.

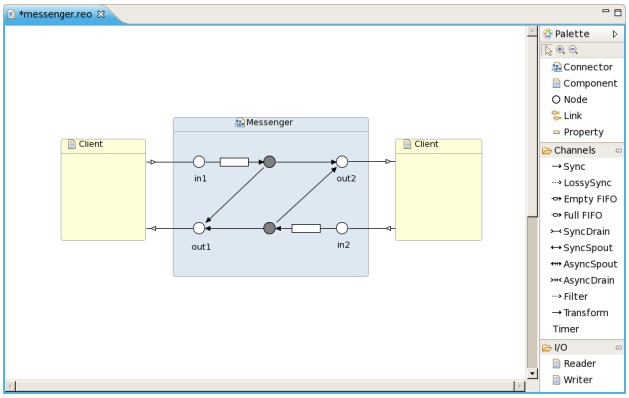

Figure 1: example network in the Reo editor.

2.1 Connectors and components

Connectors and components are the root elements in a Reo diagram. Connectors serve as containers for nodes and channels. On the other hand, components have no internal structure. They have a name and public interface, which consists of a set of input ports and output ports. Components may be further initialised using *properties*, which are basically key-value pairs. For verification purposes (see also Section 3), the editor includes *readers* and *writers* as special primitive components with predefined interfaces and a fixed behaviour.

Defining a network is done by connecting the boundary nodes of connectors with ports of components. These connections are called *links*. However, note that links are not channels–they just indicate that a port of a component is connected to a bounday node of a connector.

Messenger example. The example network in Figure 2 consists of one connector, called *Messenger* and to components, both called *Client*. Each of the clients has two ports, one input port and one output port. These ports are connected to the boundary nodes of the connector *Messenger*. To

identify the behaviour of the connector and the network on the whole, we first discuss the semantics of channels and nodes in the next section.

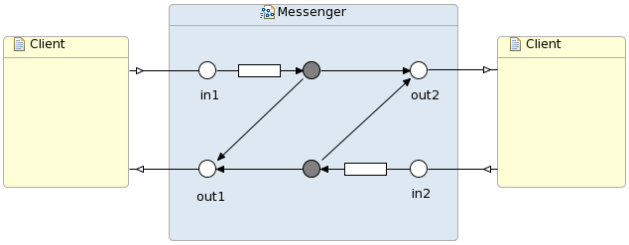

Figure 2: messenger circuit with two clients.

2.2 Channels and nodes

Channels are user-defined primitives in Reo. Their interface always consist of exactly two channel ends, which are either source ends or sink ends. Source ends accept data into and sink ends dispense data out of a channel. Even though channels are user-defined primitives, the Reo editor includes a number of standard channel types that are useful in a number of scenarios.

Standard channel types. The *Sync* channel synchronously reads data items from its source end and writes them to its sink end without buffering them. The *LossySync* behaves similarly, except that it does not block its source when its sink end cannot accept data. Instead, it accepts and loses the data item taken from the source. The *SyncDrain* has two source ends, and no sink end. If there are data items available at each of the source ends, this channel consumes both of them synchronously. On the other hand, the *AsyncDrain* accepts data items from one of its sources only if there is no data at its other end. The *SyncSpout* and *AsyncSpout* can be considered as the duals of these two channels. They both have two sink ends instead of two source ends. Accordingly, they produce data items instead of consuming them. The value of the data items is not further specified. Another directed channel is the $FIFO_1$. Unlike the other channels introduced so far, this channel is stateful, having a buffer of size one. If its buffer is empty and a data item is available at its source end, the I/O operation succeeds and the item is stored in the buffer. The channel blocks any further write requests

until the data item is delivered through its sink end. It then returns back to its empty state. All channels introduced up to now are behave independly from the content of the transferred data. Two data dependent channels are the *Filter* and the *Transform* channel. The former filters data items based on a regular expression, which is given as an additional parameter. The latter converts data items using a transformation rule parameter.

Reo nodes. Reo distinguishes between different kinds of nodes. Nodes where only source ends of channels coincide are called *source nodes*, nodes where only sink ends coincide are called *sink nodes*. Collectively, they form the *boundary nodes* of a connector, which interact with its environment. Nodes where both source ends and sink ends meet are called *mixed nodes*. Mixed nodes are internal and not accessible from the outside. In the editor and the animation tool, mixed nodes are represented as black circles and boundary nodes as white circles.

In contrast to channels, the behaviour of nodes is fixed in Reo. Source nodes replicate available data items to all coinciding source ends, which is why they are sometimes also referred to as *replicators*. On the other hand, sink nodes merge the input from their sink ends. Sink nodes accept data from one of their sink ends and prohibit data flow at all other sink ends at the same time. They are also referred to as (nondeterministic) *mergers*. Mixed nodes are a combination of a merger and a replicator. They merge data on the sink ends and replicate on the source ends. It is important to note here that nodes never buffer data.

Messenger example. The connector in our running example in Figure 2 has two source nodes and two sink nodes. Each of the clients is connected to one source and one sink node (respectively *in1,out1* and *in2,out2*). The idea is that the clients exchange text messages via these interfaces. The connector serves as glue code between the clients, implementing a simple instant messaging protocol. For each client, the messenger can buffer exactly one message in one of the $FIFO_1$ channels, until the client on the opposite side is ready to receive the message via the node *out1*/*out2*. At the same time, a copy of the message is also sent back to the sender to acknowledge the successful transfer. After that, the $FIFO_1$ buffers are empty again and new messages can be exchanged.

3 Network animation

To verify the behaviour of connectors, ECT includes an animation tool which generates Flash animations from Reo diagrams on the fly. These animations are generated from the so-called colouring semantics of Reo. In the first part of this section, we briefly introduce colourings as a compositional model for Reo, without giving formal definitions. We refer to [3] for a detailed discussion. In the second part, we illustrate the use of the colouring semantics for generating network animations.

3.1 Colouring semantics

The idea of the colouring semantics is to assign data flow colours to nodes in a connector. In the basic case, two colours are used to indicate the presence or absence of data flow. Every channel provides a list of valid colourings, which is called *colouring table*. To compute the colouring table of a connector, the colouring tables of all channels of the connector are composed using a join operation. This join operation basically merges colourings that are compatible with each other, i.e., which assign the same colour to common nodes. The resulting colouring table describes the data flow in the connector, or–if the components are included–of the complete network.

The colouring scheme with two data flow colours is a model that captures synchronisation and mutual exclusion constraints on the data flow. However, by adding a third colour, one can further express context-dependent behaviour that is required for a correct modeling of some primitives, such as the *LossySync* channel.

3.2 Generating animations

The animation tool in ECT is based on the colouring semantics with three colours. In the animations, the presence of data flow at a channel end is depicted as . The absence of data flow is represented by and . The animations are rendered directly from a Reo diagram. No extra information is required.

Messenger network. Two screenshots of an animation of the messenger network from before are depicted in Figure 3. Note that, since it is currently not possible to specify the semantics of components, the clients are modeled

using (independent) readers and writers here. Figure 3a) shows the first step of this animation. The client on the left-hand side sends a message which is then stored in the $FIFO_1$ buffer. The animation abstracts the text message to a token flowing through the channels. In the second step of the animation, the message is replicated at a mixed node. One copy is sent to the other client, the other is sent back to the first client to acknowledge the successful transfer. Note that, depending on the state of the clients and the connector, different animations are possible. However, the animation tool generates the complete state space and hence produces *all* possible animations, as long there are no loops involved. With loops, usually there are infinitely many possible behaviours. In this case, the tool has to make a choice which animations are 'interesting'. Alternatively, the animations can be executed stepwise. The user then has to decide which step should be executed next.

Figure 3: an animation of the messenger network.

4 Code generation

In this section we present a code generator and an interpreter that are implemented using Constraint Automata [2] (CA) semantics for Reo. We transform a Reo connector into a constraint automaton representing its behaviour. The CA is used to generate code for an executable state machine that yields an efficient centralised implementations, but, requires all the primitives of the connector need to be known apriori.

The framework also supports loading and interpreting constraint automata at runtime. This facility is useful for deploying Reo coordinators in environments where code loading may be restricted, such as Java application servers.

4.1 Constraint automata

Constraint automata semantics can be used to specify coordination when the complete Reo connector is defined. A transition in a constraint automaton is labelled with a set of *ports* and a *synchronisation constraint* on these ports.

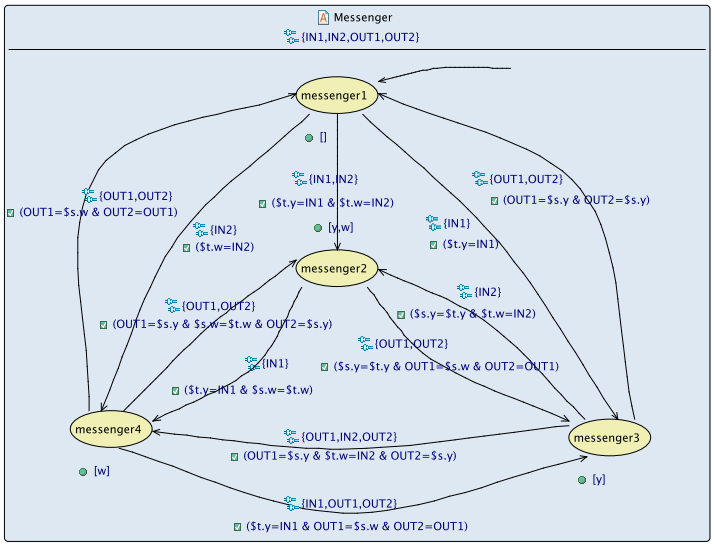

Figure 4: CA generated from Figure 1

Transforming a Reo connector to a CA is a two-step process. Each of the primitives (i.e. channels) and mixed nodes is first mapped to its CA representation. The product of these CAs gives a CA defining the complete behaviour of the Reo connector. Next, a *hide* operation is performed on the port names corresponding to primitive ends that are internal to the connector. Port names in the CA resulting from the hide operation correspond to the I/O links that attach components to source and sink nodes of the Reo connector. Note that the $-prefixed variables in Figure 4 are memory cells corresponding to FIFO channels in the connector.

4.2 Generating code from constraint automata

Code generation from finite automata is a well-known technique from language tokenisers like Lex, which can produce code for recognising a regular

language. The CA code generator uses this very same principle, specifying the set of ports that must be active to trigger a transition from a state. Once a transition is triggered, synchronous data transfer specified by the its constraint occurs between the synchronisation points corresponding to the port names.

Constraint automata port semantics are implemented by so-called *synchronisation points*, which are equivalent to Hoare's CSP channels. Synchronisation points can be implemented using common concurrency primitives such as mutexes and condition variables. Components communicate via `put` and `get` operations on synchronisation points. Both the generated code and the interpreter depend on a runtime library that implements these synchronisation points, using language-specific synchronisation primitives.

The code generator and interpreter support predicates in constraints needed to implement filter channels. It also permits adding Transformer channels — Sync channels with a data transformation function that acts on data flowing through it. This facility is used to implement the SABRE [4] web service mashup environment.

Code generation is implemented using the ANTLR[2] parser-generator and Stringtemplate[3] template engine. Currently the code-generator produces Java code, but the entire code generation framework is retargetable. Defining a new code generation target involves defining a set of code generation templates and porting the runtime library implementing SyncPoints. Support for C is currently being added.

4.3 Writing components

The coordinator code produced by the code generator can interface with external components written in the same language. The Java code generator expects components to implement the `cwi.ea.runtime.ReoComponent` interface shown in Listing 1. The code for the client components in Figure 1 is given in Listing 3 in Appendix A. In future components will also be able to use Web-service protocols like SOAP to commnicate with the coordinator in a language-independent fashion.

Listing 1: Java interfaces for components

```
public interface ReoComponent extends Runnable {
```

[2] `http://antlr.org`

[3] `http://stringtemplate.org`

```
    // Takes one or more Source ports and returns "this"
    public ReoComponent withSourcePorts(Source...  sources);

    // Takes one or more Sink ports and returns "this"
    public ReoComponent withSinkPorts(Sink...  sinks);
}

//A port corresponding to a source end of a Reo primitive.
public interface Sink<T> {
    T take() throws InterruptedException;
    boolean take(T o, long nanos) throws InterruptedException;
}

//A port corresponding to a sink end of a Reo primitive.
public interface Source<T> {
    void write(T data) throws InterruptedException;
    boolean write(T o, long nanos) throws InterruptedException;
}
```

In addition to the coordination code, the code generator also produces a `main` method to instantiate the coordinator and components, and wire them together as specified in the Reo editor. Listing 2 in Appendix A shows the wiring that is generated for the connector in Figure 1.

5 Conclusions

The Eclipse Coordination Tools provide functionality for developing component-based software, based on formal methods. The Reo coordination model serves as the language for implementing the protocols that orchestrate the autonomous components in a network.

Future work includes a comparison and integration of the underlying formal models, which will ensure the consistency between simulations and derived implementations. In the near future, we plan to integrate a model checker and a distributed implementation of Reo into the ECT. The integration with other modelling languages such as UML and BPMN is another branch of our research.

References

[1] Farhad Arbab. Reo: a Channel-based Coordination Model for Component Composition. *Mathematical Structures in Computer Science*, 14:329–366, 2004.

[2] C. Baier, M. Sirjani, F. Arbab, and J. Rutten. Modeling component connectors in Reo by Constraint Automata. *Sci. Comput. Program.*, 61(2):75–113, 2006.

[3] Dave Clarke, David Costa, and Farhad Arbab. Connector Colouring I: Synchronisation and Context Dependency. *Science of Computer Programming*, 66(3):205–22, 2007.

[4] Z. Maraikar, A. Lazovik, and F. Arbab. Building enterprise mashups with SABRE. In *Conf. on Service-Oriented Computing (ICSOC-08)*, LNCS 5364, pages 70–83. Springer, 2008.

A Code Generation Example

Listing 2: Example code for Client component in Figure 1

```
package messenger;

import cwi.ea.runtime.ReoComponent;
import cwi.ea.runtime.Sink;
import cwi.ea.runtime.Source;

public class Client implements ReoComponent{
    Source<String> out;
    Sink<String> in;

    @Override
    public ReoComponent withSinkPorts(Sink... sinks) {
        in = sinks[0];
        return this;
    }

    @Override
    public ReoComponent withSourcePorts(Source... sources) {
        out = sources[0];
        return this;
```

```
    }

    @Override
    public void run() {
        new Thread() {
            @Override
            public void run() {
                while(true)
                    try {
                        System.out.println("\t"+
                                in.take());
                    } catch (InterruptedException e) {
                        break;
                    }
            }
        }.start();

        String msg;
        do {
            msg = System.console().readLine();
            try {
                out.write(msg);
            } catch (InterruptedException e) {
                break;
            }
        } while(!msg.isEmpty()) ;
    }
}
```

Listing 3: Generated component wiring code for connector in Figure 1

```
package messenger;

import cwi.ea.runtime.primitives.TimeoutPort;
import cwi.ea.runtime.ReoComponent;

import cwi.ea.runtime.components.Writer;
import cwi.ea.runtime.components.Reader;


@SuppressWarnings("unchecked")
public class Main {
    public static void main(String[] args) throws Exception {
        TimeoutPort OUT1 = new TimeoutPort("OUT1"),
        OUT2 = new TimeoutPort("OUT2"),
        IN1 = new TimeoutPort("IN1"),
```

```
        IN2 = new TimeoutPort("IN2");

        ReoComponent client1 = new Client()
        .withSourcePorts(OUT1)
        .withSinkPorts(IN1);

        ReoComponent client2 = new Client()
        .withSourcePorts(OUT2)
        .withSinkPorts(IN2);

        Thread[] components = new Thread[] {
            new Thread(client1),new Thread(client2)
        };

        Thread coordinator = new Thread(new Messenger(OUT1,OUT2,IN1,IN2));
        coordinator.start();
        for (Thread _thr_:components)
            _thr_.start();

        for (Thread _thr_:components)
            _thr_.join();

        coordinator.interrupt();
    }
}
```