Modeling, Testing and Executing Reo Connectors with the Eclipse Coordination Tools[2]

F. Arbab, C. Koehler[1], Z. Maraikar, Y.-J. Moon, J. Proenca

CWI, P.O. Box 94079, 1090 GB Amsterdam, The Netherlands

Abstract

We present in this paper the Eclipse Coordination Tools (ECT), a set of visual tools for modeling, testing and executing Reo connector in the Eclipse development environment.

Keywords: Reo, Eclipse Coordination Tools

1 Introduction

Coordination models and languages are a way of dealing with the growing complexity of concurrent systems by separating the active parts performing the computations, i.e. components or services, from those parts that glue these pieces of software together, i.e. component or service connectors. The task of connectors is to coordinate the interaction of the components, by enforcing some kind of protocol from outside, which is also referred to as *exogenous* coordination. The coordination language *Reo* [2] instantiates these very abstract notions with a compositional channel-based calculus. As a counterpart to this formal framework we present in this paper the Eclipse Coordination Tools [1]: a set of integrated tools for Reo, implemented as plug-ins for the Eclipse[3] development environment.

The rest of this paper is organized as follows. We introduce Reo briefly in Section 2. An overview of the tools is given in Section 3. Conclusions and future work are presented in Section 4.

[1] Corresponding author, e-mail: christian.koehler@cwi.nl.

[2] The work in this paper is supported by NWO/GLANCE projects WoMaLaPaDIA and SYANCO.

[3] Eclipse platform: http://www.eclipse.org

2 Reo Overview

Reo is a channel-based coordination language wherein channels are primitive, user-defined entities that have two ends which can be either *sources* or *sinks*. Source ends accept data into and sink ends dispense data out of channels. Each channel type imposes its own constraints on the possible data flow at its ends, e.g. synchrony or mutual exclusion. For instance, the *Sync* channel (——►) synchronously reads data items from its source end and writes it to its sink end without buffering them. The *LossySync* (- - - - -►) behaves similarly, except that it looses the data when no one is ready to read it from its sink end. While these two channels are both unbuffered, the $FIFO_1$ (—□►) can store a single data item and release it again when someone tries to read from it. The *SyncDrain* (►——◄) has–unlike the channels introduced so far–two source ends and hence is not directed. If there are data items available at each of its ends, this channel consumes both of them synchronously.

Channels can be composed by joining them into nodes. Such a composite structure is called a *connector* in Reo and has some similarities with electrical circuits. As mentioned before, channels are user-defined. However, nodes have a fixed behaviour in Reo. A node atomically does a destructive read at one of the sink ends and replicates the data item at all of its attached source ends. With this simple behaviour of nodes and a small set of basic channels, complex protocols can be implemented compositionally, i.e. by just plugging together channels and nodes.

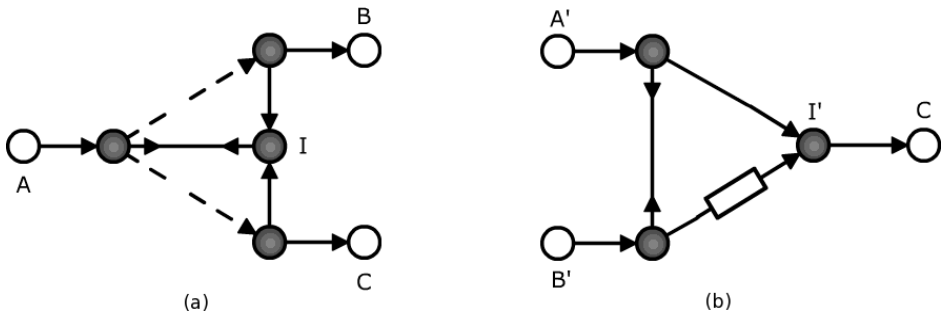

Fig. 1. a) *exclusive router*, b) *ordering* connector.

For instance, the *exclusive router*, shown in Fig. 1a, routes data from A to either B or C. If both, B and C, are able to receive data the choice is made nondeterministically. The mutual exclusion that is implemented by this connector is due to the fact that the internal node I allows input only from exactly one of its sink ends. The second connector, shown in Figure 1b, imposes an ordering on the data flow from the input nodes A' and B' to the output node C'. The *SyncDrain* synchronises the two inputs and the $FIFO_1$ delays the items from B'.

The minimalistic set of core concepts that Reo introduces is surprisingly powerful. A number of formalisms have been invented to model the different facets of Reo connectors (see, e.g., [3,4,5]). These formalisms are not only nice from the theoretical point of view, but they further serve as a strong basis for implementing Reo. This includes tools for the modeling and verification of connectors (i.e. interaction protocols) as well as actual runtime engines.

3 Tools

The tool suite for Reo consists of development tools for designing and verification of connectors on the one hand and runtime engines for multiple platforms on the other. The development tools are implemented as plug-ins for the Eclipse platform. Front-ends of these plug-ins are referred to as *views* in following. We now introduce the core tools separately and give references for further reading where possible.

Editor

One of the key tools in our framework is the graphical Reo editor, which is depicted in Fig. 2. It not only allows us to design connectors by simple drawing operations, but it serves also as a bridge to a number of other tools that can be either invoked from the context menu or directly interact with it. Reo files are saved using an XML format. The model classes and the parser for this format can be used also in stand-alone applications which is possible for most of the tools presented here.

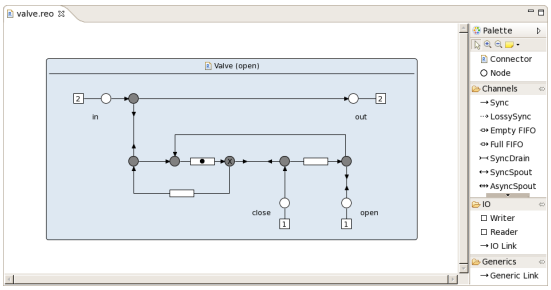

Fig. 2. *Valve* connector in the Reo editor. This connector models a synchronous communication channel that can be opened and closed by a third party.

Animation

The *animation view* generates Flash animations of connectors on-the-fly. Fig. 3 depicts the connector that was shown in the editor before in the animation view. The parts of the connector with synchronous data flow are highlighted. Tokens move along these synchronous regions. On the left side there is a list of possible animations for this connector and the attached writers and readers (white boxes), modeling the environment of the connector. The underlying formalism for the synthesis of such animations is an extension of the colouring semantics [5], introduced by Clarke et al.

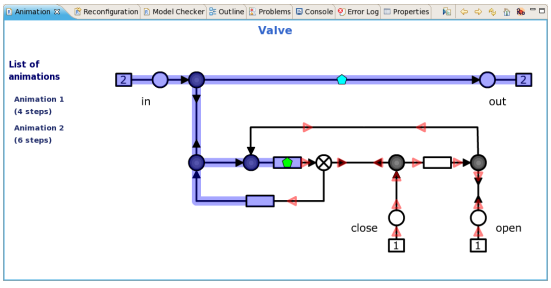

Fig. 3. *Valve* connector in the animation view. In this state, the communication channel is open and data flows through it.

Model checker

For model checking, connectors are translated to a semantic model, called Constraint Automata [3], which elegantly describe the synchronisation performed by connectors depending on their internal states. Moreover, these automata are compositional in the sense that it is possible to derive the automaton of a connector by composing the automata of its constituent channels. The core automata tools include a graphical editor and a conversion tool to generate them from connectors.

Our model checker uses a symbolic model and a CTL-like logic as specification format. Properties like deadlock-freeness and behavioural equivalence of connectors can be verified as well. The model checker for Reo has been implemented by the group of Christel Baier at the Technical University of Dresden [7] and is integrated into the rest of the ECT as a *model checker view*.

Performance analysis

Another semantic model for connectors is the Quantitative Intentional Automata (QIA), which are an extension of Constraint Automata with quantitative properties, such as arrival rates at ports and average delays of dataflows between ports. For performance analysis, these automata are translated to Continuous-Timed Markov Chains [6] and fed into the PRISM model checker [9]. With this approach, we can reason about properties like average response times and meetings of deadlines, and further analyse worst/best case scenarios. Moreover, the QIA model is compositional and the generated Markov Chains are very compact, enabling an efficient model checking. As an example, the QIA of the *Ordering* is shown in Fig. 4.

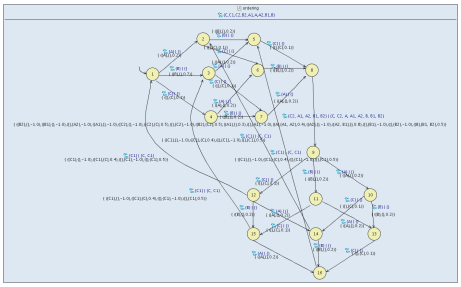

Fig. 4. Quantitative Intentional Automata of the *Ordering* connector.

Code generation

Constraint automata are not only used for model checking, but they also serve as the basis for generating executable Java code. Similar to the model checker, the code generation is a two-step process. First, the connector is translated to a constraint automaton. Then, the automaton is used to generate a Java state-machine *coordinator*. A runtime library implements constraint automaton ports as blocking zero-length buffers. Additionally, a graphical wiring editor lets users connect Java components to the coordinator by pairing corresponding ports together. Although the existing code-generator is limited to Java, a new retargetable generator capable of generating C, BPEL, etc. is currently under construction. The new framework

also supports loading and interpreting constraint automata at runtime, which is useful for deploying Reo coordinators in Java application servers etc.

Distributed implementation

A distributed Reo engine allows for the creation of Reo primitives within that engine, and can be connected to primitives in other distributed Reo engines. This is performed automatically by allowing a single node to be scattered in more than one engine. The synchronisation aspect inherent in Reo requires these primitives to play an active role when searching for an agreement on the global behaviour of the distributed connector. The *deployment view* provides the possibility of interacting with the Reo engines, deploying channels on different engines, and reconfiguring deployed systems. Higher level operations, such as the discovery of the best engine for each subpart of a Reo connector, are still out of the scope of this plug-in.

Reconfiguration

We follow an approach to modeling reconfigurations, i.e., changes in the topology of Reo connectors using algebraic graph transformation [8]. The reconfiguration model and engine are already implemented together with a basic *reconfiguration view*. Up to now, reconfiguration rules are used only inside the editor and for dynamic creation of connectors from templates. An integration with the distributed engine is planned in our future work for dynamic reconfiguration triggered by data flow.

4 Conclusion

The Eclipse Coordination Tools is a powerful tool suite for Reo. Key features are its versatility and strong underlying formal models. Future work includes a formal comparison and integration of these models, the further development of the existing tools and an integration with modelling languages such as UML and BPMN.

References

[1] *Eclipse Coordination Tools*, http://www.cwi.nl/ koehler/ect.

[2] Arbab, F., *Reo: a Channel-based Coordination Model for Component Composition*, Mathematical Structures in Computer Science **14** (2004), pp. 329–366.

[3] Arbab, F., C. Baier, J. Rutten and M. Sirjani, *Modeling Component Connectors in Reo by Constraint Automata*, Science of Computer Programming **61** (2006), pp. 75–113.

[4] Arbab, F. and J. Rutten, *A Coinductive Calculus of Component Connectors*, in: *Proceedings of WADT*, 2002, pp. 34–55.

[5] Clarke, D., D. Costa and F. Arbab, *Connector Colouring I: Synchronisation and Context Dependency*, Science of Computer Programming **66** (2007), pp. 205–22.

[6] Hermanns, H., "Interactive Markov Chains," Lecture Notes in Computer Science **2428**, Springer, 2002.

[7] Klüppelholz, S. and C. Baier, *Symbolic model checking for channel-based component connectors*, Electronic Notes in Theoretical Computer Science **175** (2007), pp. 19–37.

[8] Koehler, C., A. Lazovik and F. Arbab, *Connector Rewriting with High-Level Replacement Systems*, in: *Proceedings of FOCLASA'07*, Electronic Notes in Theoretical Computer Science, 2007, pp. 77–92.

[9] Kwiatkowska, M., G. Norman and D. Parker, *PRISM: Probabilistic symbolic model checker*, in: T. Field, P. Harrison, J. Bradley and U. Harder, editors, *Proc. of TOOLS'02*, LNCS **2324** (2002), pp. 200–204.