Constraint-based analysis of business process models

Changizi, B.

3

Reo Coordination Language

3.1 Introduction

In the realm of service-oriented programming that is a current trend in software development, the behavior of a software system is not only defined by the functionalities of its underlying services, but also in terms of their interactions. The code written to realize the latter is often referred to as *glue code*.

Writing and maintaining glue code is a tedious task, especially in complex systems wherein the size and rigidity of the glue code tend to increase over time. This makes these systems hard to modify and maintain. Coordination languages offer a more manageable alternative for generating glue code.

Reo [Arb04] is a channel-based coordination language for composition of software components and services. Using a small and open-ended set of predefined and user-defined constructs, Reo supports modeling of complex coordination behavior in terms of synchronization, buffering, mutual exclusion, priority, etc.

The primitive constructs of Reo are *channel*s. Each *channel* has two *end*s, also called *port*s. Channel ends are either of type *source* that read data into the channel or *sink* that write the channel's data out.

Channels can connect to each other on their ends to form compound elements. Reo *connector*s, also called *network*s are constructed this way. A Reo *node* is formed by one or more channel ends.

Furthermore, Reo provides a mechanism for hierarchical modeling and abstracting from inner structures by means of *component*s [Arb04]. A connector can turn into a *component*. In this case it will exhibit (part of) its inner logic as an observable behavioral interface.

Reo emphasizes on the connectors and their compositions rather than the entities that connect to the connectors to coordinate with each others. A Reo connector imposes a specific coordination pattern on interactions occurring between entities. This happens without the entities controlling or being necessarily aware of this pattern. This type of coordination is called *exogenous*, as it is performed from the outside.

According to a survey of coordination languages [Arb06], Reo belongs to the class of dataflow-oriented coordination languages, which is between the data-oriented and the control-oriented classes.

While the main concern of data-oriented coordination languages is consistency among shared data, control-driven languages focus on the flow of control. In comparison, dataflow-oriented languages define the communicating entities, the points of data-flow, and exchanging data-items.

3.2 Reo

In this section, we present an informal overview of the pre-defined set of Reo constructs. Following is the list of Reo channels:

A *sync* channel has a source and a sink end. It accepts data from its source end iff it can dispense it simultaneously through its sink end.

A *lossySync* has a source and a sink end. It reads a data-item from its source end and writes it simultaneously to its sink end. If the sink end is not ready to accept the data-item, the channel loses it.

A *syncDrain* has two source ends and no sink end. It reads data through its two source ends iff both ends are ready to interact simultaneously. The channel discards the received data-items.

A *syncSpout* has two sink ends and no source end. For each sink end, the channel generates a data-item out of the underlying data domain and writes them simultaneously to the corresponding ends.

An *asyncDrain* has two source ends and no sink end. It accepts and discards a data-item from either of its source ends that offers data. If both ends offer data-items simultaneously, the channel chooses one of the ends non-deterministically.

A *blockSourceSync* channel is a *Sync* channel that blocks the propagation of priority from its source end toward the sink end.
This channel and the two next priority blocking channels are used to limit the scope affected by priority, which originates from a *PrioritySync* channel.

A *blockSinkSync* channel is a *Sync* channel that stop spreading of priority from its sink end toward the source end.

A *blockSync* channel is a combination of *BlockSourceSync* and *BlockSinkSync*. It stops the propagation of priority in both directions.

The following is a list of pre-defined Reo components that are abstracted connectors.

A *replicator* has one source end and one or more sink ends. It replicates data-items coming from its source to its sink ends simultaneously.

A *merger* has one or more source ends and a sink end. It chooses one of its source ends that is ready to communicate in a non-deterministic way, receives the incoming data-item, and writes it to its sink end simultaneously.

A *router* has one source end and one or more sink ends. It accepts a data-item from its source end and simultaneously replicates it on one of its sink end that is non-deterministically chosen from its set of sink ends, which are ready to accept data.

A *cross-product* has one or more source ends and a sink end. It accepts a data-item from each of its source ends. Furthermore, it forms a tuple of the data-items that are set in the counter-clockwise order with respect to the sink node. It writes the tuple on its sink end. All of these operations occur simultaneously.

As mentioned, Reo nodes are created from channel ends. In case that the node only consists of source ends, it is called a *source* node. A node is *sink*, if it is formed by merely sink ends. Otherwise, if a mixture of source and sink ends collide, the created node is called a *mixed* node.

A mixed node is an atomic combination of a replicator and a non-deterministic merger. Each read and write action needs all of its involved source and sink ends to be able to interact synchronously. Otherwise, the action cannot take place.

3.3 Examples

Example 3.3.1 *Figure 3.3.1 shows a Reo network that is composed of a* lossySync *and a* FIFO_1 *channel. When the* FIFO_1 *channel is empty, the* lossySync *reads a value from its source end and passes it to its sink end that coincides with the source end of the* FIFO_1 *channel. Therefore, the* FIFO_1 *channel becomes full. The data stored in the* FIFO_1 *channel can be read and consumed via its sink channel. Before that the* FIFO_1 *channel loses its data, the* lossySync *channel accepts but loses all its incoming data.*

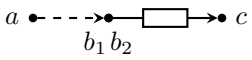

Figure 3.3.1: An example of a context-dependent Reo network

Example 3.3.2 *Figure 3.3.2 depicts a Reo network consisting of two* filter *channels with negating conditions. The first channel reads a data item from its source end and writes it on its sink end if it matches its condition, otherwise it loses the data. In the former case, the data item will not satisfy the condition corresponding to the second channel, so it is lost by the second channel. In both cases, there won't be any write operation on the sink end of the second channel.*

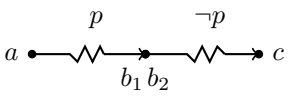

Figure 3.3.2: An example of a data-aware Reo network

Example 3.3.3 *Figure 3.3.3 illustrates a Reo network containing two* FIFO_1 *channels. The network behaves as a* FIFO_2 *buffer. In the beginning, both channels are empty. If there is an incoming data item on the source end of the first channel, the*

channel accepts the data and becomes full. Then, by an internal transition the data item is moved to the second channel. It makes it possible for the first channel to read another data item and/or to writes out the stored data through the sink end of the second channel.

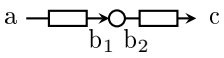

Figure 3.3.3: A Reo network for a $FIFO_2$ buffer

3.4 Extensible Coordination Tools (ECT)

A variety of Reo related tools are bundled together in a common framework, called Extensible Coordination Tools (ECT) [AKM+08a]. The tools in the framework are integrated as Eclipse plugins and operate based on the operational semantics of Reo, most notably, connector coloring and variations of constraint automata. ECT includes tools to design, transform, animate, model check, test, perform QoS analysis, and generate executable code from Reo connectors.

The ECT tools can be chained together to enable analysis on business process models. Here, we briefly overview these tools:

- *Graphical editor*: The graphical editor provides facilities to design Reo networks. The editor has been implemented based on the Eclipse Modeling Framework (EMF) [SBPM09] and Eclipse Graphical Modeling Framework (GMF). As a requirement of the model-driven approach and to work with EMF, Reo meta-model has been defined in [Kra11] [KMLA11].

- *Animation tool*: The animation tool produces simulation of Reo networks in the format of Adobe Flash [fla]. The tool is based on the animation semantics introduced in [Cos10] and visualizes the token game in Reo connectors [Kra11].

- *Verification tool*: Vereofy [BBK+10] is a model checker for Reo networks developed at the Technical University of Dresden. It can be used independently or from the ECT.

- *mCRL2 conversion tool*: Another model checker for Reo networks that is integrated into ECT is the mCRL2 [GMR+06]. The mCRL2 to Reo converter tool translates constraint automata specifications of Reo into mCRL2 specifications.

- *Execution engines*: ECT includes two execution engines: i) The centralized execution engine of Reo is a code generation framework based on constrained automata [BSAR06]. ii) The distributed execution engine for Reo is implemented based on constraint-based semantics of Reo [Pro11].

- *The Extensible Automata (EA) framework*: Extensible Automata (EA) framework is a unified framework for generating automata-based semantics of Reo networks. The framework comes with a graphical automata editor, which also can be used outside of the context of Reo. It includes functionality to generate automata models with stochastic information from graphical Reo models. From these models, it is possible to extract Continuous Time Markov Chains (CTMCs) that can be analyzed by the external tools such as PRISM probabilistic model checker [KNP02] or ECT stochastic simulation tool [Kan10].

- *BPMN 2 to Reo conversion tool*: In the context of this thesis, we have implemented a plugin to convert BPMN 2 models into Reo connectors [CKA10]. The converter deals with transactions, whose behavior is relatively more complex to map, in a two phases manner.

  The first phase is refinement, wherein transactions are substituted by a group of BPMN 2 elements, which collectively presents the same behavior as the transaction, yet they are easier to be mapped to Reo. In the second phase, the BPMN 2 constructs are being matched against some patterns to generate corresponding Reo elements. Chapter 5 elaborates on the converter.

- *Constraint-based semantics calculator*: As part of this thesis, we have implemented a tool to generate data-dependent, context-sensitive, and priority-aware formal semantics of Reo. To generate the automata-based formal semantics of Reo networks, we express the behavior of the Reo network in term of constraint satisfaction problem. From the solutions to this problem, we build the automata model.

  Our approach in using constraint solving to get the semantics of a Reo network is similar to the one used to generate the distributed execution engine for Reo [CPLA10]. However, unlike [CPLA10] [Pro11] , we support data, time, and priority. Another difference is that we calculate the all the possible behavior, while the mentioned tool has a step-wise approach that find the next possible behavior at a time. In Chapter 6, we present our approach in details.

  Our work is the first tool support for priority in Reo. Chapter 7 elaborates on our approach in obtaining a priority-aware formal semantics of Reo from the

solutions of constraints generated from each of Reo elements in a compositional manner.