Specification, Simulation, and Verification of Component Connectors in Reo

MohammadReza Mousavi[1], Marjan Sirjani[2,3], Farhad Arbab[2,4]

[1] Eindhoven University of Technology,
Eindhoven, The Netherlands

[2] Center for Mathematics and Computer Science (CWI),
Amsterdam, The Netherlands

[3] Sharif University of Technology,
Tehran, Iran

[4] Leiden University,
Leiden, The Netherlands

Abstract

Coordination and composition of components is an essential concern in component-based software engineering. In this paper, we present an operational semantics for a component composition language called Reo. Reo connectors exogenously compose and coordinate the interactions among individual components, that unawarely comprise a complex system, into a coherent collaboration. The formal semantics we present here paves the way for studying the behavior of component composition mechanisms rigorously. To demonstrate the feasibility of such a rigorous approach, we give a faithful translation of Reo semantics into the Maude term rewriting language. This translation allows us to exploit the rewriting engine and the model-checking module in the Maude tool-set to symbolically run and model-check the behavior of Reo connectors.

1 Introduction

Component-based software development has been proposed as a means to tackle the increasing complexity of software development [16, 22]. Components are assumed to be separate independent units of functionality and deployment out of which complete applications can be constructed using a mechanism for component composition.

An important aspect of component composition is that a piece of connecting code must match different requirements of the composed components. This piece of code is often referred to as *glue code*. The complexity of the glue code in a system can range from simple synchronization and ordering primitives to complicated distributed coordination protocols. It is often necessary to be able to specify and design these connecting devices and analyze and reason about their behavior individually, as well as in orchestration with (abstract) behavioral models of components. Little has been done in this direction and component connectors are usually left unspecified or under-specified using textual or graphical notations without a precise semantics.

Reo [4, 3] addresses this problem. Reo offers an expressive model and a graphical language for building coordinating component connectors by composition of primitive channels. It can be used to model the behavior of such connectors and to formally reason about them. Because the constructed Reo circuits directly constitute the so-called glue code, once proven correct, they can be readily used as connectors in a system. Thus, using Reo enables a correct-by-construction method for building component connectors.

In this paper, we specify a formal semantics for Reo in Plotkin's style of Structural Operational Semantics (SOS) [19]. Using this style of semantics, we benefit from the results of research and tools available for SOS. To show the usefulness of our semantics, we have implemented it in the rewriting logic language of Maude [1]. This implementation paves the way for symbolic execution of connectors specified in Reo and further on, model checking of their properties using Linear Temporal Logic (LTL).

The rest of this paper is structured as follows. In Section 2, we define the syntax and the semantics of a subset of Reo, as well as notions of equality and refinement for Reo connectors. In Section 3, we generalize our modeling framework by presenting a generic way of defining components and connectors. In Section 4, we present our implementation of Reo in Maude together with a few examples of our experiments with this implementation. Subsequently, we compare our approach to other related approaches for modeling component connectors and elaborate on other existing semantics for Reo in Section 5. Finally, Section 6 concludes the paper by summarizing our contributions.

The implementation code in Maude and its accompanying documentation with several examples are available at `http://www.win.tue.nl/~mousavi/reo_maude.tar.gz`.

2 Reo: Syntax and Operational Semantics

Reo [3] is a channel-based exogenous coordination model wherein complex coordinators, called *connectors*, are compositionally built out of simpler ones. The basic connectors in Reo, called channels, have well-defined behavior supplied by users. Components can instantiate, compose, connect to, and perform I/O operations through connectors.

Reo's notion of channel is far more general than its common interpretation and allows for any primitive communication medium with exactly two ends. The channel ends are classified as *source* ends through which data enter and *sink* ends through which data leave a channel. In a composed connector, internal or *hidden nodes* result from juxtaposing a source end of one channel on a sink end of another. Reo allows for an open-ended set of channel-types with user-defined semantics, each with different characteristics for ordering, synchronization, buffering, computation, and data-loss. In this paper, we use the notion of *basic connector* which is a generalization of the notion of channel in Reo, in that it allows for a set of (thus possibly more than one) nodes at each end (while Reo channels have only one node at each channel-end). [1] For our purposes in this paper, we restrict ourselves to the basic connector types described in Section 2.1.

Reo connectors are constructed in the same spirit as logic and electronics circuits: take basic elements (e.g., wires, diodes, and transistors) and compose them to build a circuit. A complex connector has a graphical representation, called a *Reo circuit* (also called a *system* in this paper), which can be produced by applying certain composition operators. A Reo circuit coordinates the data-flow through its basic connectors which interconnect the input/output ports of some components. In this paper, we do not consider the dynamic creation, composition, and reconfiguration of connectors by components that is an inherent aspect of Reo. We restrict our attention to connectors that have static graphical representations as Reo circuits.

Nodes constitute an important logical concept in Reo and they should not be confused with components or locations. Nodes may move around and reside on various physical locations in Reo, thus providing a basic and natural notion of mobility. However, we do not deal with mobility in this paper. Intuitively, a circuit itself can also be considered as a component, wherein its source nodes correspond to the input ports, and its sink nodes to the output ports of a component, while

[1] To be precise, our notion of basic connector is the same as the notion of channel in Reo with the addition of two connectors: *Fork* and *Merge*. These two connectors model replication of data items and choosing a data item among several available ones in Reo nodes, respectively. This addition simplifies the given semantics in that nodes are reduced to connecting points rather than pumping and choice points. Nevertheless, for every Reo circuit in the original presentation, there exists a simplified Reo circuit with Fork and Merge connectors and vice versa.

$$\begin{array}{lcl} \mathit{Sys} & ::= & \{\mathit{BCI}\} \mid \mathit{Sys} \cup \mathit{Sys} \\ \mathit{BCI} & ::= & \langle \mathit{NodeSet}\ \mathit{BCT}\ \mathit{NodeSet} \rangle \\ \mathit{NodeSet} & ::= & \emptyset \mid \mathit{NodeSet} \cup \{\mathit{Node}\} \\ \mathit{BCT} & ::= & \longrightarrow \mid \succ\!\prec \mid \dashrightarrow \mid -\!\square\!\!\rightarrow \mid -\!a\!\!\rightarrow \mid \\ & & -\![u]\!\!\rightarrow \mid -\!\{pat\}\!\!\rightarrow \mid -\!\!\prec \mid \succ\!\!- \end{array}$$

Figure 1: Reo Syntax

hidden nodes and internal basic connectors constitute its hidden internal structure. Components cannot connect to, read from, or write to hidden nodes which are results of juxtaposing sink nodes on source nodes. Instead, data-flow through hidden nodes is totally specified by the circuits they belong to.

Component behavior can be modeled as a side specification to Reo so that one can also analyze the interaction of components with a Reo connector. In this section, we assume that the output values of components are available as initial data sequences that are used as the input to Reo connectors. This assumption can be easily relaxed in our semantics and to show this, we sketch a proposal in Section 3 for unifying component and connector definitions.

2.1 Reo Syntax

Abstract Syntax The abstract syntax of a connector in the subset of Reo that we consider in this paper is given in Figure 1. In this figure, a Reo circuit *Sys* consists of a set of basic connector instances *BCI*. Each basic connector instance is instantiated from a basic connector type *BCT*, connecting two *node sets*. For simplicity in presentation, we gather the source nodes of a basic connector instance on the left-hand side of the basic connector type and its sink nodes on its right-hand side, each forming (a possibly empty) *node set*. We identify each node with a name, taken from a set *Names*, with typical members $A, B, C, \ldots$ and variables $a, b, c, \ldots$ ranging over them. Variables $ci, ci_0, \ldots$ range over basic connector instances and $sys, sys_0, \ldots$ range over terms from the syntax of Reo circuits. Where there is no confusion and for more brevity in presentation, we may skip the braces around systems and nodes. In such cases, one must bear in mind that the ordering and repetition of channel instances and channel ends are irrelevant.

Basic connector types in *BCT* stand for the following intuitions:

1. *Synchronous connector* ($\longrightarrow$): A synchronous connector instance has a source- and a sink-node at each end. It synchronizes its source and sink by communicating the data item from its source to its sink atomically (thus, synchronously).

2. *Synchronous drain connector* ($\succ\!\prec$): A synchronous drain connector instance reads data from its two source nodes synchronously. It has no sink node, so it loses all data items it obtains from its ends.

3. *Synchronous lossy connector* ($\dashrightarrow$): A synchronous lossy connector has a source and a sink node and synchronizes the sink with the source but not vice versa. In other words, it blocks the reader component/connector on its sink end until a writer writes a data item on the source, but if a reader is not present, the writer performs its write operation and the data item is lost.

4. *One place FIFO connector*: An empty one place FIFO connector ($-\!\square\!\!\rightarrow$) is a basic connector to define asynchronous architectures. When a data item is present at the only source node of this connector, it is taken into the FIFO buffer and the buffer becomes full (represented by $-\!a\!\!\rightarrow$), thus blocking further write operations. The reader can read the data from the buffer through its sink node whenever it is not empty.

5. *Unbounded FIFO connector* ($-\![u]\!\!\rightarrow$): An unbounded FIFO connector allows asynchronous operations on its source and sink nodes by accepting an arbitrary number of consecutive writes and allowing reads as long as its buffer is not empty. The (possibly empty) sequence of data items currently residing inside the buffer is denoted by u.

6. *Filter connector* ($-\!\{pat\}\!\!\rightarrow$): A filter connector, parameterized by the pattern $pat \subseteq Data$ (which designates a set of data items), communicates a data item from its source to its sink node if the data item is in (i.e., matches) the pattern pat, otherwise the data item is accepted from the source and is lost.

7. *Fork connector* ($-\!\!\!<$): A fork connector synchronously replicates a data from its only source node to all its sink nodes. In this paper, we only consider fork connector with one source node and two sink nodes. However, using this connector, fork connectors with more sink nodes can be added as a syntactic sugar to our set of basic connector types.

8. *Merge connector* ($>\!\!\!-$): A merge connector synchronously transfers a data item from one of its source nodes to its only sink node. If more than one source node has a suitable data item to offer, one of them is chosen nondeterministically. Again, we only consider merge connectors with two source nodes and one sink node in the remainder.

Observe that the above fork and merge connectors are *not* Reo channels. We use them in this paper to explicitly represent the replication and the merge aspects that are inherent in the behavior of Reo nodes (with more than one coincident source or sink channel ends). Because we do not deal with dynamic reconfiguration of Reo circuits in this paper, any Reo circuit that involves nodes with more than one coincident source or sink channel ends can always be transformed into another Reo circuit with equivalent behavior, where instances of the above fork and merge connectors make their respective inherent replication and merge node behavior explicit. The resulting circuits involve nodes (the only kinds we deal with in this paper) with no more that one coincident source and/or sink channel end.

Constraints on Abstract Syntax The concrete syntax of our subset of Reo imposes some additional constraints on the abstract syntax given in Figure 1. These constraints are categorized as follows:

- *Source and sink cardinalities:* Basic connectors are of different types. Basic connectors $\longrightarrow$, $-\!\square\!\!\rightarrow$, $-\!a\!\!\rightarrow$, $-\![u]\!\!\rightarrow$ and $-\!\{pat\}\!\!\rightarrow$ are of type $1to1$ meaning that they have a single source and a single sink nodes. The synchronous drain connector $>\!\!\!-\!\!\!<$ is of type $2to0$ meaning that it has two source nodes and no sink node. The fork connector $-\!\!\!<$ is of type $1to2$ and its dual, the merge connector $>\!\!\!-$, is of type $2to1$ ($1toN$ fork connectors and $Nto1$ merge connectors can trivially be added to our language as syntactic sugar).

- *Plugging principle:* Connector instances can be "plugged" into each other (i.e., connected) by combining a sink node of one connector to the source node of another. Combining nodes is represented by sharing of names, i.e., when the sink of one connector bears the same name as the source of another, the two are connected. No other connection scheme is allowed in our subset of Reo. Combined nodes are *hidden* in our circuits (notation $hid(Sys)$) and cannot be used to plug other nodes.

- *Congestion freedom:* Hidden nodes of a circuit can only pass data. As such, they cannot initially (or in any stable state of the circuit) hold a non-empty data sequence. In other words, there should be no congestion in the internal nodes of Reo connectors.

Note that the above constraints are required to be valid only in the initial specification of a Reo connector and our SOS semantics preserves them as an invariant during an execution of the circuit.

Definition 1 (Source/Sink/Hidden node sets) Based on their intuitive meaning, source, sink, and hidden node sets of a Reo connector are defined inductively as follows.

1. For a basic connector instance $ci = \langle nos_0\ ct\ nos_1 \rangle$ $(ct \in BCT)$, we define $hid(ci) \triangleq nos_0 \cap nos_1$, $source(ci) \triangleq nos_0$ and $sink(ci) \triangleq nos_1$.

2. For a circuit $Sys = ci \cup Sys'$:

   - $hid(Sys) \triangleq hid(ci) \cup hid(Sys') \cup (source(ci) \cap sink(Sys')) \cup (source(Sys') \cap sink(ci))$;
   - $source(Sys) \triangleq (source(ci) \cup source(Sys')) - hid(Sys)$ and
   - $sink(Sys) \triangleq (sink(ci) \cup sink(Sys')) - hid(Sys)$.

2.2 Reo Semantics

The operational state of a Reo system consists of a pair $\langle Sys, Val \rangle$, where Sys is a Reo system term with the syntax defined before and Val is a valuation of data on nodes. Data valuation at each node is a sequence of data taken from a set $DataSeq : (\mathbb{N} \to Data) \cup \{[]\}$ (where $\mathbb{N}$ is the set of natural numbers and $[]$ represents an empty sequence). Variables ranging over data sequences are denoted by $u, v, w, \ldots$. We use $d \frown u$ (similarly, $u \frown d$) to denote the concatenation of a data item d to the head (tail) of a sequence u. Data valuation $Val : Names \to DataSeq$ is a function that defines the data value of each node. Variables ranging over data valuations are denoted by $\sigma, \sigma', \ldots$.

Definition 2 (Consistency and Data Values) A system is consistent under a data valuation if that data valuation assigns an empty sequence to each of its hidden nodes. Observe that basic connector instances are mostly consistent, because they usually do not have a hidden node. A system resulting from the union of two connectors is consistent under a data valuation if each connector is individually consistent under that data valuation and the valuation assigns an empty sequence to each of their shared (hidden) nodes. For a consistent system sys when the data valuation is understood, the data value of a node x is denoted by $sys(x)$.

The first part of the Structural Operational Semantics of a Reo connector is defined in Figure 2. This part is concerned with the semantics of our basic connector instances. The first rule **(Syn)** defines the behavior of a synchronous connector by copying data from its source node to its sink node. Note that the data are processed in a first come first served manner: the data are taken from the end of the sequence of the source node (the oldest data item is taken) and are put at the beginning of the corresponding sink sequence. The expression $\sigma \uplus \sigma'$ represents the union of σ and σ' as two disjoint parts of a data valuation function. Rule **(Synd)** specifies that a synchronous drain connector reads data from its two source nodes when they both offer a data item each. Presence of data at both source nodes is the only necessary condition and the two data items need not be the same. In rules **(LSyn0)** and **(LSyn1)** we specify the two possible courses of behavior of a lossy synchronous connector, namely, copying data from its source to its sink, or alternatively, removing data from its source and losing it. The behavior of the one-place FIFO and the unbounded FIFO connectors are described by rules **(OFifo0)-(OFifo1)** and **(IFifo0)-(IFifo1)**, respectively. Rule **(Filter0)** specifies that a filter can communicate data items present in *pat* and rule **(Filter1)** shows that a data item will be lost if it is not contained in *pat*. The behavior of the Fork connector is defined in rule **(Fr)** as copying an available data item from its source to its sink nodes. Similarly, rules **(Mr0)** and **(Mr1)** state that the merge connector copies a data item available on one of its source nodes (chosen nondeterministically if both have available data items) to its sink node.

The second part of our SOS Reo semantics is presented in Figure 3. In this part, we specify how the semantics of a system is composed from the semantics of its subsystems (ultimately, its basic connector instances). This composition is presented in a layered fashion comprising of three

$$\textbf{(Syn)}\,\frac{}{\begin{array}{l}\langle A \longrightarrow B, \{A \mapsto u\frown d, B \mapsto v\} \uplus \sigma\rangle \to \\ \langle A \longrightarrow B, \{A \mapsto u, B \mapsto d\frown v\} \uplus \sigma\rangle\end{array}}$$

$$\textbf{(Synd)}\,\frac{}{\begin{array}{l}\langle (A,B) \succ\!\!\prec \emptyset, \{A \mapsto u\frown d, B \mapsto v\frown d'\} \uplus \sigma\rangle \to \\ \langle (A,B) \succ\!\!\prec \emptyset, \{A \mapsto u, B \mapsto v\} \uplus \sigma\rangle\end{array}}$$

$$\textbf{(LSyn0)}\,\frac{}{\begin{array}{l}\langle A \dashrightarrow B, \{A \mapsto u\frown d, B \mapsto v\} \uplus \sigma\rangle \to \\ \langle A \dashrightarrow B, \{A \mapsto u, B \mapsto d\frown v\} \uplus \sigma\rangle\end{array}}$$

$$\textbf{(LSyn1)}\,\frac{}{\begin{array}{l}\langle A \dashrightarrow B, \{A \mapsto u\frown d, B \mapsto v\} \uplus \sigma\}\rangle \to \\ \langle A \dashrightarrow B, \{A \mapsto u, B \mapsto v\} \uplus \sigma\rangle\end{array}}$$

$$\textbf{(OFifo0)}\,\frac{}{\begin{array}{l}\langle A \mathrel{-\!\square\!\!\rightarrow} B, \{A \mapsto u\frown d\} \uplus \sigma\rangle \to \\ \langle A \mathrel{-\!d\!\!\rightarrow} B, \{A \mapsto u\} \uplus \sigma\rangle\end{array}}$$

$$\textbf{(OFifo1)}\,\frac{}{\begin{array}{l}\langle A \mathrel{-\!d\!\!\rightarrow} B, \{B \mapsto u\} \uplus \sigma\rangle \to \\ \langle A \mathrel{-\!\square\!\!\rightarrow} B, \{B \mapsto d\frown u\} \uplus \sigma\rangle\end{array}}$$

$$\textbf{(IFifo0)}\,\frac{}{\begin{array}{l}\langle A \mathrel{-\![u]\!\!\rightarrow} B, \{A \mapsto v\frown d\} \uplus \sigma\rangle \to \\ \langle A \mathrel{-\![d\frown u]\!\!\rightarrow} B, \{A \mapsto v\} \uplus \sigma\rangle\end{array}}$$

$$\textbf{(IFifo1)}\,\frac{}{\begin{array}{l}\langle A \mathrel{-\![u\frown d]\!\!\rightarrow} B, \{B \mapsto v\} \uplus \sigma\rangle \to \\ \langle A \mathrel{-\![u]\!\!\rightarrow} B, \{B \mapsto d\frown v\} \uplus \sigma\rangle\end{array}}$$

$$\textbf{(Filter0)}\,\frac{d \in pat}{\begin{array}{l}\langle A \mathrel{-\!\{pat\}\!\!\rightarrow} B, \{A \mapsto u\frown d, B \mapsto v\} \uplus \sigma\rangle \to \\ \langle A \mathrel{-\!\{pat\}\!\!\rightarrow} B, \{A \mapsto u, B \mapsto d\frown v\} \uplus \sigma\rangle\end{array}}$$

$$\textbf{(Filter0)}\,\frac{d \notin pat}{\begin{array}{l}\langle A \mathrel{-\!\{pat\}\!\!\rightarrow} B, \{A \mapsto u\frown d, B \mapsto v\} \uplus \sigma\rangle \to \\ \langle A \mathrel{-\!\{pat\}\!\!\rightarrow} B, \{A \mapsto u, B \mapsto v\} \uplus \sigma\rangle\end{array}}$$

$$\textbf{(Fr)}\,\frac{}{\begin{array}{l}\langle A \mathrel{-\!\!\prec} (B,C), \{A \mapsto u\frown d, B \mapsto v, C \mapsto w\} \uplus \sigma\rangle \to \\ \langle A \mathrel{-\!\!\prec} (B,C), \{A \mapsto u, B \mapsto d\frown v, C \mapsto d\frown w\} \uplus \sigma\rangle\end{array}}$$

$$\textbf{(Mr0)}\,\frac{}{\begin{array}{l}\langle (A,B) \mathrel{\succ\!\!-} C, \{A \mapsto u\frown d, B \mapsto v, C \mapsto w\} \uplus \sigma\rangle \to \\ \langle (A,B) \mathrel{\succ\!\!-} C, \{A \mapsto u, B \mapsto v, C \mapsto d\frown w\} \uplus \sigma\rangle\end{array}}$$

$$\textbf{(Mr1)}\,\frac{}{\begin{array}{l}\langle (A,B) \mathrel{\succ\!\!-} C, \{A \mapsto u, B \mapsto v\frown d, C \mapsto w\} \uplus \sigma\rangle \to \\ \langle (A,B) \mathrel{\succ\!\!-} C, \{A \mapsto u, B \mapsto v, C \mapsto d\frown w\} \uplus \sigma\rangle\end{array}}$$

Figure 2: Reo Semantics: Part 1

$$\textbf{(Join)}\ \frac{\begin{array}{l}\langle sys_0, \sigma\rangle \rightarrow \langle sys_0', \sigma'\rangle \\ \langle sys_1, \sigma'\rangle \rightarrow \langle sys_1', \sigma''\rangle \\ sys_0 \cap sys_1 = \emptyset \quad \forall_{x \in hid(sys\cup)} \sigma''(x) = []\end{array}}{\langle sys_0 \cup sys_1, \sigma\rangle \rightarrow \langle sys_0' \cup sys_1', \sigma''\rangle}$$

$$\textbf{(Subsys)}\ \frac{\begin{array}{l}\langle sys_0, \sigma\rangle \rightarrow \langle sys_0', \sigma'\rangle \\ sys_0 \subseteq sys \quad \forall_{x \in hid(sys)} \sigma'(x) = []\end{array}}{\langle sys_0, \sigma\rangle \rightarrow_{\subseteq sys} \langle sys_0', \sigma'\rangle}$$

$$\textbf{(Sys)}\ \frac{\begin{array}{l}\langle sys_0, \sigma\rangle \rightarrow_{\subseteq sys_0 \cup sys_1} \langle sys_0', \sigma'\rangle \\ sys_0 \cap sys_1 = \emptyset \\ \forall_{sys_2 \subseteq sys_0 \cup sys_1} sys_1 \subset sys_2 \Rightarrow sys_2 \not\rightarrow_{\subseteq sys_0 \cup sys_1}\end{array}}{\langle sys_0 \cup sys_1, \sigma\rangle \rightsquigarrow \langle sys_0' \cup sys_1, \sigma'\rangle}$$

Figure 3: Reo Semantics: Part 2

levels. The first level is described by rule **(Join)**. This rule specifies that a system can perform a *total transition*, denoted by $\rightarrow$, if the system can be decomposed into two disjoint parts such that the first part makes a total transition and in turn, provides input for the second subsystem to perform its total transition. As the congestion freedom principle must be maintained by our semantics, we also check in the premise of **(Join)** that the result of this total transition contains no data item in hidden nodes. However, a total transition is not always possible in a Reo connector due to its blocking and synchronization constraints. Thus, as the second layer, **(Subsys)** defines the criteria under which a subsystem of Sys can perform a *consistent (partial) transition*, denoted by $\rightarrow_{\subseteq Sys}$. Finally, the third layer, defined by **(Sys)**, chooses a *maximal (partial) transition*, denoted by $\rightsquigarrow$ and defines it as a transition of the system. Note that a maximal transition is not necessarily unique due to the nondeterminism which is inherent in some basic Reo connectors (i.e., merge). The operational semantics of Reo is the smallest relation $\rightsquigarrow$, satisfying the deduction rules of Figures 2 and 3.

To better illustrate the idea of our syntax and semantics we specify two typical Reo connectors in the following examples and derive their transitions using our semantics.

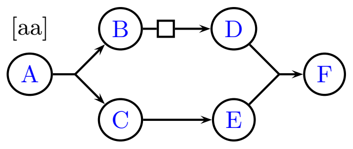

Figure 4: A Replicator Connector

Example 1 Consider the system depicted in Figure 4. In this figure, all data sequences at the nodes are initially empty but the one of node A which contains the sequence $[aa]$.

According to the semantics of Figures 2 and 3, the first step of the system can be deduced as follows:

$$\begin{array}{l|c|r}
\langle A -\!\!\!-\!\!< (B,C), \{A \mapsto [aa], B \mapsto [], C \mapsto [], D \mapsto [], E \mapsto [], F \mapsto []\}\rangle \rightarrow & & \\
\langle A -\!\!\!-\!\!< (B,C), \{A \mapsto [a], B \mapsto [a], C \mapsto [a], , D \mapsto [], E \mapsto [], F \mapsto []\}\rangle & (1) & \textbf{(Fr)} \\
& & \\
\langle B \xrightarrow{\square} D, \{A \mapsto [a], B \mapsto [a], C \mapsto [a], D \mapsto [], E \mapsto [], F \mapsto []\}\rangle \rightarrow & & \\
\langle B \xrightarrow{a} D, \{A \mapsto [a], B \mapsto [], C \mapsto [a], D \mapsto [], E \mapsto [], F \mapsto []\}\rangle & (2) & \textbf{(OFifo0)} \\
& & \\
\langle C \longrightarrow E, \{A \mapsto [a], B \mapsto [a], C \mapsto [a], D \mapsto [], E \mapsto [], F \mapsto []\}\rangle \rightarrow & & \\
\langle C \longrightarrow E, \{A \mapsto [a], B \mapsto [a], C \mapsto [], D \mapsto [], E \mapsto [a], F \mapsto []\}\rangle & (3) & \textbf{(Syn)} \\
& & \\
\langle C \longrightarrow E, B \xrightarrow{\square} D, \{A \mapsto [a], B \mapsto [a], C \mapsto [a], D \mapsto [], E \mapsto [], F \mapsto []\}\rangle \rightarrow & & (2),(3) \\
\langle C \longrightarrow E, B \xrightarrow{a} D, \{A \mapsto [a], B \mapsto [], C \mapsto [], D \mapsto [], E \mapsto [a], F \mapsto []\}\rangle & (4) & \textbf{(Join)} \\
& & \\
\langle \{A -\!\!\!-\!\!< (B,C), C \longrightarrow E, B \xrightarrow{\square} D\}, & & \\
\{A \mapsto [aa], B \mapsto [], C \mapsto [], D \mapsto [], E \mapsto [], F \mapsto []\}\rangle \rightarrow & & \\
\langle \{A -\!\!\!-\!\!< (B,C), C \longrightarrow E, B \xrightarrow{a} D\}, & & (1),(4), \\
\{A \mapsto [a], B \mapsto [], C \mapsto [a], , D \mapsto [], E \mapsto [], F \mapsto []\}\rangle & (5) & \textbf{(Join)} \\
& & \\
\langle (D,E) >\!\!-\!\!\!- F, \{A \mapsto [a], B \mapsto [], C \mapsto [], D \mapsto [], E \mapsto [a], F \mapsto []\}\rangle \rightarrow & & \\
\langle (D,E) >\!\!-\!\!\!- F, \{A \mapsto [a], B \mapsto [], C \mapsto [], D \mapsto [], E \mapsto [], F \mapsto [a]\}\rangle & (6) & \textbf{(Mr1)} \\
& & \\
\langle \{A -\!\!\!-\!\!< (B,C), B \xrightarrow{\square} D, C \longrightarrow E, (D,E) >\!\!-\!\!\!- F\}, & & \\
\quad \{A \mapsto [aa], B \mapsto [], C \mapsto [], D \mapsto [], E \mapsto [], F \mapsto []\}\rangle \quad \rightarrow & & \\
\langle \{A -\!\!\!-\!\!< (B,C), B \xrightarrow{a} D, C \longrightarrow E, (D,E) >\!\!-\!\!\!- F\}, & & (5),(6), \\
\quad \{A \mapsto [a], B \mapsto [], C \mapsto [], , D \mapsto [], E \mapsto [], F \mapsto [a]\}\rangle & (7) & \textbf{(Join)} \\
& & \\
\langle \{A -\!\!\!-\!\!< (B,C), B \xrightarrow{\square} D, C \longrightarrow E, (C,D) >\!\!-\!\!\!- E\}, & & \\
\quad \{A \mapsto [aa], B \mapsto [], C \mapsto [], D \mapsto [], E \mapsto [], F \mapsto []\}\rangle \quad \rightarrow_{\subseteq Sys} & & \\
\langle \{A -\!\!\!-\!\!< (B,C), B \xrightarrow{a} D, C \longrightarrow E, (C,D) >\!\!-\!\!\!- E\}, & & (7), \\
\quad \{A \mapsto [a], B \mapsto [], C \mapsto [], , D \mapsto [], E \mapsto [], F \mapsto [a]\}\rangle & (8) & \textbf{(Subsys)} \\
& & \\
\langle \{A -\!\!\!-\!\!< (B,C), B \xrightarrow{\square} D, C \longrightarrow E, (D,E) >\!\!-\!\!\!- F\}, & & \\
\quad \{A \mapsto [aa], B \mapsto [], C \mapsto [], D \mapsto [], E \mapsto [], F \mapsto []\}\rangle \quad \rightsquigarrow & & \\
\langle \{A -\!\!\!-\!\!< (B,C), B \xrightarrow{a} D, C \longrightarrow E, (D,E) >\!\!-\!\!\!- F\}, & & (8), \\
\quad \{A \mapsto [a], B \mapsto [], C \mapsto [], , D \mapsto [], E \mapsto [], F \mapsto [a]\}\rangle & & \textbf{(System)}
\end{array}$$

Note that in the above transitions Sys is used as a shorthand for the specification of the whole connector in its initial state. Starting from the new state, the next transition of the system can be deduced as follows:

$$\begin{array}{l|c|r}
\langle B \xrightarrow{a} D, \{A \mapsto [a], B \mapsto [], C \mapsto [], D \mapsto [], E \mapsto [], F \mapsto [a]\}\rangle \rightarrow & & \\
\langle B \xrightarrow{\square} D, \{A \mapsto [a], B \mapsto [], C \mapsto [], D \mapsto [a], E \mapsto [], F \mapsto [a]\}\rangle & (1) & \textbf{(OFifo1)} \\
& & \\
\langle (D,E) >\!\!-\!\!\!- F, \{A \mapsto [a], B \mapsto [], C \mapsto [], D \mapsto [a], E \mapsto [], F \mapsto [a]\}\rangle \rightarrow & & \\
\langle (D,E) >\!\!-\!\!\!- F, \{A \mapsto [a], B \mapsto [], C \mapsto [], D \mapsto [], E \mapsto [], F \mapsto [aa]\}\rangle & (2) & \textbf{(Mr0)} \\
& & \\
\langle B \xrightarrow{a} D, (D,E) >\!\!-\!\!\!- F, \{A \mapsto [a], B \mapsto [], C \mapsto [], D \mapsto [], E \mapsto [], F \mapsto [a]\}\rangle \rightarrow & & (1),(2), \\
\langle B \xrightarrow{\square} D, (D,E) >\!\!-\!\!\!- F, \{A \mapsto [a], B \mapsto [], C \mapsto [], D \mapsto [], E \mapsto [], F \mapsto [aa]\}\rangle & (3) & \textbf{(Join)} \\
& & \\
\langle B \xrightarrow{a} D, (D,E) >\!\!-\!\!\!- F, \{A \mapsto [a], B \mapsto [], C \mapsto [], D \mapsto [], E \mapsto [], F \mapsto [a]\}\rangle \rightarrow_{\subseteq Sys} & & (3), \\
\langle B \xrightarrow{\square} D, (D,E) >\!\!-\!\!\!- F, \{A \mapsto [a], B \mapsto [], C \mapsto [], D \mapsto [], E \mapsto [], F \mapsto [aa]\}\rangle & (4) & \textbf{(Subsys)} \\
& & \\
\langle A -\!\!\!-\!\!< (B,C), B \xrightarrow{a} D, C \longrightarrow E, (D,E) >\!\!-\!\!\!- F, & & \\
\quad \{A \mapsto [a], B \mapsto [], C \mapsto [], D \mapsto [], E \mapsto [], F \mapsto [a]\}\rangle \rightsquigarrow & & \\
\langle A -\!\!\!-\!\!< (B,C), B \xrightarrow{\square} D, C \longrightarrow E, (D,E) >\!\!-\!\!\!- F, & & (4), \\
\quad \{A \mapsto [a], B \mapsto [], C \mapsto [], D \mapsto [], E \mapsto [], F \mapsto [aa]\}\rangle & & \textbf{(System)}
\end{array}$$

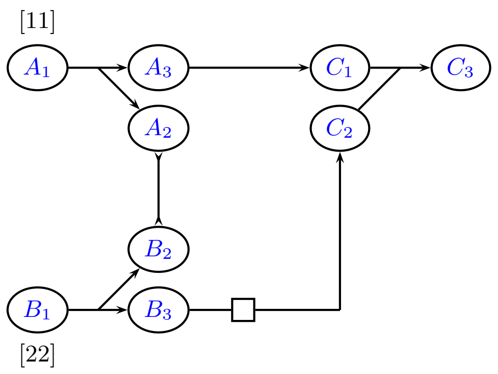

Figure 5: An Interleaving Connector

Form the above two simulation steps, we get the impression that the above connector duplicates it source sequence data on its sink node.

Example 2 A more complex Reo system is depicted in Figure 5. In this connector, whenever a data item is read from one input, this read has to be synchronized with the other input due to the synchronous drain connectors between nodes A_2 and B_2. Thus, the first transition of this connector results in a synchronous read from each of the two source nodes, filling in the FIFO buffer with the data item 2 and communicating the data item 1 to the sink node C_3:

$\langle A_1 -\!\!\!< (A_2, A_3), \{(A_1, [11]), (A_2, []), (A_3, [1]), \ldots\}\rangle \rightarrow$ $\langle A_1 -\!\!\!< (A_2, A_3), \{(A_1, [1]), (A_2, [1]), (A_3, [1]), \ldots\}\rangle$	(1)	**(Fr)**
$\langle B_2 >\!\!-\!\!\!< A_2, \{(B_2, [2]), (A_2, [1]), \ldots\}\rangle \rightarrow$ $\langle B_2 >\!\!-\!\!\!< A_2, \{(B_2, []), (A_2, []), \ldots\}\rangle$	(2)	**(Synd)**
$\langle A_1 -\!\!\!< (A_2, A_3), B_2 >\!\!-\!\!\!< A_2,$ $\{(A_1, [11]), (A_2, []), (A_3, []), (B_2, [2]), \ldots\}\rangle \rightarrow$ $\langle A_1 -\!\!\!< (A_2, A_3), B_2 >\!\!-\!\!\!< A_2,$ $\{(A_1, [1]), (A_2, []), (A_3, [1]), (B_2, []), \ldots\}\rangle$	(3)	(1), (2), **(Join)**
$\langle B_1 -\!\!\!< (B_2, B_3), \{(B_1, [22]), (B_2, []), (B_3, []), \ldots\}\rangle \rightarrow$ $\langle B_1 -\!\!\!< (B_2, B_3), \{(B_1, [2]), (B_2, [2]), (B_3, [2]), \ldots\}\rangle$	(4)	**(Fr)**
$\langle A_1 -\!\!\!< (A_2, A_3), B_2 >\!\!-\!\!\!< A_2, B_1 -\!\!\!< (B_2, B_3),$ $\{(A_1, [11]), (A_2, []), (A_3, []), (B_1, [22]), (B_2, []), (B_3, []), \ldots\}\rangle \rightarrow$ $\langle A_1 -\!\!\!< (A_2, C_1), B_2 >\!\!-\!\!\!< A_2, B_1 -\!\!\!< (B_2, B_3),$ $\{(A_1, [1]), (A_2, []), (A_3, [1]), (B_1, [2]), (B_2, []), (B_3, [2]), \ldots\}\rangle \rightarrow$	(5)	(3), (4), **(Join)**
$\langle B_3 \xrightarrow{\square} C_2, \{(B_2, [2]), \ldots\}\rangle \rightarrow$ $\langle B_3 \xrightarrow{2} C_2, \{(B_2, []), \ldots\}\rangle$	(6)	**(OFifo0)**
$\langle A_1 -\!\!\!< (A_2, A_3), B_2 >\!\!-\!\!\!< A_2, B_1 -\!\!\!< (B_2, B_3), B_3 \xrightarrow{\square} C_2,$ $\{(A_1, [11]), (A_2, []), (A_3, []), (B_1, [22]), (B_2, []), (B_3, []), (C_2, []), \ldots\}\rangle \rightarrow$ $\langle A_1 -\!\!\!< (A_2, C_1), B_2 >\!\!-\!\!\!< A_2, B_1 -\!\!\!< (B_2, B_3), B_3 \xrightarrow{2} C_2,$ $\{(A_1, [1]), (A_2, []), (A_3, [1]), (B_1, [2]), (B_2, []), (B_3, []), (C_2, []), \ldots\}\rangle$	(7)	(5), (6), **(Join)**

$$\begin{array}{l}\langle A_3 \longrightarrow C_1, \{(A_3,[1]),(C_1,[]),\ldots\}\rangle \rightarrow \\ \langle A_3 \longrightarrow C_1, \{(A_3,[]),(C_1,[1]),\ldots\}\rangle\end{array} \qquad (8) \qquad \textbf{(Syn)}$$

$$\begin{array}{l}\langle A_1 \mathrel{-\!\!<} (A_2,A_3), B_2 \mathrel{>\!\!<} A_2, B_1 \mathrel{-\!\!<} (B_2,B_3), B_3 \xrightarrow{\square} C_2, A_3 \longrightarrow C_1, \\ \{(A_1,[11]),(A_2,[]),(A_3,[]),(B_1,[22]),(B_2,[]),(B_3,[]),(C_1,[]),(C_2,[]),\ldots\}\rangle \rightarrow \\ \langle A_1 \mathrel{-\!\!<} (A_2,C_1), B_2 \mathrel{>\!\!<} A_2, B_1 \mathrel{-\!\!<} (B_2,B_3), B_3 \xrightarrow{2} C_2, A_3 \longrightarrow C_1, \\ \{(A_1,[1]),(A_2,[]),(A_3,[]),(B_1,[2]),(B_2,[]),(B_3,[]),(C_1,[1]),(C_2,[]),\ldots\}\rangle\end{array} \qquad (9) \qquad (7),(8),\ \textbf{(Join)}$$

$$\begin{array}{l}\langle (C_1,C_2) \mathrel{>\!\!-} C_3, \{(C_1,[1]),(C_2,[]),(C_3,[]),\ldots\}\rangle \rightarrow \\ \langle (C_1,C_2) \mathrel{>\!\!-} C_3, \{(C_1,[]),(C_2,[]),(C_3,[1]),\ldots\}\rangle\end{array} \qquad (10) \qquad \textbf{(Mr0)}$$

$$\begin{array}{l}\langle A_1 \mathrel{-\!\!<} (A_2,A_3), B_2 \mathrel{>\!\!<} A_2, B_1 \mathrel{-\!\!<} (B_2,B_3), B_3 \xrightarrow{\square} C_2, A_3 \longrightarrow C_1, (C_1,C_2) \mathrel{>\!\!-} C_3 \\ \{(A_1,[11]),(A_2,[]),(A_3,[]),(B_1,[22]),(B_2,[]),(B_3,[]),(C_1,[]),(C_2,[]),(C_3,[])\}\rangle \rightarrow \\ \langle A_1 \mathrel{-\!\!<} (A_2,A_3), B_2 \mathrel{>\!\!<} A_2, B_1 \mathrel{-\!\!<} (B_2,B_3), B_3 \xrightarrow{2} C_2, A_3 \longrightarrow C_1, (C_1,C_2) \mathrel{>\!\!-} C_3 \\ \{(A_1,[1]),(A_2,[]),(A_2,[]),(B_1,[2]),(B_2,[]),(B_3,[]),(C_1,[]),(C_2,[]),(C_3,[1])\}\rangle\end{array} \qquad (11) \qquad (9),(10)\ \textbf{(Join)}$$

$$\begin{array}{l}\langle A_1 \mathrel{-\!\!<} (A_2,A_3), B_2 \mathrel{>\!\!<} A_2, B_1 \mathrel{-\!\!<} (B_2,B_3), B_3 \xrightarrow{2} C_2, A_3 \longrightarrow C_1, (C_1,C_2) \mathrel{>\!\!-} C_3 \\ \{(A_1,[11]),(A_2,[]),(A_3,[]),(B_1,[22]),(B_2,[]),(B_3,[]),(C_1,[]),(C_2,[]),(C_3,[])\}\rangle \rightarrow_{\subseteq Sys} \\ \langle A_1 \mathrel{-\!\!<} (A_2,A_3), B_2 \mathrel{>\!\!<} A_2, B_1 \mathrel{-\!\!<} (B_2,B_3), B_3 \xrightarrow{2} C_2, A_3 \longrightarrow C_1, (C_1,C_2) \mathrel{>\!\!-} C_3 \\ \{(A_1,[1]),(A_2,[]),(A_2,[]),(B_1,[2]),(B_2,[]),(B_3,[]),(C_1,[]),(C_2,[]),(C_3,[1])\}\rangle\end{array} \qquad (12) \qquad (11)\ \textbf{(Subsys)}$$

$$\begin{array}{l}\langle A_1 \mathrel{-\!\!<} (A_2,A_3), B_2 \mathrel{>\!\!<} A_2, B_1 \mathrel{-\!\!<} (B_2,B_3), B_3 \xrightarrow{2} C_2, A_3 \longrightarrow C_1, (C_1,C_2) \mathrel{>\!\!-} C_3 \\ \{(A_1,[11]),(A_2,[]),(A_3,[]),(B_1,[22]),(B_2,[]),(B_3,[]),(C_1,[]),(C_2,[]),(C_3,[])\}\rangle \rightsquigarrow \\ \langle A_1 \mathrel{-\!\!<} (A_2,A_3), B_2 \mathrel{>\!\!<} A_2, B_1 \mathrel{-\!\!<} (B_2,B_3), B_3 \xrightarrow{2} C_2, A_3 \longrightarrow C_1, (C_1,C_2) \mathrel{>\!\!-} C_3 \\ \{(A_1,[1]),(A_2,[]),(A_3,[]),(B_1,[2]),(B_2,[]),(B_3,[]),(C_1,[]),(C_2,[]),(C_3,[1])\}\rangle\end{array} \qquad (13) \qquad (12)\ \textbf{(System)}$$

The next step cannot allow any read because the FIFO buffer is full and can involve only the flushing of the FIFO buffer to the sink node:

$$\begin{array}{l}\langle B_3 \xrightarrow{2} C_2, (C_1,C_2) \mathrel{>\!\!-} C_3 \\ \{(B_3,[]),(C_1,[]),(C_2,[]),(C_3,[]),\ldots\}\rangle \rightarrow \\ \langle B_3 \xrightarrow{\square} C_2, (C_1,C_2) \mathrel{>\!\!-} C_3 \\ \{(B_3,[]),(C_1,[]),(C_2,[]),(C_3,[2]),\ldots\}\rangle\end{array} \qquad \textbf{(OFifo0)}\ \textbf{(Mr1)}\ \textbf{(Join)}$$

$$\begin{array}{l}\langle B_3 \xrightarrow{2} C_2, (C_1,C_2) \mathrel{>\!\!-} C_3 \\ \{(B_3,[]),(C_1,[]),(C_2,[]),(C_3,[])\}\rangle \rightarrow_{\subseteq Sys} \\ \langle B_3 \xrightarrow{\square} C_2, (C_1,C_2) \mathrel{>\!\!-} C_3 \\ \{(B_3,[]),(C_1,[]),(C_2,[]),(C_3,[2])\}\rangle\end{array} \qquad \textbf{(Subsys)}$$

$$\begin{array}{l}\langle A_1 \mathrel{-\!\!<} (A_2,A_3), B_2 \mathrel{>\!\!<} A_2, B_1 \mathrel{-\!\!<} (B_2,B_3), B_3 \xrightarrow{2} C_2, A_3 \longrightarrow C_1, (C_1,C_2) \mathrel{>\!\!-} C_3 \\ \{(A_1,[1]),(A_2,[]),(A_3,[]),(B_1,[2]),(B_2,[]),(B_3,[]),(A_2,[]),(C_1,[]),(C_2,[]),(C_3,[])\}\rangle \rightsquigarrow \\ \langle A_1 \mathrel{-\!\!<} (A_2,A_3), B_2 \mathrel{>\!\!<} A_2, B_1 \mathrel{-\!\!<} (B_2,B_3), B_3 \xrightarrow{\square} C_2, A_3 \longrightarrow C_1, (C_1,C_2) \mathrel{>\!\!-} C_3 \\ \{(A_1,[1]),(A_2,[]),(A_3,[]),(B_1,[2]),(B_2,[]),(B_3,[]),(A_2,[]),(C_1,[1]),(C_2,[]),(C_3,[21])\}\rangle\end{array} \qquad \textbf{(System)}$$

Thus, the intuitive behavior of this connector can be summarized as the interleaving of the input sequences at its two source nodes into an output sequence at its sink node.

Instantaneous Behavior of FIFO Connectors In our semantics of basic connectors, we allowed a FIFO connector to perform only either a read or a write at each step. This reflects the semantics of asynchronous FIFO channels in Reo. However, Reo also allows for the synchronous FIFO connector which both reads and stores a new data item at the same time as it writes and removes another item from its buffer. This can be summarized in the following rule:

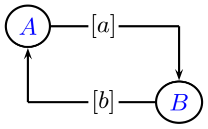

Figure 6: Instantaneous Behavior of FIFOs

$$\textbf{(OFifo2)}\ \frac{}{\begin{array}{l}\langle A \xrightarrow{d} B, \{A \mapsto u \frown d', B \mapsto v\} \uplus \sigma\rangle \rightarrow \\ \langle A \xrightarrow{d'} B, \{A \mapsto u, B \mapsto d \frown v\} \uplus \sigma\rangle\end{array}}$$

Similarly, for an infinite FIFO connector, we get the following extra rule:

$$\textbf{(IFifo2)}\ \frac{}{\begin{array}{l}\langle A \xrightarrow{[u \frown d]} B, \{A \mapsto v \frown d', B \mapsto w\} \uplus \sigma\rangle \rightarrow \\ \langle A \xrightarrow{[d' \frown u]} B, \{A \mapsto v, B \mapsto d \frown w\} \uplus \sigma\rangle\end{array}}$$

However, allowing for such basic connectors calls for a new joining scheme. In other words, the change is not localized to the basic rules and the rule for composing two connectors should also be changed to the following rule:

$$\textbf{(Join')}\ \frac{\begin{array}{l}\langle sys_0, \sigma_0\rangle \rightarrow \langle sys'_0, \sigma'_0\rangle \\ \langle sys_1, \sigma_1\rangle \rightarrow \langle sys'_1, \sigma'_1\rangle \\ \sigma_0 = \sigma \uparrow \langle sys'_1, \sigma'_1\rangle \quad \sigma_1 = \sigma \uparrow \langle sys'_0, \sigma'_0\rangle \\ \sigma' = \langle sys'_0, \sigma'_0\rangle \oplus \langle sys'_1, \sigma'_1\rangle \\ sys_0 \cap sys_1 = \emptyset \quad \forall_{x \in hid(sys_0 \cup sys_1)} \sigma'(x) = []\end{array}}{\langle sys_0 \cup sys_1, \sigma\rangle \rightarrow \langle sys'_0 \cup sys'_1, \sigma'\rangle}$$

In the above rule $\sigma \uparrow \langle sys', \sigma'\rangle$ stands for updating the data valuation σ by values of sink nodes of sys' in σ' and $\langle sys, \sigma\rangle \oplus \langle sys', \sigma'\rangle$ is merging σ and σ' by giving priority to source nodes of sys and sys' on the shared variables. The above rule has a circular structure in that the transition of sys_0 depends on the transition sys_1 and vice versa. This circularity makes the reasoning about the behavior more difficult. The following example explains the essence of this change.

Example 3 Consider the connector depicted in Figure 6. Suppose that FIFO connectors are synchronous as specified by rules **(OFifo2)** and **(IFifo2)**. According to our current semantics with rule **(Join)**, in order for this system to make a total transition, there should exist a decomposition of the system into two parts, in which the first part makes an independent move and then the second part uses the result of the first and makes its transition. However, such a decomposition does not exist in the connector of Figure 6 since both upper and lower FIFO connectors depend on the input of each other for their transition. No partial transition is possible, because any partial transition (involving a single synchronous FIFO connector) results in a congestion of a data item at a hidden node. Hence, it seems very intuitive and natural to extend our **(Join)** to Rule **(Join')** in order to allow for such cyclic dependencies. Using this new semantic rule, we can deduce an infinite behavior for the connector of Figure 6 which results in a and b changing their positions in the upper and lower FIFOs in each turn.

Due to the cyclic nature of Rule **(Join')**, the semantics of synchronous FIFO connectors is less efficient in automated reasoning (since the choice of updates cannot be known before the transitions). Thus, we will use the old semantics without such an instantaneous behavior in this paper, unless explicitly mentioned otherwise. It remains the designer's choice to use one of these two alternative semantics depending on particular application area. To be more precise, the more involved semantics of **(Join')** is needed for basic connectors that can synchronously (instantaneously) write and read on different ports and change their state (evolve to a different basic connector) after this transition. Synchronous FIFO connectors are just examples of such connectors.

2.3 (Bi-)Simulation of Reo Circuits

Defining a notion of equality and refinement is standard practice in reasoning about formalisms with a transition system semantics. Several different notions of equality and refinement exist in the literature that have been used for different semantics for different purposes [23]. However, to find the right notion, one must consider the kind of properties that need to be preserved under equality and refinement. For example, since the notions of nondeterministic choice and deadlock play an important role in the semantics of Reo, we go for a bisimulation-like notion of equality (and similarly a notion of simulation for refinement). Another essential property for the notions of equality and refinement is compositionality. In compositional reasoning about systems, we must ensure that if we prove an equality or refinement relation, this relation is preserved when the systems are composed in a larger context. Formally speaking, we must find a notion of (bi-)simulation that is a congruence. For this property, we go for a robust notion of (bi-)simulation, called initially stateless (bi-)simulation, which is defined as follows.

Definition 3 A relation R is called a simulation relation on Reo configurations if and only if for all pairs $(\langle Sys_0, \sigma\rangle, \langle Sys_1, \sigma'\rangle) \in R$, $\sigma = \sigma'$, σ is consistent with both Sys_0 and Sys_1, and if for some consistent σ'', $\langle Sys_0, \sigma\rangle \rightsquigarrow \langle Sys_0', \sigma''\rangle$ then there exists a Sys_1' such that $\langle Sys_1, \sigma\rangle \rightsquigarrow \langle Sys_1', \sigma''\rangle$ and $(\langle Sys_0', \sigma''\rangle, \langle Sys_1', \sigma''\rangle) \in R$. A symmetric simulation relation is called a bisimulation relation.

Two Reo connectors Sys and Sys' are called initially stateless (bi-)similar, denoted as $Sys \leq Sys'$ ($Sys \underline{\leftrightarrow} Sys'$), if and only if they have the same source and sink node sets and there exists a (bi-)simulation relation R such that for all σ, $(\langle Sys_0, \sigma\rangle, \langle Sys_1, \sigma\rangle) \in R$.

Theorem 1 (Congruence (Robustness)) If $Sys_0 \underline{\leftrightarrow} Sys_0'$ then for all consistent Sys_1, $Sys_0 \cup Sys_1 \underline{\leftrightarrow} Sys_0' \cup Sys_1$. In other words, initially stateless bisimulation is a congruence relation for composing Reo connectors.

Proof. Our rules are in `sfisl` format of [15], thus initially stateless bisimulation is a congruence (a similar statement holds for pre-congruence of initially stateless simulations). ⊠

Congruence and pre-congruence are very essential properties in reasoning about system equalities and refinement. To put it in the context of Reo, they allow for replacing Reo connectors with their equal (refined) connectors in arbitrary environments of components and still expect the same (refined) behavior from the overall system.

3 Orchestrating Components and Connectors

Thus far, we have given an operational semantics for a basic set of connectors and their compositions. However, on the one hand, we have already pointed out that the set of basic connectors can be extended (changed) by the designer and on the other hand, we have also mentioned that reasoning about connectors in orchestration with a behavioral model of components can be of essential importance. We have not yet introduced a systematic way neither to define basic connectors, nor to specify the abstract behavior of components. In this section, we present some initial thoughts on modeling components and basic connectors in a unified framework. In this framework, components and connectors can both be specified in a process algebraic language. The formalism that we propose in this section is inspired by Milner's Calculi of Synchrony and Asynchrony [14].

The basic syntax of a component is defined in Figure 7. In this syntax a component is defined with a name and a process specification. A process specification can either be an atom or a name (for recursive specifications) or a composition of two processes. Atomic processes are *read* or *write* statements with a node and a variable or a constant data item as their parameters, or a *delay* (of one unit). Process composition operators consist of sequential composition ;, synchronized sequential composition $\circ$, and the nondeterministic choice operator $+$. Synchronized sequential

$$\begin{array}{lll}
Comp & ::= & Name = Proc \\
Proc & ::= & Atom \mid Name \mid \\
 & & \quad Proc; Proc \mid Proc \circ Proc \mid Proc + Proc \\
Atom & ::= & read(a, [x|d]) \mid write(b, [x|d]) \mid delay \\
 & & \\
d \in D, x \in nodes & &
\end{array}$$

Figure 7: Syntax of a Component Specification Language

composition composes synchronous transitions of its two arguments in a single synchronized transition.

The semantics of this component specification language is given in Figure 8. In the above semantics nos_0 and nos_1 are source and sink nodes of the basic connector instance being defined, respectively. Rule **(Write0)** specifies how a write operation with a constant data item behaves. Rule **(Write1)** specifies that if the data variable is not bound, it will cause an arbitrary write in the sink node. Similarly, rules **(Read0)** and **(Read1)** specify the behavior of the read operation with fixed and variable data arguments. Rule **(Delay)** specifies an idling transition to the term *skip*. In all the rules, *skip* represent a terminated process. Rule **(SSeq)** specifies that if both arguments of a synchronous sequential composition can perform a synchronous transition (and thus terminate, with the second one benefiting from substitutions caused by the first) then the two transitions are combined in a single synchronous transition. In this rule, $\rho' + \rho$ stands for the substitution resulting from merging ρ' and ρ with priority for substitutions in ρ' on common variables. Rule **(Seq0)** specifies that if the first component of a sequential composition can perform a transition to a non-terminating state then the composition can make the same transition by keeping the composition structure and applying the resulting substitution (due to reading data from ports) to the remainder of the process term. Substitution of a variable by its corresponding data item can be defined inductively on process terms in a natural way. Rule **(Seq1)** presents a similar behavior of parallel composition if the first argument terminates. Rules **(Choice0)** and **(Choice1)** are standard rules for nondeterministic choice.

To illustrate the use of our specification language, we specify our set of basic connector types in the following example.

Example 4 (Basic Connectors Specification) The following examples show how our component specification language can be used to model our set of basic connectors:

$$\begin{array}{lll}
Syn & = & (read(a, x) \circ write(b, x)); Syn \\
Synd & = & (read(a, x) \circ read(b, y)); Synd \\
Lossy & = & ((read(a, x) \circ write(b, x)) + \\
 & & \quad read(a, x)); Lossy \\
OFifo & = & (read(a, x); write(b, x)); Fifo \\
IFifo(u) & = & read(a, x); ((write(a, x); IFifo(u)) + \\
 & & \quad IFifo(u \frown x)) \\
Filter(pat) & = & (\sum_{d \in pat}(read(a, d) \circ write(a, d)) + \\
 & & \quad \sum_{d \notin pat} read(a, d)); Filter(pat) \\
Fork & = & (read(a, x) \circ \\
 & & \quad (write(b, x) \circ write(c, x))); Fork \\
Merge & = & ((read(a, x) + read(b, x)) \circ \\
 & & \quad write(c, x)); Merge
\end{array}$$

Note that in the above examples, $\sum_d$ is a syntactic shorthand for a nondeterministic choice over a finite (and non-empty) domain of data values (assuming associativity and commutativity of choice).

The next example specifies an exclusive router connector which is not a basic connector type in our semantics. Then, we show how we can implement this connector using our existing set of

$$\textbf{(Read0)}\ \frac{a \in nos_0}{\begin{array}{l}\langle nos_0\ read(a,d)\ nos_1, \{a \mapsto u \frown d\} \uplus \sigma\rangle \xrightarrow{[]} \\ \langle nos_0\ skip\ nos_1, \{a \mapsto u, \ldots\}\rangle\end{array}}$$

$$\textbf{(Read1)}\ \frac{a \in nos_0}{\begin{array}{l}\langle nos_0\ read(a,x)\ nos_1, \{a \mapsto u \frown d\} \uplus \sigma\rangle \xrightarrow{[d/x]} \\ \langle nos_0\ skip\ nos_1, \{a \mapsto u, \ldots\}\rangle\end{array}}$$

$$\textbf{(Write0)}\ \frac{b \in nos_1}{\begin{array}{l}\langle nos_0\ write(b,d)\ nos_1, \{a \mapsto u\} \uplus \sigma\rangle \xrightarrow{[]} \\ \langle nos_0\ skip\ nos_1, \{a \mapsto d \frown u\} \uplus \sigma\rangle\end{array}}$$

$$\textbf{(Write1)}\ \frac{b \in nos_1}{\begin{array}{l}\langle nos_0\ write(b,x)\ nos_1, \{a \mapsto u\} \uplus \sigma\rangle \xrightarrow{[]} \\ \langle nos_0\ skip\ nos_1, \{a \mapsto d \frown u\} \uplus \sigma\rangle\end{array}}$$

$$\textbf{(Delay)}\ \frac{b \in nos_1}{\begin{array}{l}\langle nos_0\ delay\ nos_1, \sigma\rangle \xrightarrow{[]} \\ \langle nos_0\ skip\ nos_1, \sigma\rangle\end{array}}$$

$$\textbf{(SSeq)}\ \frac{\begin{array}{c}\langle nos_0\ p\ nos_1, \sigma\rangle \xrightarrow{\rho} \langle nos_0\ skip\ nos_1, \sigma'\rangle \\ \langle nos_0\ q[\rho]\ nos_1, \sigma'\rangle \xrightarrow{\rho'} \langle nos_0\ skip\ nos_1, \sigma''\rangle\end{array}}{\begin{array}{l}\langle nos_0\ p \circ q\ nos_1, \sigma\rangle \xrightarrow{\rho' + \rho} \\ \langle nos_0\ skip\ nos_1, \sigma'\rangle\end{array}}$$

$$\textbf{(Seq0)}\ \frac{\langle nos_0\ p\ nos_1, \sigma\rangle \xrightarrow{\rho} \langle nos_0\ p'\ nos_1, \sigma'\rangle \quad p' \neq skip}{\langle nos_0\ p; q\ nos_1, \sigma\rangle \xrightarrow{\rho} \langle nos_0\ p'; q[\rho]\ nos_1, \sigma'\rangle}$$

$$\textbf{(Seq1)}\ \frac{\langle nos_0\ p\ nos_1, \sigma\rangle \xrightarrow{\rho} \langle nos_0\ skip\ nos_1, \sigma'\rangle}{\langle nos_0\ p; q\ nos_1, \sigma\rangle \xrightarrow{\rho} \langle nos_0\ q[\rho]\ nos_1, \sigma'\rangle}$$

$$\textbf{(Choice0)}\ \frac{\langle nos_0\ p\ nos_1, \sigma\rangle \xrightarrow{\rho} \langle nos_0\ p'\ nos_1, \sigma'\rangle}{\langle nos_0\ p + q\ nos_1, \sigma\rangle \xrightarrow{\rho} \langle nos_0\ p'\ nos_1, \sigma'\rangle}$$

$$\textbf{(Choice1)}\ \frac{\langle nos_0\ p\ nos_1, \sigma\rangle \xrightarrow{\rho} \langle nos_0\ p'\ nos_1, \sigma'\rangle}{\langle nos_0\ q + p\ nos_1, \sigma\rangle \xrightarrow{\rho} \langle nos_0\ p'\ nos_1, \sigma'\rangle}$$

Figure 8: Semantics of the Component Specification Language

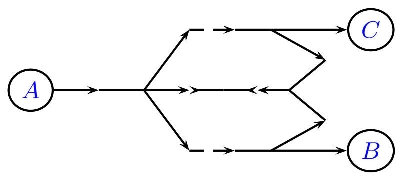

Figure 9: ExRouter Connector

basic connector types.

Example 5 (ExRouter) Consider a connector that is supposed to route the input data (present at its single source node) nondeterministically to one of its two sink nodes. This behavior can be specified as a basic connector with the following process expression:

$$\begin{aligned} ExRouter = & (read(a,x) \circ \\ & (write(b,x) + write(c,x))); ExRouter \end{aligned}$$

Note that with our set of basic connectors defined before, there is no need to define a new ExRouter connector since it can be specified as a composed Reo connector. Figure 9 depicts this composed connector.

4 Tool Support

In order to mechanize reasoning about Reo models, we have translated our operational semantics to Maude rewriting logic. The translation is made possible due to the operational nature of our semantics and allows for symbolic execution and model checking of Reo connectors in the Maude tool-set. In this section, we explain the outline of this translation together with examples of simulation and property checking on the previously presented Reo models.

4.1 A Maude Primer

Maude [1, 10] is a term rewriting language based on a rewriting logic. A specification in Maude can be divided into two parts: functional and system modules. The functional module specifies underlying sorts, basic operations on them, a set of (conditional) equations among terms, and (conditional) memberships between terms and sorts. Operations on a sort may also be defined with their inherent attributes such as associativity and commutativity. Following is an example of a simple functional module from our implementation, defining sequences of natural numbers. The following example is a simple functional module of a vending machine specification taken from [1].

Example 6 (Vending Machine: Functional Module) The following code defines a functional module `VENDING-MACHINE-STRUCTURE`.

```
fmod  VENDING-MACHINE-SIGNATURE is

    sorts Coin Item Marking .
    subsorts Coin Item < Marking .
    op null : -> Marking [ctor] .
    op $    : -> Coin [ctor] .
    op q    : -> Coin [ctor] .
    op a    : -> Item [ctor] .
    op c    : -> Item [ctor] .
    op _ _  : Marking Marking ->
                  Marking [ctor assoc comm id : null] .
endfm
```

In the above code, first basic sorts of the vending machine specification `Coin`, `Item`, and `Marking` are defined. Then, it is specified that a `Coin` or an `Item` are both `Marking`s using the subsort construct. Then, constants of types coin and item are defined. Namely, `$` stands for a dollar, `q` for a quarter, `a` represents an apple and `c` denotes a cake. Furthermore, composition of two markings are defined to be another marking and composition operation is defined to be commutative and associative with identity element `null`.

Note that operations in functional modules are terminating, confluent, and deterministic. We use functional modules to define the static part of our system. Built upon the preciously specified functional module, the next example defines the dynamic (nondeterministic and reactive) behavior of the wending machine as a system module.

Example 7 (Vending Machine: System Module) The following system module defines the nondeterministic behavior of a vending machine based on the signature defined in Example 6:

```
mod  VENDING-MACHINE is

    protecting VENDING-MACHINE-SIGNATURE .

    var M : Marking            .

    rl [add-q]  : M => M q      .
    rl [add-$]  : M => M $      .
    rl [buy-c]  : $ => c        .
    rl [buy-a]  : $ => a q      .
    rl [change] : q q q q => $ .

endm
```

4.2 Reo in Maude

As all other Maude specifications, the specification of our operational semantics in Maude consists of two types of modules: functional and system modules. For Reo semantics, we implemented three functional modules: `Node`, `Channel` and `System`. Module `Node` defines the concepts of node, node set, data sequence, and valuation. Since these are straightforward implementations of concepts such as sequence and set, we dispense with presenting and explaining detailed code of this module. Basic connector types, their cardinality constraints and the concepts of sink and source nodes are defined in module `Channel`. Here, we give a summary of the implementation of this module in Maude.

```
fmod CHANNEL is

pr NODE                                         .

sort CT                                         .

sort 1to1CT                                     .
ops Syn 1FifoE  Lossy : -> 1to1CT [ctor]        .
op  1Fifo             : Data -> 1to1CT [ctor]   .
op  Fifo              : Seq  -> 1to1CT [ctor]   .

sort 2to0CT                                     .
op Synd : -> 2to0CT [ctor]                      .

sort 1to2CT                                     .
op Fork : -> 1to2CT [ctor]                      .

sort 2to1CT                                     .
op Merge : -> 2to1CT [ctor]                     .

subsorts 1to1CT 2to0CT 1to2CT 2to1CT < CT       .
```

```
sort CI                                 .

var nos nos0           : NodeSet        .

op _ _ _  : NodeSet CT NodeSet -> [CI]                                  .

cmb ( nos0 ct  nos1 ) : CI
                if (ct :: 1to1CT) /\
                   (size ( nos0 ) == 1) /\ (size ( nos1 ) == 1) .
.
.
.

op source_ : CI -> NodeSet                      .
op sink_   : CI -> NodeSet                      .
op hidden_ : CI -> NodeSet                      .

.
.
.
eq hidden ( nos0 ct nos1 )      = nos0 cap nos1                  .

endfm
```

The above code, first defines the basic connector type sort `CT` and its different subsorts `1to1CT`, etc. and defines basic connector types as constants in these sorts. Afterwards, it defines the sort basic connector instance `CI` and constraints on the cardinality of node sets by using the conditional membership construct `cmb`. Finally, operations defining source, sink, and hidden nodes of a basic connector instance are declared and specified using equations.

The last functional module of our implementation, defines the notion of a system (connector) as follows.

```
fmod SYSTEM is

pr NODE                       .
pr CHANNEL                    .

sort Sys                      .
subsort  CI < Sys             .


op _;_   : Sys Sys -> Sys [ctor comm assoc]                     .


op source_ : Sys -> NodeSet                                     .
op sink_   : Sys -> NodeSet                                     .
op hidden_ : Sys -> NodeSet                                     .


.
.
.
eq hidden ( sys0 ; sys1 ) = hidden ( sys0 ) , hidden ( sys1 ) ,
                        ( ( source ( sys0 ) cap sink   ( sys1 ) ) ,
                          ( sink   ( sys0 ) cap source ( sys1 ) ) )     .

endfm
```

The most important part of this module, is to define a system as a basic connector instance or composition of systems. Furthermore, the notions of source, sink, and hidden are lifted to systems as defined in Definition 1.

The second part of the Reo specification in Maude is the definition of system modules. This part specifies the dynamic non-deterministic behavior of systems as a rewrite theory. In our case, the original behavior of our system is specified in terms of SOS rules and thus we have to turn deduction rules into conditional rewrite rules. For the axioms of our semantics, this is a straightforward translation: almost the same SOS rules can be used as Maude conditional rewrite rules. Following rewrite rules are translations of our SOS specification for synchronous, synchronous-drain, and one-place FIFO basic connector types.

```
crl [Syn] :      * < (a Syn b) -
                        (((a mapsto (u ; d) ) , (b mapsto w)) , sig ) >
                 =>
                 < (a Syn b) -
                        (((a mapsto u) , (b mapsto ( d ; w )))  , sig) >

                 if

                (d =/= emptyEl ) .

crl [Synd] :     * < ( ( a , b )  Synd NoSEmptyset ) -
                        (((a mapsto (u ; d) ) , (b mapsto ( w ; dp ))) , sig ) >
                 =>
                 < ( ( a , b )  Synd NoSEmptyset ) -
                        (((a mapsto u ) , (b mapsto  w )) , sig ) >

                 if

                (d =/= emptyEl ) .

crl [OFifo0] :   * < (a 1FifoE b) -  ((a mapsto (u ; d))  , sig ) >
                        =>
                 < (a 1Fifo(d) b) - ((a mapsto u)  , sig) >

                 if

                (d =/= emptyEl ) .

crl [OFifo1] :   * < (a 1Fifo(d) b) -  ((b mapsto u) , sig ) >
                        =>
                 < (a 1FifoE  b) - ((b mapsto (d ; u)) , sig) >

                 if

                (d =/= emptyEl ) .
```

Rewriting in Maude is modulo reflexivity, congruence, and transitivity, all three of which are harmful for implementation of our SOS semantics. In other words, it is not true that for any state, a self transition is possible in our semantics (thus, contradicting reflexivity). Similarly, it is not the case that if a subsystem of a Reo circuit can perform a total transition, it can perform it in any context (due to congestion freedom constraint, thus contradicting congruence). By the same token, transitivity is also harmful to our semantics. To overcome this, we annotate each state before a transition with a `*` so that we can distinguish between total transitions due to SOS rules and those due to reflexivity. We use the same trick to distinguish between total transitions and partial ones. Rules **(Join)** and **(Subsys)** are defined as follows:

```
crl [Join] :    * < ( sys0  ;  sys1 )  - sig >
                        =>
                < ( sysp0 ; sysp1 )  - sigp  >

                if

                * < sys0  - sig > => < sysp0 - sigp0 >           /\

                * < sys1  - sigp0 > => < sysp1 - sigp >          /\

                ( hidden ( sys0 ; sys1 ) isEmptyIn sigp )         .


crl [Subsys] :  ( <  sys0  - sig >
                        subtrans sys )
                        =>
                <  sysp0  - sigp >

                if

                ( sys0 subseteq sys )                           /\

                * < sys0  - sig > => < sysp0 - sigp >           /\

                ( hidden ( sys ) isEmptyIn sigp )               .
```

To translate the rule **(System)**, we need a way to specify negative premises (impossibility of a rewrite) in Maude. It is not possible in core Maude to do this. Thus, we must specify a meta-level operation for this purpose. In the release version available at the time of preparing this document, the transformation of terms between these levels is not yet implemented. We used an alpha version (alpha83) of Maude with this support (thanks to the Maude development and support team). A summary of the code for the meta level operation and **(System)** rule is given below.

```
crl [System] : $ < sys0 ; sys1  - sig >
                =>
               < sysp0 ; sys1  - sigp >

               if
```

Reo Model	Basic Connector Instances	Single Step Rewrites / Time	Total Behavior Rewrites / Time
Example 1	4	3.0×10^2 / .04s	1.8×10^4 / .22s
Example 2	6	2.1×10^5 / 2.9s	1.2×10^6 / 19.3s
Example 5	8	2.0×10^7 / 350s	4.1×10^7 / 818s

Table 1: Comparison of Simulation Results

```
            ( < sys0  - sig >   subtrans (sys0 ; sys1 ) ) => < sysp0 - sigp >   /\

            ( cannotMove < sys0  - sig > with sys1 in  (sys0 ; sys1) )          .

op cannotMove _ with  _ in _  : Conf Sys Sys -> Bool .

eq cannotMove  < sys0  - sig > with   (ci  ; sys1)  in sys =
        ( ( cannotRewrite ( < sys0 ; ci  - sig > subtrans sys ) ) and
        ( ( cannotMove < sys0 ; ci  - sig > with sys1 in sys )  and
          ( cannotMove < sys0  - sig >      with sys1 in sys ) ) ) .


eq cannotMove < sys0  - sig > with  ci  in sys =
        cannotRewrite  ( < sys0 ; ci  - sig > subtrans sys ).

op sysMove : Term -> Bool .

ceq sysMove ( T ) =
        canMove? :: Result4Tuple
        if canMove? :=
                metaXapply(['ReoTotal], T , 'Subsys , none , 0, unbounded, 0  ) .
```

In the above code, we specify that a Reo system can make a transition if either all of its parts can participate in the transition or it can make a maximal move. The maximal move predicate is then specified using the meta-level function `sysMove`.

4.3 Simulation and Model Checking

We implemented the Reo connectors specified in Examples 1, 2, and 5 in Maude and simulated their behavior. Table 1 summarizes the number of rewrites and the amount of time used for simulating a single step and the total behavior of these components on input sequences of size 2. The timing is measured on a personal computer with Pentium 700 processor and 128 megabytes of RAM running Redhat Linux 7.3.

We also applied model checking techniques to verify the behavior of the *exclusive router* connector of Example 5. The specification of this connector states that if a data item appears at its source node, the data item should be communicated in a single step to one of its sink nodes (and not to both). This correctness criteria can be specified in Linear Temporal Logic as follows.

$$\begin{array}{l}\Box((ExRouter(A) = u \frown d \wedge \\ \quad (ExRouter(B) = v \wedge \\ \quad (ExRouter(C) = w \wedge) \Rightarrow \\ \quad \bigcirc\, ((ExRouter(A) = u \wedge \\ \quad (ExRouter(B) = d \frown v \ \texttt{xor} \\ \quad (ExRouter(C) = d \frown w)))\end{array}$$

One can deduce from the semantics of Reo that the behavior of this component is symmetric with respect to the value of data item d. Thus, we can safely generalize the above property, once we prove it correct for a particular data value (e.g., $d = 1$). Furthermore, we assumed that the data values u, v, and w at the sink and the source nodes are empty since they do not influence the behavior of the system at each step. We model-checked this simplified property on the connector of Example 5. The rewriting engine performed 7.4×10^7 rewrites to exhaust the state space and it took about 25 minutes on the same computer to model-check this property.

4.4 Lessons Learned from the Implementation

The Maude implementation of our operational semantics helped us to gain insight and confidence in its underlying SOS semantics. Using the simulation toolkit, we were able to observe the behavior of different connectors and match them with the intuition behind them. In several cases, we were able to find errors or shortcomings in our initial SOS semantics. Since formalizing the semantics is the first step into the formal world, there is often no complete way of checking the correctness of this formalization except for checking it against the underlying intuition. Thus, we believe that prototyping languages in a simulation and model-checking environment, such as Maude, is of major help and importance in this regard.

Maude was a very convenient choice for our purpose since we could obtain a faithful translation of our SOS rules into Maude rewrite rules. This way, we saved a huge effort in proving the correctness of our translation. Thus, we can recommend Maude as a rapid prototyping environment for formalisms and languages with Structural Operational Semantics.

However, as it can be seen from our simulation results, the infamous *combinatorial explosion*, disallows using model (checking) based techniques for analyzing any practical system in its entirety. Thus, using compositional techniques to reason about parts of the system and then using congruence results for compositional construction of a correct system is inevitable.

5 Related Work

Coordination and Components Coordination languages offer abstraction layers for specification of component interaction. Coordination paradigms can be categorized into two main classes of *data-driven* and *control-driven* [18]. Data-driven coordination is about providing an abstraction layer for data communication among components. Such an abstraction layer is usually in the form of a shared data space. Linda [9] and Gamma [7] are typical examples of data-driven coordination languages. Control-driven coordination languages are concerned with imposing an extraneous control strategy on the interactions of black-box components. Synchronizers [11] and Manifold [2, 8] are instances of control-driven coordination languages. Reo can be regarded as a successor to Manifold. One of the main advantages of Reo is that it supports compositional construction of connectors (and architectural styles). Alfa [12] is an architectural description language that follows a connector metaphor similar to that of Reo and uses the automata-based semantics of Reo to verify the behavior of its composed software architectures. The ideas that we presented here, can be used for mechanization and formalization of Alfa, as well.

Reo Semantics In [5], a coalgebraic formal semantics for Reo connectors is developed in terms of relations on infinite *timed data streams*. We regard this semantics as *the reference semantics* for Reo, for it precisely specifies the initial intuition behind Reo connectors. The declarative, relational nature of this semantics is one of its strengths; nevertheless, it also makes it difficult to operationalize and execute directly for applications such as simulation or model checking. In [20, 21], an application of the general theory of coalgebraic stream calculus is presented. Ultimately, this work may yield analysis-like methods and tools for solving the timed-data-stream equations of Reo connectors and their composition.

In [6], an automata-based formalism, called *constraint automata*, is proposed for modeling Reo connectors. In constraint automata the transitions are labeled with the names of the nodes that exhibit data-flow activity (e.g., a read or write) and a constraint equation that must be satisfied by the data items involved. By going for a transition system semantics and bisimulation as an equivalence on Reo connectors, we are close to the most distinguishing end of Van Glabeek's semantic spectrum [23]. While automata and language-equivalence based semantics are placed on the other (least distinguishing) end of this spectrum, the case is not evident for constraint automata of [6] since for example, both language equivalence and bisimulation are reflected upon in [6]. An advantage of our semantics, compared to that of [6], is that it uses the de-facto standard of Structural Operational Semantics. This makes the semantics both more accessible for the rest of the research community and allows utilization of existing theories and implementation tools available for SOS semantics (as already shown in Section 4). Furthermore, modeling unbounded primitives or even bounded primitives with unbounded data domains is impossible with Constraint Automata. Bounded large data domains cause an explosion in the Constraint Automata model which becomes problematic. In the SOS semantics, however, we abstract away from actual data domains, and therefore large or even unbounded data domains present no problem.

SOS in Maude There have been a few other attempts to translate structural operational semantics into Maude rewriting logic. In [25] and [24], SOS semantics of CCS and LOTOS are translated into Maude, respectively. In an unpublished note [13], Meseguer and Braga present a general framework for implementing SOS semantics in Maude. Introduction of negative premises to the framework of [13] can be considered as our contribution to implementing SOS semantics in Maude.

6 Conclusion

In this paper, we presented a structural operational semantics for Reo. This semantics is then translated to Maude rewriting logic in order to benefit from the existing tools available around Maude. Due to the close similarities in the underlying formal theories of SOS and Maude the presented translation is rather straightforward and proves to be a faithful representation of the original semantics. The translation allows a system designer to evaluate component-based software architectures formally by animating and model checking their corresponding Reo connector models in the Maude tool-set.

Acknowledgment The authors thank Steven Eker and the members of the Maude development and support team. We also acknowledge Michel Reniers' valuable comments on early versions of this paper.

References

[1] The Maude system. Available from `http://maude.cs.uiuc.edu/`.

[2] Farhad Arbab. The IWIM Model for Coordination of Concurrent Activities. In Paolo Ciancarini and Chris Hankin, editors, *Proceedings of the First International Conference on Coordination Models and Languages*, volume 1061 of *Lecture Notes in Computer Science*, pages 34–56. Springer-Verlag, Berlin, Germany, 1996.

[3] Farhad Arbab. Reo: A channel-based coordination model for component composition. *Mathematical Structures in Computer Science*, 14(3):329–366, 2004.

[4] Farhad Arbab and Farhad Mavaddat. Coordination through channel composition. In Farhad Arbab and Carolyn L. Talcott, editors, *Proceedings of 5th International Conference on Coordination Models and Languages (COORDINATION'02)*, volume 2315 of *Lecture Notes in Computer Science*, pages 22–39. Springer-Verlag, Berlin, Germany, 2002.

[5] Farhad Arbab and Jan J. M. M. Rutten. A coinductive calculus of component connectors. In *Proceedings of 16th International Workshop on Algebraic Development Techniques (WADT'02)*, Lecture Notes in Computer Science. Springer-Verlag, Berlin, Germany, to appear.

[6] Christel Baier, Marjan Sirjani, Farhad Arbab, and Jan J. M. M. Rutten. Modeling component connectors in Reo by constraint automata. *Science of Computer Programming*, 2004. To appear. Current draft available at `http://web.informatik.uni-bonn.de/I/baier/papers/SCPJournal04.pdf`.

[7] Jean-Pierre Banâtre, Pascal Fradet, and Daniel Le Métayer. Gamma and the chemical reaction model: Fifteen years after. In Cristian S. Calude, Gheorghe Paun, Grzegorz Rozenberg, and Arto Salomaa, editors, *Multiset Processing: Mathematical, Computer Science, and Molecular Computing Points of View*, volume 2235 of *Lecture Notes in Computer Science*, pages 17–44. Springer-Verlag, Berlin, Germany, 2001.

[8] Marcello M. Bonsangue, Farhad Arbab, Jaco W. de Bakker, Jan J. M. M. Rutten, Adriano Scutellá, and Gianluigi Zavattaro. A transition system semantics for the control-driven coordination language Manifold. *Theoretical Computer Science*, 240(1):3–47, 2000.

[9] Antonio Brogi and Jean-Marie Jacquet. On the expressiveness of Linda-like concurrent languages. In Ilaria Castellani and Catuscia Palamidessi, editors, *Proceedings of Fifth International Workshop on Expressiveness in Concurrency (EXPRESS'98)*, volume 16 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science, Dordrecht, The Netherlands, 1998.

[10] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn Talcott. The Maude 2.0 system. In Robert Nieuwenhuis, editor, *Rewriting Techniques and Applications (RTA'03)*, volume 2706 of *Lecture Notes in Computer Science*, pages 76–87. Springer-Verlag, Berlin, Germany, 2003.

[11] Svend Frølund and Gul Agha. A language framework for multiobject coordination. In Oscar Nierstrasz, editor, *Proceedings of the European Conference on Object Oriented Programming (ECOOP'93)*, volume 707 of *Lecture Notes in Computer Science*, pages 346–360. Springer-Verlag, Berlin, Germany, 1993.

[12] Nikunj R. Mehta and Nenad Medvidovic. Composing architectural styles from architectural primitives. In *Proceedings of 9th European Software Engineering Conference and 11th ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC-FSE03)*, pages 347–350. ACM SIGSOFT Press September 2003.

[13] José Meseguer and Christiano O. Braga. Modular rewriting semantics of programming languages. Unpulished note, available from `http://maude.cs.uiuc.edu/papers/`, 2003.

[14] Robin Milner. Calculi for synchrony and synchrony. *Theoretical Computer Science*, 25:267–310, 1983.

[15] MohammadReza Mousavi, Michel A. Reniers, and Jan Friso Groote. Congruence for SOS with data. Technical Report 04-05, Department of Computer Science, Eindhoven University of Technology, 2004.

[16] Rob van Ommering, Frank van der Linden, Kramer Jeff, and Jeff Magee. The Koala component model for consumer electronics software. *Computer*, 33(3):78–85, March 2000.

[17] Prakash Panangaden and Franck van Breugel, editors. *Mathematical Techniques for Analyzing Concurrent and Probabilistic Systems.* CRM Monograph Series. American Mathematical Society, 2004.

[18] George A. Papadopoulos and Farhad Arbab. Coordination models and languages. In Marvin Zelkowitz, editor, *The Engineering of Large Systems*, volume 46 of *Advances in Computers*, pages 330–396. Academic Press, Netherlands, 1998.

[19] Gordon D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, Aarhus, Denmark, September 1981.

[20] Jan J. M. M. Rutten. A case study in coinductive stream calculus. In *Proceedings of Second International Symposium on Formal Methods for Components and Objects (FMCO'03)*, Lecture Notes in Computer Science. Springer-Verlag, Berlin, Germany, 2004, to appear.

[21] Jan J. M. M. Rutten. Component connectors. In *[17]*, chapter 5, pages 73–87. 2004.

[22] Clemens Szyperski. Component technology: what, where, and how? In *Proceedings of the 25th International Conference on Software Engineering (ICSE'03)*, pages 684–693. IEEE Computer Society, 2003.

[23] Rob J. van Glabbeek. The linear time - branching time spectrum II. In Eike Best, editor, *International Conference on Concurrency Theory (CONCUR'93)*, volume 715 of *Lecture Notes in Computer Science*, pages 66–81. Springer-Verlag, Berlin, Germany, 1993.

[24] Alberto Verdejo. Building tools for LOTOS symbolic semantics in Maude. In Doron Peled and Moshe Vardi, editors, *Proceedings of 22nd IFIP International Conference on Formal Techniques for Networked and Distributed Systenms (FORTE'02)*, volume 2529 of *Lecture Notes in Computer Science*, pages 292–307. Springer-Verlag, Berling, Germany, 2002.

[25] Alberto Verdejo and Narciso Martí-Oliet. Implementing CCS in Maude 2. In Fabio Gadducci and Ugo Montanari, editors, *Proceedings of Fourth International Workshop on Rewriting Logic and its Applications (WRLA'02)*, volume 71 of *Electronic Notes on Theoretical Computer Science*, pages 239–257. Elsevier Science, Dordrecht, The Netherlands, 2002.