Formal Semantics and Analysis of Component Connectors in Reo

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

[1] *TU/ Eindhoven, Eindhoven, The Netherlands*

[2] *CWI, Amsterdam, The Netherlands*

[3] *Tehran University and IPM, Tehran, Iran*

Abstract

We present an operational semantics for a component composition language called Reo. Reo connectors exogenously compose and coordinate the interactions among individual components that comprise a complex system, into a coherent collaboration. The formal semantics we present here paves the way for a rigorous study of the behavior of component composition mechanisms. 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

Massively parallel and distributed systems provide a platform for building such large and complex applications that they introduce new challenges for software technology. Component-based software development has been proposed as a means to tackle the increasing complexity of software development. Components are assumed to be separate and 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 pieces of connecting code must match different requirements of the composed components, while implementing various aspects of the system behavior that lie outside of individual component boundaries. This code is 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 the context of the (abstract) behavioral models of components. Nevertheless, 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, often incarnated as arcane communication and specialized scripting language code.

Reo [2] is a coordination language that addresses this problem by offering an expressive model and a graphical language for building component connectors through composition of primitive channels. Reo can be used to model and construct connectors that implement some specified behavior and to formally reason about them. Because the constructed Reo circuits directly constitute the so-called glue code, once proved 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 present a formal semantics for Reo in Plotkin's style of Structural Operational Semantics (SOS) [15]. Using this semantics, we benefit from the results of research and tools available for SOS. Particularly, we can formally observe and reason about operational behavior of component connectors. To realize this potential, we have implemented our SOS semantics 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 give a concise and informal introduction to Reo. Then, in Section 3, we define the syntax and the semantics of a workable subset of Reo, as well as notions of equality and refinement for Reo connectors. Our implementation of Reo in Maude is presented subsequently in Section 4 together with a few examples of our experiments with this implementation. 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.

Due to space limitation, we do not present the details of our implementation here. A more detailed version of this paper can be found in [14]. The implementation code in Maude and its accompanying documentation with several examples are available at the following URL:
http://www.win.tue.nl/~mousavi/reo_maude.tar.gz.

2 Reo: A Coordination Language

Reo [2] is a channel-based exogenous coordination language 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 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*, which can be produced by applying Reo's `join` composition operator. 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, which are inherent in Reo. We restrict our attention to connectors that have static graphical representations as Reo circuits.

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. 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. A composed connector consists of channels and nodes. A set of channel ends coincide on a node, and each channel end coincides on exactly one node. A node on which only *source* channel ends coincide is a *source* node and a node where only *sink* channel ends coincide is a *sink* node. Nodes with both *source* and *sink* coincident channel ends are called *mixed* nodes, which for our purposes in this paper, are internal or *hidden nodes* of a connector.

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 mixed nodes and internal basic connectors constitute its hidden internal structure. Components cannot connect to, read from, or write to mixed nodes. Instead, data-flow through mixed nodes is totally specified by the circuits they belong to.

A component can write data items to a source node that it is connected to. The write operation succeeds only if all (source) channel ends coincident on the node accept the data item, in which case the data item is transparently written to every source end coincident on the node. A source node, thus, acts as a *replicator*. A component can obtain data items from a sink node that it is connected to through input operations. An input operation succeeds only if at least one of the (sink) channel ends coincident on the node offers a suitable data item; if more than one coincident channel end offers suitable data items, one is selected nondeterministically. A sink node, thus, acts as a nondeterministic *merger*. A mixed node is a self-contained "pumping station" that combines the behavior of a sink node (merger) and a source node (replicator) in an atomic iteration of an endless loop: in every iteration a mixed node nondeterministically selects and takes a suitable data item offered by one of its coincident sink channel ends and replicates it into all of its coincident source

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

Fig. 1. Reo Syntax

channel ends.

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 the remainder, we first 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 as we illustrate in the extended version of this paper [14]. There, we also present initial ideas on how to systematically define new channel types and use them within our framework.

3 Syntax and Operational Semantics of Reo

Every channel represents a simple connector with two ends. In this paper, we use the notion of *basic connector* to represent the notion of channel in Reo, with the addition of two basic 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 replication, pumping, and choice points. In such simplified circuits, each node has at most one coincident source and at most one coincident sink channel ends. This simplification is justified here because in this paper, we restrict ourselves to static Reo circuits only, and for every static Reo circuit, there exists a simplified Reo circuit with simplified nodes and Fork and Merge connectors, and vice versa. In this section, we restrict ourselves to a fixed set of the basic connector types which is expressive enough to specify most practical systems (in fact, this subset is Turing complete [2]). In [14], we show how to generalize this subset.

3.1 Syntax

Abstract Syntax

The abstract syntax of a connector in the subset of Reo that we consider in this section is given in Figure 1. In this figure, a Reo connector *Sys* (also called circuit or system) 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 systems. 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 nodes are irrelevant.

Basic connector types in BCT stand for the following intuitions:

(i) *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).

(ii) *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.

(iii) *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.

(iv) *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.

(v) *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.

(vi) *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.

(vii) *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.

(viii) *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 $\longrightarrow$, $-\!\Box\!\!\rightarrow$, $-\!a\!\!\rightarrow$, $-\![u]\!\!\rightarrow$ 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 buffer data (i.e., contain a non-empty sequence of data). 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.

(i) For a basic connector instance $ci = \langle nos_0\ ct\ nos_1 \rangle$ $(ct \in BCT)$, we define $hid(ci) \stackrel{\triangle}{=} nos_0 \cap nos_1$, $source(ci) \stackrel{\triangle}{=} nos_0 \setminus hid(ci)$ and $sink(ci) \stackrel{\triangle}{=} nos_1 \setminus hid(ci)$.

(ii) For a circuit $Sys = ci \cup Sys'$:

- $hid(Sys) \stackrel{\triangle}{=} hid(ci) \cup hid(Sys') \cup (source(ci) \cap sink(Sys')) \cup (source(Sys') \cap sink(ci))$;
- $source(Sys) \stackrel{\triangle}{=} (source(ci) \cup source(Sys')) - hid(Sys)$ and
- $sink(Sys) \stackrel{\triangle}{=} (sink(ci) \cup sink(Sys')) - hid(Sys)$.

Note that in the above definition, we do not exclude the possibility of specifying cyclic Reo circuits (even for a basic connector).

3.2 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 on each node is a finite (possibly empty) sequence of data, denoted by $DataSeq$ (the empty data sequence is denoted by $[]$). 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 $Val : Names \rightarrow DataSeq$ 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

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

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

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

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

$$\textbf{(OFifo0)}\;\frac{}{\begin{array}{l}\langle a \multimap\!\!\!\square\!\!\!\rightarrow b, \{a \mapsto u\frown d\} \uplus \sigma\rangle \to \\ \langle a \,\text{-}\!d\!\!\rightarrow b, \{a \mapsto u\} \uplus \sigma\rangle\end{array}} \qquad \textbf{(OFifo1)}\;\frac{}{\begin{array}{l}\langle a \,\text{-}\!d\!\!\rightarrow b, \{b \mapsto u\} \uplus \sigma\rangle \to \\ \langle a \multimap\!\!\!\square\!\!\!\rightarrow b, \{b \mapsto d\frown u\} \uplus \sigma\rangle\end{array}}$$

$$\textbf{(IFifo0)}\;\frac{}{\begin{array}{l}\langle a \,\text{-}\![u]\!\!\rightarrow b, \{a \mapsto v\frown d\} \uplus \sigma\rangle \to \\ \langle a \,\text{-}\![d\frown u]\!\!\rightarrow b, \{a \mapsto v\} \uplus \sigma\rangle\end{array}} \qquad \textbf{(IFifo1)}\;\frac{}{\begin{array}{l}\langle a \,\text{-}\![u\frown d]\!\!\rightarrow b, \{b \mapsto v\} \uplus \sigma\rangle \to \\ \langle a \,\text{-}\![u]\!\!\rightarrow b, \{b \mapsto d\frown v\} \uplus \sigma\rangle\end{array}}$$

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

$$\textbf{(Filter1)}\;\frac{d \notin pat}{\begin{array}{l}\langle a \,\text{-}\!\{pat\}\!\!\rightarrow b, \{a \mapsto u\frown d, b \mapsto v\} \uplus \sigma\rangle \to \\ \langle a \,\text{-}\!\{pat\}\!\!\rightarrow b, \{a \mapsto u, b \mapsto v\} \uplus \sigma\rangle\end{array}}$$

$$\textbf{(Fork)}\;\frac{}{\begin{array}{l}\langle a \,{-}\!\!\prec (b,c), \{a \mapsto u\frown d, b \mapsto v, c \mapsto w\} \uplus \sigma\rangle \to \\ \langle a \,{-}\!\!\prec (b,c), \{a \mapsto u, b \mapsto d\frown v, c \mapsto d\frown w\} \uplus \sigma\rangle\end{array}}$$

$$\textbf{(Merge0)}\;\frac{}{\begin{array}{l}\langle (a,b) \succ\!\!{-}\, c, \{a \mapsto u\frown d, b \mapsto v, c \mapsto w\} \uplus \sigma\rangle \to \\ \langle (a,b) \succ\!\!{-}\, c, \{a \mapsto u, b \mapsto v, c \mapsto d\frown w\} \uplus \sigma\rangle\end{array}}$$

$$\textbf{(Merge1)}\;\frac{}{\begin{array}{l}\langle (a,b) \succ\!\!{-}\, c, \{a \mapsto u, b \mapsto v\frown d, c \mapsto w\} \uplus \sigma\rangle \to \\ \langle (a,b) \succ\!\!{-}\, c, \{a \mapsto u, b \mapsto v, c \mapsto d\frown w\} \uplus \sigma\rangle\end{array}}$$

Fig. 2. Reo Semantics: Part 1, Basic Connector Semantics

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

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

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

Fig. 3. Reo Semantics: Part 2

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. 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 **(Fork)** as copying an available data item from its source to its sink nodes. Similarly, rules **(Merge0)** and **(Merge1)** 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 consisting of three levels. The first level is described by rule **(Join)**. This rule specifies that a system can perform a *total transition*, denoted by $\to$, 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 $\to_{\subseteq Sys}$. 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 describe their transitions using our semantics.

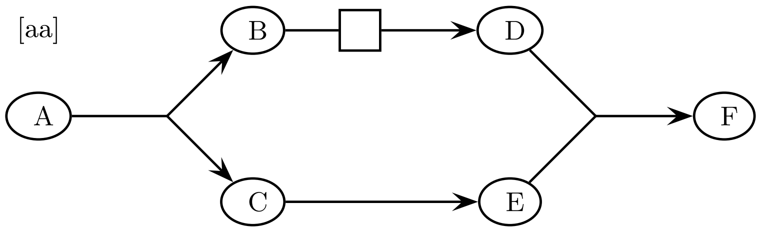

Fig. 4. A Replicator Connector

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 by (1) communicating the first data items to nodes B and C and subsequently, (2) filling the FIFO buffer, and finally (3) moving the pending data item from C to E and finally to F. Thus, in the first step the whole system is involved and this results in having a copy of the first item in the FIFO buffer and another copy in node F (the sink node). By studying the semantics, one can find out that in the next step only two connectors of the system (namely, FIFO and merge connectors) are involved. Therefore, the second step results in copying the data item inside the FIFO buffer to node F. We have simulated the behavior of this connector using our implementation described in Section 4.

3.3 (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 [10]. In this section, we define the following notion of initially stateless (bi-)similarity (following [12]) which relate the behavior of connectors with respect to all consistent initializations of data sequences. The notion should provide us with sufficient tools for replacing (parts of) Reo circuits with their more concrete implementations.

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 $Sys \leq Sys'$ ($Sys \leftrightarroweq Sys'$), 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 consistent σ, $(\langle Sys_0, \sigma\rangle, \langle Sys_1, \sigma\rangle) \in R$.

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 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.

4.1 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. Functional modules define the basic data types of our specification (sets, sequences, etc.) and operations on them (intersection, concatenation, etc.). For Reo semantics, we implemented three functional modules: `Node`, `Channel` and `System`. These modules, apart from defining the above mentioned basic sorts and operations, define concepts such as data valuation and configuration and operations such as extracting source, sink and hidden nodes of a connector.

The second part of the Maude specification of Reo is the definition of system modules. This part specifies the dynamic nondeterministic behavior of a system as a rewrite theory. In our case, the original behavior of our system is specified in terms of SOS rules and thus we must turn the 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. For example, the following rewrite rule is the specification of rule **(Syn)** in Maude.

```
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 ) .
```

The above rule, is a conditional rewrite rule specifying that if a synchronous channel has a non-empty sequence of data items on its source node, it will be rewritten to a synchronous channel with the first data item moved to its sink node (`crl` keyword stands for conditional rewrite rule and `=>` is the symbol for rewriting). Similarly, all other axioms are copies of their corresponding rules in the operational semantics (modulo syntactic changes).

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 the congestion freedom constraint), thus contradicting congruence. Analogously, the transitivity of rewrite is also harmful to our semantics. To overcome this, we annotate each operational state before the transition with a `*` so that we can distinguish between total transitions due to SOS rules and those due to reflexivity and to prevent rewrites due to congruence and transitivity. We use the same trick to distinguish between total transitions and partial ones, namely, by augmenting source terms with special symbols. For instance, the operational rule **(Subsys)** is implemented as follows.

```
crl [Subsys]    :* < ( sys0 ; sys1 ) - sig > subtrans sys
                => <( sysp0 ; sysp1 ) - sigp >
                if ( sys0 subseteq sys )                       /\
                   * < sys0 - sig > => < sysp0 - sigp0 >   /\
                   ( hidden ( sys ) isEmptyIn sigp )          .
```

To translate the rule **(System)**, we need a way to specify negative premises (the impossibility of a transition or rewrite). To this, we have to use the new reflective feature of Maude, which allows us to interpret rewrite theories as ordinary objects. This way, we can check, from meta-level, whether a particular rewrite is allowed by a rewrite theory or not. A summary of the code for the meta level operation and the **(System)** rule are given below.

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
Interleaver	6	2.1×10^5 / 2.9s	1.2×10^6 / 19.3s
ExRouter	8	2.0×10^7 / 350s	4.1×10^7 / 818s

Table 1
Comparison of Simulation Results

```
 crl [System]:* < sys0 ; sys1 - sig >
                => <( sysp0 ; sys1 ) - sigp >
                if * < sys0 - sig > subtrans (sys0 ; sys1)
                => < sysp0 - sigp >                              /\
                cannotMove < sys0 - sig > with sys1 in (sys0 ; sys1)   .
op sysMove : Term -> Bool .
ceq sysMove ( T ) =
   canMove? :: Result4Tuple
   if canMove? := metaXapply(['ReoTotal], T , 'Subsys , none , 0, unbounded, 0 ) .
```

4.2 Simulation and Model Checking

After embedding our operational semantics in Maude, we implemented the Reo connectors specified in Examples 1, and two other examples 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 (the transitive closure of the single step semantics) 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 described in details in [3] and verified its characterizing LTL property correct. 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 the correctness property. We refer to the extended version of this paper [14] for the details of the examples and the verified property.

4.3 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. Thus, we believe that prototyping languages semantics in a simulation and model-checking environment, such as Maude, is of major help and importance.

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. A viable method is to use compositional verification techniques and combining model checking and theorem proving techniques.

5 Related Work

5.1 Coordination and Components

Reo can be regarded as a successor to the control-driven coordination language Manifold. One of the main advantages of Reo is that it supports compositional construction of connectors (and architectural styles). Alfa [11] 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.

In [6], for a language of *stateless connectors* (excluding FIFO connectors, for example), a categorical semantics and a sound and complete equational theory (with respect to tile bisimilarity of [9]) are given. It is an interesting topic for future research to find sound and complete axiomatizations for Reo connectors (for example, with respect to the notion of initially stateless bisimilarity given in this paper).

5.2 Reo Semantics

In [4], 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 [5], 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. An advantage of our semantics, compared to that of [5], 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.

5.3 SOS in Maude

There have been other attempts to translate structural operational semantics into Maude rewriting logic. In [17] and [16], SOS semantics of CCS and LOTOS are translated into Maude, respectively. Also, a translation of Modular SOS (SOS with a structure on labels) to Maude is defined in [8] and implemented in [7]. Inspired by our initial attempt in this paper, we have recently generalized our translation of SOS to Maude and prototype a general-purpose tool for SOS Meta-theory in Maude [13]

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.

Acknowledgements. Michel Reniers provided useful comments on an earlier version of this paper. Comments of the anonymous referees of the FOCLASA workshop are also gratefully acknowledged.

References

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

[2] F. Arbab. Reo: A channel-based coordination model for component composition. *Mathematical Structures in Computer Science*, 14(3):1–38, 2004.

[3] F. Arbab, C. Baier, J. J. Rutten, and M. Sirjani. Modeling component connectors in Reo by constraint automata. In *Proceedings of FLOCASA'03*, volume 97 of ENTCS, pp. 25–46, Elsevier Science, 2004.

[4] F. Arbab and J. J. Rutten. A coinductive calculus of component connectors. In *Proceedings of WADT'02*, volume 2755 of LNCS. pp. 34–55, Springer, 2002.

[5] C. Baier, M. Sirjani, F. Arbab, and J. Rutten. Modeling component connectors in Reo by constraint automata. *Science of Computer Programming*, 2004. to appear.

[6] R. Bruni, I. Lanese and U. Montanari, Complete Axioms for Stateless Connectors. In *Proceedings of CALCO'05*, 2005. to appear.

[7] C. d. O. Braga. *Rewriting Logic as a Semantic Framework for Modular Structural Operational Semantics.* PhD thesis, Departamento de Informática, Pontifícia Universidade Católica de Rio de Janeiro, Brasil, 2001.

[8] C. de O. Braga, E. H. Haeusler, J. Meseguer, and P. D. Mosses: Mapping Modular SOS to Rewriting Logic. *Proceedings of LOPSTR'02*, volume 2664 of *LNCS*, pages 262–277, Springer, 2003.

[9] F. Gadducci and U. Montanari. The tile model. In *Proof, Language and Interaction: Essays in Honour of Robin Milner*, pp. 133166. MIT Press, 2000.

[10] R. J. van Glabbeek. The linear time - branching time spectrum II. In E. Best, editor, *International Conference on Concurrency Theory (CONCUR'93)*, volume 715 of *LNCS*, pages 66–81. Springer, 1993.

[11] N. R. Mehta and N. Medvidovic. Composing architectural styles from architectural primitives. In *Proceedings of the ESEC-FSE03*, pages 347–350. ACM SIGSOFT, 2003.

[12] M. R. Mousavi, M. Reniers, and J. F. Groote. Congruence for SOS with data. In *Proceedings of LICS'04*, pages 302–313. IEEE CS, 2004.

[13] M. R. Mousavi and M. A. Reniers. Prototyping SOS meta-theory in Maude. *Proceedings of SOS'05*, ENTCS, Elsevier Science, 2005. to appear.

[14] M. R. Mousavi, M. Sirjani, and F. Arbab. Specification and verification of component connectors. Technical Report CSR-04-15, Eindhoven University of Technology, 2004.

[15] G. D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University, 1981.

[16] A. Verdejo. Building tools for LOTOS symbolic semantics in Maude. *Proceedings of FORTE'02*, volume 2529 of *LNCS*, pages 292–307. Springer, 2002.

[17] A. Verdejo and N. Martí-Oliet. Implementing CCS in Maude 2. *Proceedings of WRLA'02*, volume 71 of *ENTCS*, pages 239–257. Elsevier Science, 2002.