Universiteit
Leiden
The Netherlands

Model checking of component connectors

Izadi, M.

Citation
Izadi, M. (2011, November 6). *Model checking of component connectors*. *IPA Dissertation Series*. Retrieved from https://hdl.handle.net/1887/18189

Version:	Corrected Publisher's Version
License:	Licence agreement concerning inclusion of doctoral thesis in the Institutional Repository of the University of Leiden
Downloaded from:	https://hdl.handle.net/1887/18189

Note: To cite this publication please use the final published version (if applicable).

3 Formal Modeling of Component Connectors

Reo is an exogenous coordination language for compositional construction of coordination systems. Constraint automaton has been defined as the operational semantics of Reo. In the first two sections of this chapter, we describe Reo and constraint automata. In the third section, we briefly present the other semantic formalisms that have been introduced for Reo.

3.1 Reo: A Channel Based Coordination Language

Reo is a coordination language which can model and specify coordination of a set of components trough networks of channels or compositional connector built out of primitive channels. In this section, we introduce Reo primitives and compositional coordination constructions which can be obtained by using it as introduced in [13, 14, 19, 25].

3.1.1 Reo Primitives

Reo is a coordination language which is based on a calculus of channels [13, 14, 19, 30]. By using Reo specifications, complex component connectors can be organized in a network of channels and build in a compositional manner. The simplest connectors in Reo are a set of *channels* with well-defined behavior supplied by users. Reo can be used as a coordination language for concurrent processes or as a "glue language" for compositional construction of connectors that orchestrate component instances in a component based system. The emphasis in Reo is on connectors and their composition only, not on the entities that connect to, communicate and cooperate through these connectors.

Reo uses a simple notion of channels and can model any kind of peer-to-peer communication. The only requirements for a channel used in a Reo network are that the channel should have two channel ends, called as *sink* or *source* ends, and a well-defined semantics which constraints or relates the flow of data through these ends. At a source end data items enter the channel by performing corresponding write operations. Data items are received from a channel at its sink end by performing corresponding read operations. Reo allows for an open ended set of channel types with user defined semantics. Some primitive channels relevant for this thesis are shown in Figure 3.1 by their graphical representations.

Every synchronous or FIFO channel has a source and a sink end. A *synchronous channel* (abbreviated by *Sync*) has no buffer and accepts a data item through its source end if and only if it can simultaneously dispense it through its sink. A *FIFO1 channel* is represented graphically by a small box in the middle of an arrow. Writing a data item at the source end of a FIFO1 is enabled as long as the buffer is empty. The effect of writing d is that d will be stored in the buffer. Reading at the sink end is enabled if the buffer is full, in which case the data item is taken off from the buffer. FIFO channels with two or more buffer cells can be produced by composing several FIFO1 channels [30].

A *lossy synchronous channel* (abbreviated as *LossySync*) is similar to synchronous channel, except that it always accepts all data items through its source end. If it is possible for it to simultaneously dispense the data item through its sink (e.g. there is a take operation pending on its sink) the channel transfers the data item, otherwise the data item is lost. For a

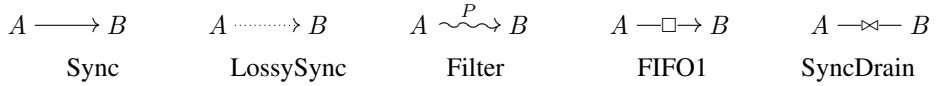


Figure 3.1: Some useful channel-types in Reo

synchronous filter channel, its pattern P (for our purpose here, formalized as a subset P of data set $\mathcal{D}$) specifies the type of data items that can be transmitted through the channel. Any value $d \in P$ is accepted through its source end iff its sink end can simultaneously dispense d. All data items $d \notin P$ are always accepted through the source end but are immediately lost. The *P-producer* is a variant of a synchronous channel whose source end accepts any data item $d \in \mathcal{D}$, but the value dispensed through its sink end is always a data element $d \in P$.

Two very useful channels for the design of complex coordination principles in Reo are the *synchronous and asynchronous drains*. Because a drain has no sink end, no data value can ever be obtained from these channels. Thus, a synchronous drain accepts a data item through one of its ends iff a data item is also available for it to simultaneously accept through the other end as well. All data accepted by this channel are lost. An asynchronous drain accepts and loses data items through its two source ends, but never simultaneously. *synchronous and asynchronous spout* are duals of their corresponding drain channel types, as they have two sink ends.

3.1.2 Compositional Connectors

Every channel represents a simple connector with two ends. More complex connectors are constructed in Reo out of the simpler ones using its *join* operation. Joining of two connectors is plugging their ends together into nodes such that the data exchanged through the ports that coincide on the same node are synchronized, but the I/O behaviors of the other ports remain as they were before.

Reo defines a ***connector*** as a set of channel ends and their connecting channels organized in a graph of ***nodes*** and ***edges*** such that:

- Zero or more channel ends coincide on every node.

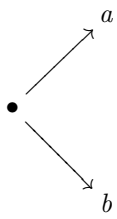
- Every channel end coincides on exactly one node.

- There is an edge between two (not necessarily distinct) nodes if and only if there is a channel end which coincides on each of those nodes.

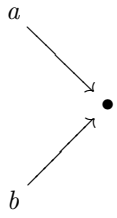
Let x be a channel end and N be a node. We use $x \mapsto N$ to denote that x coincides on N, and $\hat{x}$ to denote the unique node on which the channel end x coincides. For a node N, the set $[N] = \{x | x \mapsto N\}$ is the set of all channel ends coincide on N and is partitioned into the disjoint sets $Src(N)$ and $Snk(N)$, denoting the sets of source and sink channel ends that coincide on N, respectively. The nodes of a Reo network represent sets of channel ends. They arise through Reo's *join* operator and can be classified into *source, sink* and *mixed* nodes, depending on whether all channel ends that coincide on a node N are source ends (then N is a source node), sink ends (then N is a sink node) or whether N combines sink and source ends (then N is a mixed node). Source and sink nodes represent input and output ports where components might connect to the network. The mixed nodes serve as routers where

data items can be transmitted through the network. More formally, a node N is a ***source node*** if $Scr(N) \neq \emptyset \wedge Snk(N) = \emptyset$. Analogously, N is a ***sink node*** if $Snk(N) \neq \emptyset \wedge Scr(N) = \emptyset$. A node N is a ***mixed node*** if $Scr(N) \neq \emptyset \wedge Snk(N) \neq \emptyset$.

Components are connected only to sink or source nodes, they cannot connect to mixed nodes. At most one component can be connected to a source or a sink node. A source node acts as a *replicator*, while a sink node acts as a nondeterministic *merger*. Consider the following source node:



A write operation succeeds on the node if the source ends a and b are both ready to accept the data item, in which case it is written to both source ends. Thus, the data item is replicated. On the other hand, take the following sink node:



A read or take operation succeeds on the node only if at least one of the sink ends a or b is ready to offer a suitable data item into the node. If both of them are ready to offer data, one is selected non-deterministically. Thus, the sink node acts as a nondeterministic merger.

A complex connector has a graphical representation, called *Reo circuit or network*. The graph representing a connector is not directed. However, for each channel end x_c of a channel c, we use the directionality of x_c to assign a local direction on the neighborhood of $\hat{x_c}$ to the edge that represents c. Complex connectors are constructed out of simpler ones using the *join* operation. The join operation is defined only on nodes. Joining two nodes N_1 and N_2 destroys both nodes and produces a new node N, in which, $[N] = [N_1] \cup [N_2]$. This operation allows construction of arbitrary complex connector graphs involving any combination of channels picked from an open-ended set of channel types. The semantics of a connector is defined as a composition of the semantics of its constituent channels and nodes. Reo does not provide any channels, thus, it dose not define their semantics either. What Reo defines is the composition of channels into connectors and the semantics of this composition through the semantics of its three types of nodes.

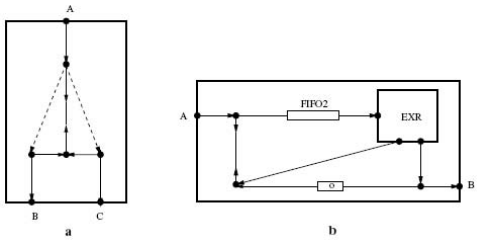


Figure 3.2: Exclusive router (a) and shift-lossy (b) channels designed by primitive channels of Reo [19]

As examples of Reo networks, Figure 3.2 shows the implementations of an exclusive router and a shift-lossy FIFO1 connectors in Reo. The intuitive behavior of the *exclusive router* is such that through its source node A, it obtains a data item d from its environment and delivers d to one of its sink nods B or C. If both B and C are ready to accept d, the exclusive router nondeterministically chooses one of them for delivery. A *shift-lossy FIFO1 channel* behaves the same as a FIFO1 channel, except that writing to its source end never blocks. If at the time of a write operation its buffer is full, the stored data item in the buffer is lost and the new data item replaces it in the buffer. For more examples and details of Reo circuits see [13, 14, 19, 30].

3.2 Basic Theory of Constraint Automata

Constraint automata were introduced as the semantics of Reo first in [30]. The semantics of constraint automata is based on timed data streams and their languages. In other words, constraint automata are defined as acceptors of the tuples of timed data streams and two constraint automata are (language - theoretically) equivalent if they accept exactly the same set of tuples of timed data streams.

In this section, we describe the basic theory of constraint automata, their semantics and their composition operators as introduced in [30].

3.2.1 Timed Data Streams

Before introducing the notion of constraint automata, we need to introduce some preliminary notations and the notions of timed data streams, their tuples and languages.

Definition 3.1 Let V be any set. We define the sets V^* and V^ω as the sets of all finite and infinite sequences (words or strings) over V, respectively. Obviously, we can define the set V^ω of all streams (infinite sequences) over V as the set of functions $w{:}\mathbb{N} \rightarrow V$. For a stream $w \in V^\omega$ we call $w(0)$ the initial value of w. The (stream) *derivative* w' of a stream w is defined as $w'(k) = w(k+1)$. We write $w^{(i)}$ for the i-th derivative of w which is defined by $w^{(0)} = w$ and $w^{(i+1)} = w^{(i)\prime}$. Note that $w^{(i)}(k) = w(i+k)$, for all $k, i \geq 0$.

Definition 3.2 Let $\mathcal{D}$ be a fixed and nonempty set of data that can be sent or received via channels. A *timed data stream* is an ordered pair $\langle \alpha, a \rangle$, in which, α is an infinite sequence of data and a is a time stream consisting of increasing positive real numbers that go to infinity. We denote the set of all timed data streams by TDS. Thus, more formally we have:

$$TDS = \{\langle \alpha, a \rangle \in \mathcal{D}^\omega \times \mathbb{R}_+^\omega |\ \forall n \geq 0\, (a(n) \langle a(n+1)\ and\ \lim_{n\to\infty} a(n) = \infty)\},$$

where $\mathbb{R}_+ = [0, \infty)$ is the set of all positive real numbers including zero.

A timed data stream $A = \langle \alpha, a \rangle$ represents occurrences of events at a port A and consists of a data stream $\alpha \in \mathcal{D}^\omega$ and a time stream $a \in \mathbb{R}_+^\omega$ of increasing positive real numbers. The time stream a indicates the moments $a(n)$ at which their respective data items $\alpha(n)$ occur at port A.

Let $\mathcal{N} = \{A_1, \ldots, A_n\}$ be a countable set of port names. With each port $A_i \in \mathcal{N}$, we associate a timed data stream recording both the data communicated and the time when the communication happens. That is, we define $TDS^{\mathcal{N}}$ as the set of all TDS-tuples consisting of one timed data stream for each port in $\mathcal{N}$.

Definition 3.3 Let $\mathcal{N} = \{A_1, \ldots, A_n\}$ be a countable set of port names. Then,

$$TDS^{\mathcal{N}} = \{(\langle \alpha_1, a_1 \rangle, \ldots, \langle \alpha_n, a_n \rangle) \mid \langle \alpha_i, a_i \rangle \in TDS,\ i = 1, \ldots, n\}$$

contains all *TDS-tuples* consisting of one timed data stream for each port.
A *TDS-language* L is a subset of $TDS^{\mathcal{N}}$, namely $L \subseteq TDS^{\mathcal{N}}$.

We use a family-notation $\theta = (\theta|_i)_{A_i \in \mathcal{N}}$ for the elements of $TDS^{\mathcal{N}}$, where $\theta|_i$ stands for the projection of θ along the port A_i. Simultaneous exchange of data between a set of ports can be detected by inspecting the time when communications happen. For this purpose, for $\theta \in TDS^{\mathcal{N}}$ we define $\theta.time$ to be a stream in $\mathbb{R}_+^\omega$ obtained by merging the streams $(\theta|_n)_r$ in increasing order. More formally,

Definition 3.4 Let $\theta = (\langle \alpha_1, a_1 \rangle, \ldots, \langle \alpha_n, a_n \rangle) \in TDS^{\mathcal{N}}$ be a TDS-tuple. $\theta.time$ is the time stream which, can be derived by merging the time streams $a_1, \ldots, a_n$ in an increasing order. Thus, for $\theta.time$ we have:

$$\theta.time(0) = min\{a_i(0) \mid i = 1, \ldots, n\},$$

and for all $j \geq 1$:

$$\theta.time(j) = min\{a_i(k) \mid a_i(k) > \theta.time(j-1),\ i = 1, \ldots, n,\ \ k = 0, 1, \ldots\}.$$

Also, $\theta.N$ is a *name-sets stream* over $2^{\mathcal{N}}$, in which, $\theta.N = \theta.N(0), \theta.N(1), \ldots$ and

$$\theta.N(k) = \{A_i \in \mathcal{N} \mid a_i(l) = \theta.time(k)\ for\ some\ l \in \{0, 1, 2, \ldots\}\}.$$

Let $\delta{:}N \to \mathcal{D}$ be a data assignment function, where, $N \neq \emptyset$ and $N \subseteq \mathcal{N}$. The notation of $\delta = [A \mapsto \delta_A : A \in N]$ designates the data assignment function that assigns to any TDS name $A \in N$ the value $\delta_A \in \mathcal{D}$. Now, $\theta.\delta = \theta.\delta(0), \theta.\delta(1), \ldots$, is a *data-assignments stream*, in which, $\theta.\delta(k)$ represents the observed data flow at time point $\theta.time(k)$, namely,

$$\theta.\delta(k) = [A_i \mapsto \alpha_i(l_i) : A_i \in \theta.N(k)]$$

where $l_i \in \{0, 1, 2, \ldots\}$ is the unique index with $a_i(l_i) = \theta.time(k)$.

In the rest of this section, we assume that the data set $\mathcal{D}$ is fixed and predefined. Thus, we do not mention it in the definitions.

3.2.2 Constraint Automata: the Operational Semantics of Reo

Constraint automata can be viewed as acceptors for tuples of timed data streams that are observed at certain ports $A_1, ..., A_n$. The rough idea is that such an automaton observes the data occurring at $A_1, ..., A_n$ and either changes its state according to the observed data or rejects the data if there is no corresponding transition in the automaton. Further, constraint automata are augmented with the names of their ports $A_1, ..., A_n$, where A_i stands for the ith TDS. Each transition in a constraint automaton is labeled with a pair N, g such that N is a non-empty subset of $\mathcal{N} = \{A_1, ..., A_n\}$, and g is a guard which, constrains data in the TDS of ports referenced in N.

Definition 3.5 Let $\mathcal{D}$ be a set of data and $\mathcal{N}$ be a set of port names. A *data constraint* g over sets $\mathcal{D}$ and $\mathcal{N}$ is a proposition that can be constructed using the following abstract grammar:

$$g ::= \mathit{true} \mid d_A = d \mid g_1 \vee g_2 \mid \neg g \qquad d \in \mathcal{D}, \quad A \in \mathcal{N}.$$

In the above grammar, for every port name $A \in \mathcal{N}$, d_A is a variable whose value at each time instance is the value of the data item exchanged through the port A at that time. Thus, $d_A = d$ means that the value of data on port A is d.

Let DC be the set of all data constraints over the names set $\mathcal{N}$ and the data set $\mathcal{D}$, defined by the above grammar. Obviously, DC contains some logically equivalent propositions. We use $DC(\mathcal{N}, \mathcal{D})$ as the set of all logically different data constraints over the names set $\mathcal{N}$ and the data set $\mathcal{D}$. In other words, $DC(\mathcal{N}, \mathcal{D})$ is the partitioned version of DC using the logical equivalence relation. Thus, for all data constraints equivalent to g, we use the notion of $g \in DC(\mathcal{N}, \mathcal{D})$. Obviously, if $\mathcal{N}$ and $\mathcal{D}$ are finite then $DC(\mathcal{N}, \mathcal{D})$ is finite.

Now, we introduce the notion of constraint automaton as originally has been defined in [30]:

Definition 3.6 A *constraint automaton* is a quadruple $C = \langle Q, \mathcal{N}, \longrightarrow, Q_0 \rangle$ where,
Q is a set of states,
$\mathcal{N}$ is a set of names,
$\longrightarrow \subseteq Q \times 2^{\mathcal{N}} \times DC \times Q$ is a set of transitions,
$Q_0 \subseteq Q$ is the set of initial states.
We write $p \xrightarrow{N,g} q$ instead of $(p, N, g, q) \in \longrightarrow$ and call N the name set and g the guard of the transition. For every transition $p \xrightarrow{N,g} q$ it is supposed that $N \neq \emptyset$ and $g \in DC(N, \mathcal{D})$.

A constraint automaton $C = \langle Q, \mathcal{N}, \longrightarrow, Q_0 \rangle$ is finite, if the sets $Q, \mathcal{N}$ and $\mathcal{D}$ are finite. Also, C is said to be *deterministic* if its set of initial states Q_0 is a singleton and for each state q, a set of port names N and a data assignment $\delta{:}N \rightarrow \mathcal{D}$ there is at most one transition $q \xrightarrow{N,g} q'$ with $\delta \models g$.

The intuitive operational behavior of a constraint automaton is as follows. It starts in its initial state q_0. If the current state is q, then C waits until data items occur at some of its ports

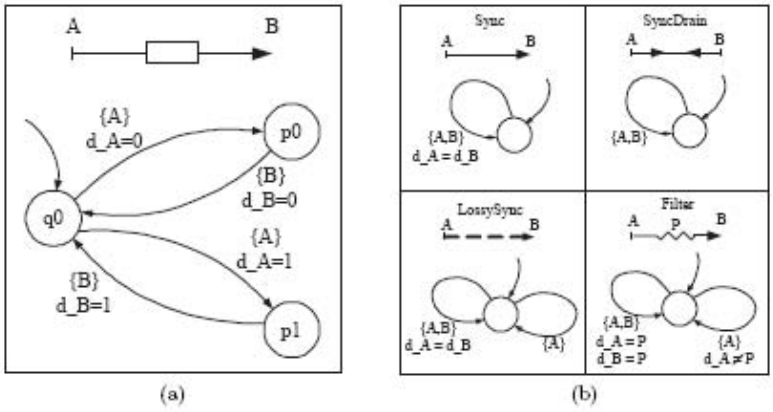


Figure 3.3: Constraint automata for some basic channels in Reo [30]

$A_1, ..., A_n$. Suppose data item d_1 occurs at A_1 and data item d_2 at A_2 while (at this moment) no data is observed at the other ports $A_3, ..., A_n$. This triggers the automaton to check the data constraints of the outgoing transitions of state q with a name set $\{A_1, A_2\}$ to choose a transition t, such that its guard is satisfied by $A_1 \mapsto d_1$ and $A_2 \mapsto d_2$ resulting in state p. If there is no $\{A_1, A_2\}$-transition from q whose data constraint is fulfilled then C remains waiting (perhaps indefinitely).

Example 3.1 Figure 3.3 shows the constraint automata models of the set of Reo channels which we introduced in the previous section [30]. Figure 3.3(a) is the constraint automaton model of a FIFO1 channel from a source node A to sink B over the data set $\{0, 1\}$. Figure 3.3(b) shows the constraint automata models of the remaining channels with the consideration that $d_A = d_B$ is an abbreviation for $\vee_{d \in \mathcal{D}}(d_A = d \wedge d_B = d)$ where $\mathcal{D}$ is the set of data $\{0, 1\}$. In the case of the filter channel it must be that $P \subseteq \mathcal{D}$.

Like ordinary automata are acceptors of finite strings, constraint automata are acceptors of tuples of timed data streams. Informally, each element of the tuple is associated with a port of the system and corresponds to the streams of observed data communicated through this port together with the time when the data has been observed.

Definition 3.7 Let $C = \langle Q, \mathcal{N}, \longrightarrow, Q_0 \rangle$ be a constraint automaton and φ be a TDS-tuple, $\varphi \in TDS^{\mathcal{N}}$.
- An *infinite run* for φ in C is an infinite sequence of states $r = q_0, q_1, ...,$ in which, $\forall i, q_i \in Q$ and there exists transition $q_0 \xrightarrow{N,g} q_1$ with $N = \varphi.N(0)$, $\varphi.\delta(0) \models g$ and r' is an infinite run for φ'. The infinite run $r = q_0, q_1, ...$ is an *initial infinite run*, if $q_0 \in Q_0$.
- TDS-tuple φ is accepted by a constraint automaton C if and only if there is an initial infinite run for φ in C. The language of constraint automaton C is

$$L_{TDS}(C) = \{\varphi \in TDS^{\mathcal{N}} \mid C \text{ accepts } \varphi\}.$$

The above definition of $L_{TDS}(C)$ can also be formally defined by means of the greatest fixed point of a suitably chosen monotone operator [30]. For the purpose of this thesis we found it easier to reason with the accepted language characterized by means of the (standard) notion of accepted runs.

As for the case of finite automata, using Rabin-Scott powerset construction for each nondeterministic constraint automaton there is a deterministic one which accepts the same language [30].

Further, in a finite constraint automaton C all transitions with unsatisfiable guards can be removed without any effect on the TDS language accepted by C, where a guard g of a transition $p \xrightarrow{N,g} q$ is said to be semantically unsatisfiable for N if there is no data assignment for elements of N which satisfies g (take, for example, g to be $\neg true$). In the rest of this thesis we assume without any loss of generality that all guards in a constraint automaton are satisfiable with respect to the set of names of the transition they belong to.

Remark 3.1 As we mentioned in Definition 3.6, in [30] it is presupposed that for every transition $p \xrightarrow{N,g} q$ the set of ports N is non-empty. This condition makes constraint automata the operational models of the *observable* behavior of systems' components. In other words, with this condition constraint automata are the operational models of the *interfaces* of components. If we ignore this condition, a constraint automaton may also have transitions of the form $p \xrightarrow{\emptyset,true} q$. We abbreviate this form of transitions by $p \xrightarrow{\tau} q$ and call them by τ-transitions. We refer to constraint automata that are allowed to have τ-transitions by the term *constraint automata with τ-transitions*. τ-transitions can be considered as models of internal behaviors of the components that are not exactly observable in the interfaces.

Remark 3.2 Constraint automata are, in general, not closed under complement. Informally this is due to the fact that constraint automata do not have *final* states. If we augment the definition of constraint automaton by a set of final states and use Büchi acceptance condition (a timed data stream is accepted if at least one of the correspondent runs for it contains one of the final states infinitely many times), we refer to the resulting automaton as a *Büchi constraint automaton*. Obviously, a constraint automaton is a Büchi constraint automaton in which all states are accepting. Its complement, however does not need to satisfy this property.

3.2.3 Composing of Constraint Automata

In the literature on constraint automata, there are two operators for composing them: join of two constraint automata and hiding a name in all transitions of a constraint automaton [30].

Join

Constraint automata can be composed by means of a join operator, the semantic counterpart of the join operator in Reo [30]. Different than the ordinary product for finite automata, the composition of two constraint automata is allowed even if they have different alphabets. In fact, the resulting constraint automaton has transitions when data occur at the ports belonging to only one of the automata, without involving the transitions or states that it inherits from the other automaton (because at that point in time, there is no data on any of its corresponding ports). More formally, the join operation for constraint automata is defined as follows:

Definition 3.8 Let $C_1 = \langle Q_1, \mathcal{N}_1, \longrightarrow_1, Q_{01}\rangle$ and $C_2 = \langle Q_2, \mathcal{N}_2, \longrightarrow_2, Q_{02}\rangle$ be two constraint automata (with or without τ-transitions). The *join* of C_1 and C_2 is the constraint automaton:

$$C_1 \bowtie_C C_2 = \langle Q_1 \times Q_2, \mathcal{N}_1 \cup \mathcal{N}_2, \longrightarrow, Q_{01} \times Q_{02}\rangle$$

where, transition relation $\longrightarrow$ is defined by the following rules:
Rule 1:

$$\frac{q \xrightarrow{N_1, g_1}_1 q', p \xrightarrow{N_2, g_2}_2 p', N_1 \cap \mathcal{N}_2 = N_2 \cap \mathcal{N}_1}{\langle q, p\rangle \xrightarrow{N_1 \cup N_2, g_1 \wedge g_2} \langle q', p'\rangle}$$

Rule 2:

$$\frac{q \xrightarrow{N_1, g_1}_1 q', N_1 \cap \mathcal{N}_2 = \emptyset}{\langle q, p\rangle \xrightarrow{N_1, g_1} \langle q', p\rangle},$$

Rule 3 (dual of rule 2):

$$\frac{p \xrightarrow{N_2, g_2}_2 p', N_2 \cap \mathcal{N}_1 = \emptyset}{\langle q, p\rangle \xrightarrow{N_2, g_2} \langle q, p'\rangle}.$$

Basically, the two automata have to agree on the data exchanged on the common ports (that is, the names used in the transition of the first automaton known to the second automaton are exactly the same as the names used by the transition of the second automaton known to the first one), and each maintains its own behavior on the other ports (as described by the last two rules).

The join of two constraint automata using the operation defined in Definition 3.8 is correct with respect to the join of their accepted TDS-languages, where the join of two TDS-languages is basically the same as defined in the theory of relational databases [30]: the projection on the common indexes (port names) of the resulting language must agree with that of the two original languages, while the projection on each indexes in one language but not in the other must agree with the projection on the same index of the language where the index belongs to. Thus, it can be shown that [30]:

Lemma 3.1 Let $C_1 = \langle Q_1, \mathcal{N}_1, \longrightarrow_1, Q_{01}\rangle$ and $C_2 = \langle Q_2, \mathcal{N}_2, \longrightarrow_2, Q_{02}\rangle$ be two constraint automata. Then:

1) $L_{TDS}(C_1 \bowtie_C C_2) = L_{TDS}(C_1) \bowtie L_{TDS}(C_2)$,

2) If $\mathcal{N}_1 = \mathcal{N}_2$ then $L_{TDS}(C_1 \bowtie_C C_2) = L_{TDS}(C_1) \cap L_{TDS}(C_2)$,

where for TDS-languages L_1 and L_2, $L_1 \bowtie L_2$ means the standard join of the languages of tuples in the theory of databases.

Example 3.2 Let us join the constraint automata representing two FIFO1 channels, one from source A to sink B and the other from source B to sink C. The automata models of the two channels and the resulting automaton are illustrated in Figure 3.4. For simplicity we assume that the data set is $\mathcal{D} = \{d\}$. Thus, every transition label contains only the set of port names that participate in firing the transition.

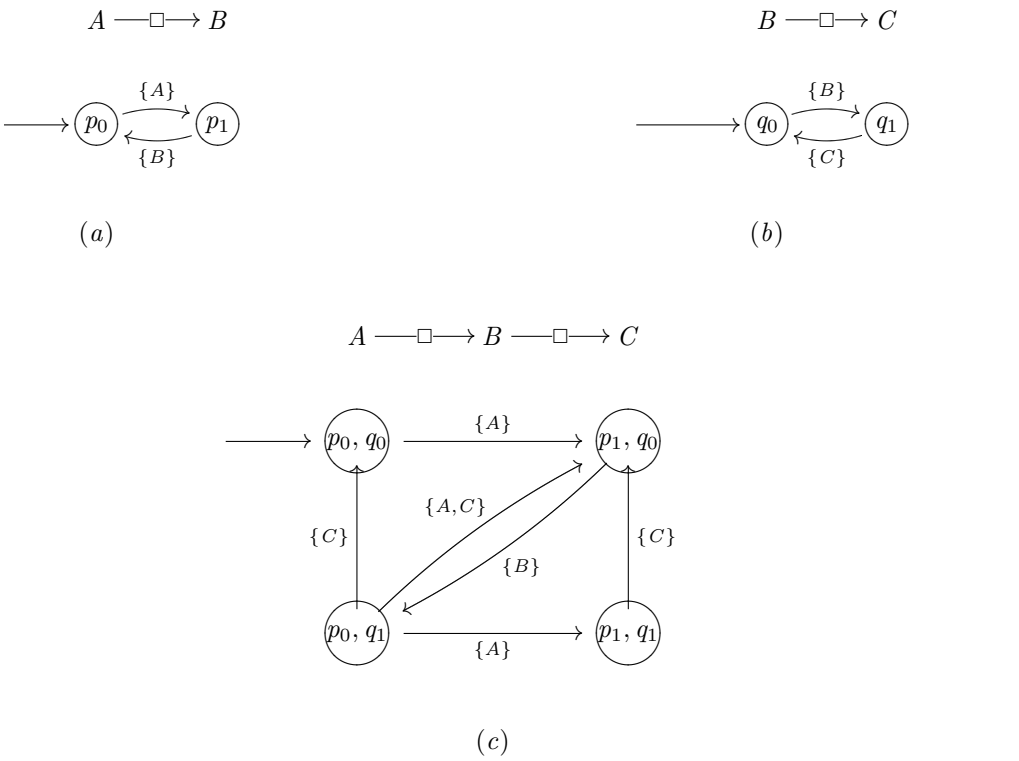


Figure 3.4: Joining of constraint automata models of two FIFO1 channels

Hiding of port Names

Now, we define hiding of a port name in all transitions of a constraint automaton. First consider constraint automata without τ-transitions. The hiding operation is defined as follows [30]:

Definition 3.9 Let $C = \langle Q, \mathcal{N}, T, Q_0 \rangle$ be a constraint automaton and $B \in \mathcal{N}$. The constraint automaton resulted by *hiding* of B in C is constraint automaton

$$\exists B[C] = \langle Q, \mathcal{N} \backslash \{B\}, \longrightarrow_B, Q_{0,B} \rangle$$

where transition relation $\longrightarrow_B$ is defined as follows:
Let $\longrightarrow^*$ be the transition relation such that $q \longrightarrow^* p$ if and only if there exists a finite path $q \overset{\{B\},g_1}{\longrightarrow} q_1 \overset{\{B\},g_2}{\longrightarrow} q_2 \ldots \overset{\{B\},g_n}{\longrightarrow} q_n$, in which, $q_n = p$ and $g_1, g_2, \ldots, g_n$ are satisfiable. Then,

$$Q_{0,B} = Q_0 \cup \{p \in Q | q_0 \rightarrow^* p, \textit{ for some } q_0 \in Q_0\}.$$

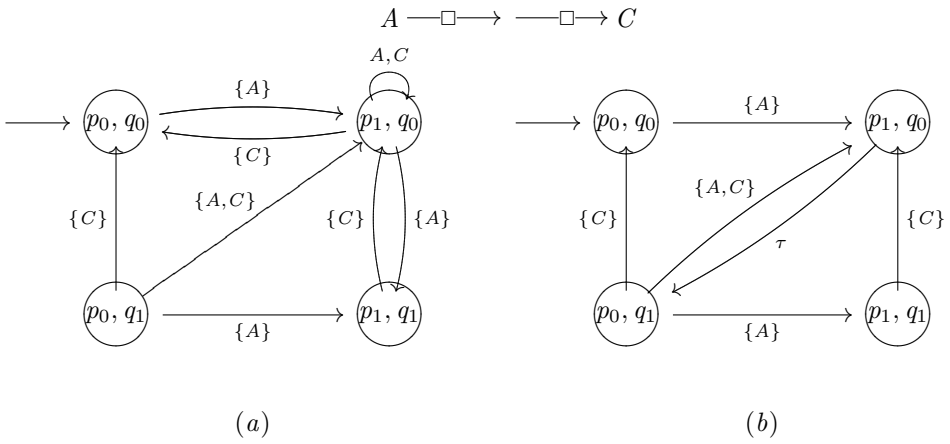


Figure 3.5: Hiding of port B in constraint automaton of Figure 3.4(c)

The transition relation $\rightarrow_B$ is defined by the rule:

$$\frac{q \rightarrow^* p, p \xrightarrow{N,g} r, N' = N \backslash \{B\} \neq \emptyset, g' = \exists B[g]}{q \xrightarrow{N',g'}_B r}$$

where $\exists B[g] = \vee_{d \in \mathcal{D}} g[d_B/d]$.

Example 3.3 Let us hide the intermediate port B in the constraint automaton illustrated in Figure 3.4(c) using Definition 3.9. The result is illustrated in Figure 3.5(a). As we expect the resulting automaton is the operational model of the observable behaviors of a FIFO channel with capacity of two. Also, observe that in this case the state $\langle p_0, q_1 \rangle$ is unreachable from the initial state, and can thus be removed.

Now, let us consider the more general case of constraint automata with τ-transitions. In this case, the definition of hiding is simpler, because it is allowed for the hiding operation to generate τ-transitions.

Definition 3.10 Let $C = \langle Q, \mathcal{N}, T, Q_0 \rangle$ be a constraint automaton (with or without τ-transitions) and $B \in \mathcal{N}$. The constraint automaton (possibly with τ-transitions) resulting from *hiding* of B in C is the constraint automaton

$$\exists B[C] = \langle Q, \mathcal{N} \backslash \{B\}, \longrightarrow_B, Q_{0,B} \rangle$$

where the transition relation $\longrightarrow_B$ is defined as follows:

$$\frac{q \xrightarrow{N,g} p, N' = N \backslash \{B\}, g' = \exists B[g]}{q \xrightarrow{N',g'}_B p}$$

and $\exists B[g] = \vee_{d \in \mathcal{D}} g[d_B/d]$.

Example 3.4 Let us hide the intermediate port B in the constraint automaton illustrated in Figure 3.4.c using Definition 3.10. The result is illustrated in Figure 3.5.b. As we expect, the resulting automaton is the operational model of a FIFO channel with capacity of two, considering its internal actions.

Remark 3.3 Obviously, if we hide a port name of a constraint automaton using Definition 3.9, the resulting automaton is language theoretically equivalent with the automaton that is the result of hiding the same port using Definition 3.10 and then eliminating all τ-transitions using the standard algorithm of eliminating all τ-transitions (ϵ-transitions) of ordinary finite automata (for this algorithm see [66]). For example, the constraint automaton illustrated in Figure 3.5.a is the result of eliminating the τ-transition in the constraint automaton illustrated in Figure 3.5.b.

3.3 Other Semantic Models for Reo

In recent years, several models have been proposed in the literature to formally capture the semantics of Reo connectors. The first formal operational model for Reo was constraint automaton that we introduced in the previous section. In the next chapters, we will introduce our proposed operational semantics for Reo using standard notion of Büchi automata. Before introducing our proposed model, in this section, we briefly review the other important semantic models of Reo and discusse some of their shortcomings in the context of the questions and aims of this thesis.

3.3.1 Co-algebraic Model of Connectors

The *co-inductive calculus of timed data streams* first proposed in [27] and then extended to the notion of *Abstract Behavior Types (ABT in short)* [14], is a simple and transparent relational model of Reo. In this calculus the behavior of a connector port is modeled as a timed data stream. As we showed in Definition 3.2, an element of the time-stream is the time at which its respective data value in the data-stream is observed at its corresponding port. The interactions of connectors are modeled as relations on timed data streams. More formally, let TDS be the set all timed data streams, then each basic connector, namely channel, is defined as a binary relation

$$R \subseteq TDS \times TDS$$

over timed data streams. For example, the synchronous channel of Reo (Sync) is modeled by the following relation:

$$Sync = \{(\langle \alpha, a \rangle, \langle \beta, b \rangle) \,|\, \alpha = \beta \wedge a = b\}$$

where $\langle \alpha, a \rangle$ and $\langle \beta, b \rangle$ respectively are the timed data streams over input and output ports of the channel.

Because the connectors are relations, their composition is modeled by relational composition. For example, the composition of two copies of the synchronous channel yields the

following binary relation over the set TDS:

$$Sync \circ Sync = \{(\langle \alpha, a\rangle, \langle \beta, b\rangle) \mid \exists \langle \gamma, c\rangle : (\alpha = \gamma \wedge a = c) \wedge (\gamma = \beta \wedge c = b)\}$$

This co-inductive model abstracts away from the connector topology, and the direction of the dataflow within the connector. Connectors are reduced to a collection of ports, and the behavior of the component-based system is expressed as a relation. Based on the relational nature of this semantics of Reo, if someone be interested to verify some properties of the communication protocol modeled by composed connectors, it should be done using a deductive verification method, as for a simple example it has been done in [27].

These models of Reo were shown to be equivalent to constraint automata, and thus unable to express several fairness constraints and context dependencies [36, 37]. Also, because the model is not an operational model it is not suitable for model checking based verification.

3.3.2 Connector Coloring Models

Connector coloring is an intuitive semantics for Reo to model dataflow behavior of connectors [47]. Colors are used to denote the presence of dataflow and its absence in connected ports. Colorings with two colors suffice to express the same class of behavior as constraint automata can express [52]. Each coloring of a connector is a solution to the synchronization constraints imposed by its channels and nodes.

By refining the set of colors to three colors and propagating the negative information about the absence of dataflow in some ports, a set of context dependencies can be expressed.

Coloring a connector in a specific state with given boundary conditions (I/O requests) provides a means to determine the routing alternatives for dataflow. The circuit representation of connectors are used to describe the dataflow behavior: each coloring corresponding to a dataflow behavior of the connector can be overlaid on top of the circuit representation to provide insight into the dataflow behavior of the individual primitives of the circuit [52]. The product composition operator for colorings has been defined such that it is associative, commutative, and idempotent. These properties make the coloring scheme with its composition operator suitable for distributed implementations [52]. In a recent work, Jongmans investigates the relationships between 2/3-color coloring models and constraint automata through a set of operators that transform one model to the other [84]. In another paper, he establishes an encoding of context sensitivity expressed in 3-coloring semantics within constraint automata [83].

Coloring based semantics is not suitable for the purpose of verification, especially by model checking which is the main purpose of this thesis. This type of semantics for Reo and its extension called tile logic [21] suffer from a number of other problems. For a survey of these problems see [36].

3.3.3 Intentional Automata

The main reason for introducing *intentional automata* [52] as an operational semantic model of Reo was to overcome the shortcoming of constraint automata in expressing context dependent behaviors of connectors. An intentional automaton explicitly models the arrival of

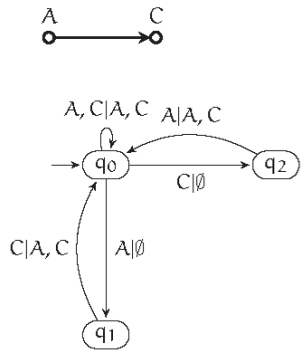


Figure 3.6: Intentional automaton model of a synchronous channel [52].

communication (or I/O) requests, and distinguishes between communication requests and the actual communications. This gives the model an extra degree of expressiveness that becomes useful in modeling systems that need to behave differently depending on the presence or absence of pending requests in their context/environment. Intentional automata are transition systems over the set of port names where the context dependencies are expressed by labeling transitions with a request set and a firing set. The request set models the context and the firing set models the subsequent behavior. More formally, in an intentional automaton over a set of ports $\mathcal{N}$, each transition from state q into state q' is of the form $q \xrightarrow{R \mid F} q'$, where $R \subseteq \mathcal{N}$ is the set of ports requesting data communication and $F \subseteq \mathcal{N}$ is the set of ports that actually participate in data communication during this transition. Also, the pending (arrived but not fired) communication requests are modeled by expanding the set of states.

For example, the intentional automaton model of a synchronous channel from input port A to output port C with the ability to suspend data communication when one of the ports is not ready to communicate, as modeled in [52], is illustrated in Figure 3.6. In this model, q_1 is the state in which port A is requesting communication while C is not ready, and similarly, q_2 is the state in which port C is requesting communication while A is not ready. The transition $q_0 \xrightarrow{A,C \mid A,C} q_0$ means that whenever both ports A and C are requesting communication, the communication is fired on them simultaneously.

Intentional automata can be composed using a product and a hiding operators very similar to their counterparts for constraint automata (see Definitions 3.8 and 3.10). The only defined semantics (equivalence relations) for intentional automata are weak and strong bisimulation relations.

Because of the strategy of modeling pending request by expanding the set of states, intentional automata have quiet a large number of states to manage the buffering and firing of such requests, which rapidly become difficult to manipulate, making intentional automata not suitable for model checking purposes [36]. While intentional automaton tries to overcome the shortcomings of constraint automaton for context dependent behaviors of components, based on the absence of the notion of final (accepting) states or sets of fairness constraints in its

syntax, it fails to express several fairness requirements.

3.3.4 Guarded and Reo Automata

Following the basic idea of intentional automaton (namely expressing request and firing sets explicitly in each transition) and similar to our guarded strings semantics for augmented Büchi automata of records (will be introduced in Chapter 5), Bonsangue *et al.* introduced a new model for Reo connectors. In general, this model is called *guarded automaton* and when it is used with some constraint to model context dependent connectors, it is called *Reo automaton* [36, 37].

Guarded automata are transition systems over a set of port names (an alphabet set in a more general view) where the context dependencies are expressed by labeling transitions with a boolean guard expression over the alphabet as their atoms and a firing set. The guard expresses the context and the firing set models the subsequent behavior. More formally, in a guarded automaton over a set of ports $\mathcal{N}$, each transition from a state q into a state q' is of the form $q \xrightarrow{g \mid f} q'$, where g is a boolean expression over the set of ports (as the atoms) and $f \subseteq \mathcal{N}$ is the set of ports that actually participate in data communication during this transition. Guarded automata are defined as acceptors of finite guarded strings, where each guarded string is a string of alternating sets of atoms of the boolean algebra of the guard expressions and sets of port names. Two guarded automata are equivalent if their languages of guarded strings are the same. In addition to this equivalence, a notion of bi-simulation relation between guarded automata has been defined such that it also implies the language equivalence. Furthermore, a product operation as the counterpart of the intersection of the languages of guarded automata has been defined.

Reo automata are guarded automata such that each transition label $g \mid f$ satisfies two criteria: *reactivity* which guaranties that data flow only on ports where I/O requests are made; and *uniformity* which captures two properties: first, that the condition of the request set corresponding precisely to the firing set is sufficient to cause firing, and second, that removing additional unfired requests from a transition will not affect the behavior of the connector [37]. It has been shown that the set of Reo automata is closed under the product of guarded automata [37].

In comparison with an intentional automaton, a Reo automaton is more dense, namely, to model context dependencies and suspended communications, it does not expand the set of state as much as an intentional automaton does. Also the product of Reo automata is more difficult to compute than the product of intentional automata and the information represented in the states of intentional automata is represented in Reo automata using compound transition labels. It can be shown that the expressive power of Reo automata is at least as much as that intentional automata. Both intentional and Reo automata have been proposed to express more semantics of Reo connectors without significant attempts to use them in verification and model checking. Because of their complicated syntax and non-standard semantics (from standard automata theory point of view) defining temporal logics over them and designing model checking algorithms will not be so simple. Also, similar to intentional automaton and based on the absence of the notion of final (accepting) states or sets of fairness constraints in the syntax of Reo automaton, it fails to express several fairness requirements.

3.3.5 Process Algebraic and Structural Operational Semantics

In addition to the above mentioned models, there are some other attempts to present formal semantics for component connectors. In the work of Barbosa et al. [32] the semantics of components are defined by the expressions of a process algebra. These expressions carry out both positive (presence) and negative (absence) information about data in ports. Complex connectors are built from basic ones using some composition operators such as join, parallel composition, and interleaving. This work needs more investigation to before it can be used for the purpose of deductive verification and is not suitable for model checking. The model needs more enhancements to express some important fairness constraints.

In the joint works of Kokash, Krause et al., reported in [91, 92, 93, 94], and [95], they map some semantic models for Reo, namely, constraint automata, timed constraint automata, and coloring semantics to the process algebraic specification language of mCRL2 [61] and show the correctness of their mappings. Also, they have build a tool for these mappings that is now a part of the tool-set *Extensible (Eclipse) Coordination Tools* (ECT) [2].

In another work, Mousavi et al. [116] express the formal semantics of Reo connectors by a set of structural operational semantic rules. In general, this semantics is not a context dependent semantics, it carries only positive information. However, to express some context dependencies, specially for the lossy synchronous channel, the authors used extra more rules to remove undesired behavior. To be able to express more fairness or context dependent behavior the set of rules needs to be expanded.

3.4 Tool Support for Reo

In addition to a mostly updated homepage for Reo [5] that helps its user to have access to the research and tools supporting Reo, there is a tool-set, called *Extensible (Eclipse) Coordination Tools* (ECT), for Reo [2]. It is a set of integrated plug-ins for the *Eclipse platform*, which offer a graphical development environment for the specification, analysis and execution of component-based software systems using the coordination language Reo [95]. It contains a set of tools implemented by a variety of researchers and programmers. Based on the last report of the tool-set ECT, presented in [95], it now contains the following tools:

- *Graphical Reo editor* by which the user can graphically construct Reo networks out of the basic channel types.
- *Animator tool* that generates animations for the flow of data in Reo nets.
- *mCRL2 conversion tool* which encodes Reo specifications generated by the graphical Reo editor to the mCRL2 specification language. This conversion is based on the works reported in detail in [91, 92, 93, 94], and [95].
- *Extensible automata editor*. Because there are several automata based models for Reo, ECT contains a tool for deriving automata based models from Reo in general. The framework contains a graphical automata editor and can also be used outside of the context of Reo [95].

- Tools for *stochastic modeling and analysis*. There are two tools for stochastic modeling and analysis in ECT: one that generates a sort of intentional automata augmented by stochastic information, called *quantitative intensional automata*, from the graphical Reo models [113, 114, 22, 23, 26], and the other which simulates the behavior of Reo nets [150, 87] using their coloring semantics as described in [47].

- *Execution engines* that execute Reo connectors. Currently there exist two implemented execution engines: one that generates the codes based on constraint automata models and the other that executes Reo connectors in a distributed platform (for more see [125, 95]).

- *Conversion tools* that convert some other modeling languages such as UML2 and BPMN to Reo.

- *Vereofy*. As we mentioned before, *Vereofy* is a model checker implemented based on the work of Baier et al. reported in [99, 98]. Briefly, it does symbolic model checking of a CTL-like temporal logic (called *BTSL* temporal logic) by using ordered binary decision diagrams (OBDD) as the data structures representing constraint automata models. The *Vereofy* tool is available through its own web-page [7] and now is also accessible through the ECT tool-set [2].