Centrum voor Wiskunde en Informatica

REPORT*RAPPORT*

SEN

Software Engineering

Software ENgineering

Modeling component connectors in Reo by constraint automata

F. Arbab, C. Baier, J.J.M.M. Rutten, M. Sirjani

REPORT SEN-R0304 JULY 31, 2003

CWI is the National Research Institute for Mathematics and Computer Science. It is sponsored by the Netherlands Organization for Scientific Research (NWO).
CWI is a founding member of ERCIM, the European Research Consortium for Informatics and Mathematics.

CWI's research has a theme-oriented structure and is grouped into four clusters. Listed below are the names of the clusters and in parentheses their acronyms.

Probability, Networks and Algorithms (PNA)

Software Engineering (SEN)

Modelling, Analysis and Simulation (MAS)

Information Systems (INS)

P.O. Box 94079, 1090 GB Amsterdam (NL)
Kruislaan 413, 1098 SJ Amsterdam (NL)
Telephone +31 20 592 9333
Telefax +31 20 592 4199

ISSN 1386-3711

Modeling Component Connectors in Reo by Constraint Automata

Farhad Arbab[a], Christel Baier[b], Jan Rutten[a], Marjan Sirjani[a,c]

[a] *Centrum voor Wiskunde en Informatica, Department of Software Engineering*
Kruislaan 413, P.O.Box 94079, 1090 GB Amsterdam, The Netherlands
{*Farhad.Arbab,Jan.Rutten,Marjan.Sirjani*}*@cwi.nl*

[b] *Institut für Informatik I, University of Bonn*
Römerstraße 164, D-53117 Bonn, Germany
baier@cs.uni-bonn.de

[c] *Sharif University of Technology*
Azadi Ave., Tehran, Iran

Abstract

Reo is an exogenous coordination language for compositional construction of component connectors based on a calculus of channels. Building automated tools to address such concerns as equivalence or containment of the behavior of two given connectors, verification of the behavior of a connector, etc. requires an operational semantic model suitable for model checking. In this paper we introduce *constraint automata* and propose them as a semantic model for Reo.

2000 ACM Computing Classification: C.2.4, D.1.3, D.1.m, D.3.2, D.3.3, F.1.2, F.3
Keywords and Phrases: Coordination, Components, Composition, Constraint Automata, Reo, Coalgebraic semantics, Streams

1 Introduction

Reo is a channel-based exogenous coordination model wherein complex coordinators, called *connectors*, are compositionally built out of simpler ones. The simplest connectors in Reo are a set of *channels* with well-defined behavior supplied by users [Arb03]. The emphasis in Reo is on connectors, their behavior, and their composition, not on the entities that connect, communicate, and cooperate through them. The behavior of every connector in Reo imposes a specific coordination pattern on the entities that perform normal I/O operations through that connector, without the knowledge of those entities. This makes Reo a powerful "glue language" for compositional construction of connectors to combine component instances into a software system and exogenously orchestrate their mutual interactions.

A coalgebraic formal semantics for Reo is developed in [AR02] in terms of relations on infinite *timed data streams*. With this semantics as our starting point, in this paper we introduce *constraint automata* and use them to present an operational model for the behavior of connectors in Reo. Constraint automata can be thought of as conceptual generalizations of probabilistic automata where simple constraints, instead of probabilities, influence applicable state transitions. The single most important composition operator in Reo, `join`, amounts to a product of automata in this model.

Our notion of constraint automata is in the spirit of I/O-automata and timed port automata [LT89, GR95]. In contrast to these, we do not distinguish between input and output ports (and hence, we do not require input enabledness) and use constraints rather than specific data values. Unlike I/O- or timed port automata, we do not follow a strictly time-synchronous approach, which becomes important when we compose constraint automata. Instead, the composition of constraint automata $\mathcal{A}_1$ and $\mathcal{A}_2$ allows transitions when data occur at the input/output ports that the resulting automaton inherits from only one of the automata $\mathcal{A}_i$, 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). Such transitions do not exist in the "one-to-many-composition" of timed port automata.
The rest of this paper is organized as follows. Section 2 is a brief overview of Reo. In Section 3, we define constraint automata and show their utility through describing the behavior of a number of simple Reo channels as examples. The product of constraint automata and hiding, which are necessary to model Reo's join operator, are defined and applied to a few examples in Section 4. In Section 5 we study various notions of equivalence, forming a foundation for algorithms and tools for verification and derivation of properties of Reo connectors. We conclude in Section 6, hinting at our current and future work on model checking and automated tools for reasoning about Reo connectors.

2 A Reo primer

Reo defines a number of operations for components to (dynamically) compose, connect to, and perform I/O through *connectors*. Atomic connectors are *channels*. The notion of channel in Reo is far more general than its common interpretation.
A channel is a primitive communication medium with exactly two ends, each with its own unique identity. There are two types of channel ends: *source* end through which data enters and *sink* end through which data leaves a channel. A channel must support a certain set of primitive operations, such as I/O, on its ends; beyond that, Reo places no restriction on the behavior of a channel. This allows an open-ended set of different channel types to be used simultaneously together in Reo, each with its own policy for synchronization, buffering, ordering, computation, data retention/loss, etc.
A connector is a set of channel ends 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 iff there is a channel one end of which coincides on each of those nodes.

A node is an important concept in Reo. Not to be confused with a location or a component, a node is a logical construct representing the fundamental topological property of coincidence of a set of channel ends, which has specific implications on the flow of data among and through those channel ends.

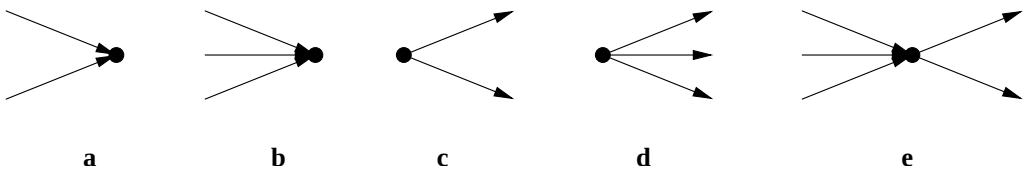

Figure 1: Nodes in Reo

The set of channel ends coincident on a node N is disjointly partitioned into the sets $Src(N)$ and $Snk(N)$, denoting the sets of source and sink channel ends that coincide on N, respectively. A node is called a *source node* if $Src(N) \neq \emptyset \wedge Snk(N) = \emptyset$. Analogously, N is called a *sink node* if $Src(N) = \emptyset \wedge Snk(N) \neq \emptyset$. A node N is called a *mixed node* if $Src(N) \neq \emptyset \wedge Snk(N) \neq \emptyset$. Figures 1.a and b show sink nodes with, respectively, two and three coincident channel ends. Figures 1.c and d show source nodes with, respectively, two and three coincident channel ends. Figure 1.e shows a mixed node where three sink and two source channel ends coincide.
Reo provides operations that enable components to connect to and perform I/O on source and sink nodes only; components cannot connect to, read from, or write to mixed nodes. At most one component can be connected to a (source or sink) node at a time. 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 destructive (take) and non-destructive (read) input operations. A take 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 ends offer suitable data items, one is selected nondeterministically. A sink node, thus, acts as a (fair) 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 channel ends. A data item is suitable for selection in an iteration only if it can be accepted by all source channel ends that coincide on the mixed node.
It follows that every channel represents a (simple) connector with two nodes. More complex connectors are constructed in Reo out of simpler ones using its `join` operation. Joining two nodes destroys both nodes and produces a new node on which all of their coincident channel ends coincide. This single operation allows construction of arbitrarily complex connectors involving any combination of channels picked from an open-ended assortment of user-defined channel types. The semantics of a connector is defined as a composition of the semantics of its (1) constituent channels, and (2) nodes. The semantics of channels are defined by the users who provide them. Reo defines the semantics of its three types of nodes, mentioned above.

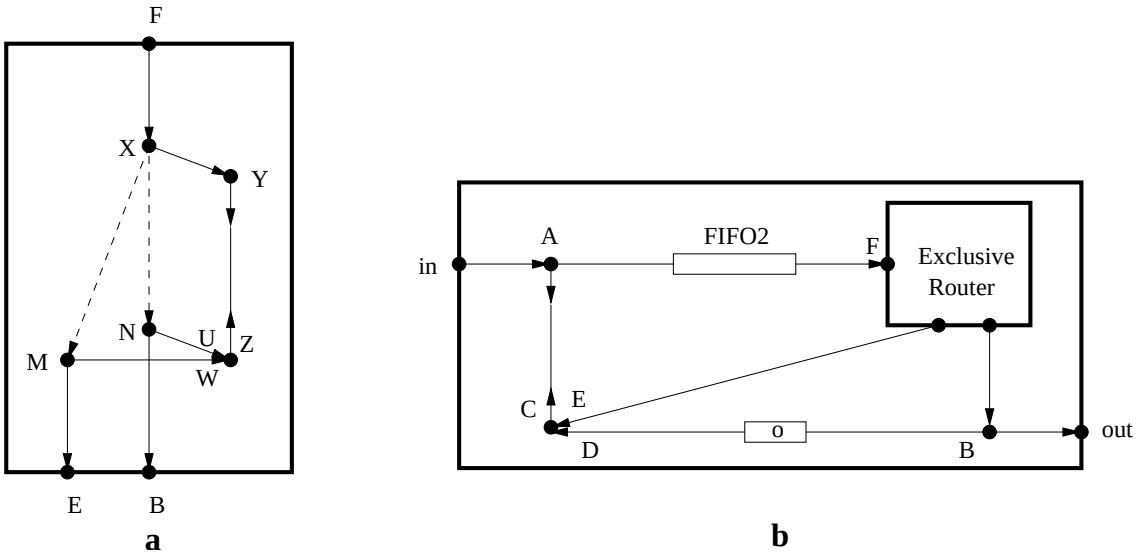

Figure 2: Exclusive router and shift-lossy FIFO1

Figures 2.a and b show two Reo connectors. We consider these connectors in more detail in Examples 4.0.7 and 4.0.8, respectively, in Section 4. Here, we use them to introduce our visual syntax for presenting Reo connector graphs and some frequently useful channel types. The enclosing thick boxes in these figures represent *hiding*: the topologies of the nodes (and their edges) inside the box are hidden and cannot be modified, yielding a connector with a number of input/output *ports*, represented as nodes on the border of the bounding box, which can be used by other entities outside the box to interact with and through the connector.
The simplest channels used in these connectors are synchronous (*Sync*) channels, represented as simple solid arrows. A Sync channel has a source and a sink end, and no buffer. It accepts a data item through its source end iff it can simultaneously dispense it through its sink. A lossy synchronous (*LossySync*) channel is similar to a Sync 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. LossySync channels are depicted as dashed arrows, e.g., in Figure 2.a. The edge BD in Figure 2.b represents an asynchronous channel with the bounded capacity of 1 (*FIFO1*), with the small box in the middle of the arrow representing its buffer. This channel can have an initially empty buffer, or as in Figure 2.b, contain an initial data value (in this case, the "o" in the box representing its buffer). Analogously, the edge AF in Figure 2.b represents an asynchronous FIFO channel with the bounded capacity of 2 (*FIFO2*), with its obvious semantics.
An example of more exotic channels permitted in Reo is the synchronous drain channel (*SyncDrain*), whose

visual symbol appears as the edges YZ and AC in Figures 2.a and b, respectively. A SyncDrain channel has two source ends. Because it has no sink end, no data value can ever be obtained from this channel. It accepts a data item through one of its ends iff a data item is also available for it to simultaneously accept through its other end as well. All data accepted by this channel are lost. A close kin of SyncDrain is the asynchronous drain (*AsyncDrain*) channel (not shown in Figure 2): it has two source ends through which it accepts and loses data items, but never simultaneously.

3 Modeling connectors by constraint automata

The semantics of Reo connectors can be defined in terms of relations on *timed data streams* (TDSs) as presented in [AR02]. In this section we introduce the notion of *constraint automata* and show how they can serve as operational models for the behavior of Reo connectors by relating the languages of these automata with timed data streams.

Let V be any set. We define the set V^ω of all streams (infinite sequences) over V as $V^\omega = \{\alpha \mid \alpha : \{0,1,2,\ldots\} \to V\}$. For convenience, we consider only infinite streams (which will correspond to infinite "runs" of our automata), but finite streams can easily be included as well (at no extra cost other than some additional case distinctions in Section 3.2). We denote individual streams as $\alpha = (\alpha_0, \alpha_1, \alpha_2, \ldots)$ (or $a = (a_0, a_1, a_2, \ldots)$). We call α_0 the *initial value* of α. The *(stream) derivative* α' of a stream α is defined as $\alpha' = (\alpha_1, \alpha_2, \alpha_3, \ldots)$. Note that $(\alpha')_n = \alpha_{n+1}$, for all $n \geq 0$. We also recall the definition of timed data streams from [AR02]:

$$TDS = \left\{ \langle \alpha, a \rangle \in Data^\omega \times \mathbb{R}_+^\omega \mid \forall n \geq 0 : \; a_n < a_{n+1} \text{ and } \lim_{n\to\infty} a_n \; = \; \infty \right\}$$

A timed data stream $\langle \alpha, a \rangle$ consists of a *data stream* $\alpha \in Data^\omega$ and a *time stream* $a \in \mathbb{R}_+^\omega$ consisting of increasing positive real numbers. The time stream a indicates for each data item α_n the moment a_n at which it is being input or output.

Constraint automata can be viewed as acceptors for tuples $(\langle \alpha_1, a_1 \rangle, \ldots, \langle \alpha_n, a_n \rangle)$ of timed data streams that are observed at certain input/output ports $A_1, \ldots, A_n$ of components. The rough idea is that such an automaton observes the data occurring at $A_1, \ldots, A_n$ and either changes its state according to the observed data or rejects it if there is no corresponding transition in the automaton. We use constraint automata as a semantic model to describe the TDS-language induced by Reo connector networks.

3.1 Definition of constraint automata

In the sequel, *Data* is a fixed and finite set of data that can be sent (and received) via channels. Constraint automata are augmented with a finite set of *names*, e.g., $A_1, \ldots, A_n$ where A_i stands for the i-th input/output port of a connector or component. A name-data-assignment denotes a function $\delta : N \to Data$ where $N \subseteq \mathcal{N}$. We use notations like $\delta = [data(A) = d_A : A \in N]$ to describe the name-data-assignment that assigns to any TDS-name $A \in N$ the value $d_A \in Data$. The transitions of the automata are labeled with pairs consisting of a subset N of $\{A_1, \ldots, A_n\}$ and a data constraint g. Data constraints are defined by the following grammar:

$$g \;::=\; \mathsf{false} \;\Big|\; \mathsf{true} \;\Big|\; data(A) = d \;\Big|\; g_1 \vee g_2 \;\Big|\; g_1 \wedge g_2$$

Here, A, B are names and $d \in Data$.[1] Data constraints (DCs) can be viewed as sets of name-data-assignments. In the sequel, we write $DC(\mathcal{N}, Data)$ to denote the set of data constraints. We often use derived DCs such as $data(A) \neq d$ or $data(A) = data(B)$ which stand for the DCs

$$\bigvee_{d' \in Data \setminus \{d\}} (data(A) = d') \quad \text{and} \quad \bigvee_{d \in Data} \big(\, (data(A) = d) \, \wedge \, (data(B) = d) \, \big),$$

respectively. The symbol $\models$ stands for the obvious satisfaction relation which results from interpreting DCs over name-data-assignments. For instance,

$$\begin{array}{lll} [data(A) = d_1, data(B) = d_2, data(C) = d_1] & \models & data(A) = data(C), \\ [data(A) = d_1, data(B) = d_2, data(C) = d_1] & \not\models & data(A) = data(B) \end{array}$$

[1] We assume a global data domain *Data* for all names. Alternatively, we can assign a data domain $Data_A$ to every name A and require type-consistency in the definition of data constraints.

if $d_1 \neq d_2$. Satisfiability and validity, logical equivalence $\equiv$ and logical implication $\leq$ of DCs are defined as usual; e.g.:

$$g_1 \equiv g_2 \quad \text{iff} \quad \text{for all name-data-assignments } \delta\text{:} \quad \delta \models g_1 \iff \delta \models g_2$$
$$g_1 \leq g_2 \quad \text{iff} \quad \text{for all name-data-assignments } \delta\text{:} \quad \delta \models g_1 \implies \delta \models g_2$$

Definition 3.1.1 [Constraint automata] A constraint automaton (over the data domain *Data*) is a tuple $\mathcal{A} = (Q, \mathcal{N}, \longrightarrow, Q_0)$ where

- Q is a finite set of states,
- $\mathcal{N}$ is a finite set of names,
- $\longrightarrow$ is a finite subset of $Q \times 2^{\mathcal{N}} \times DC \times Q$, called the transition relation of $\mathcal{A}$,
- $Q_0 \subseteq Q$ is the set of initial states.

We write $q \xrightarrow{N,g} p$ instead of $(q, N, g, p) \in \longrightarrow$. We call N the name set and g the guard of the transition. For every transition

$$q \xrightarrow{N,g} p$$

we require that (1) $N \neq \emptyset$ and (2) $g \in DC(N, Data)$. $\square$

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 $\mathcal{A}$ waits until data items occur at some of the input/output ports $A_i \in \mathcal{N}$. 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 $A_3, \ldots, A_n$. This triggers the automaton to check the data constraints of the outgoing $\{A_1, A_2\}$-transitions of state q to choose a transition

$$q \xrightarrow{\{A_1,A_2\},g} p$$

where $\left[data(A_1) = d_1, data(A_2) = d_2\right] \models g$ and move to state p. If there is no $\{A_1, A_2\}$-transition from q whose data constraint is fulfilled then $\mathcal{A}$ rejects.
Having this behavior in mind, the intuitive meaning of conditions (1) and (2) in Definition 3.1.1 is as follows. Condition (1) stands for the requirement that automata-transitions can fire only if some data occur at $A_1, \ldots, A_n$ while condition (2) formalizes the notion that the behavior of an automaton may depend only on its observed data (and not on data that will occur sometime in the future).
Figure 3 shows the constraint automata for some of the basic Reo connectors. The merger automaton in this figure models the merge behavior inherent in sink and mixed nodes in Reo.
Definition 3.1.1 allows for nondeterministic constraint automata since for a fixed state q, a nonempty subset N of $\mathcal{N}$, and a given name-data-assignment δ, there may be several transitions[2]

$$q \xrightarrow{N,g_1} q_1, \;\; q \xrightarrow{N,g_2} q_2, \;\; \ldots \;\; \text{with } \delta \models g_i, \; i = 1, 2, \ldots.$$

However, for modeling the TDS-languages induced by Reo connector networks, deterministic constraint automata are sufficient, because all nondeterministic choices in Reo are internal, and hence, can be assumed to be resolved at the level of constraint automata.[3] A constraint automaton $\mathcal{A}$ is called *deterministic* iff (1) it has a unique initial state and (2) for every state q, every N, and every name-data-assignment δ there is *at most* one transition

$$q \xrightarrow{N,g} q'$$

such that $\delta \models g$. In fact, all automata in Figure 3.1.1 are deterministic.

[2] Observe that if N' is a proper subset of N and data occur exactly at the input/output ports $A_i \in N'$ then only N'-transitions (but no N-transitions) can fire.

[3] For instance, if data simultaneously occur on both input ports A and B of the merger in Figure 3.1.1, then an internal choice decides which one to take for output. Thus, this constraint automaton must reject any TDS-tuple where data on A and B occur simultaneously, since the Reo merger (i.e., a sink or mixed node) can never observe more than one input at a time. This explains why the merger constraint automaton in this figure has no transition for the name set $\{A, B\}$ or $\{A, B, C\}$.

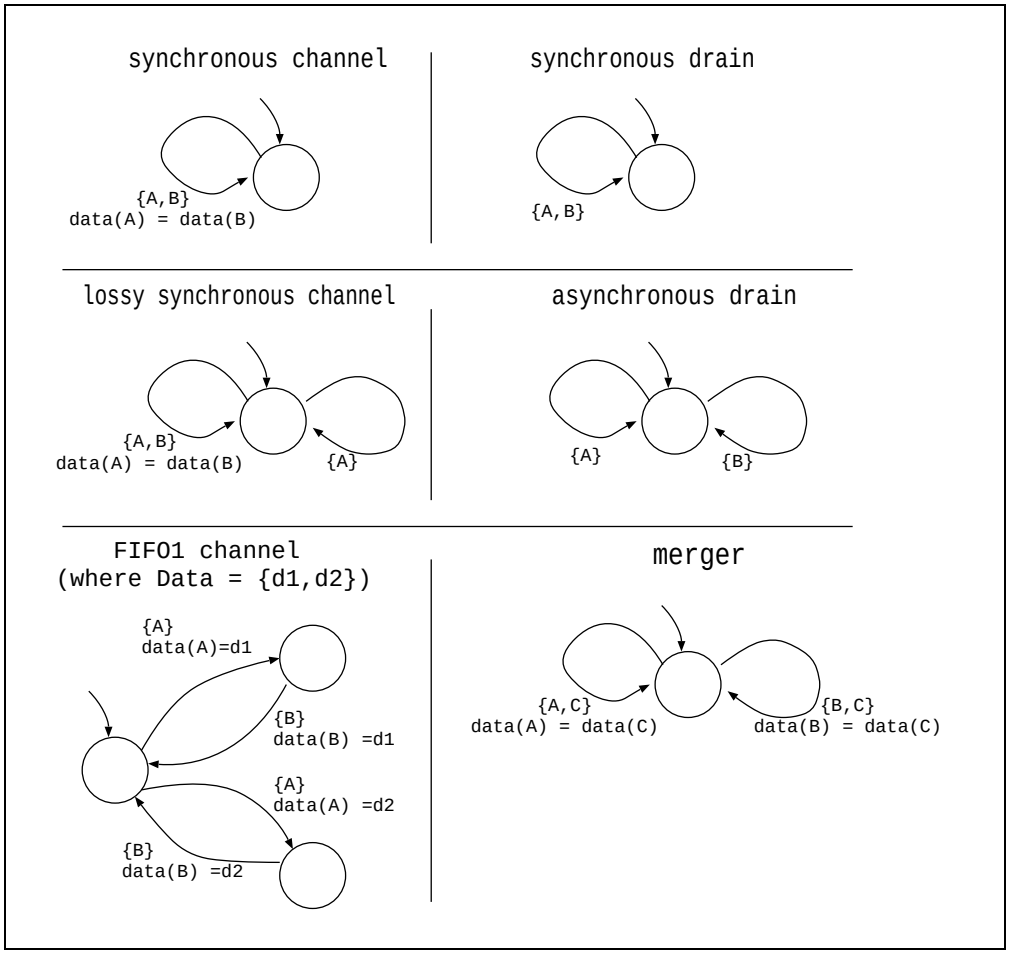

Figure 3: Deterministic constraint automata for some basic connectors

3.2 From automata to streams

In this section we show how to define a language of the so-called *timed runs* for a constraint automaton. Rather than being fully general (which is not more difficult but would require a bit more text), we look at a simple yet representative example. We consider a constraint automaton $\mathcal{A} = (Q, \mathcal{N}, \rightarrow, Q_0)$ that models the behavior of a Reo channel through which data elements flow from input port A to output port B. Thus, we set $\mathcal{N} = \{A, B\}$ and we associate with A and B timed data streams $\langle \alpha, a \rangle$ and $\langle \beta, b \rangle$ in *TDS*. We define the *language accepted by* $\mathcal{A}$ as follows:

$$\mathcal{L}_{TDS}(\mathcal{A}) \;=\; \bigcup_{q_0 \in Q_0} \mathcal{L}_{TDS}(\mathcal{A}, q_0)$$

where $\mathcal{L}_{TDS}(\mathcal{A}, q)$ denotes the language accepted by the state q of automaton $\mathcal{A}$:

$$\mathcal{L}_{TDS}(\mathcal{A}, q) = \big\{ (\langle \alpha, a \rangle, \langle \beta, b \rangle) \in TDS \times TDS \mid (\langle \alpha, a \rangle, \langle \beta, b \rangle) \text{ is a timed run for } (\mathcal{A}, q) \big\}$$

and $(\langle \alpha, a \rangle, \langle \beta, b \rangle)$ is a timed run for $(\mathcal{A}, q)$ iff there exists a transition $q \xrightarrow{N,g} \bar{q}$ such that

$$\begin{array}{lllll} & a_0 < b_0 & \wedge & N = \{A\} \wedge [data(A) = \alpha_0] \models g \wedge (\langle \alpha', a' \rangle, \langle \beta, b \rangle) \in \mathcal{L}_{TDS}(\mathcal{A}, \bar{q}), \\ \text{or} & b_0 < a_0 & \wedge & N = \{B\} \wedge [data(B) = \beta_0] \models g \wedge (\langle \alpha, a \rangle, \langle \beta', b' \rangle) \in \mathcal{L}_{TDS}(\mathcal{A}, \bar{q}), \\ \text{or} & a_0 = b_0 & \wedge & N = \{A, B\} \wedge [data(A) = \alpha_0, data(B) = \beta_0] \models g \wedge \\ & & & \qquad (\langle \alpha', a' \rangle, \langle \beta', b' \rangle) \in \mathcal{L}_{TDS}(\mathcal{A}, \bar{q}) \end{array}$$

Note that although this definition of $\mathcal{L}_{TDS}(\mathcal{A}, q)$ is circular (i.e., $\bar{q}$ may be equal to q) it can be formally defined by means of the *greatest fixed point* of a suitably chosen monotone operator. The data streams α

and β in a timed run $(\langle \alpha, a\rangle, \langle \beta, b\rangle)$ contain the data elements that are being input and output by the channel ends A and B. The time streams a and b contain for each of them the times at which these input and output actions take place. The relevance of this timing information is restricted to the particular connector, in this case the channel, at hand: what matters is only the relative order of the initial values a_0 and b_0, which determines which channel ends will be active next. A pair of timed data streams is a timed run for a state $q \in Q$ of the automaton $\mathcal{A}$ if at any moment both the set of names of active channel ends and the values of the incoming and outgoing data items 'match' the name sets and constraints of the subsequent transitions of q.

There is more to be said about the relation between the automata model of Reo and the model based on timed data streams than we have space for here. For instance, one can prove that the language accepted by the constraint automaton for a 1-bounded FIFO channel equals the set

$$\big\{(\langle \alpha, a\rangle, \langle \beta, b\rangle) \in TDS \times TDS \mid \alpha = \beta \ \wedge \ a < b < a'\big\}$$

In the present paper, we concentrate on the automata model only, and defer such observations to another occasion.

For the operators on constraint automata, the names used in the automata play an important role. For this reason, we consider $\mathcal{L}_{TDS}(\mathcal{A})$ as a set of functions $\mathcal{N} \to TDS$ (or as a subset of $TDS^{\mathcal{N}}$) rather than just an n-ary relation of TDS.

Notation 3.2.1 For a constraint automaton $\mathcal{A}$ as before, q a state in $\mathcal{A}$, $N \subseteq \mathcal{N}$ and $P \subseteq Q$, we define

$$dc_{\mathcal{A}}(q, N, P) \ = \ \bigvee \ \{ \ g : \ q \xrightarrow{N,g} \ p \text{ for some } p \in P \ \}.$$

If $\mathcal{A}$ is understood from the context, we simply write $dc(q,N,P)$. Intuitively, $dc(q,N,P)$ is the weakest DC that ensures there is an N-transition from state q to P. Note that $dc(q,N,P) = \mathsf{false}$ if there is no N-transition from q to a P-state. We use $dc(q,N)$ as an abbreviation for $dc(q,N,Q)$. □

Remark 3.2.2 [*Deriving deterministic constraint automata*] As for standard finite automata, deterministic constraint automata are as powerful as their nondeterministic variants, if we are interested only in their accepted stream-languages.[4] More precisely, given a nondeterministic constraint automaton $\mathcal{A} = (Q, \mathcal{N}, \longrightarrow, Q_0)$, one can use the standard powerset construction to obtain a deterministic constraint automaton

$$\mathsf{det}(\mathcal{A}) \ = \ \big(2^Q \setminus \{\emptyset\}, \mathcal{N}, \longrightarrow_{\mathsf{det}}, Q_0\big)$$

where the transition relation $\longrightarrow_{\mathsf{det}}$ is defined as follows.[5] For $P, P' \subseteq 2^Q$ with $P \neq \emptyset$ and $P' \neq \emptyset$ and $N \subseteq \mathcal{N}$:

$$P \xrightarrow{N,g}_{\mathsf{det}} P' \quad \text{iff} \quad g \ = \ \bigvee_{p \in P} dc(p, N, P')$$

It can be shown that $\mathcal{L}_{TDS}(\mathcal{A}) = \mathcal{L}_{TDS}(\mathsf{det}(\mathcal{A}))$. □

4 Product and hiding

The composition of TDS relations is defined to be similar to the join operator in relational data bases. For instance, given two binary TDS relations $R_1(A,B)$ and $R_2(B,C)$[6] the binary relation $(R_1 \bowtie R_2)(A,B,C)$ is given by

$$R_1 \bowtie R_2 \ = \ \big\{(\langle \alpha, a\rangle, \langle \beta, b\rangle, \langle \gamma, c\rangle) : (\langle \alpha, a\rangle, \langle \beta, b\rangle) \in R_1 \text{ and } (\langle \beta, b\rangle, \langle \gamma, c\rangle) \in R_2 \ \big\}.$$

[4] Nevertheless, as for ordinary finite automata, using nondeterministic automata has the advantage that they may be exponentially smaller than their deterministic equivalents.

[5] Of course, we can use the same ideas as for standard finite automata and apply an on-the-fly construction of the reachable part of $\mathsf{det}(\mathcal{A})$. This may lead to a smaller state space, but cannot avoid the exponential blowup in the worst-case.

[6] We assume that an n-ary TDS relation is a function $\{A_1, \ldots, A_n\} \to TDS$ rather than just a subset of TDS^n. The notation $R_1(A,B)$ suggests that R_1 is a binary relation that uses name A for its first argument and B for its second.

Definition 4.0.3 [Product-automaton (join)] The product-automaton of the two constraint automata $\mathcal{A}_1 = (Q_1, \mathcal{N}_1, \longrightarrow_1, Q_{0,1})$ and $\mathcal{A}_2 = (Q_2, \mathcal{N}_2, \longrightarrow_2, Q_{0,2})$, is:

$$\mathcal{A}_1 \bowtie \mathcal{A}_2 \;=\; (Q_1 \times Q_2, \mathcal{N}_1 \cup \mathcal{N}_2, \longrightarrow, Q_{0,1} \times Q_{0,2})$$

where $\longrightarrow$ is defined by the following rules:

$$\frac{q_1 \xrightarrow{N_1, g_1}_1 p_1,\;\; q_2 \xrightarrow{N_2, g_2}_2 p_2,\;\; N_1 \cap \mathcal{N}_2 = N_2 \cap \mathcal{N}_1}{\langle q_1, q_2 \rangle \xrightarrow{N_1 \cup N_2,\, g_1 \wedge g_2} \langle p_1, p_2 \rangle}$$

and

$$\frac{q_1 \xrightarrow{N, g}_1 p_1,\;\; N \cap \mathcal{N}_2 = \emptyset}{\langle q_1, q_2 \rangle \xrightarrow{N, g} \langle p_1, q_2 \rangle}$$

and latter's symmetric rule. □

Hiding of a name C in a TDS-relation $R(A_1, \ldots, C, \ldots, A_n)$ means existential quantification over the C-component. Formally, e.g. for a ternary relation $R = R(A, B, C)$:

$$\exists C[R(A,B,C)] \;=\; \big\{ (\alpha, a\rangle, \langle \beta, b\rangle) : \exists \text{ TDS } \langle \gamma, c\rangle \text{ with } (\alpha, a\rangle, \langle \beta, b\rangle, \langle \gamma, c\rangle) \in R \big\}$$

In constraint automata, the hiding operator removes all information about C.

Definition 4.0.4 [Hiding on constraint automata] Let $\mathcal{A} = (Q, \mathcal{N}, \longrightarrow, Q'_0)$ be a constraint automaton and $C \in \mathcal{N}$. The constraint automaton

$$\exists C[\mathcal{A}] \;=\; \big(Q, \mathcal{N} \setminus \{C\}, \longrightarrow_C, Q_{0,C}\big)$$

is defined as follows. Let $\rightsquigarrow^*$ be the (transition) relation such that $q \rightsquigarrow^* p$ iff there exists a finite path

$$q \xrightarrow{\{C\}, g_1} q_1 \xrightarrow{\{C\}, g_2} q_2 \xrightarrow{\{C\}, g_3} \cdots \xrightarrow{\{C\}, g_n} q_n$$

where $q_n = p$ and $g_1, \ldots, g_n$ are satisfiable (i.e., $g_i \not\equiv \mathsf{false}$). (Note that the g_i's depend only on C.) The set Q'_0 of initial states is $Q_0 \cup \big\{ p \in Q : q_0 \rightsquigarrow^* p \text{ for some } q_0 \in Q_0 \big\}$. The transition relation $\longrightarrow_C$ is given by:

$$\frac{q \rightsquigarrow^* p, \quad p \xrightarrow{N, g} r,\;\; N' = N \setminus \{C\} \neq \emptyset,\;\; g' = \exists C[g]}{q \xrightarrow{N', g'}_C r}$$

where $\exists C[g] \;=\; \bigvee_{d \in Data} g[data(C)/d]$.[7] □

For instance, if $\mathcal{A}_{merger}$ denotes the merger automaton in Figure 3, then $\exists C\big[\mathcal{A}_{merger}\big]$ is the same as the automaton for the asynchronous drain.
Note that the product of two deterministic constraint automata is always a deterministic automaton, while hiding can turn a deterministic constraint automaton into a nondeterministic one. However, when modeling Reo networks with constraint automata, one can derive from $\exists C[\mathcal{A}]$ a $\mathsf{det}(\exists C[\mathcal{A}])$; see Remark 3.2.2.

Example 4.0.5 [*Composition of two FIFO1 channels*] *Figure 4 shows how a FIFO2 channel can be obtained from two FIFO1 channels $\mathcal{A}_{FIFO1}(A,C)$ and $\mathcal{A}_{FIFO1}(C,B)$ via product and hiding:*

$$\mathcal{A}_{FIFO2}(A,B) \;=\; \exists C\big[\mathcal{A}_{FIFO1}(A,C) \bowtie \mathcal{A}_{FIFO1}(C,B)\big]$$

For simplicity, we deal with a singleton data domain $Data = \{d\}$ which allows us to skip the DCs of the transitions. Note that the state $\langle q_1, p_2\rangle$ is not reachable in $\mathcal{A}_{FIFO2}(A,B)$. The reason is that $\langle q_1, p_2\rangle$ is entered through C when the data element moves from the buffer of the first channel to that of the second. As we abstract away from the activities of C, state $\langle q_1, p_2\rangle$ can be skipped in $\mathcal{A}_{FIFO2}(A,B)$ (or alternatively, it can be identified with the state $\langle p_1, q_2\rangle$). □

[7] $g[data(C)/d]$ denotes the DC obtained by syntactically replacing all occurrences of $data(C)$ in g with d. More precisely, we replace the atoms $data(C) = d'$ with true if $d = d'$ and with false if $d \neq d'$.

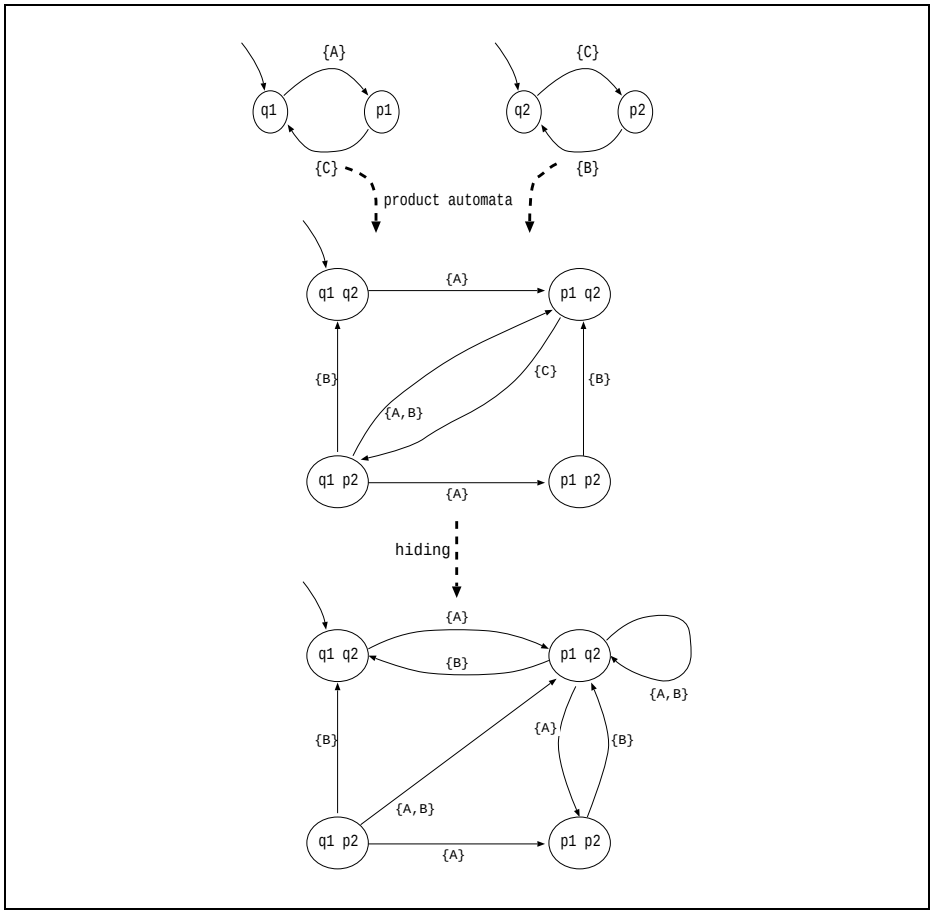

Figure 4: Composition of two FIFO1 channels

Lemma 4.0.6 [Correctness of join] *Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be two constraint automata as above. Then:*

(a) $\mathcal{L}_{TDS}(\mathcal{A}_1 \bowtie \mathcal{A}_2) = \mathcal{L}_{TDS}(\mathcal{A}_1) \bowtie \mathcal{L}_{TDS}(\mathcal{A}_2)$.

(b) If $\mathcal{N}_1 = \mathcal{N}_2$ then $\mathcal{L}_{TDS}(\mathcal{A}_1 \times \mathcal{A}_2) = \mathcal{L}_{TDS}(\mathcal{A}_1) \cap \mathcal{L}_{TDS}(\mathcal{A}_2)$. □

The equality $\mathcal{L}_{TDS}(\exists C[\mathcal{A}]) = \exists C[\mathcal{L}_{TDS}(\mathcal{A})]$ does not hold in general. For instance, hiding B in the merger automaton in Figure 3 yields a constraint automaton with a single state q, one $\{A,C\}$-transition, and one $\{C\}$-transition. Hence, any TDS-pair $(\langle \alpha, a\rangle, \langle \gamma, c\rangle)$ with $\alpha = \gamma$ and $a = c$ belongs to the accepted language of $\exists B[\mathcal{A}_{merger}]$. On the other hand, none of the pairs $(\langle \alpha, a\rangle, \langle \gamma, c\rangle)$ with $a = c$ is contained in the language $\exists B[\mathcal{L}_{TDS}(\mathcal{A}_{merger})]$ because in every timed run of $\mathcal{A}_{merger}$ data occur infinitely often on B and C but not on A. To remedy the situation we need to add *fairness* conditions that declare which automata-transitions must be taken infinitely often (similar to Büchi automata, see e.g. [Tho90]). To keep this extended abstract short, we skip this detail in the sequel.

Example 4.0.7 [*Exclusive router*] *Figure 2.a shows the Reo network for an* exclusive router *connector. A data item arriving at the input port F flows through to only one of the output ports B or E, depending on which one is ready to consume it. If both output ports are prepared to consume a data item, then one is selected nondeterministically. The input data is never replicated to more than one of the output ports.*[8] *Figure 2.a shows that the exclusive router is obtained by composing two LossySync channels, a SyncDrain (YZ) channel, a merger (inherent in the mixed node of Z), and six Sync channels:*

[8] The behavior of this connector is the counterpart of the primitive nondeterministic selection inherent in the merge that a Reo (sink or mixed) node performs on its multiple input, modeled by the merger in Figure 3.

$$\begin{aligned}\mathcal{A}_{XRouter}(F,E,D) \quad = \quad & \exists M,N,U,W,X,Y,Z\big[\mathcal{A}_{LossySync}(X,M)\times\mathcal{A}_{LossySync}(X,N)\times\\ & \mathcal{A}_{SyncDrain}(Y,Z)\times\mathcal{A}_{merger}(U,W,Z)\times\\ & \mathcal{A}_{Sync}(F,X)\times\mathcal{A}_{Sync}(X,Y)\times\mathcal{A}_{Sync}(N,U)\times\\ & \mathcal{A}_{Sync}(M,W)\times\mathcal{A}_{Sync}(M,E)\times\mathcal{A}_{Sync}(N,B)\big]\end{aligned}$$

Figure 5 shows how the constraint automaton for our exclusive router is obtained as the product of the constraint automata of its constituent channels followed by hiding of its internal transitions. □

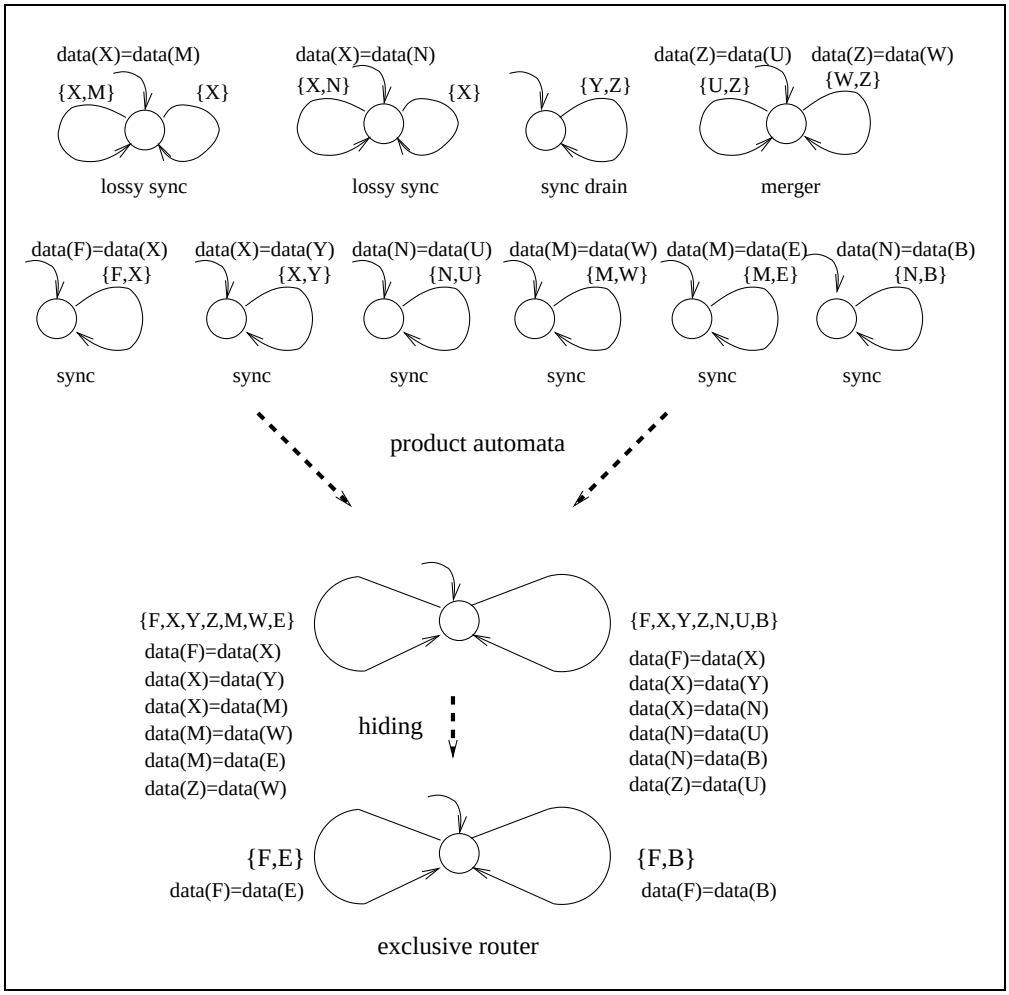

Figure 5: Exclusive router obtained through composition of other Reo channels

Example 4.0.8 [*Shift-lossy FIFO1 channel*] *Figure 2.b shows a Reo network for a connector that behaves as a lossy FIFO1 channel with a shift loss-policy. This a channel is called shift-lossy FIFO1 (*ShiftFIFO1*). It behaves as a normal FIFO1 channel, except that if its buffer is full then the arrival of a new data item deletes the existing data item in its buffer, making room for the new arrival. As such, this channel implements a "shift loss-policy" losing the oldest contents in its buffer in favor of the latest arrivals. This is in contrast to the behavior of an* overflow-lossy FIFO1 *channel, whose "overflow loss-policy" loses the new arrivals when its buffer is full.*

The connector in Figure 2.b is composed of an exclusive router (shown in Figure 2.a and explained in Example 4.0.7), a merger (inherent in the mixed node of C), a SyncDrain (AC), an initially full FIFO1 channel (BD), an initially empty FIFO2 channel (AF), and four Sync channels. Figure 6 shows how the constraint automaton for our ShiftFIFO1 channel is obtained from the constraint automata of its constituents through product and hiding:

$$\mathcal{A}_{ShiftFIFO1}(A,B) \quad = \quad \exists C,D,E,F\,\big[\mathcal{A}_{XRouter}(F,E,B) \times \mathcal{A}_{merger}(E,D,C) \times \mathcal{A}_{SyncDrain}(A,C) \\ \times \mathcal{A}_{FIFO1}(B,D) \times \mathcal{A}_{FIFO2}(A,F)\big]$$

As in our other examples, for simplicity we assume a singleton data domain Data $= \{d\}$ *which allows us to skip the DCs of the transitions.* □

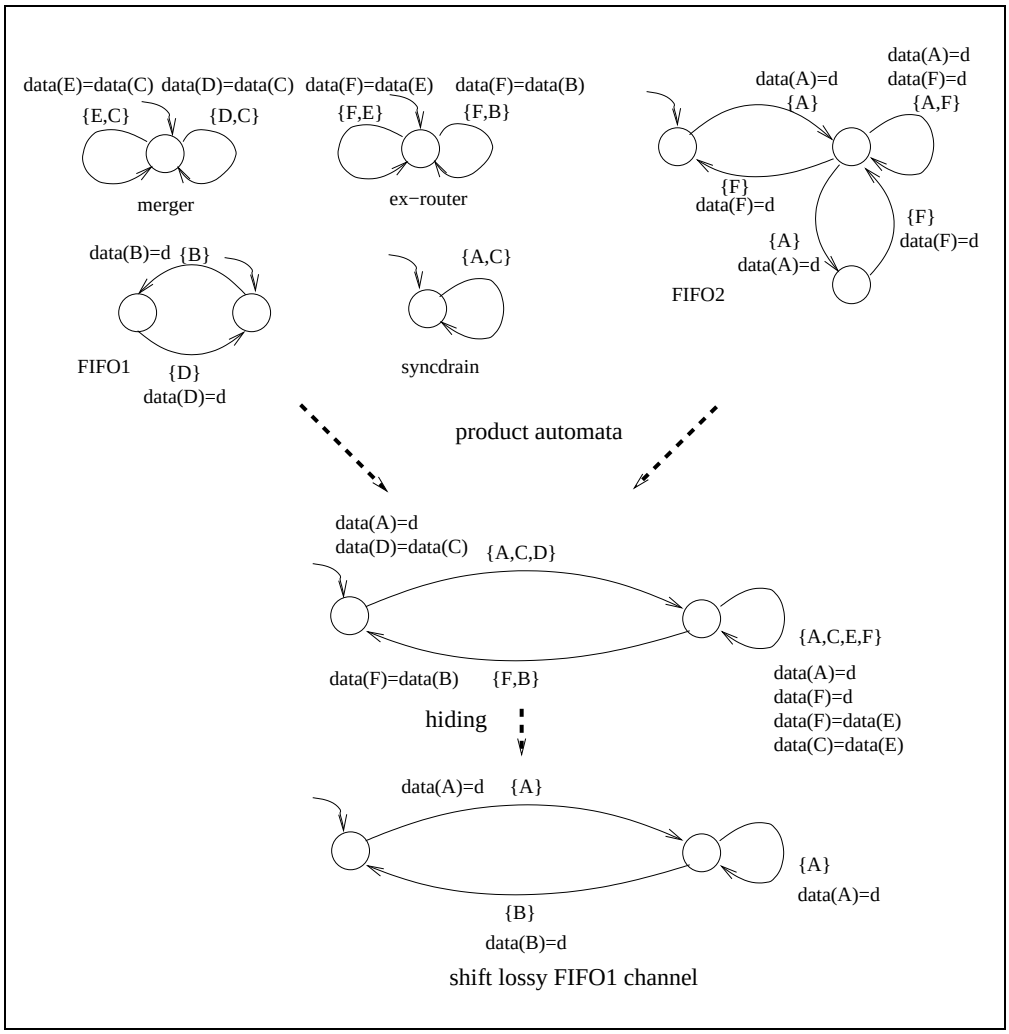

Figure 6: Shift-lossy FIFO1 channel obtained through composition of other Reo channels

Our product operator relies on the standard construction for building finite automata for intersection and has similarities with composition operators for similar models, e.g., the one-to-many composition of port automata [GR95]. On the other hand, the hiding operator for timed port automata is totally different from our construction. The former does not change the structure of the automata but makes certain output ports invisible. In contrast, our construction removes all information about the hidden names (similar to the deletion of ε-transitions in ordinary nondeterministic finite automata).

5 Bisimulation and simulation

As for standard labeled transition systems, branching time relations like bisimulation and simulation can be defined for constraint automata. In the context of Reo, we are interested only in the languages induced by Reo networks (or constraint automata) rather than their branching behavior. Nevertheless, branching time relations are important because they yield an alternative characterization of language equivalence/inclusion,

and a simple way to verify if two automata are language equivalent, or if the language of one is contained in the language of the other.

Definition 5.0.9 [Bisimulation] Let $\mathcal{A} = (Q, \mathcal{N}, \longrightarrow, Q_0)$ be a constraint automaton and $\mathcal{R}$ an equivalence relation on Q. $\mathcal{R}$ is called a bisimulation for $\mathcal{A}$ if for all pairs $(q_1, q_2) \in \mathcal{R}$, all $\mathcal{R}$-equivalence classes $C \in Q/\mathcal{R}$, and every $N \subseteq \mathcal{N}$:

$$dc(q_1, N, C) \equiv dc(q_2, N, C).$$

States q_1 and q_2 are called bisimulation equivalent (denoted $q_1 \sim q_2$) iff there exists a bisimulation $\mathcal{R}$ with $(q_1, q_2) \in \mathcal{R}$. □

As usual, two constraint automata $\mathcal{A}_1$ and $\mathcal{A}_2$ with the same set of names are called bisimulation equivalent (denoted $\mathcal{A}_1 \sim \mathcal{A}_2$) iff for every initial state $q_{0,1}$ of $\mathcal{A}_1$ there is an initial state $q_{0,2}$ of $\mathcal{A}_2$ such that $q_{0,1}$ and $q_{0,2}$ are bisimulation equivalent, and vice versa. Here, $\mathcal{A}_1$ and $\mathcal{A}_2$ must be combined into a "large" automaton obtained through the disjoint union of (the state spaces of) $\mathcal{A}_1$ and $\mathcal{A}_2$.

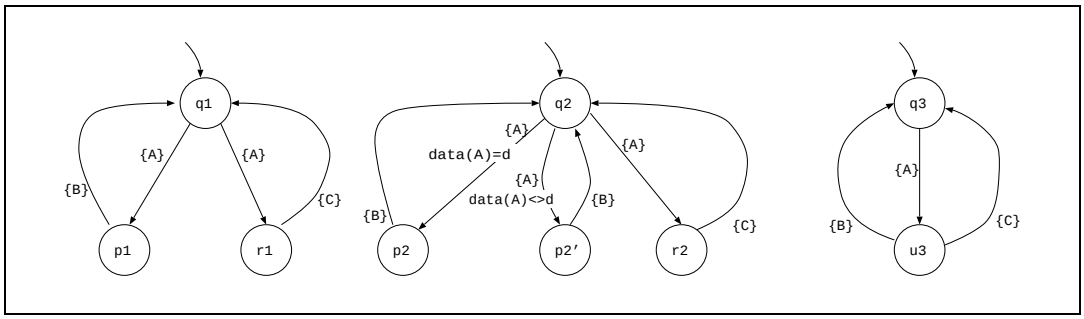

Figure 7: Similarity and bisimilarity

Example 5.0.10 *In the constraint automata of Figure 7, states q_1 and q_2 are bisimilar while $q_1, q_2 \not\sim q_3$. To see why q_1 and q_2 are bisimilar it suffices to establish a bisimulation which contains (q_1, q_2). In fact, the equivalence $\mathcal{R}$ induced by the partition*

$$Q/\mathcal{R} = \{\{q_1, q_2\}, \{q_3\}, \{p_1, p_2, p_2'\}, \{r_1, r_2\}, \{u_3\}\}$$

can be shown to be a bisimulation. Note that, for instance,

$$dc(q_1, \{A\}, \{p_1, p_2, p_2'\}) = \mathsf{true} \equiv dc(q_2, \{A\}, \{p_1, p_2, p_2'\}).$$

On the other hand, q_1 and q_2 are not bisimilar to q_3. The reason is that there is no state reachable from q_1 or q_2 which is bisimilar to u_3 because $dc(u_3, \{B\}) = dc(u_3, \{C\}) = \mathsf{true}$, while $dc(r_1, \{B\}) = dc(r_2, \{B\}) = \mathsf{false}$ and $dc(p_1, \{C\}) = dc(p_2, \{C\}) = \mathsf{false}$. □

In the example of Figure 7, states q_1, q_2, and q_3 are language equivalent (i.e., we have $\mathcal{L}_{TDS}(\mathcal{A}, q_1) = \mathcal{L}_{TDS}(\mathcal{A}, q_2) = \mathcal{L}_{TDS}(\mathcal{A}, q_3)$) but bisimulation distinguishes them as non-equivalent. For nondeterministic constraint automata bisimulation is strictly finer than language equivalence. However, for deterministic constraint automata, bisimulation and language equivalence coincide.

Theorem 5.0.11 [Bisimulation versus language equivalence] *Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be two constraint automata with the same name set $\mathcal{N}$.*

(a) If $\mathcal{A}_1 \sim \mathcal{A}_2$ then $\mathcal{L}_{TDS}(\mathcal{A}_1) = \mathcal{L}_{TDS}(\mathcal{A}_2)$.

(b) If $\mathcal{A}_1$ and $\mathcal{A}_2$ are deterministic then $\mathcal{A}_1 \sim \mathcal{A}_2$ iff $\mathcal{L}_{TDS}(\mathcal{A}_1) = \mathcal{L}_{TDS}(\mathcal{A}_2)$.

Proof 1 *(a) is an easy verification. The proof for (b) can be established by showing that given a deterministic constraint automaton $\mathcal{A} = (Q, \mathcal{N}, \longrightarrow, Q_0)$, the relation*

$$\mathcal{R} = \{(q_1, q_2) \in Q \times Q : \mathcal{L}_{TDS}(\mathcal{A}, q_1) = \mathcal{L}_{TDS}(\mathcal{A}, q_2])\}$$

is a bisimulation. □

We now provide an alternative characterization of language inclusion by means of the simulation preorder which can be viewed as a uni-directional version of bisimulation:

Definition 5.0.12 [Simulation] Let $\mathcal{A} = (Q, \mathcal{N}, \longrightarrow, Q_0)$ be a constraint automaton and $\mathcal{R}$ a binary relation on Q. $\mathcal{R}$ is called a simulation for $\mathcal{A}$ if for all pairs $(q_1, q_2) \in \mathcal{R}$, all $\mathcal{R}$-upward closed sets $P \subseteq Q$, and every $N \subseteq \mathcal{N}$:

$$dc(q_1, N, P) \;\leq\; dc(q_2, N, P).$$

P is called $\mathcal{R}$-upward closed iff for all states $p \in P$ and $(p, p') \in \mathcal{R}$ we have $p' \in P$. A state q_1 is simulated by another state q_2 (and q_2 simulates q_1), denoted as $q_1 \preceq q_2$, iff there exists a simulation $\mathcal{R}$ with $(q_1, q_2) \in \mathcal{R}$. A constraint automaton $\mathcal{A}_2$ simulates another constraint automaton $\mathcal{A}_1$ (denoted as $\mathcal{A}_1 \preceq \mathcal{A}_2$) iff every initial state of $\mathcal{A}_1$ is simulated by an initial state of $\mathcal{A}_2$.[9] □

As the logical or ($\vee$) is idempotent, we have that $\mathcal{R}$ is a simulation iff $dc(q_1, N, p) \;\leq\; dc(q_2, N, p \uparrow_{\mathcal{R}})$ for all pairs $(q_1, q_2) \in \mathcal{R}$, states $p \in Q$, and $N \subseteq \mathcal{N}$. Here, $p \uparrow_{\mathcal{R}}$ denotes the $\mathcal{R}$-upward closure of $\{p\}$, i.e., the set $\{p' \in Q : (p, p') \in \mathcal{R}\}$.

Example 5.0.13 State q_3 in Figure 7 simulates states q_1 and q_2 in the same figure. Other examples include, in Figure 3:

- the automaton for the synchronous drain which simulates the automaton for the synchronous channel,
- the automaton for the asynchronous drain which simulates the automaton for the FIFO1 channel, and
- the automaton for the synchronous channel which is simulated by the automaton for the lossy synchronous channel. □

Analogous to Theorem 5.0.11, we obtain:

Theorem 5.0.14 [Simulation versus language inclusion] *Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be two constraint automata with the same name set $\mathcal{N}$.*

(a) If $\mathcal{A}_1 \preceq \mathcal{A}_2$ then $\mathcal{L}_{TDS}(\mathcal{A}_1) \subseteq \mathcal{L}_{TDS}(\mathcal{A}_2)$.

(b) If $\mathcal{A}_1$ and $\mathcal{A}_2$ are deterministic then $\mathcal{A}_1 \preceq \mathcal{A}_2$ iff $\mathcal{L}_{TDS}(\mathcal{A}_1) \subseteq \mathcal{L}_{TDS}(\mathcal{A}_2)$. □

As for ordinary labeled transition systems, bisimulation equivalence is finer than simulation equivalence $\preceq \cap \preceq^{-1}$.

Lemma 5.0.15 [Bisimulation versus simulation] *For all constraint automata $\mathcal{A}_1$ and $\mathcal{A}_2$:*

(a) If $\mathcal{A}_1 \sim \mathcal{A}_2$ then $\mathcal{A}_1 \preceq \mathcal{A}_2$ (and $\mathcal{A}_2 \preceq \mathcal{A}_1$).

(b) If $\mathcal{A}_1$ and $\mathcal{A}_2$ are deterministic and $\mathcal{A}_1 \preceq \mathcal{A}_2$, $\mathcal{A}_2 \preceq \mathcal{A}_1$ then $\mathcal{A}_1 \sim \mathcal{A}_2$.

Proof 2 *(a) follows from the fact that any bisimulation is a simulation. (b) follows by observing that for deterministic automata, simulation equivalence is a bisimulation.* □

Lemma 5.0.16 [Compositionality of join and hiding]

(a) If $\mathcal{A}_1 \preceq \mathcal{A}_1'$ and $\mathcal{A}_2 \preceq \mathcal{A}_2'$ then $\mathcal{A}_1 \bowtie \mathcal{A}_2 \preceq \mathcal{A}_1' \bowtie \mathcal{A}_2'$.

(b) If $\mathcal{A}_1 \sim \mathcal{A}_1'$ and $\mathcal{A}_2 \sim \mathcal{A}_2'$ then $\mathcal{A}_1 \bowtie \mathcal{A}_2 \sim \mathcal{A}_1' \bowtie \mathcal{A}_2'$.

(c) If $\mathcal{A}_1 \preceq \mathcal{A}_2$ then $\exists C[\mathcal{A}_1] \preceq \exists C[\mathcal{A}_2]$.

(d) If $\mathcal{A}_1 \sim \mathcal{A}_2$ then $\exists C[\mathcal{A}_1] \sim \exists C[\mathcal{A}_2]$.

[9] Here, we assume that $\mathcal{A}_1$ and $\mathcal{A}_2$ rely on the same set of names.

Proof 3 *To prove (a) and (b), consider the relations*

$$\begin{aligned} \mathcal{R}_{sim} &= \big\{(\langle q_1,q_2\rangle,\langle q_1',q_2'\rangle) : q_1 \preceq q_1', q_2 \preceq q_2'\big\}, \\ \mathcal{R}_{bis} &= \big\{(\langle q_1,q_2\rangle,\langle q_1',q_2'\rangle) : q_1 \sim q_1', q_2 \sim q_2'\big\}. \end{aligned}$$

Then, $\mathcal{R}_{sim}$ is a simulation and $\mathcal{R}_{bis}$ a bisimulation on the product-automata.
We provide the proof for (c) and observe that the proof for (d) is similar. To prove (c) it suffices to show that given a constraint automaton $\mathcal{A} = (Q,\mathcal{N},\longrightarrow,q_0)$, any simulation $\mathcal{R}$ for $\mathcal{A}$ is a simulation for $\exists C[\mathcal{A}]$. By considering the $\{C\}$-transitions in $\mathcal{A}$, we obtain:

$$(q_1,q_2) \in \mathcal{R} \wedge q_1 \rightsquigarrow^* q_1' \implies q_2 \rightsquigarrow^* q_2' \text{ for some state } q_2' \text{ with } (q_1',q_2') \in \mathcal{R}. \qquad (*)$$

Let $(q_1,q_2) \in \mathcal{R}$, N a nonempty subset of $\mathcal{N} \setminus \{C\}$, and P an $\mathcal{R}$-upward closed subset of Q. Then, for all states $q \in Q$:

$$dc_{\exists C[\mathcal{A}]}(q,N,P) = \bigvee_{q' \in q^*} \big(dc_{\mathcal{A}}(q',N,P) \vee dc_{\mathcal{A}}(q',N \cup \{C\},P)\big)$$

where $q^* = \{q' \in Q : q \rightsquigarrow^* q'\}$. From (*), we obtain that for every state $q_1' \in q_1^*$ there exists a state $q_2' \in q_2^*$ with $(q_1',q_2') \in \mathcal{R}$. Because

$$\begin{aligned} dc_{\mathcal{A}}(q_1',N,P) &\leq dc_{\mathcal{A}}(q_2',N,P), \\ dc_{\mathcal{A}}(q_1',N \cup \{C\},P) &\leq dc_{\mathcal{A}}(q_2',N \cup \{C\},P), \end{aligned}$$

we get $dc_{\exists C[\mathcal{A}]}(q_1,N,P) \leq dc_{\exists C[\mathcal{A}]}(q_2,N,P)$. □

6 Concluding remarks

Connector construction in Reo is conceptually analogous to the design of asynchronous electronic circuits. Among other things, this analogy emphasizes the importance of visual environments for design, analysis, verification, and optimization of Reo connectors, as counterparts of tools and facilities available in modern electronic CAD systems. In this context, issues such as whether two Reo connectors have the same observable behavior, or whether one's behavior is only a refinement of that of the other arise naturally and frequently. Constraint automata enable us to formally phrase such issues in terms of language equivalence or language containment and check, e.g., whether $\mathcal{L}_{TDS}(\mathcal{A}_1) = \mathcal{L}_{TDS}(\mathcal{A}_2)$ or $\mathcal{L}_{TDS}(\mathcal{A}_1) \subseteq \mathcal{L}_{TDS}(\mathcal{A}_2)$. Known methods for analysis and model checking with ordinary finite state automata and labeled transition systems can be adapted to work with our constraint automata.
Given deterministic constraint automata $\mathcal{A}_1$ and $\mathcal{A}_2$, the simplest way to check language equivalence is to build the bisimulation quotient of the constraint automaton $\mathcal{A} = \mathcal{A}_1 \uplus \mathcal{A}_2$, which we obtain by taking the disjoint union of the state spaces of $\mathcal{A}_1$ and $\mathcal{A}_2$, and check whether the initial states of $\mathcal{A}_1$ and $\mathcal{A}_2$ belong to the same equivalence class. To compute the bisimulation equivalence classes of $\mathcal{A}$, we may apply the prominent partitioning-splitter technique [KS83, PT87]. Similarly, to check language inclusion for two deterministic constraint automata $\mathcal{A}_1$ and $\mathcal{A}_2$, we may check whether $\mathcal{A}_2$ simulates $\mathcal{A}_1$ by a technique based on the same idea as for labeled transition systems (e.g. [HHK95]).
Nondeterministic constraint automata offer a useful semantic model for Reo connector networks which, e.g., avoids the exponential blowup that may result from applying the powerset construction to an automaton $\exists C[\mathcal{A}]$. The algorithms to compute the bisimulation quotient or simulation preorder can be applied here as a sound (but incomplete) verification method to show language equivalence or inclusion.
Our future work includes the development of temporal logics and model checking algorithms based on constraint automata.

References

[AR02] F. Arbab, J.J.M.M. Rutten. A Coinductive Calculus of Component Connectors. Report SEN-R0216, CWI, 2002. Available at URL: www.cwi.nl. To appear in the proceedings of WADT'02.

[Arb03] F. Arbab. Reo: A Channel-based Coordination Model for Component Composition, *Mathematical Structures in Computer Science*, Cambridge University Press, to appear in 2003.

[GR95] R. Grosu, B. Rumpe. Concurrent Timed Port Automata. Techn. Report Techn. Universität München TUM-I9533, 1995. http://www4.informatik.tu-muenchen.de/reports/TUM-I9533.html.

[HHK95] M. Henzinger, T. Henzinger, P. Kopke. Computing Simulations on Finite and Infinite Graphs, Proc. FOCS'95, pp 453-462, 1995.

[KS83] P. Kannelakis, S. Smolka. CCS Expressions, Finite State Processes and Three Problems of Equivalence, Proc. 2nd ACM Symposium on the *Principles of Distributed Computing*, pp 228-240, 1983. The full version with the same title has appeared in *Information and Computation*, Vol. 86, pp 43-68, 1990.

[LT89] N. Lynch, M.R. Tuttle. An introduction to input/output automata, CWI Quarterly 2(3), pp 219-246, 1989.

[Mil80] R. Milner. A Calculus of Communicating Systems. *Lecture Notes in Computer Science*, Vol. 92, 1980.

[PT87] R. Paige, R. Tarjan. Three Partition Refinement Algorithms. *SIAM Journal of Computing*, Vol. 16, No. 6, pp 973-989, 1987.

[Tho90] W. Thomas. Automata on Infinite Objects. In J. van Leuwen, editor, *handbook of Theoretical Computer Science*, vol. B, chapter 4, pp 135-191. Elsevier Science Publishers, Amsterdam, 1990.