Available online at www.sciencedirect.com

SCIENCE DIRECT®

Science of Computer Programming 61 (2006) 75–113

Science of Computer Programming

www.elsevier.com/locate/scico

Modeling component connectors in Reo by constraint automata

Christel Baier[a,*], Marjan Sirjani[b,c], Farhad Arbab[d,e,f], Jan Rutten[d,g]

[a] *Institut für Informatik I, University of Bonn, Römerstraße 164, D-53117 Bonn, Germany*
[b] *Department of Electrical and Computer Engineering, University of Tehran, Karegar Avenue, Pardis #2, Tehran, Iran*
[c] *School of Computer Science, IPM, Niavaran Square, Tehran, Iran*
[d] *Department of Software Engineering, Centrum voor Wiskunde en Informatica, Kruislaan 413, P.O. Box 94079, 1090 GB Amsterdam, The Netherlands*
[e] *Leiden Institute for Advanced Computer Science, Leiden University, Niels Bohrweg 1, 2333 CA Leiden, The Netherlands*
[f] *School of Computer Science, University of Waterloo, 200 University Avenue West, Waterloo, ON N2L 3G1, Canada*
[g] *Vrije Universiteit, De Boelelaan 1081, 1081 HV Amsterdam, The Netherlands*

Received 30 January 2004; received in revised form 24 May 2005; accepted 10 October 2005
Available online 30 March 2006

Abstract

In this paper we introduce *constraint automata* and propose them as an operational model for Reo, an exogenous coordination language for compositional construction of component connectors based on a calculus of channels. By providing composition operators for constraint automata and defining notions of equivalence and refinement relations for them, this paper covers the foundations for building tools to address concerns such as the automated construction of the automaton for a given component connector, equivalence checking or containment checking of the behavior of two given connectors, and verification of coordination mechanisms.

Keywords: Constraint automata; Reo; Timed data streams; Coordination; Components; Composition; Bisimulation; Simulation; Verification

1. Introduction

Coordination models and languages emerged in the 1990s as the linguistic counterpart of the so-called *middle-ware* layer of software that consisted of ad-hoc libraries of functions providing higher-level inter-process communication support in parallel and, especially, distributed applications. Coordination models and languages close the conceptual gap between the cooperation model used by the constituent parts of an application and the lower-level communication model used in its implementation. They provide a clean separation between individual software components and their interactions within their overall software organization. This separation, together with the higher-level abstractions offered by coordination models and languages, improve software productivity, enhance maintainability, advocate modularity, promote reusability, and lead to software organizations and architectures that are more tractable and more amenable to verification and global analysis.

* Corresponding author.
E-mail addresses: baier@cs.uni-bonn.de (C. Baier), msirjani@ut.ac.ir (M. Sirjani), Farhad.Arbab@cwi.nl (F. Arbab), Jan.Rutten@cwi.nl (J. Rutten).

0167-6423/$ - see front matter
doi:10.1016/j.scico.2005.10.008

The current interest in constructing applications out of independent software components necessitates paying attention to the so-called *glue-code*. The purpose of glue-code is to compose a set of components by filling the significant interface gaps that naturally arise among them, simply because they are not (supposed to be) tailor-made to work with one another. Using components thus means understanding how they interact individually with their environment, and specifying how they should engage in mutual, cooperative interactions in order for their composition to behave as a coordinated whole. Many of the core issues involved in component composition have already been identified and studied as key concerns in work on coordination. Coordination models and languages address key issues in Component Based Software Engineering such as specification, interaction, and dynamic composition of components. Specifically, *exogenous coordination models and languages*, which enable third-party entities to wield coordination control over the interaction behavior of mutually anonymous entities involved in a collaboration *from outside* of its participants, provide a very promising basis for the development of effective glue-code languages.

Constraint automata. In this paper, we introduce constraint automata as a formalism to describe the "behavior" and possible data flow in coordination models that connect anonymous components to enable their coordinated interaction. The theory of constraint automata thus yields a basis for the formal verification of coordination mechanisms (e.g., model checking against temporal-logical specifications or equivalence checking). Constraint automata can be thought of as conceptual generalizations of probabilistic automata where data constraints, instead of probabilities, influence applicable state transitions. We show that constraint automata can serve as an *operational model* for the coordination language Reo, introduced in [1]. 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. 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 input/output (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.

Using constraint automata as an operational model for Reo connectors, the automata states stand for the possible configurations (e.g., the contents of the FIFO channels of a Reo connector) while the automata transitions represent the possible data flow and its effect on these configurations. In fact, the operational semantics for Reo presented in [1] can be reformulated in terms of constraint automata. However, in this paper we follow a different approach and define the constraint automaton of a given Reo connector in a *compositional* way. For this, we introduce composition operators for constraint automata corresponding to the Reo connector primitives, and thus provide the basis for the algorithmic construction of constraint automata for Reo connectors.

The paper [2] presents a coalgebraic formal semantics for Reo connectors that assigns to any Reo connector a relation over infinite *timed data streams* (called TDS languages in this paper). In fact, many interesting properties of Reo connectors, as well as notions of equivalence or refinement for Reo connectors, can be formulated in terms of TDS languages. To reason about TDS languages, we may regard constraint automata as *acceptors* for timed data streams. The rough idea behind the use of constraint automata as language acceptors is that such an automaton observes the data occurring at certain input/output ports of components and either changes its state according to the observed data or rejects it if there is no corresponding transition in the automaton. From this point of view, constraint automata serve as a formalism for describing TDS languages, in a similar way that ordinary finite automata (or, alternatively, ω-automata) can be used as a formalism to describe languages of finite (respectively, infinite) words (see, e.g., [14, 26]). In particular, they can serve as a specification formalism for a coordination mechanism that is yet to be designed, or as interface specifications for the component instances that are (to be) glued together.

To solve typical verification problems, e.g., checking whether a given Reo connector meets its automata specification or whether two Reo connectors are language-equivalent (in the sense that they induce the same TDS language), the fact that constraint automata are close to both ordinary finite or ω-automata and labeled transition systems allows us to modify known methods for the analysis of reactive systems (modeled by labeled transition systems) or formal languages (represented by finite or ω-automata) to work with constraint automata. As checking language equivalence or language inclusion for non-deterministic automata is computationally hard, we introduce notions of bisimulation equivalence and a simulation relation for constraint automata. Being refinements of the language relations with simpler decision algorithms, these branching time relations provide sound (but incomplete) proof methods for checking language equivalence or language inclusion.

Related models. Of course, the use of automata-based models (including variants of labeled transition systems) as operational models for coordination principles is not new. Our notion of constraint automata is most in the spirit of I/O automata [17], timed port automata [11] and interface automata [8]. We briefly summarize the major differences and similarities:

- While transitions in I/O automata are labeled with action names, transitions in timed port and constraint automata are data-dependent. However, timed port automata label the transitions with specific data values, whereas we use a symbolic representation by means of data constraints (Boolean expressions for the 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. 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 suitable data on any of its corresponding ports). Such transitions do not exist in the "one-to-many composition" of timed port automata.
- As for interface automata, we do not assume input enabledness, as is the case for I/O or timed port automata. In fact, in our setting, there is no need to distinguish between input and output ports, unlike in interface automata.
- Constraint automata, like I/O-automata, are based on transition systems. Interface automata are based on game theory, and their main purpose is to allow automatic checking of compatibility between interfaces.

Used as acceptors for TDS languages (e.g., to specify the "legal" data flow of a coordination mechanism that is yet to be designed or for an interface specification of a component), constraint automata are in the spirit of ordinary finite automata and ω-automata. For the purposes of this paper, where we do not consider finite behavior — which may occur, for example, if configurations are reached where data flow at certain ports is blocked — there is no need for final states. Thus, acceptance of a timed data stream by constraint automata requires only the existence of an infinite run in the automata. However, this difference between standard automata and constraint automata cannot be understood as an advantage of the latter, as it can be explained by our decision not to consider finite behavior. To reason about finite timed data streams or assuming fairness for certain Reo-connector primitives, constraint automata would have to be extended with final states, leading to a different notion of acceptance. To keep the presentation of the basic concepts of constraint automata simple and clear and to avoid overloading with notation, we decided to restrict ourselves in this paper to infinite behavior, without fairness assumptions.

In summary, constraint automata are close to various other automata models, which yields the advantange that known validation technique can be adapted for our purposes. The characteristics of constraint automata are chosen in a way that fits best in the Reo framework where the focus is on reasoning about the observable data flow at nodes in a channel network by means of the relation of timed data streams. This is in contrast to other automata models that were designed for slightly different tasks and rely on concepts (such as action names for the activities of individual agents, input enabledness, and compatibility checking) that are not relevant for Reo component connectors.

Organization of the paper. The rest of this paper is organized as follows. In Section 2, we recall the definition of timed data streams and introduce some notation. In Section 3, we present the definition of constraint automata and their accepted TDS languages. The use of constraint automata as an operational semantics for Reo connectors is explained in Section 4. This section starts with a brief overview of Reo. We then provide the definition of composition operators (join and hiding) on constraint automata corresponding to the Reo connector primitives and demonstrate the compositional construction of constraint automata for given Reo connectors through a series of examples. In Section 5, we introduce notions of bisimulation and simulation for constraint automata, discuss their relationship to the language-based relations, and provide congruence results for the composition operators defined in Section 4. Section 6 is concerned with algorithms for checking the equivalence of two constraint automata and whether one automaton can be viewed as a refinement of another. We conclude in Section 7, hinting at our current and future work on model checking and automated tools for reasoning about constraint automata and Reo connectors.

2. Timed data streams

In this section, we recall the definition of timed data streams (TDS for short) and explain our notations.

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 behavior and infinite streams that correspond

to infinite "runs" of our automata, omitting final states including deadlocks. We denote individual streams as $\alpha = \alpha(0), \alpha(1), \alpha(2), \ldots$ (or $a = a(0), a(1), a(2), \ldots$). We call $\alpha(0)$ the *initial value* of α. The *(stream) derivative* α' of a stream α is defined as $\alpha' = \alpha(1), \alpha(2), \alpha(3), \ldots$. We write $\alpha^{(i)}$ for the i-th derivative of α, which is defined as $\alpha^{(0)} = \alpha$ and $\alpha^{(i+1)} = (\alpha^{(i)})'$. Note that $\alpha'(k) = \alpha(k+1)$ and $\alpha^{(i)}(k) = \alpha(i+k)$, for all $k, i \geq 0$.

Timed data streams. We now recall the definition of timed data streams from [2]. In the sequel, *Data* is a fixed, non-empty and finite set of data that can be sent (and received) via channels.[1] The set of all (infinite) timed data streams over *Data* is given by:

$$TDS = \big\{\langle \alpha, a\rangle \in Data^\omega \times \mathbb{R}_+^\omega : \forall k \geq 0 : a(k) < a(k+1) \text{ and } \lim_{k\to\infty} a(k) = \infty\big\}.$$

Thus, 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 that go to infinity. The time stream a indicates, for each data item $\alpha(k)$, the moment $a(k)$ at which it is being input or output.

TDS-tuples. To formalize the input/output behavior of a coordination model by means of timed data streams, we use names, say $A_1, \ldots, A_n$, for the input or output ports that connect the component instances with other component instances or the environment of the whole system. With each port A_i, we associate a timed data stream. That is, for a given name-set $\mathcal{N}ames = \{A_1, \ldots, A_n\}$, we define

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

as the set of all TDS-tuples consisting of one timed data stream for each port. When writing the elements of $TDS^{\mathcal{N}ames}$ as tuples of timed data streams, we assume a fixed enumeration of the port names in $\mathcal{N}ames$, say $A_1, \ldots, A_n$, such that the i-th timed data stream of the TDS-tuple θ stands for the timed data stream of the i-th port A_i. If no enumeration of the port names is given, then we use a family notation $\theta = (\theta|_A)_{A\in\mathcal{N}ames}$ for the elements of $TDS^{\mathcal{N}ames}$, where $\theta|_A$ stands for the timed data stream for port A.

Data assignments. By a data assignment, we mean a function $\delta : N \to Data$ where $\emptyset \neq N \subseteq \mathcal{N}ames$. We use notations like

$$\delta = \big[A \mapsto \delta_A : A \in N\big]$$

to describe the data assignment that assigns to any TDS name $A \in N$ the value $\delta_A \in Data$.

Notations for TDS-tuples. If $\theta = (\langle \alpha_1, a_1\rangle, \ldots, \langle \alpha_n, a_n\rangle) \in TDS^{\mathcal{N}ames}$, then we write $\theta.time$ to denote the time stream obtained by merging the timed data streams $a_1, \ldots, a_n$ in increasing order. That is,

$$\begin{aligned}
\theta.time(0) &= \min\big\{a_i(0) : i = 1, \ldots, n\big\},\\
\theta.time(1) &= \min\big\{a_i(k) : a_i(k) > \theta.time(0),\ i = 1, \ldots, n,\ k = 0, 1, 2, \ldots\big\},\\
\theta.time(2) &= \min\big\{a_i(k) : a_i(k) > \theta.time(1),\ i = 1, \ldots, n,\ k = 0, 1, 2, \ldots\big\},\\
&\vdots
\end{aligned}$$

Next we define $\theta.N = \theta.N(0), \theta.N(1), \theta.N(2), \ldots$ as a stream over $2^{\mathcal{N}ames}$ by

$$\theta.N(k) = \big\{A_i \in \mathcal{N}ames : a_i(\ell) = \theta.time(k) \text{ for some } \ell \in \{0, 1, 2, \ldots\}\big\}.$$

Intuitively, $\theta.N(k)$ is the name-set consisting of the ports $A \in \mathcal{N}ames$ at which a data item is observed at time point $\theta.time(k)$. Moreover, we define $\theta.\delta = \theta.\delta(0), \theta.\delta(1), \theta.\delta(2), \ldots$ as a stream over the set of data assignments where $\theta.\delta(k)$ represents the observed data flow at time point $\theta.time(k)$. Formally,

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

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

[1] The finiteness of *Data* is irrelevant in most of this paper. In a few examples, we also consider infinite data domains.

We write θ' for the TDS-tuple that is obtained by the first derivatives of the timed data streams $\theta|_A$ for $A \in \theta.N(0)$ together with the timed data streams $\theta|_A$ for $A \notin \theta.N(0)$. For instance, if $\theta.N(0) = \{A_1, A_2\}$, then

$$\theta' = (\langle \alpha_1', a_1' \rangle, \langle \alpha_2', a_2' \rangle, \langle \alpha_3, a_3 \rangle, \ldots, \langle \alpha_n, a_n \rangle).$$

The $(i+1)$-st derivative is given by $\theta^{(i+1)} = (\theta^{(i)})'$.

Remark 2.1 (*Infinite Data Flow at all Ports*). The requirement that all timed data streams $\langle \alpha_i, a_i \rangle$ in a TDS-tuple $\theta = (\langle \alpha_1, a_1 \rangle, \ldots, \langle \alpha_n, a_n \rangle)$ are *infinite* (together with the assumption on time streams a that $\lim_{k\to\infty} a(k) = \infty$) implies that, for any port $A \in \mathcal{N}ames$, there are infinitely many indices k with $A \in \theta.N(k)$. Hence, we assume that at *any* port A there is an infinite data flow. This assumption simplifies the notations but, on the other hand, lacks the possibility to describe, e.g., deadlock situations where a certain coordination mechanism blocks the data flow at port A. □

Remark 2.2 (*Distinguishing Inputs and Outputs*). Timed data streams, as defined here, do not distinguish between input and output actions; instead, they merely report the "observed" data at a port A, but not whether it is a write or read operation that occurs at A. However, we can assume a fixed classification of the ports into input or output ports and — using this classification — derive the information whether an observed data item d at port A stands for "reading d" or "writing d".[2] Alternatively, we can deal with a data domain that distinguishes between written and read values. □

A TDS language (for $\mathcal{N}ames$) denotes any subset of $TDS^{\mathcal{N}ames}$. Following the approach of [2] where a compositional semantics for Reo circuits is provided using coinductive reasoning with timed data streams, we shall use TDS languages as a formalism to describe the possible data flow of a coordination model. For instance, the language for a 1-bounded FIFO channel (viewed as a connector that sends values from input port A to output port B) equals the TDS language

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

where, for time streams a and b, the ordering $<$ is given by $a < b$ iff $a(k) < b(k)$ for all $k \geq 1$.

3. Constraint automata

Constraint automata use a finite set $\mathcal{N}$ of *names*, e.g., $\mathcal{N} = \{A_1, \ldots, A_n\}$ where A_i stands for the i-th input/output port of a connector or component. The transitions of constraint automata are labeled with pairs consisting of a non-empty subset N of $\{A_1, \ldots, A_n\}$ and a data constraint g. Data constraints can be viewed as a symbolic representation of *sets* of data assignments. Formally, *data constraints* are propositional formulae built from the atoms "$d_A = d$", where data item d is assigned to port A. Data constraints are given by the following grammar:

$$g ::= \mathsf{true} \;\big|\; d_A = d \;\big|\; g_1 \vee g_2 \;\big|\; \neg g$$

where $A \in \mathcal{N}$ is a name and $d \in Data$. In the sequel, we write $DC(N, Data)$, for a non-empty subset N of $\mathcal{N}$, to denote the set of data constraints using only atoms "$d_A = d$" for $A \in N$. We use DC as an abbreviation for $DC(\mathcal{N}, Data)$. The Boolean connectors $\wedge$ (conjunction), $\oplus$ (exclusive or), $\rightarrow$ (implication), $\leftrightarrow$ (equivalence), and so on, can be derived as usual. We often use derived data constraints such as $d_A \neq d$ or $d_A = d_B$, which stand for the data constraints $\neg(d_A = d)$ and $\bigvee_{d \in Data} \big((d_A = d) \wedge (d_B = d)\big)$, respectively.

Remark 3.1 (*Some Comments on the Data Domains*). 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.

The assumption that *Data* is finite allows us to derive data constraints as "$d_A = d_B$" or "$d_A \in D$" or "$(d_A, d_B) \in E$" for $D \subseteq Data$ and $E \subseteq Data \times Data$. However, as long as we do not speak about algorithmic aspects, we can allow for

[2] In the context of Reo, some input and output ports can also be internal to a component instance. Any data flow at such an internal port A stands for the transmission of data inside the corresponding component instance via port A. Thus, observing data item d at A has the meaning of "writing d" and "reading d".

an infinite data domain as well. In this case, to derive data constraints as above, we enrich the syntax of data constraints by infinite disjunctions/conjunctions, or simply add "$d_A = d_B$", "$d_A \in D$" etc., as atomic data constraints. □

The symbol $\models$ stands for the obvious satisfaction relation which results from interpreting data constraints over data assignments (which were introduced in Section 2). For instance,

$$\begin{aligned}\big[A \mapsto d_1, B \mapsto d_2, C \mapsto d_1\big] &\models d_A = d_C,\\ \big[A \mapsto d_1, B \mapsto d_2, C \mapsto d_1\big] &\not\models d_A = d_B\end{aligned}$$

if $d_1 \neq d_2$. With this satisfaction relation, we may identify any data constraint g with the set δ of all data assignments where $\delta \models g$ holds.

Satisfiability and validity, logical equivalence $\equiv$, and logical implication $\leq$ of data constraints are defined as usual, e.g.:

$$\begin{aligned}g_1 \equiv g_2 \quad &\text{iff} \quad \text{for all data assignments } \delta\text{: } \delta \models g_1 \Longleftrightarrow \delta \models g_2\\ g_1 \leq g_2 \quad &\text{iff} \quad \text{for all data assignments } \delta\text{: } \delta \models g_1 \Longrightarrow \delta \models g_2.\end{aligned}$$

3.1. Definition of constraint automata

We now present the definition of constraint automata which can be viewed as acceptors for TDS-tuples (see Section 3.2) and which can serve as an operational model for channel-based coordination languages (see Section 4).

Definition 3.2 (*Constraint Automata*). A constraint automaton (over the data domain *Data*) is a tuple $\mathcal{A} = (Q, \mathcal{N}ames, \longrightarrow, Q_0)$ where

- Q is a set of states,
- $\mathcal{N}ames$ is a finite set of names,
- $\longrightarrow$ is a subset of $Q \times 2^{\mathcal{N}ames} \times DC \times Q$, 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) $g \in DC(N, Data)$. $\mathcal{A}$ is called finite iff Q, $\longrightarrow$ and the underlying data domain *Data* are finite. □

We do not generally assume that $\mathcal{A}$ is finite, because modeling connectors that use channels with unbounded capacity leads to constraint automata with an infinite state-space. In fact, except for algorithmic aspects (see Section 6), assuming that $\mathcal{A}$ is finite is not important. (Even the requirement that $\mathcal{N}$ is finite can be relaxed.)

The intuitive meaning of a constraint automaton $\mathcal{A}$ as an operational model for connectors of a coordination language is similar to the interpretation of labeled transition systems as formal models for reactive systems. The states represent the configurations of the connector, the transitions, the possible one-step behavior where the meaning of

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

is that, in configuration q, the ports $A_i \in N$ have the possibility of performing I/O operations that meet the guard g and that lead from configuration q to p, while the other ports $A_j \in \mathcal{N}ames \setminus N$ do not perform any I/O operation.

Example 3.3 (*1-Bounded FIFO Channel*). Fig. 1 shows a constraint automaton for a 1-bounded FIFO channel with input port A and output port B. Here, we assume that the data domain consists of two data items 0 and 1. Intuitively, the initial state q_0 stands for the configuration where the buffer is empty, while the states p_0 and p_1 represent the configurations where the buffer is filled with one of the data items. □

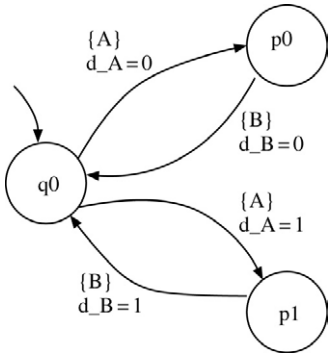


Fig. 1. Constraint automaton for a 1-bounded FIFO channel.

The intuitive behavior of a constraint automaton $\mathcal{A}$ viewed as an acceptor for TDS-tuples is as follows. We assume that the automaton gets a TDS-tuple $\theta \in TDS^{\mathcal{N}ames}$ as input and tries to find out whether θ describes a possible data flow of $\mathcal{A}$ viewed as an operational model, in a similar way that a finite automaton (or ω-automaton) obtains a finite (infinite) word as input and tries to find an accepting run. (However, as constraint automata do not have final states, accepting runs are always infinite.) That is, $\mathcal{A}$ starts in one of its initial states q_0. If the current state is q, then $\mathcal{A}$ waits until data items occur at some of the input/output ports $A_i \in \mathcal{N}ames$. Suppose that data item d_1 occurs at A_1 and data item d_2 occurs 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 $\left[A_1 \mapsto d_1, A_2 \mapsto d_2\right] \models g$ 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. In general, if data occur exactly at the input/output ports $A_i \in N$, then only N-transitions (but no N'-transitions, where N' is a subset or superset of N) where the data constraint is fulfilled can fire.

Having this behavior in mind, the intuitive meaning of conditions (1) and (2) in Definition 3.2 is as follows. Condition (1) stands for the requirement that automata transitions can fire only if some data occurs at one or more of the ports $A_1, \ldots, A_n$, while condition (2) formalizes that the behavior of an automaton may depend only on its observed data (and not on data that will occur sometime in the future).

The constraint automaton for the FIFO1 channel (Example 3.3) is *deterministic*, in the sense that (1) there is a unique initial state and (2) for every state q, every non-empty subset N of $\mathcal{N}ames$ and every data assignment δ, there is *at most* one transition

$$q \xrightarrow{N,g} q' \quad \text{such that } \delta \models g.$$

As for ordinary finite or ω-automata, deterministic constraint automata have a "unique" behavior (formalized as a "run" in the next section) for a given input stream θ. However, Definition 3.2 allows for *non-deterministic* constraint automata since, for a fixed state q, a non-empty subset N of $\mathcal{N}ames$, and a given data-assignment δ, there may be several transitions:

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

Later, in Remark 3.9, we see that (as for ordinary finite automata) for any constraint automaton there exists a language-equivalent deterministic constraint automaton.

Example 3.4 (*Non-deterministic Behavior of Constraint Automata*). The constraint automaton in Fig. 2 can be viewed as an operational model for a connector with ports A and B that allows A first to consume an arbitrary (but finite) number of data items without any effect for the current configuration (represented by the self-loop with name-set $\{A\}$ at the initial state q_0), followed by an I/O operation at A that leads to configuration q_1 in which A and B are forced to synchronize (e.g., in a handshaking mechanism via a synchronous channel). Recall that we assume infinite data flow at all ports (see Remark 2.1). Here and in the sequel, valid guards are skipped in the pictures for constraint automata.

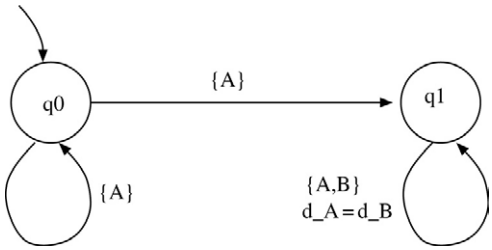


Fig. 2. A non-deterministic constraint automaton.

We now consider the same automaton as an acceptor for TDS pairs $\theta \in TDS^{\{A,B\}}$. The automaton starts in the initial state q_0 and waits there until data flow at A and/or B is observed. If there is only some data value at A, then the automaton has the non-deterministic choice to move to state q_1 or to stay in its initial state. If $\mathcal{A}$ is in q_0 and data flow is observed simultaneously at A and B, the automaton finds no matching transition and rejects. The same holds for the case where, in state q_0, data flow occurs only at B, and for state q_1, when data flow occurs at only one of the ports A or B, or different data values are observed at A and B.

As for ordinary non-deterministic finite automata or ω-automata, the accepted language — which is formally defined in Section 3.2 — covers all input streams that have at least one 'successful' (non-rejecting) run in the automaton. Hence, the existence of a rejecting run does *not* mean that the input stream is not included in the accepted language. Thus, for the above automaton in a situation where the current state is q_0 and data flow is observed at A, the "correct choice" for an input stream $\theta = (\langle\alpha, a\rangle, \langle\beta, b\rangle)$ with $\langle\beta, b\rangle = \langle\alpha^{(i)}, a^{(i)}\rangle$ for some $i \geq 1$ requires an oracle that knows the index i in advance. □

3.2. From automata to streams

In this section, we give the formal definition of the accepted TDS language of a constraint automaton which was informally described in the previous section. In the sequel, we consider constraint automata as acceptors for TDS-tuple that get an "input-stream" $\theta \in TDS^{\mathcal{N}ames}$ and (try to) generate an infinite run for θ, i.e., a sequence $q_0, q_1, q_2, \ldots$ of automaton states that can be obtained via transitions whose name-sets and guards match θ.

We first look at a simple yet representative example. We consider a constraint automaton $\mathcal{A} = (Q, \mathcal{N}ames, \rightarrow, Q_0)$ that models the behavior of connector or component instance through which data elements flow from input port A to output port B. Thus, we set $\mathcal{N}ames = \{A, B\}$ 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:

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

where $\mathcal{L}_{TDS}(\mathcal{A}, q)$ denotes the language accepted by the state q (viewed as the starting state) of automaton $\mathcal{A}$ which is defined as the set of all TDS-tuples $(\langle\alpha, a\rangle, \langle\beta, b\rangle)$ that have an infinite run in $\mathcal{A}$ starting in state q. Intuitively, the data streams α and β in the input stream $\theta = (\langle\alpha, a\rangle, \langle\beta, b\rangle)$ contain the data elements that are being input and output by the ports A and B. The time streams a and b contain, for each of them, the time moments at which these input and output actions take place. The relevance of this timing information is restricted to the particular connector 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. Then, $(\langle\alpha, a\rangle, \langle\beta, b\rangle) \in \mathcal{L}_{TDS}(\mathcal{A}, q)$ if, at any moment $\theta.time(k)$, both the set of names of active ports (the name-set $\theta.N(k)$) and the values of their incoming and outgoing data items (given by the data assignment $\theta.\delta(k)$) "match" the name-sets and constraints of the subsequent transitions of q.

The formal definition of $\mathcal{L}_{TDS}(\mathcal{A}, q)$ can be given by means of a recursive equation system. $\mathcal{L}_{TDS}(\mathcal{A}, q)$ consists of all TDS pairs $\theta = (\langle\alpha, a\rangle, \langle\beta, b\rangle)$ such that there exists a transition

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

that satisfies the following condition:

$$\begin{array}{ll} & a(0) < b(0) \wedge N = \{A\} \wedge [A \mapsto \alpha(0)] \models g \wedge (\langle \alpha', a' \rangle, \langle \beta, b \rangle) \in \mathcal{L}_{TDS}(\mathcal{A}, \bar{q}), \\ \text{or} & b(0) < a(0) \wedge N = \{B\} \wedge [B \mapsto \beta(0)] \models g \wedge (\langle \alpha, a \rangle, \langle \beta', b' \rangle) \in \mathcal{L}_{TDS}(\mathcal{A}, \bar{q}), \\ \text{or} & a(0) = b(0) \wedge N = \{A, B\} \wedge [A \mapsto \alpha(0), B \mapsto \beta(0)] \models g \wedge (\langle \alpha', a' \rangle, \langle \beta', b' \rangle) \in \mathcal{L}_{TDS}(\mathcal{A}, \bar{q}). \end{array}$$

Although the above 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.

Definition 3.5 (*Fixed-point Definition of the Accepted TDS language*). For a given constraint automaton $\mathcal{A} = (Q, \mathcal{N}ames, \longrightarrow, Q_0)$, we define the operator

$$\Omega_{\mathcal{A}} : \left(Q \to 2^{TDS^{\mathcal{N}ames}}\right) \to \left(Q \to 2^{TDS^{\mathcal{N}ames}}\right)$$

as follows. Let $L : Q \to 2^{TDS^{\mathcal{N}ames}}$ be a function and $q \in Q$. Then, $\Omega_{\mathcal{A}}(L)(q)$ consists of all TDS-tuples $\theta \in TDS^{\mathcal{N}ames}$ for which there exists a transition

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

with $\theta' \in L(\bar{q})$, $\theta.N(0) = N$ and $\theta.\delta(0) \models g$. We then define $\mathcal{L}_{TDS}(\mathcal{A}, \cdot)$ as the greatest-fixed-point of $\Omega_{\mathcal{A}}$. As before, $\mathcal{L}_{TDS}(\mathcal{A})$ denotes the union of the TDS languages $\mathcal{L}_{TDS}(\mathcal{A}, q_0)$ for the initial states $q_0 \in Q_0$. □

The above fixed-point definition of the accepted TDS language is often useful for providing simple proofs for language-based properties of automata. However, in some cases, it is easier to reason with the accepted language characterized by means of the (standard) notion of runs:

Definition 3.6 (*Runs in Constraint Automata*). Given a TDS-tuple $\theta \in TDS^{\mathcal{N}ames}$, the set of infinite q-runs for θ in $\mathcal{A}$ is the greatest set of streams $\mathbf{q} = q_0, q_1, \ldots$ over Q such that $q_0 = q$ and there is a transition

$$q_0 \xrightarrow{N,g} q_1$$

with $N = \theta.N(0)$, $\theta.\delta(0) \models g$ and $\mathbf{q}'$ is an infinite q_1-run for θ' in $\mathcal{A}$. By a rejecting q-run for θ in $\mathcal{A}$, we mean a finite sequence of automaton states $q_0, \ldots, q_n$ such that

- $q_0 = q$,
- if $n \geq 1$, then there is a transition $q_0 \xrightarrow{N,g} q_1$ with $N = \theta.N(0)$, $\theta.\delta(0) \models g$ and $q_1, \ldots, q_n$ is a rejecting q_1-run for θ',
- if $n = 0$, then there is no transition $q_0 \xrightarrow{N,g} q_1$ with $N = \theta.N(0)$, $\theta.\delta(0) \models g$.

By an accepting run for θ in $\mathcal{A}$ we mean an infinite q_0-run for θ where q_0 is an initial state. Similarly, a rejecting run for θ in $\mathcal{A}$ denotes a rejecting q-run for θ in $\mathcal{A}$ where $q \in Q_0$. □

It is easy to see that

$$\mathcal{L}_{TDS}(\mathcal{A}, q) = \left\{\theta \in TDS^{\mathcal{N}ames} : \text{there exists an infinite } q\text{-run for } \theta \text{ in } \mathcal{A}\right\}.$$

Example 3.7 (*Accepted TDS language*). The language accepted by the constraint automaton for a 1-bounded FIFO channel (Example 3.3) equals the set

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

Because this automaton is deterministic, any TDS pair has a unique (accepting or rejecting) run. However, this is not the case for non-deterministic constraint automata. For instance, the non-deterministic constraint automaton in Example 3.4 whose accepted TDS language is

$$\left\{(\langle \alpha, a \rangle, \langle \alpha^{(i)}, a^{(i)} \rangle) : \langle \alpha, a \rangle \in TDS, i \geq 1\right\},$$

has infinitely many rejecting runs for any input stream $(\langle \alpha, a \rangle, \langle \alpha^{(i)}, a^{(i)} \rangle)$ (namely, the runs q_0^{i+k}, for $k \geq 1$, where the automaton stays too long in its initial state) and exactly one accepting run, namely q_0^i, q_1^ω. □

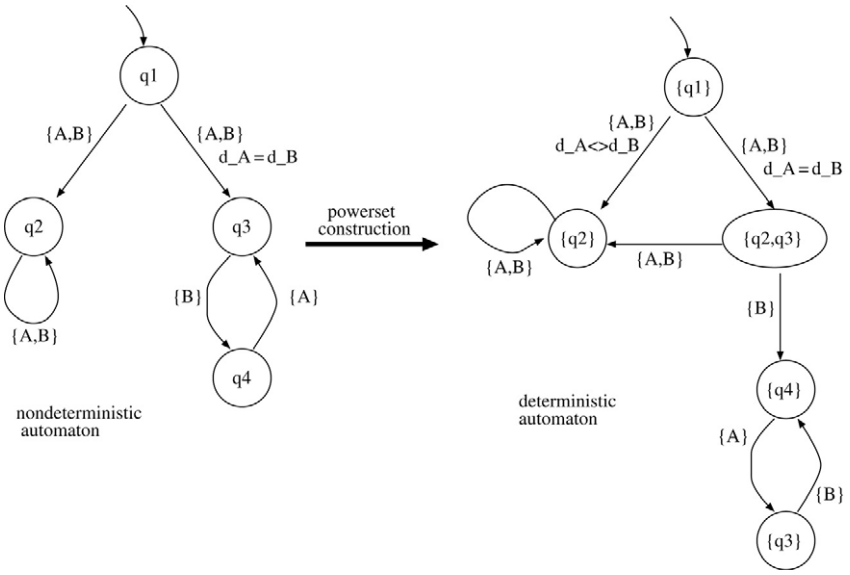


Fig. 3. Example for the powerset construction.

We now show that any non-deterministic constraint automaton can be transformed into a language-equivalent deterministic constraint automaton. For the construction, we need the following notation:

Notation 3.8 (*Data Constraints dc*(...)). For a constraint automaton $\mathcal{A}$ as before, q a state in $\mathcal{A}$, $N \subseteq \mathcal{N}ames$ 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 $dc(q, N, P)$. We use $dc(q, N)$ as an abbreviation for $dc(q, N, Q)$ and $dc(N, P)$ for $\bigvee_{q \in Q} dc(q, N, P)$. □

Intuitively, $dc(q, N, P)$ is the weakest data constraint that ensures the existence of 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.

Remark 3.9 (*Deriving Deterministic Constraint Automata*). As for standard finite automata, deterministic constraint automata are as powerful as their non-deterministic variants, if we are interested only in their accepted stream languages.[3] More precisely, given a non-deterministic constraint automaton $\mathcal{A} = (Q, \mathcal{N}ames, \longrightarrow, Q_0)$, one can use the standard powerset construction to obtain a deterministic constraint automaton

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

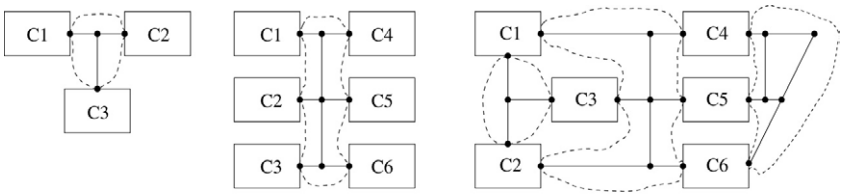
where the transition relation $\longrightarrow_{\mathsf{det}}$ is defined as follows.[4] For $P, P' \subseteq 2^Q$ with $P \neq \emptyset$ and $P' \neq \emptyset$ and $N \subseteq \mathcal{N}ames$:

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

Using similar arguments as in the correctness proof of the powerset construction in ordinary finite automata, it can be shown that $\mathcal{L}_{TDS}(\mathcal{A}) = \mathcal{L}_{TDS}(\mathsf{det}(\mathcal{A}))$. Fig. 3 shows an example. □

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

[4] 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.



(a) a 3-way connector (b) a 6-way connector (c) two 3-way connectors and a 6-way connector

Fig. 4. Components and connectors.

4. Constraint automata as operational model for Reo circuits

In this section, we show how constraint automata can serve as an operational semantics for the coordination language Reo [1]. We start with a brief introduction to Reo (Section 4.1) and then define composition operators for constraint automata that correspond to the Reo connector primitives (Sections 4.2–4.4). Section 4.5 illustrates the compositional construction of the constraint automaton for a given Reo connector through a few examples.

4.1. A Reo primer

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 [1]. 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.

Reo's notion of components and connectors is depicted in Fig. 4, where component instances are represented as boxes, channels as straight lines, and connectors are delineated by dashed lines. Each connector in Reo is, in turn, constructed compositionally out of simpler connectors, which are ultimately composed of primitive channels. For instance, the connector in Fig. 4a may in fact be a flow-regulator (if its three constituent channels are of the right type, as described in [1]). Fig. 4a would then represent a system composed of two *writer* component instances (C1 and C3), plus a *reader* component instance (C2), glued together by our flow-regulator connector. Every component instance performs its I/O operations following its own timing and logic, independently of the others. None of these component instances is aware of the existence of the others, the specific connector used to glue it to the rest, or even of its own role in the composite system. Nevertheless, the protocol imposed by our flow-regulator glue code (see [1] and [2]) ensures that a data item passes from C1 to C2 only whenever C3 writes a data item (whose actual value is ignored): the "tokens" written by C3 thus serve as cues to regulate the flow of data items from C1 to C2. The behavior of the connector, in turn, is independent of the components that it connects: without their knowledge, it imposes a coordination pattern among C1, C2, and C3 that regulates the precise timing and/or the volume of the data items that pass from C1 to C2, according to the timing and/or the volume of tokens produced by C3. The other connectors in Fig. 4 implement more complex coordination patterns.

Channels. 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.

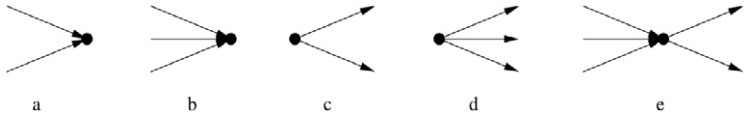


Fig. 5. Nodes in Reo.

Connectors. 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 the coincidence of a set of channel ends, which has specific implications on the flow of data among and through those channel ends.

The set of channel ends coincident on a node A is disjointly partitioned into the sets $\mathsf{Src}(A)$ and $\mathsf{Snk}(A)$, denoting the sets of source and sink channel ends that coincide on A, respectively. A node A is called

- a *source node* if $\mathsf{Src}(A) \neq \emptyset \wedge \mathsf{Snk}(A) = \emptyset$,
- a *sink node* if $\mathsf{Src}(A) = \emptyset \wedge \mathsf{Snk}(A) \neq \emptyset$,
- a *mixed node* if $\mathsf{Src}(A) \neq \emptyset \wedge \mathsf{Snk}(A) \neq \emptyset$.

Fig. 5a and b show sink nodes with, respectively, two and three coincident channel ends. Fig. 5c and d show source nodes with, respectively, two and three coincident channel ends. Fig. 5e 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 to which it is connected. 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 to which it is connected 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 end offers suitable data items, one is selected non-deterministically. A sink node thus acts as a non-deterministic *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 non-deterministically 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 each channel is defined by the user who provides it. Reo defines the semantics of its three types of nodes, mentioned above.

Fig. 6a and b show two Reo connectors. We consider these connectors in more detail in Examples 4.6 and 4.7, respectively, in Section 4.3. 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.

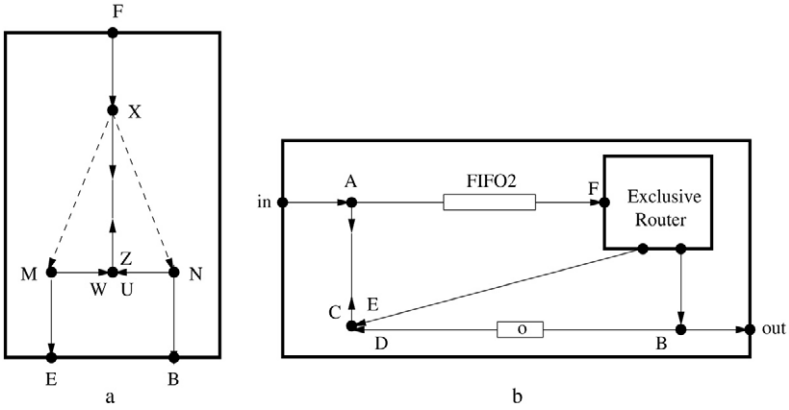


Fig. 6. Exclusive router and shift-lossy FIFO1.

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 depending 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 Fig. 6a. The edge BD in Fig. 6b 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 type of channel can have an initially empty buffer or, as in Fig. 6b, contain an initial data value (in this case, the "o" in the box representing its buffer). Analogously, the edge AF in Fig. 6b represents an asynchronous FIFO channel with the bounded capacity of 2 (*FIFO2*), with its obvious semantics.

An example of the more exotic channels permitted in Reo is the synchronous drain channel (*SyncDrain*), whose visual symbol appears as the edges XZ and AC in Fig. 6a 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 Fig. 6): it has two source ends through which it accepts and loses data items, but never simultaneously. *SyncSpout* and *AsyncSpout* are dual to the drain channel types, as they have two sink ends.

In this paper, as in [2], we do not consider the dynamic behavior of components in creating and composing connectors. Our focus is on the Reo circuits, built from basic connectors (channels and merger) via join and hide operations, without considering the split-operation, which may abolish the effect of previous join-operations and can be followed by further join-operations (yielding a network of Reo circuits).

We now explain how constraint automata can be used to model the possible data flow of a given Reo circuit. The nodes of a Reo circuit play the role of the ports in the constraint automata.

The operational semantics presented in [1] describes the configurations in which a set of I/O operations for certain nodes can take place and which successor configurations can be reached. Hence, we can reformulate the semantics presented in [1] in terms of a constraint automaton whose states are the configurations and whose transitions correspond to the possible I/O operations. Instead, we follow another approach in this paper and provide a *compositional* semantics for Reo circuits. Thus, we need constraint automata for each of the basic channel connectors and automata operations to mimic the behavior of the Reo operations for join and hiding.

4.2. Constraint automata for the basic channels

Fig. 7 shows the constraint automata for some of the standard basic channel types: synchronous channels with source A and sink B (or vice versa), (a)synchronous drain with the sources A, B, (a)synchronous spout with the sinks A, B, and lossy synchronous channels with source A and sink B. In every case, one single state is sufficient. Moreover, the automata are deterministic.

A constraint automaton for the FIFO1 channel was shown in Example 3.3. For FIFO channels with capacity ≥ 2, similar constraint automata can be used. However, the number of states grows exponentially with the capacity. For

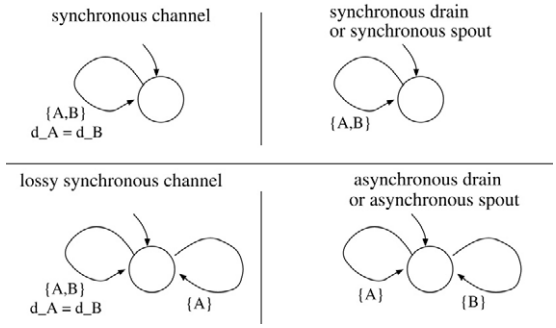


Fig. 7. Deterministic constraint automata for some basic connectors.

instance, for a FIFO2 channel with the data domain $\{0, 1\}$, we need seven states representing the configurations where the buffer is empty or the buffer contains one element (0 or 1) or is full (00, 01, 10 or 11). For unbounded FIFO channels, we even get constraint automata with an infinite state-space.

Of course, for compositional reasoning, we must assume that other user-defined basic channel types are also specified by appropriate constraint automata.

4.3. Join: Merge and product

As constraint automata do not distinguish between input ports (source nodes in Reo) and output ports (sink nodes in Reo), we cannot expect a general join operator on constraint automata that covers both the replicator semantics of joining source nodes and the merge semantics of joining sink nodes.

Since we restrict our attention to (static) Reo circuits, we may assume that a given Reo circuit is built out of some basic channels via the join and hiding operations where the join operations are performed in an order such that any mixed node of the final circuit arises through first joining certain sink nodes and then joining the resulting node with certain source nodes. On the automata level, the join of a source node with another (sink, source or mixed) node will be realized by a *product* construction, while joining sink nodes will be modeled with the help of a *merger*.

We first consider the join operation for node pairs $\langle B, \bar{B}\rangle$ where, in each pair, at most one of the nodes is a sink or mixed node (while the other is a source node). In this case, the effect of join is that all data flow at the nodes B and $\bar{B}$ agree.

In the sequel, let us assume that two Reo circuits with node-sets $\mathcal{N}_1$ and $\mathcal{N}_2$ are given for which we want to perform a join operation for node-pairs $\langle B_i, \bar{B}_i\rangle \in \mathcal{N}_1 \times \mathcal{N}_2$, $i = 1, \ldots, k$, where, for any i, at least one of the nodes B_i or $\bar{B}_i$ is a source node. We may assume that the constraint automata $\mathcal{A}_1$ and $\mathcal{A}_2$ for both circuits have already been constructed. To simplify the notation, we assume that the names of the nodes are renamed in such a way that $B_1 = \bar{B}_1, \ldots, B_k = \bar{B}_k$ and that the two circuits/automata do not contain other common nodes. That is, we have to join all common nodes $B \in \mathcal{N}_1 \cap \mathcal{N}_2$. On the language level, join (under the above conditions) can be viewed as an analogue to the natural join (denoted $\bowtie$) for relational data bases. For instance, given two TDS languages $L_1 = L_1(A, B)$ and $L_2 = L_2(B, C)$,[5] the TDS language $(L_1 \bowtie L_2)(A, B, C)$ is given by

$$L_1 \bowtie L_2 = \big\{(\langle\alpha, a\rangle, \langle\beta, b\rangle, \langle\gamma, c\rangle) : (\langle\alpha, a\rangle, \langle\beta, b\rangle) \in L_1 \text{ and } (\langle\beta, b\rangle, \langle\gamma, c\rangle) \in L_2\big\}.$$

In a similar way, we may define the natural join for TDS languages with other name-sets. Thus, join as an operator for TDS languages can be regarded as a generalization of intersection. It is realized on the automata level by a product construction.

Definition 4.1 (*Product automaton*). The product automaton of the two constraint automata $\mathcal{A}_1 = (Q_1, \mathcal{N}ames_1, \longrightarrow_1, Q_{0,1})$ and $\mathcal{A}_2 = (Q_2, \mathcal{N}ames_2, \longrightarrow_2, Q_{0,2})$ is:

$$\mathcal{A}_1 \bowtie \mathcal{A}_2 = (Q_1 \times Q_2, \mathcal{N}ames_1 \cup \mathcal{N}ames_2, \longrightarrow, Q_{0,1} \times Q_{0,2})$$

[5] The notation $L(A, B)$ suggests that L is a TDS language for the name-set $\mathcal{N} = \{A, B\}$.

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

$$\frac{q_1 \xrightarrow{N_1,g_1}_1 p_1, \quad q_2 \xrightarrow{N_2,g_2}_2 p_2, \quad N_1 \cap \mathcal{N}ames_2 = N_2 \cap \mathcal{N}ames_1}{\langle q_1, q_2\rangle \xrightarrow{N_1 \cup N_2, g_1 \wedge g_2} \langle p_1, p_2\rangle}$$

and

$$\frac{q_1 \xrightarrow{N,g}_1 p_1, \quad N \cap \mathcal{N}ames_2 = \emptyset}{\langle q_1, q_2\rangle \xrightarrow{N,g} \langle p_1, q_2\rangle}$$

and the latter's symmetric rule. □

The following lemma shows the correctness of the product construction, in the sense that the product automaton realizes the (natural) join of the TDS languages of its arguments:

Lemma 4.2 (*Correctness of the Product*). *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 $\mathcal{N}ames_1 = \mathcal{N}ames_2$ then $\mathcal{L}_{TDS}(\mathcal{A}_1 \bowtie \mathcal{A}_2) = \mathcal{L}_{TDS}(\mathcal{A}_1) \cap \mathcal{L}_{TDS}(\mathcal{A}_2)$.* □

Proof. (b) follows by (a). We provide the proof for (a). In the sequel, let $\mathcal{N}ames = \mathcal{N}ames_1 \cup \mathcal{N}ames_2$.

"$\supseteq$": We show that, for all states $q_1 \in Q_1$, $q_2 \in Q_2$, the function $Q_1 \times Q_2 \to 2^{TDS^{\mathcal{N}ames}}$, $\langle q_1, q_2\rangle \mapsto L(q_1, q_2)$ where

$$L(q_1, q_2) = \mathcal{L}_{TDS}(\mathcal{A}_1, q_1) \bowtie \mathcal{L}_{TDS}(\mathcal{A}_2, q_2)$$

is a post-fixed-point of $\Omega_{\mathcal{A}_1 \bowtie \mathcal{A}_2}$ (as defined in Definition 3.5), i.e.,

$$L(q_1, q_2) \subseteq \Omega_{\mathcal{A}_1 \bowtie \mathcal{A}_2}(L)(q_1, q_2).$$

Recall that the greatest-fixed-point of a monotonic operator in a lattice is the greatest post-fixed-point; see, e.g., [7].

Let $\theta \in L(q_1, q_2)$, that is, θ is the "join" of two timed data streams $\theta_i \in \mathcal{L}_{TDS}(\mathcal{A}_i, q_i)$, $i = 1, 2$, with $\theta_1|_A = \theta_2|_A$ for all $A \in \mathcal{N}_1 \cap \mathcal{N}_2$. (By the "join" of θ_1 and θ_2, we mean the unique TDS-tuple for the name-set $\mathcal{N}$ with $\theta|_A = \theta_i|_A$ if $A \in \mathcal{N}_i$.)

- If $\theta.time(0) = \theta_1.time(0) < \theta_2.time(0)$, then there exists a transition $q_1 \xrightarrow{N,g}_1 p_1$ in $\mathcal{A}_1$ such that

  $$N = \theta.N(0) = \theta_1.N(0), \quad \theta.\delta(0) = \theta_1.\delta(0) \models g \quad \text{and} \quad \theta_1' \in \mathcal{L}_{TDS}(\mathcal{A}_1, p_1).$$

  Hence, $N \subseteq \mathcal{N}_1 \setminus \mathcal{N}_2$ and the above transition can be lifted to a transition

  $$\langle q_1, q_2\rangle \xrightarrow{N,g} \langle p_1, q_2\rangle.$$

  Moreover, we have $\theta' = (\theta_1', \theta_2) \in L(p_1, q_2)$, and hence, $\theta \in \Omega_{\mathcal{A}_1 \bowtie \mathcal{A}_2}(L)(q_1, q_2)$.
- The case $\theta.time(0) = \theta_2.time(0) < \theta_1.time(0)$ is symmetric.
- If $\theta.time(0) = \theta_1.time(0) = \theta_2.time(0)$, then there exist transitions $q_i \xrightarrow{N_i,g_i}_i p_i$ in $\mathcal{A}_i$, $i = 1, 2$, such that

  $$N_i = \theta_i.N(0), \quad \theta_i.\delta(0) \models g_i \quad \text{and} \quad \theta_i' \in \mathcal{L}_{TDS}(\mathcal{A}_i, p_i).$$

  Hence, the above transitions can be lifted to a transition $\langle q_1, q_2\rangle \xrightarrow{N,g} \langle p_1, p_2\rangle$ where $N = N_1 \cup N_2$ and $g = g_1 \wedge g_2$. We then have:

  $$N = \theta.N(0), \quad \theta.\delta(0) \models g \quad \text{and} \quad \theta' = (\theta_1', \theta_2') \in L(p_1, p_2).$$

  We conclude that $\theta \in \Omega_{\mathcal{A}_1 \bowtie \mathcal{A}_2}(L)(q_1, q_2)$.

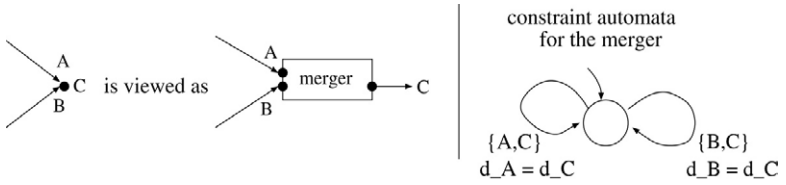


Fig. 8. The merger.

"$\subseteq$": If $\theta \in \mathcal{L}_{TDS}(\mathcal{A}_1 \bowtie \mathcal{A}_2, \langle q_1, q_2\rangle)$, then $\theta_i = (\theta|_A)_{A\in\mathcal{N}_i} \in \mathcal{L}_{TDS}(\mathcal{A}_i, q_i)$ because, for any accepting run

$$\langle q_{0,1}, q_{0,2}\rangle, \langle q_{1,1}, q_{1,2}\rangle, \langle q_{2,1}, q_{2,2}\rangle, \ldots$$

for θ in $\mathcal{A}_1 \bowtie \mathcal{A}_2$, the projection to the $\mathcal{A}_i$ states yields an accepting run for θ_i in $\mathcal{A}_i$ when the states $q_{j+1,i}$ are removed, where j is any index such that, for the taken transition

$$\langle q_{j,1}, q_{j,2}\rangle \xrightarrow{N_j, g_j} \langle q_{j+1,i}, q_{j+1,2}\rangle$$

the name-set N_j has an empty intersection with $\mathcal{N}_i$. $\boxtimes$

It remains to explain how the join of two sink nodes, say A and B, is realized with constraint automata. To capture the merge semantics of the resulting (new) node C, we use a *merger*, as shown in Fig. 8, which we then join (via the product operator $\bowtie$) with the constraint automata that contain A and B, respectively. We can then again apply the product construction to join the resulting constraint automaton (that contains C in its name-set) with another constraint automaton that contains C as a source node. In a similar way, a merger can be defined as a connector with three or more "input" nodes.

Examples for realizing join via merge and product appear in Section 4.5.

4.4. Hiding

The effect of hiding a node that is internal to some connector in a Reo circuit is that data flow at that node is no longer observable from outside. To obtain this effect for TDS languages, the hiding of a name (node) C in a TDS language $L(C, A_1, \ldots, A_n)$ is realized by existential quantification over the C-component; e.g., for $L = L(C, A, B)$:

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

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

Definition 4.3 (*Hiding on Constraint Automata*). Let $\mathcal{A} = (Q, \mathcal{N}ames, \longrightarrow, Q_0)$ be a constraint automaton and $C \in \mathcal{N}ames$. The constraint automaton

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

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_is depend only on C.) The set $Q_{0,C}$ of initial states is

$$Q_{0,C} = Q_0 \cup \big\{p \in Q : q_0 \rightsquigarrow^* p \text{ for some } q_0 \in Q_0\big\}.$$

The transition relation $\longrightarrow_C$ is given by:

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

where $\exists C[g] = \bigvee_{d\in Data}\ g[d_C/d]$. Here, we write $g[d_C/d]$ to denote the data constraint obtained by syntactically replacing all occurrences of d_C in g with d. More precisely, we replace the atoms $d_C = d'$ with true if $d = d'$ and with false if $d \neq d'$. $\square$

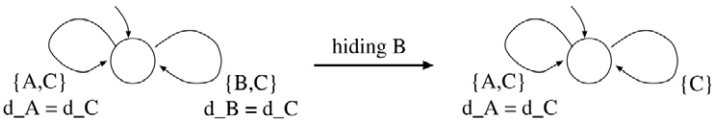


Fig. 9. Hiding a node of the merger.

For instance, if $\mathcal{A}_{merger}$ denotes the merger automaton in Fig. 8, then $\exists C\big[\mathcal{A}_{merger}\big]$ is the same as the automaton for the asynchronous drain shown in Fig. 7.

Unfortunately, the equality $\mathcal{L}_{TDS}(\exists C[\mathcal{A}]) = \exists C[\mathcal{L}_{TDS}(\mathcal{A})]$ does not hold in general (only the "$\subseteq$" relation, as shown in part (a) of Lemma 4.4, holds). For instance, hiding B in the merger automaton in Fig. 8 yields a constraint automaton shown in Fig. 9, with a single state, 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 in the language $\exists B[\mathcal{L}_{TDS}(\mathcal{A}_{merger})]$ because, in every TDS-tuple accepted by $\mathcal{A}_{merger}$, it infinitely often happens that data simultaneously occur on B and C but not A. To remedy the situation in general, we need to add *fairness* conditions that declare which automata transitions must be taken infinitely often (similar to Büchi or other ω-automata). Instead, here we show the correctness of hiding under certain conditions:

Lemma 4.4 (*Correctness of Hiding*). (a) $\exists C\big[\mathcal{L}_{TDS}(\mathcal{A})\big] \subseteq \mathcal{L}_{TDS}(\exists C[\mathcal{A}])$.

(b) *If $\mathcal{A}$ is finite and does not contain a cycle $q_0 \xrightarrow{N_1,g_1} q_1 \xrightarrow{N_2,g_2} \cdots \xrightarrow{N_k,g_k} q_k = q_0$ where $k \geq 1$, $g_1, \ldots, g_k$ are satisfiable and $C \notin N_1 \cup \cdots \cup N_k$, then*

$$\exists C\big[\mathcal{L}_{TDS}(\mathcal{A})\big] = \mathcal{L}_{TDS}(\exists C[\mathcal{A}]).$$

In part (b), we may also assume an infinite constraint automaton without infinite paths built by transitions that do *not* contain C in their name-set and that have satisfiable guards.

Proof. Part (a). With arguments that are similar to those used in the proof of Lemma 4.2, we can show that the function

$$L : Q \to 2^{TDS^{\mathcal{N}ames}} \quad \text{where } L(q) = \exists C\big[\mathcal{L}_{TDS}(\mathcal{A}, q)\big]$$

is a post-fixed-point of the operator $\Omega_{\exists C[\mathcal{A}]}$. From this, we conclude that, for any state q, $\exists C\big[\mathcal{L}_{TDS}(\mathcal{A}, q)\big] \subseteq \mathcal{L}_{TDS}(\exists C[\mathcal{A}], q)$.

Part (b). Let $\theta \in TDS^{\mathcal{N}\setminus\{C\}}$ be a TDS-tuple in $\mathcal{L}_{TDS}(\exists C[\mathcal{A}])$ and let $\mathbf{q} = q_0, q_1, q_2, \ldots$ be an infinite run for θ in $\exists C[\mathcal{A}]$ with $q_0 \in Q_{0,C}$. By our assumption, there are infinitely many transitions taken in that run which are obtained from transitions in $\mathcal{A}$ that contain C in their name-set.

We now extend $\mathbf{q}$ by inserting states and define a timed data stream $\langle\gamma, c\rangle$ such that the extended run is an infinite run for the TDS-tuple $(\theta, \langle\gamma, c\rangle) \in TDS^{\mathcal{N}}$ in $\mathcal{A}$.

- As $q_0 \in Q_{0,C}$, we have $q_0' \rightsquigarrow^* q_0$ for some $q_0' \in Q_0$. Hence, there exists a sequence of C-transitions with satisfiable data constraints in $\mathcal{A}$ that leads from q_0' to q_0, say

$$q_0' \xrightarrow{\{C\},g_1} p_1 \xrightarrow{\{C\},g_2} \cdots \xrightarrow{\{C\},g_n} p_n = q_0.$$

  Then, we replace $\mathbf{q}$ by $\mathbf{q}_0 = q_0', p_1, \ldots, p_n, q_1, q_2, \ldots$. We choose real values $c(k)$ with

$$0 < c(0) < c(1) < \cdots < c(n-1) < \theta.time(0)$$

  and data values $\gamma(k)$ such that $[C \mapsto \gamma(k)] \models g_k$, $k = 0, 1, \ldots, n-1$.
- We now assume that $\mathbf{q}_j \in Q^\omega$ and $\gamma(0), \ldots, \gamma(\ell) \in Data$, an increasing sequence $c(0), \ldots, c(\ell)$ of time points are defined (such that $c(\ell) < \theta.time(j)$). We then consider the transition

$$q_j \xrightarrow{\bar{N},\bar{h}}_C q_{j+1}$$

  which was taken in the given run $\mathbf{q}$ for θ in $\exists C[\mathcal{A}]$. That is, we have

$$\theta.N(j) = \bar{N}, \theta.\delta(j) \models \bar{g}$$

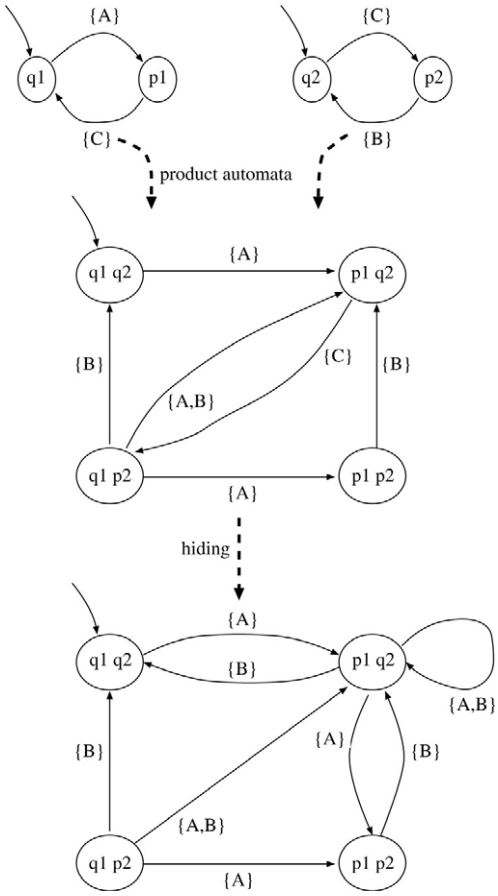


Fig. 10. Composition of two FIFO1 channels.

and there are transitions

$$q_j \xrightarrow{\{C\},h_1} r_1 \xrightarrow{\{C\},h_2} \cdots \xrightarrow{\{C\},h_m} r_m \xrightarrow{N,h} q_{j+1}$$

in $\mathcal{A}$ where $h_1, \ldots, h_m \not\equiv \mathsf{false}$ and $\bar{N} = N \setminus \{C\}$, $\bar{h} = \exists C[h]$. Hence, we may choose real numbers $c(k)$ for $k = \ell + 1, \ldots, \ell + m + 1$, with

$$c(\ell) < c(\ell + 1) < \cdots < c(\ell + m + 1) < \theta.time(j + 1)$$

and data values $\gamma(k) \in Data$ with $[C \mapsto \gamma(\ell + i)] \models h_i$ and $\delta \models h$, where δ is a data assignment for the name-set N that agrees with $\theta.\delta(j)$ for all $A \in \bar{N}$ and possibly contains a suitable data assignment for C.

In this way, we obtain an infinite run $\mathbf{q}$ for $(\theta, \langle\gamma, c\rangle)$ in $\mathcal{A}$. (Here, it is important to notice that, by our assumption, γ and c are infinite.) Hence, $(\theta, \langle\gamma, c\rangle) \in \mathcal{L}_{TDS}(\mathcal{A})$ and thus $\theta \in \exists C\big[\mathcal{L}_{TDS}(\mathcal{A})\big]$. ⊠

4.5. Examples for the construction of constraint automata via join and hiding

We now provide some simple examples to demonstrate how the constraint automaton of a Reo circuit can be obtained in a compositional way.

Example 4.5 (*Composition of Two 1-Bounded FIFO Channels*). Fig. 10 shows how a 2-bounded FIFO channel can be obtained from two 1-bounded FIFO channels $\mathcal{A}_{FIFO1}(A, C)$ and $\mathcal{A}_{FIFO1}(C, B)$ via product and hiding:

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

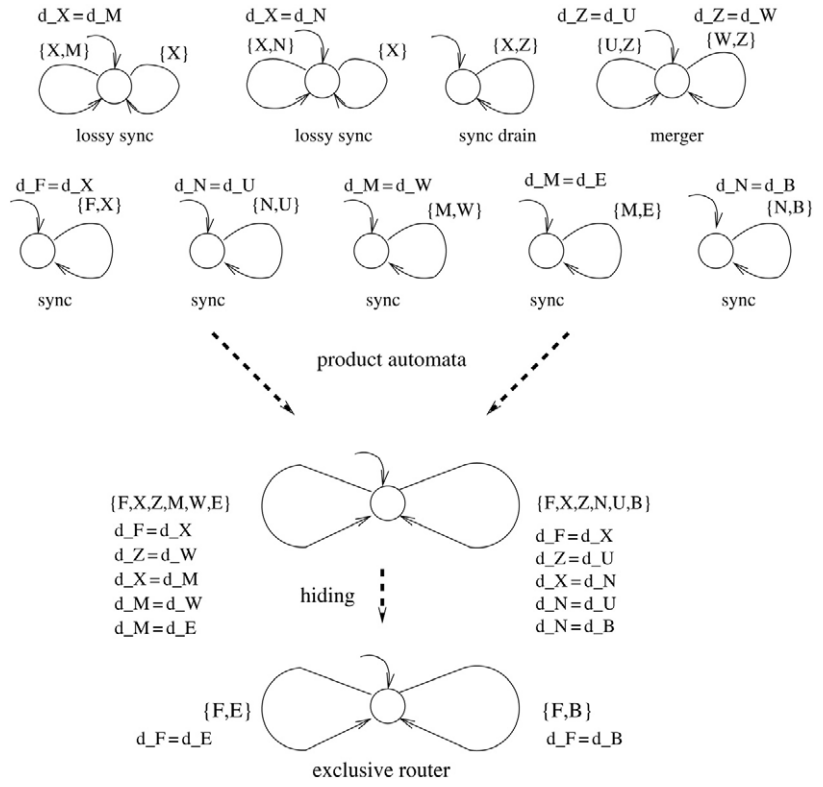


Fig. 11. Exclusive router obtained through composition of basic Reo channels.

For simplicity, we deal with a singleton data domain $Data = \{d\}$ which allows us to skip the data constraints 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 $\langle p_1, q_2\rangle$). □

Example 4.6 (*Exclusive Router*). Fig. 6a 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 non-deterministically. The input data is never replicated to more than one of the output ports.[6]

Fig. 6a shows that the exclusive router is obtained by composing two *LossySync* channels (XM, XN), a *SyncDrain* (XZ) channel, a merger (inherent in the mixed node of Z), and five *Sync* channels (FX, MW, NU, ME, NB):

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

Fig. 11 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. □

Example 4.7 (*Shift-lossy FIFO1 Channel*). Fig. 6b shows a Reo network for a connector that behaves as a lossy FIFO1 channel with a shift loss-policy. This channel is called shift-lossy FIFO1 (*ShiftFIFO1*). It behaves as a normal

[6] The behavior of this connector is the counterpart of the primitive non-deterministic selection inherent in the merge that a Reo (sink or mixed) node performs on its multiple input, modeled by the merger in Fig. 7.

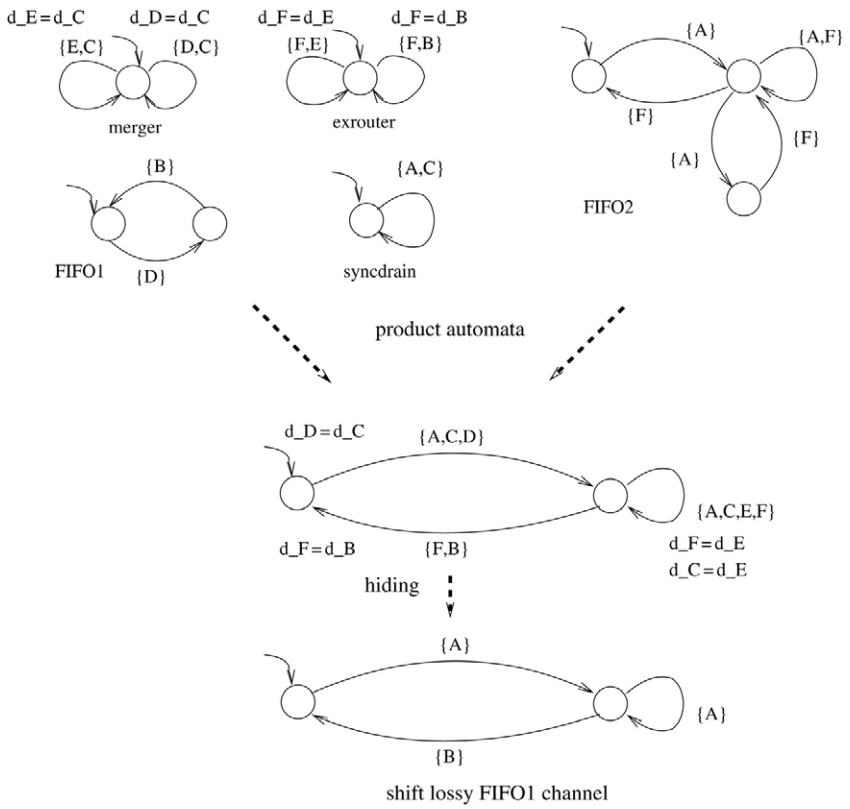


Fig. 12. Shift-lossy FIFO1 channel obtained through composition of other Reo channels.

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 Fig. 6b is composed of an exclusive router, XRouter(F,E,B) (shown in Fig. 6a and explained in Example 4.6), a merger (inherent in the mixed node of C), a *SyncDrain* (AC), an initially full FIFO1 channel (BD), and an initially empty FIFO2 channel (AF):[7]

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

Fig. 12 shows how the constraint automaton for our *ShiftFIFO1* channel is obtained from the constraint automata of its constituents through product and hiding. □

4.6. Parameterized constraint automata

In the previous examples, we concentrated on data-abstract coordination mechanisms. In many applications, the data-abstract view is too coarse, e.g., for reasoning about the functionality of the components that are glued together. Because data dependences often lead to rather complex constraint automata, we propose a parameterized notation that can simplify the picture of constraint automata with non-trivial guards. For instance, the 1-bounded FIFO channel with arbitrary data domain can be depicted as in Fig. 13.

[7] The assumption that the FIFO1 channel BD is full, while the FIFO channel AF is initially empty — as depicted in Fig. 6b — yields an initially empty shift-lossy channel.

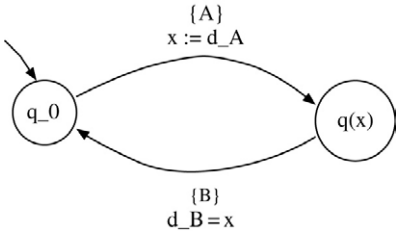


Fig. 13. Parameterized constraint automaton for a 1-bounded FIFO channel.

The automaton in Fig. 13 is *not* a constraint automaton, but an intuitive symbolic representation of the constraint automaton with state-space $Q = \{q_0\} \cup \{q(d) : d \in Data\}$, $Q_0 = \{q_0\}$, $\mathcal{N}ames = \{A, B\}$ and the transitions

$$q_0 \xrightarrow{\{A\}, d_A = d} q(d), \quad q(d) \xrightarrow{\{B\}, d_B = d} q_0$$

for any data item $d \in Data$. Formally, to reason about data-dependent coordination mechanisms, we define a *parameterized constraint automaton* as a tuple

$$\mathcal{P} = (Loc, Var, \nu, \mathcal{N}ames, \rightsquigarrow, Loc_0, init)$$

where

- Loc is a set of locations,
- $Loc_0 \subseteq Loc$ is a set of initial locations,
- Var is a set of variables,
- $\nu : Loc \to 2^{Var}$ assigns to any location ℓ a (possibly empty) set of variables,
- $init$ is a function that assigns to any initial location $\ell \in Loc_0$ a condition for the variables.

$\nu(\ell)$ can be viewed as the parameter list of location ℓ. For instance, in Fig. 13 we use $q(x)$ to denote that q is a location with parameter list $\nu(q) = \{x\}$, while q_0 is a location with an empty parameter list. The initial condition for q_0 is omitted which denotes that $init(q_0) = \mathsf{true}$.

The transition relation $\rightsquigarrow$ of a parameterized constraint automaton is a (finite) set of tuples (ℓ, N, h, X, ℓ'), written in the form

$$\ell \overset{N,h}{\rightsquigarrow}_X \bar{\ell}.$$

Here, ℓ and $\bar{\ell}$ are locations and N is a non-empty name-set. h is a (parameterized) data constraint for N, built out of atoms of the form "$d_A = expr$". The expression $expr$ is built from constants $d \in Data$, the symbols d_B for $B \in N$, variables $x \in \nu(\ell)$ and operators for the chosen data domain, e.g., Boolean operator $\vee$, $\wedge$, etc. for $Data = \{0, 1\}$ and arithmetic operators $+$, $*$, etc. for $Data = \mathbb{N}$. The subscript X of the above transition stands for a function that assigns to each variable $\bar{x} \in \nu(\bar{\ell}) \setminus \nu(\ell)$ and possibly to some of the variables in $\nu(\bar{\ell}) \cap \nu(\ell)$ an expression that is built out of the symbols d_A for nodes $A \in N$, constants $d \in Data$, variables $x \in \nu(\ell)$, and operators on $Data$. For instance, if $Data = \mathbb{N}$, the intuitive meaning of $X(\bar{x}) = d_A + x$ is the assignment "$\bar{x} := d_A + x$". For another example, if $Data = \{0, 1\}$, then we deal with assignments like "$\bar{x} := \neg d_A \wedge x$".

We use parameterized constraint automata as a *symbolic representation* of (non-parameterized) constraint automata. The states of the latter are obtained by augmenting the locations with values for the variables of their parameter list. Formally, given $\mathcal{P}$ as above, the induced constraint automaton $\mathcal{A}_\mathcal{P} = (Q, \mathcal{N}ames, \longrightarrow, Q_0)$ is defined as follows. The state-space Q of $\mathcal{A}_\mathcal{P}$ consists of the pairs $\langle \ell, \eta \rangle$, where $\ell \in Loc$ is a location and η is a variable evaluation for the variables $x \in \nu(\ell)$, i.e., η is a function from $\nu(\ell)$ to $Data$. The states $\langle \ell, \eta \rangle$ with $\ell \in Loc_0$ and $\eta \models init(\ell)$ are the initial states of $\mathcal{A}_\mathcal{P}$. The transition relation $\longrightarrow$ is derived from $\rightsquigarrow$ by the following rule:

$$\frac{\ell \overset{N,h}{\rightsquigarrow}_X \bar{\ell}, \quad \bar{\eta} = \eta[X, \delta]|_{\nu(\bar{\ell})}, \quad g = h[x/\eta(x) : x \in \nu(\ell)] \wedge g[\delta]}{\langle \ell, \eta \rangle \overset{N,g}{\longrightarrow} \langle \bar{\ell}, \bar{\eta} \rangle}$$

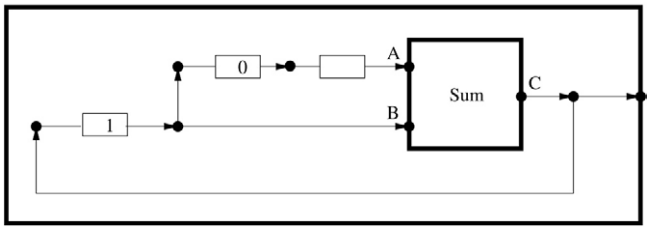


Fig. 14. Reo circuit for Fibonacci series.

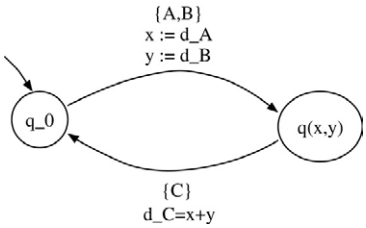


Fig. 15. Parameterized constraint automaton for Sum.

where $\delta = [A \mapsto \delta_A : A \in N_X]$ is an arbitrary data assignment for N_X, the set of names $A \in N$ where X contains an assignment "$\bar{x} := \ldots d_A \ldots$" (in which the symbol d_A occurs in the expression on the right) and $g[\delta]$ is the data constraint

$$g[\delta] = \bigwedge_{A \in N_X} (d_A = \delta_A).$$

The construct $h[x/\eta(x) : x \in v(\ell)]$ stands for the data constraint obtained from h by syntactically replacing variable $x \in v(\ell)$ with the value $\eta(x) \in \mathit{Data}$. The construct $\eta[X, \delta]$ denotes the evaluation for the variables in $v(\ell) \cup v(\bar{\ell})$ that is obtained from η by executing the assignments of X. For instance,

$$\eta[\underbrace{\bar{x} := d_A}_{X}, \underbrace{A \mapsto d}_{\delta}](y) = \begin{cases} \eta(y) & : \text{ if } y \in v(\ell) \setminus \{\bar{x}\} \\ d & : \text{ if } y = \bar{x}. \end{cases}$$

The construct $\eta[X, \delta]|_{v(\bar{\ell})}$ denotes the restriction of $\eta[X, \delta]$ to the variables in $v(\bar{\ell})$.

Note that constraint automata are special instances of their parameterized version with empty parameter lists for all their locations. (In this case, there is no difference between locations and states, and we have $\mathcal{A}_{\mathcal{A}} = \mathcal{A}$.)

The product construction (Definition 4.1) can easily be modified for parameterized constraint automata $\mathcal{P}_1$ and $\mathcal{P}_2$ with disjoint variable sets such that the unfolding of the product $\mathcal{P}_1 \bowtie \mathcal{P}_2$ into a (non-parameterized) constraint automaton $\mathcal{A}_{\mathcal{P}_1 \bowtie \mathcal{P}_2}$ generates the same TDS language as the product $\mathcal{A}_{\mathcal{P}_1} \bowtie \mathcal{A}_{\mathcal{P}_2}$ of the constraint automata for $\mathcal{P}_1$ and $\mathcal{P}_2$.

Example 4.8 (*A Component connector for the Fibonacci numbers*). We consider the Reo circuit in Fig. 14, which uses a component, Sum, in the context of certain channels to generate the stream of numbers in the Fibonacci series. Sum has two input ports A and B and one output port C through which it produces the sum of its input values.

Fig. 15 shows a parameterized constraint automaton $\mathcal{P}_{\mathrm{Sum}}$ that can be viewed as an interface specification for Sum. (Here, we assume that $\mathit{Data} = \mathbb{N}$.)

Joining $\mathcal{P}_{\mathrm{Sum}}$ with the constraint automaton for the Reo circuit in Fig. 14 "around" Sum (which can be obtained in a compositional way, as in the previous examples), we obtain the parameterized constraint automaton $\mathcal{P}_{\mathrm{Fib}}$ in Fig. 16.

We may now unfold $\mathcal{P}_{\mathrm{Fib}}$ into a (non-parameterized) constraint automaton, and hide the names A and B to obtain an infinite-state constraint automaton $\mathcal{A}$ (with the singleton name-set $\{C\}$) whose accepted TDS-language is the set of timed data streams $\langle \gamma, c \rangle$, where the data stream γ stands for the infinite sequence of Fibonacci numbers and c is an arbitrary time stream. □

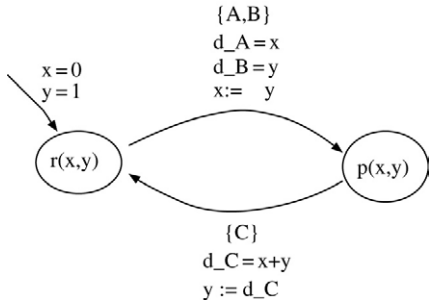


Fig. 16. Parameterized constraint automaton for Fibonacci series.

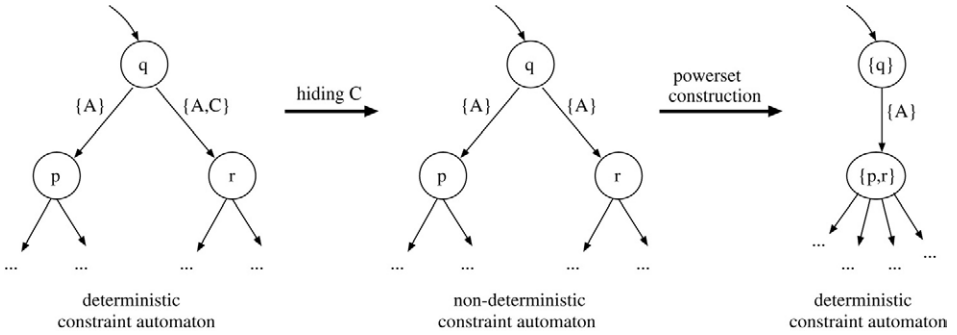


Fig. 17. Transformation of deterministic and non-deterministic constraint automata.

4.7. Remarks on the constraint automata semantics for Reo

We conclude our presentation of the constraint automata semantics for Reo with a few remarks.

Deterministic constraint automata. The product of two deterministic constraint automata is always a deterministic automaton, while hiding can turn a deterministic constraint automaton into a non-deterministic one. In particular, the constraint automaton for a Reo circuit without hidden nodes is always deterministic, provided that the user-defined basic channels are specified by deterministic constraint automata. (Recall that the automaton for the standard basic channels, such as synchronous channels, drains, spouts and FIFO channels, are deterministic.) For modeling circuits with hidden nodes, the hiding operator may yield a non-deterministic automaton, as illustrated by the left transformation in Fig. 17.

However, one can derive from $\exists C[\mathcal{A}]$ a language-equivalent deterministic automaton $\mathsf{det}(\exists C[\mathcal{A}])$; see Remark 3.9. Intuitively, the states of $\mathsf{det}(\exists C[\mathcal{A}])$ stand for *sets* of configurations in the given Reo circuit, as depicted by the right transformation in Fig. 17.

Composition operators in related models. Our product operator relies on the standard construction for building finite automata for intersection and has similarities with composition operators in similar models, e.g., TCSP-like parallel composition of labeled transition systems with synchronization over common actions and interleaving for the other actions [4] or the one-to-many composition of port automata [11]. 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 non-deterministic finite automata). In interface automata, composition is complex, because it requires a compatibility check first. Two interface automata are compatible if errors can be avoided.

Other semantics for Reo. Essentially, our compositional constraint automata semantics of Reo in this paper is consistent with the operational semantics presented in [1] (and its derived constraint automata semantics), and with the timed data stream semantics of [2], in the sense that the diagram in Fig. 18 commutes.

Because the semantics that we consider in this paper is a simplification of the full operational semantics of Reo, e.g., as informally described in [1], the left-hand side of the diagram in Fig. 18 commutes only modulo certain details. The

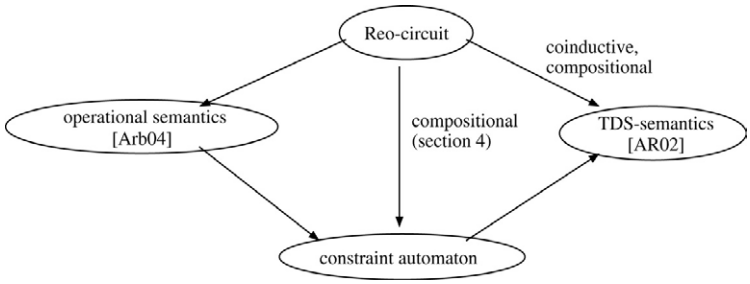


Fig. 18. Relationships among various semantics for Reo.

primary simplifications involve (1) the context-sensitive behavior of certain channels (most prominently, that of our lossy synchronous channel), and (2) the fairness of merge. The specification of the behavior of the lossy synchronous channel requires it *not* to lose the data item written to its source end, if this data item can be consumed at its sink end. This type of context-sensitive behavior can be dealt with in constraint automata by introducing the notion of priorities for their transitions. The details of this scheme are beyond the scope of this paper.

Strictly speaking, Reo itself does not require fairness: Reo is oblivious to (the fairness or other aspects of) the behavior of the channels that it composes, and its internal consistency does not depend on assuming that the non-deterministic merge inherent in the semantics of its sink and mixed nodes is fair. Nevertheless, the expressive power of channel composition in Reo and the correspondence of the formal semantics of Reo connector circuits with the intuitive interpretation of their behavior break down if this non-deterministic merge is not assumed to be fair. We do not address a formal treatment of fairness in our constraint automata semantics for Reo in this paper because, on the one hand, the fairness assumption can be formally incorporated in our basic model analogously to the way it is treated in other models. On the other hand, while it involves no real novelty, the additional formal complexity introduced by fairness becomes somewhat distracting.

The right-hand side of the diagram in Fig. 18 commutes in the sense that, for any Reo circuit R, we have

$$\mathcal{L}_{TDS}(\mathcal{A}_R) = \mathcal{L}_{TDS}^{[AR02]}(R) \qquad (*)$$

where $\mathcal{A}_R$ denotes the constraint automaton for R obtained by the compositional semantics presented in this paper and $R \mapsto \mathcal{L}_{TDS}^{[AR02]}(R)$ denotes the timed-data-stream semantics in [2]. The argument uses the greatest-fixed-point definition of the accepted TDS language, and requires showing that the equation $(*)$ holds for the basic channels, and that

$$\mathcal{L}_{TDS}(\mathcal{A}_1 \bowtie \mathcal{A}_2) = \mathcal{L}_{TDS}(\mathcal{A}_1) \bowtie_{[AR02]} \mathcal{L}_{TDS}(\mathcal{A}_2)$$

where $\bowtie_{[AR02]}$ is the semantic join operator used in [2]. The argument is the same for hiding.

5. Bisimulation and simulation

As for standard labeled transition systems, branching time relations like bisimulation and simulation à la Milner and Park (see e.g., [21]) can be defined for constraint automata. In the context of Reo, we are interested only in the TDS languages induced by Reo circuits (or constraint automata) rather than their branching behavior. Nevertheless, branching time relations are important because they yield an alternative characterization of language equivalence/inclusion, as well as a simple(r) way to verify if two automata are language equivalent, or if the language of one is contained in the language of the other.

5.1. Bisimulation

Recall the definition of $dc(q, N, P)$ introduced in Notation 3.8 in Section 3.2, which we need to define our notion of bisimulation equivalence:

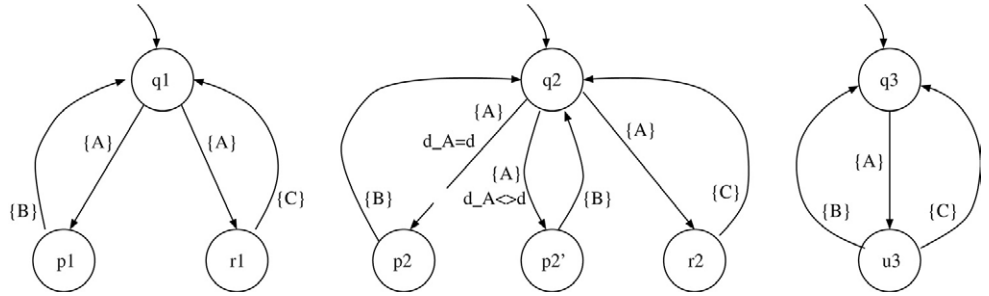


Fig. 19. Similarity and bisimilarity.

Definition 5.1 (*Bisimulation*). Let $\mathcal{A} = (Q, \mathcal{N}ames, \longrightarrow, Q_0)$ be a constraint automaton and let $\mathcal{R}$ be 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 $P \in Q/\mathcal{R}$, and every $N \subseteq \mathcal{N}ames$:

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

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 $\mathcal{A}_2$.

Example 5.2. In the constraint automata of Fig. 19, 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 that contains (q_1, q_2). In fact, the equivalence $\mathcal{R}$ induced by the partition

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

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 that 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 Fig. 19, states q_1, q_2, and q_3 are language-equivalent (i.e., $\mathcal{L}_{TDS}(\mathcal{A}, q_1) = \mathcal{L}_{TDS}(\mathcal{A}, q_2) = \mathcal{L}_{TDS}(\mathcal{A}, q_3)$) but not bisimulation-equivalent. For non-deterministic constraint automata, bisimulation is strictly finer than language equivalence. However, for deterministic constraint automata, bisimulation and language equivalence coincide, as shown in part (b) of the following theorem.

Theorem 5.3 (*Bisimulation Versus Language Equivalence*). *Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be two constraint automata with the same name-set $\mathcal{N}ames$:*

(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 and $\mathcal{L}_{TDS}(\mathcal{A}_i, q) \neq \emptyset$ for all states q in $\mathcal{A}_i$ $(i = 1, 2)$, then*

$$\mathcal{A}_1 \sim \mathcal{A}_2 \quad \textit{iff} \quad \mathcal{L}_{TDS}(\mathcal{A}_1) = \mathcal{L}_{TDS}(\mathcal{A}_2).$$

Proof. (a) follows from the observation that, if $q_1 \sim q_2$, then, for any $\theta \in \mathcal{L}_{TDS}(\mathcal{A}_1, q_1)$ and any infinite q_1-run $\mathbf{q}_1 = q_{0,1}, q_{1,1}, q_{2,1}, \ldots$ for θ in $\mathcal{A}_1$, there exists a q_2-run $\mathbf{q}_2 = q_{0,2}, q_{1,2}, q_{2,2}, \ldots$ for θ in $\mathcal{A}_2$ such that $q_{i,1} \sim q_{i,2}$ for all indices i. To see this, we may use an inductive argument to define the run $\mathbf{q}_2$. Assume that $i \geq 0$ and $q_{i,1} \sim q_{i,2}$ (where, for $i = 0$, we put $q_{i,2} = q_2$). Let

$$q_{i,1} \xrightarrow{N,g} q_{i+1,1}$$

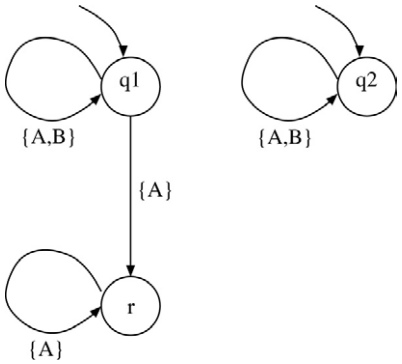


Fig. 20. Language equivalence and bisimilarity.

be the $(i+1)$-th taken transition in $\mathbf{q}_1$ (that is, $N = \theta.N(i)$ and $\theta.\delta(i) \models g$). Then,

$$\theta.\delta(i) \models g \leq dc(q_{i,1}, N, [q_{i+1,1}]) \equiv dc(q_{i,2}, N, [q_{i+1,1}]).$$

Here, we write $[p]$ to denote the bisimulation equivalence class of p. Hence, there exists a transition

$$q_{i,2} \xrightarrow{N,h} q_{i+1,2}$$

where $q_{i+1,2} \in [q_{i+1,1}]$ (i.e., where $q_{i+1,1} \sim q_{i+1,2}$) and $\theta.\delta(i) \models h$.

Part (b). Let $\mathcal{A} = (Q, \mathcal{N}ames, \longrightarrow, Q_0)$ be a deterministic constraint automaton where $\mathcal{L}_{TDS}(\mathcal{A}, q) \neq \emptyset$ for all states $q \in Q$. We show that the relation

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

is a bisimulation. Let $(q_1, q_2) \in \mathcal{R}$, N a non-empty subset of $\mathcal{N}ames$ and P an $\mathcal{R}$-equivalence class. To prove the logical equivalence of $dc(q_1, N, P)$ and $dc(q_2, N, P)$, it suffices to show that, for any data assignment δ for N with $\delta \models dc(q_1, N, P)$, there exists a transition

$$q_2 \xrightarrow{N,h} p_2$$

in $\mathcal{A}$ with $\delta \models h$ and $p_2 \in P$.

If $\delta \models dc(q_1, N, P)$, then there is a transition $q_2 \xrightarrow{N,g} p_1$ with $\delta \models g$ and $p_1 \in P$. We now choose an arbitrary TDS-tuple $\theta \in \mathcal{L}_{TDS}(\mathcal{A}_1, p_1)$ and real number t with

$$0 < t < \theta.time(0).$$

We define $\bar{\theta} = (\bar{\theta}|_A)_{A \in \mathcal{N}ames}$ as the TDS-tuple with

$$\bar{\theta}.time(0) = t, \bar{\theta}.N(0) = N, \quad \bar{\theta}.\delta(0) = \delta$$

and where $\bar{\theta}|_A = \theta_A$ if $A \in \mathcal{N}ames \setminus N$ and, for $A \in \mathcal{N}ames$, the first derivative of $\bar{\theta}|_A$ is $\theta|_A$. Then,

$$\bar{\theta} \in \mathcal{L}_{TDS}(\mathcal{A}_1, q_1) = \mathcal{L}_{TDS}(\mathcal{A}_2, q_2).$$

Hence, there exists a transition $q_2 \xrightarrow{N,h} p_2$ with $\delta = \bar{\theta}.\delta(0) \models h$ and $\theta = \bar{\theta}' \in \mathcal{L}_{TDS}(\mathcal{A}, p_2)$. As $\mathcal{A}$ is deterministic, we have

$$\mathcal{L}_{TDS}(\mathcal{A}, p_i) = \left\{\tilde{\theta}' : \tilde{\theta} \in \mathcal{L}_{TDS}(\mathcal{A}, q_i),\ \tilde{\theta}.N(0) = N,\ \tilde{\theta}.\delta(0) = \delta\right\}, \quad i = 1, 2.$$

As $\mathcal{L}_{TDS}(\mathcal{A}, q_1)$ and $\mathcal{L}_{TDS}(\mathcal{A}, q_2)$ agree, we obtain $\mathcal{L}_{TDS}(\mathcal{A}_1, p_1) = \mathcal{L}_{TDS}(\mathcal{A}_2, p_2)$, and hence the $\mathcal{R}$-equivalence of p_1 and p_2. Thus, $p_2 \in P$. ⊠

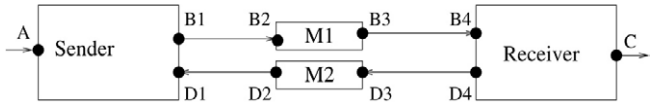


Fig. 21. ABP: components involved.

To see why in part (b) of Theorem 5.3 the assumption $\mathcal{L}_{TDS}(\mathcal{A}_i, q) \neq \emptyset$ is necessary for all states q, consider the deterministic constraint automata $\mathcal{A}_1$ and $\mathcal{A}_2$ in Fig. 20, with initial states q_1 and q_2, respectively.

We have $\mathcal{L}_{TDS}(\mathcal{A}_1, r) = \emptyset$, because of the time-divergence assumption which forces A and B to have infinite data flow. Thus, $\mathcal{A}_1$ and $\mathcal{A}_2$ are language-equivalent, as both accept the TDS language $\{(\langle\alpha, a\rangle, \langle\beta, b\rangle) : a = b\}$. On the other hand, $\mathcal{A}_1$ and $\mathcal{A}_2$ are not bisimulation-equivalent, because $dc(q_1, \{A\}) = \mathsf{true}$ while $dc(q_2, \{A\}) = \mathsf{false}$.

Example 5.4 (*Alternating Bit Protocol*). The alternating bit protocol (ABP) is a method for ensuring successful transmission of data through a faulty communication medium. Here we follow the description of ABP as suggested in [10]. The transmission success is based on the assumption that data can be re-sent an unlimited number of times, if necessary. Fig. 21 shows the components that are involved in this protocol. Data elements from a set *Msg* are communicated between a *Sender* and a *Receiver*. Once the *Sender* reads a message from its port A, it sends this datum through the communication medium M_1 to the *Receiver*, which sends the message out through its port C. The communication medium M_1 is faulty, thus a message sent through this medium can turn up as an error message (represented as !). Every time the *Receiver* receives a message via M_1, it sends an acknowledgment to the *Sender* via the communication medium M_2. The communication medium M_2 is also faulty and may change the datum it conveys.

The ABP protocol is applied to establish correct communication between the *Sender* and the *Receiver* over the faulty communication media M_1 and M_2. The *Sender* attaches a 0 or 1 bit (alternately) to the message, when it sends it through M_1. Thus, the data sent by the *Sender*, or received by the *Receiver*, are pairs $(d, 0)$ or $(d, 1)$ with $d \in Msg$. The *Receiver* sends back the attached bit via M_2, to acknowledge the reception. If the *Receiver* receives a corrupted message, then it sends the previous acknowledgment to the *Sender* once more. As long as the *Sender* receives a corrupted (i.e., !) or wrong acknowledgment (i.e., one whose value it is not expecting), it repeats sending the previous message-bit pair. Alternation of the attached bit enables the *Receiver* to determine whether the received datum is really new, and alternation of the acknowledgment enables the *Sender* to determine whether it acknowledges reception of a datum or that of a corrupted message.

The parameterized constraint automata showing the behavior of the *Sender*, the *Receiver*, the two communication media M_1 and M_2, and the synchronous channels connecting these components, namely $B_1\ B_2$, $B_3\ B_4$, $D_1\ D_2$, and $D_3\ D_4$, are shown in Fig. 22. Our ABP problem involves the following data domain:

$$Data = Msg \cup (Msg \times \{0, 1\}) \cup \{!\} \cup \{0, 1\}.$$

For $(d, b) \in Msg \times \{0, 1\}$, we define $msg(d, b) = d$. At ports A and C, we allow only data items from *Msg*. At ports $B1$, $B2$, $B3$, and $B4$, all data items are from $Msg \times \{0, 1\} \cup \{!\}$, while the channels connecting D_1 to D_2 can transmit data items in $\{0, 1, !\}$ and channels connecting D_3 and D_4 can transmit data items in $\{0, 1\}$ only. These assumptions can be formalized by data constraints. For simplicity in the figures, we skip these data constraints.

The parameterized product automata, which is the result of applying the join and hide operations to all the components in ABP, are shown in Fig. 23. As mentioned earlier, the specification of the protocol requires that the data received by the *Sender* through its port A is correctly sent out through port C of the *Receiver*. This specification is shown in Fig. 24. By comparing the unfoldings of the two parameterized automata in Figs. 23 and 24 into proper constraint automata, it can be seen that the constraint automaton that results from applying product and hiding operations to the constraint automata of the components in the ABP is bisimilar to the constraint automaton for the specification of the ABP. □

5.2. Simulation

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:

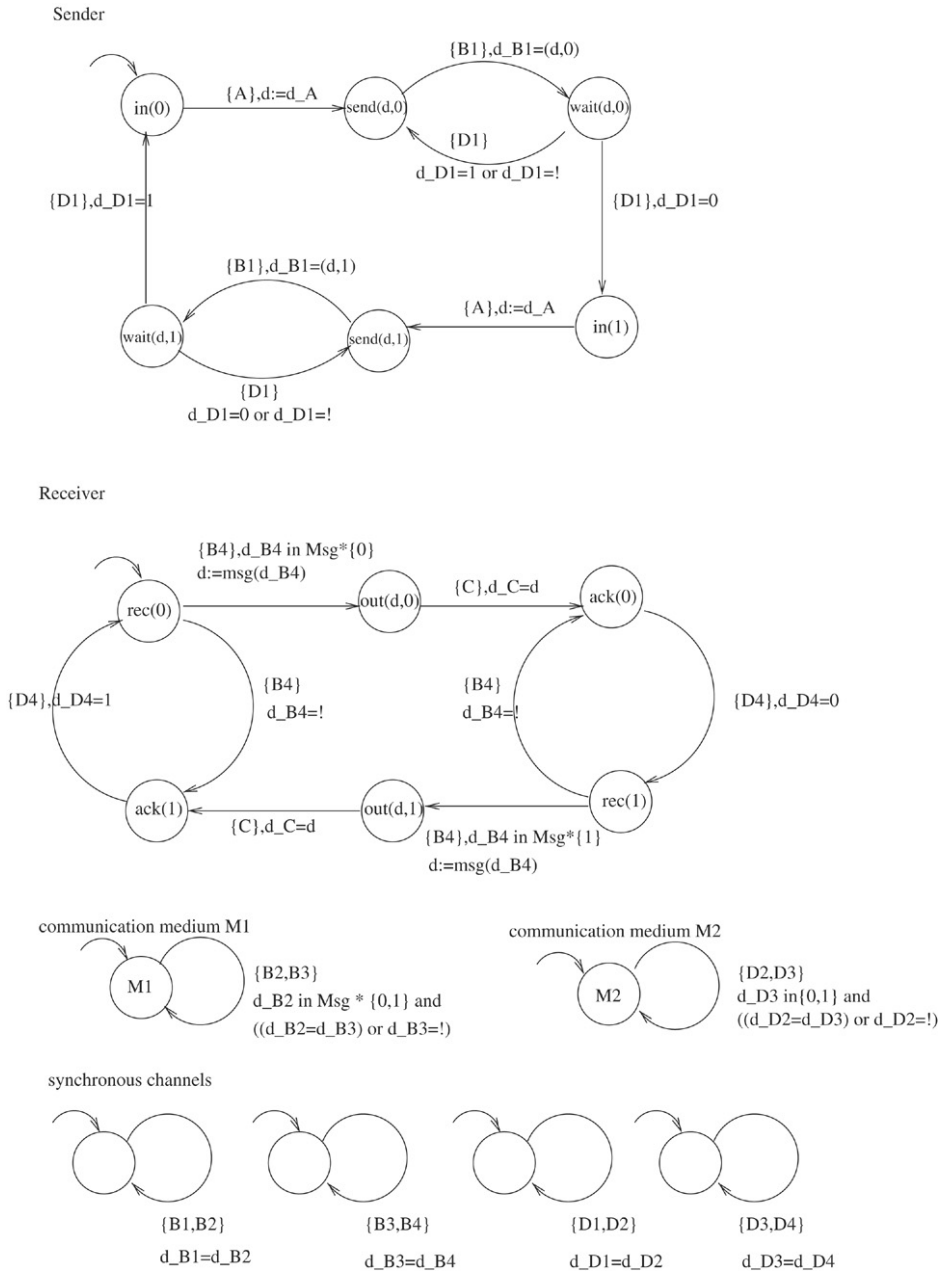


Fig. 22. ABP: constraint automata of components.

Definition 5.5 (*Simulation*). Let $\mathcal{A} = (Q, \mathcal{N}ames, \longrightarrow, Q_0)$ 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 $N \subseteq \mathcal{N}ames$:

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

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 $\mathcal{A}_2$.[8] □

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

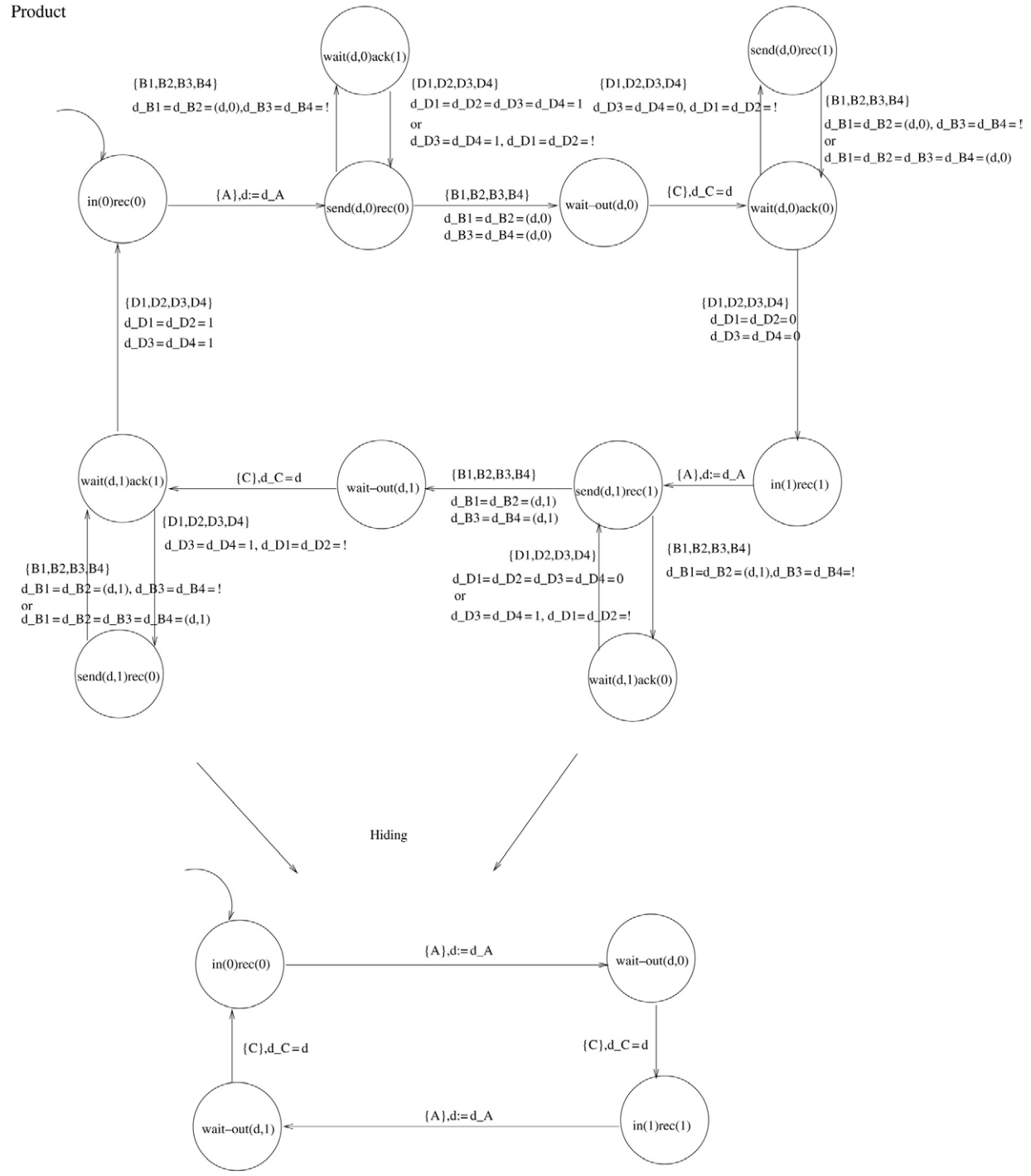


Fig. 23. ABP: product of automata.

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 $N \subseteq \mathcal{N}ames$. 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 5.6. State q_3 in Fig. 19 simulates states q_1 and q_2 in the same figure. Other examples include, in Fig. 7:

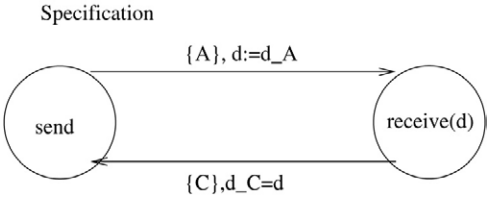


Fig. 24. ABP: specification of the protocol.

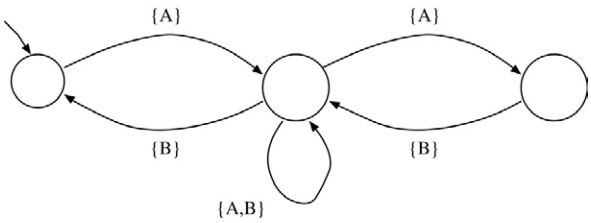


Fig. 25. Data-abstract constraint automaton for a 2-bounded FIFO channel.

- 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 1-bounded channel (Example 3.3), and
- the automaton for the synchronous channel which is simulated by the automaton for the lossy synchronous channel. □

As for ordinary transition systems, simulation is the key notion for any abstraction method. For instance, simulation covers *data abstraction* in a quite simple way. We will explain this by the example of the constraint automaton $\mathcal{A}_{FIFOn}$ for an n-bounded FIFO channel. Recall that $\mathcal{A}_{FIFOn}$ has a state-space whose size is exponential in n when the data domain *Data* contains two or more elements. When we abstract away from the data values, all states (configurations) where the buffer contains k elements (for some k with $0 \leq k \leq n$) can be collapsed into a single state. In this way, we obtain a constraint automaton that has $n+1$ reachable states and simulates the original constraint automaton $\mathcal{A}_{FIFOn}$. For instance, for $n = 2$, Fig. 25 shows the data-abstract constraint automaton for 2-bounded FIFO channels (with an arbitrary data domain).

Using analogous arguments as in the proof of Theorem 5.3, we obtain that the simulation preorder is strictly finer than language inclusion:

Theorem 5.7 (*Simulation Versus Language Inclusion*). *Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be two constraint automata with the same name-set $\mathcal{N}ames$.*

(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 such that $\mathcal{L}_{TDS}(\mathcal{A}_1, q) \neq \emptyset$ for all states q in $\mathcal{A}_1$, then*

$$\mathcal{A}_1 \preceq \mathcal{A}_2 \quad \textit{iff} \quad \mathcal{L}_{TDS}(\mathcal{A}_1) \subseteq \mathcal{L}_{TDS}(\mathcal{A}_2).$$

As for ordinary labeled transition systems, bisimulation equivalence is strictly finer than *simulation equivalence*, the kernel of the simulation preorder which identifies exactly those automata that simulate each other. Formally, $\mathcal{A}_1$ and $\mathcal{A}_2$ are simulation-equivalent iff $\mathcal{A}_1 \preceq \mathcal{A}_2$ and $\mathcal{A}_2 \preceq \mathcal{A}_1$. However, bisimulation equivalence and simulation equivalence agree for deterministic automata. Given deterministic constraint automata with non-empty TDS languages for all their states, the latter follows from the observation that simulation equivalence agrees with language equivalence (part (b) of Theorem 5.7) which, in turn, agrees with bisimulation equivalence (part (b) of Theorem 5.3). In the second part of Lemma 5.8, we provide the proof for the general case.

Lemma 5.8 (*Bisimulation Versus Simulation Equivalence*). (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, then*

$$\mathcal{A}_1 \preceq \mathcal{A}_2 \quad \textit{and} \quad \mathcal{A}_2 \preceq \mathcal{A}_1 \quad \textit{iff} \quad \mathcal{A}_1 \sim \mathcal{A}_2.$$

Proof. (a) follows from the fact that any bisimulation is a simulation. We prove (b) by showing that, for a given deterministic automaton $\mathcal{A}$, simulation equivalence is a bisimulation.

Let q_1, q_2 be two states with $q_1 \preceq q_2$ and $q_2 \preceq q_1$ and let N be a non-empty name-set and P be a simulation equivalence class. To show the logical equivalence of the data constraints $dc(q_1, N, P)$ and $dc(q_2, N, P)$, it suffices to prove that, for any transition

$$q_1 \xrightarrow{N,g} p_1$$

where $p_1 \in P$ and any data assignment δ with $\delta \models g$, there exists a transition

$$q_2 \xrightarrow{N,h} p_2$$

with $p_2 \in P$ and $\delta \models h$. (This argument shows that $dc(q_1, N, P) \leq dc(q_2, N, P)$. The symmetry yields the logical equivalence.)

Let $q_1 \xrightarrow{N,g} p_1$ be a transition with $p_1 \in P$ and δ a data assignment with $\delta \models g$. As $q_1 \preceq q_2$, we have

$$g \leq dc(q_1, N, p_1) \leq dc(q_2, N, p_1 \uparrow).$$

(Here, we write $p \uparrow$ for the set of states $\bar{p}$ with $p \preceq \bar{p}$.) Hence, $\delta \models dc(q_2, N, p_1 \uparrow)$. That is, there exists a transition

$$q_2 \xrightarrow{N,h} p_2$$

with $p_2 \in p_1 \uparrow$ and $\delta \models h$. We now use the fact that q_1 simulates q_2. Hence,

$$h \leq dc(q_2, N, p_2) \leq dc(q_1, N, p_2 \uparrow).$$

Thus, there exists a transition $q_1 \xrightarrow{N,\bar{g}} \bar{p}_1$ with $\delta \models \bar{g}$ and $p_2 \preceq \bar{p}_1$. The assumption that $\mathcal{A}$ is deterministic yields $\bar{g} = g$ and $\bar{p}_1 = p_1$. Hence,

$$p_1 \preceq p_2 \preceq \bar{p}_1 = p_1,$$

i.e., p_1 and p_2 belong to the same simulation equivalence class, namely P. ⊠

5.3. Compositionality

The following lemma provides a congruence result for bisimulation equivalence and the simulation preorder for the operators hiding and join (product). This result allows us to replace a "large" constraint automaton by a "small" bisimulation-equivalent automaton during the construction of constraint automaton with the help of join and hiding without affecting the accepted TDS language.

Lemma 5.9 (*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* $\exists C[\mathcal{A}_1] \sim \exists C[\mathcal{A}_2]$.

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

$$\mathcal{R}_{sim} = \{(\langle q_1, q_2\rangle, \langle q'_1, q'_2\rangle) : q_1 \preceq q'_1, q_2 \preceq q'_2\},$$
$$\mathcal{R}_{bis} = \{(\langle q_1, q_2\rangle, \langle q'_1, q'_2\rangle) : q_1 \sim q'_1, q_2 \sim q'_2\}.$$

Then, $\mathcal{R}_{sim}$ can be shown to be a simulation and $\mathcal{R}_{bis}$ a bisimulation on the product automata.

We now 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 $\mathcal{A} = (Q, \mathcal{N}ames, \longrightarrow, q_0)$, 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:

(*) $(q_1, q_2) \in \mathcal{R} \wedge q_1 \rightsquigarrow^* q'_1 \Longrightarrow q_2 \rightsquigarrow^* q'_2$ for some state q'_2 with $(q'_1, q'_2) \in \mathcal{R}$.

Let $(q_1, q_2) \in \mathcal{R}$, let N be a non-empty subset of $\mathcal{N}ames \setminus \{C\}$, and let P be 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. Equivalence and refinement checking

Problems like the question of whether two constraint automata have the same observable behavior or whether one's behavior is a refinement of the other one arise naturally and frequently. For instance:

- The replacement of a quite complex Reo circuit by a simpler one (e.g., which uses fewer and/or cheaper connectors) can be justified by showing that their induced constraint automata accept the same TDS language.
- Having a certain coordination mechanism in mind, it is often quite easy to depict a constraint automaton $\mathcal{A}$ that describes the allowed behavior (i.e., which rejects all timed data streams that should not occur). In this sense, $\mathcal{A}$ can serve as a specification for a Reo circuit that is to be designed. The correctness of a design can then be defined by language inclusion: a Reo circuit $\mathcal{G}$ is viewed to be correct (with respect to specification $\mathcal{A}$) iff all timed data streams that are accepted by the constraint automaton $\mathcal{A}_{\mathcal{G}}$ for $\mathcal{G}$ are also accepted by $\mathcal{A}$.

For ordinary labeled transition systems, checking language equivalence or language inclusion is computationally hard (PSPACE-complete in the case of labeled transition systems [16]), while checking bisimilarity or checking whether one system simulates another can be done in polynomial time [16,24,13]. For deterministic systems, the branching time relations (bisimulation equivalence, simulation preorder) coincide with the linear time relations (language equivalence, language inclusion); hence, any algorithm for the bisimulation (simulation) problem simultaneously also solves the language equivalence (inclusion) problem. For non-deterministic systems, the branching time relations are strictly finer than the language relations. However, the bisimulation/simulation algorithms can be used as correct, though incomplete, techniques to prove language equivalence or language inclusion.

In this section, we show that the situation for constraint automata is similar. In the sequel, let $\mathcal{A}_i = (Q_i, \mathcal{N}ames, \longrightarrow_i, Q_{0,i})$, $i = 1, 2$, be two constraint automata with the same set of TDS names. Throughout this section, the state-spaces Q_i, the data domain, and the transition relations are assumed to be finite. We now discuss the algorithmic aspects of the questions whether $\mathcal{A}_1$ and $\mathcal{A}_2$ are bisimilar, whether $\mathcal{A}_1$ is simulated by $\mathcal{A}_2$, whether the TDS language of $\mathcal{A}_1$ is contained in the TDS language of $\mathcal{A}_2$, and whether $\mathcal{A}_1$ and $\mathcal{A}_2$ are language-equivalent. For all these questions, standard algorithms for labeled transition systems (and finite automata) can be modified. (We briefly sketch the main ideas in Sections 6.1 and 6.2.) However, as we must deal with logical equivalence and implication, the algorithmic treatment of the branching time relations (bisimulation and simulation) is more difficult than for ordinary labeled transition systems where only the existence of transitions with certain target states is important.

Theorem 6.1 (*Complexity (Lower Bounds)*). *Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be two finite constraint automata with the same name-set $\mathcal{N}ames$.*

(a) *The problem of checking whether $\mathcal{A}_1 \sim \mathcal{A}_2$ is coNP-hard.*
(b) *The problem of checking whether $\mathcal{A}_1 \preceq \mathcal{A}_2$ is coNP-hard.*
(c) *The problem of checking whether $\mathcal{L}_{TDS}(\mathcal{A}_1) = \mathcal{L}_{TDS}(\mathcal{A}_2)$ is PSPACE-hard.*

Proof. (a) and (b) follow by a polynomial reduction from VALID (the validity problem for propositional logical formulae). Let f be a propositional logical formula with atoms $x_1, \ldots, x_n$. We now define two constraint automata $\mathcal{A}_1$ and $\mathcal{A}_2$ with the names $A_1, \ldots, A_n$ as follows. We use the Boolean data domain $Data = \{0, 1\}$ and identify the

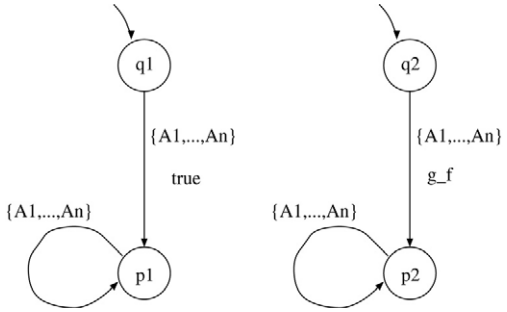


Fig. 26. CoNP constraint automata.

positive literal x_i with the atomic data constraint $d_{A_i} = 1$ and the negative literal $\neg x_i$ with the data constraint $d_{A_i} = 0$. Let g_f be the resulting data constraint, and consider the constraint automata $\mathcal{A}_1$ and $\mathcal{A}_2$ in Fig. 26.

We have: f is valid iff g_f is valid iff $\mathsf{true} \equiv g_f$ iff $\mathcal{A}_1 \sim \mathcal{A}_2$. Similarly, f is valid iff $\mathsf{true} \leq g_f$ iff $\mathcal{A}_1 \preceq \mathcal{A}_2$.

The proof of (c) follows by a polynomial reduction from the language equivalence problem for ordinary nondeterministic finite automata (NFA) where all states are accepting. This problem is known to be PSPACE-complete [16].

Let $\mathcal{M}$ be an NFA with the alphabet Σ and where all states are accepting. Let $\mathcal{L}(\mathcal{M})$ denote the accepted language of finite words over Σ, i.e., $\mathcal{L}(\mathcal{M})$ is the set of finite words $\sigma \in \Sigma^*$ that have a run in $\mathcal{M}$ starting in an initial state of $\mathcal{M}$. Similarly, we define $\mathcal{L}_\omega(\mathcal{M})$ to be the set of infinite words $\sigma \in \Sigma^\omega$ that have a run in $\mathcal{M}$ starting in an initial state of $\mathcal{M}$. As mentioned above, the problem of whether $\mathcal{L}(\mathcal{M}_1) = \mathcal{L}(\mathcal{M}_2)$ for NFAs (over the same alphabet) without non-accepting states is PSPACE-hard [16]. We now show that:

(i) the problem of whether $\mathcal{L}_\omega(\mathcal{M}_1) = \mathcal{L}_\omega(\mathcal{M}_2)$ for NFAs $\mathcal{M}_1$, $\mathcal{M}_2$ with the same alphabet is PSPACE-hard, by a polynomial reduction from the language equivalence problem for NFAs without non-accepting states; and
(ii) the problem of whether $\mathcal{L}_{TDS}(\mathcal{A}_1) = \mathcal{L}_{TDS}(\mathcal{A}_2)$ for constraint automata $\mathcal{A}_1$ and $\mathcal{A}_2$ with the same node-set is PSPACE-hard, by a polynomial reduction from (i).

Part (i). Given an NFA $\mathcal{M}$ where all states are accepting, we define $\hat{\mathcal{M}}$ as the NFA that results from $\mathcal{M}$ by adding a new state $\hat{q}$, a new input symbol δ, and transitions

$$q \xrightarrow{\delta} \hat{q}$$

for every state q in $\mathcal{M}$ and $q = \hat{q}$. Then, we have

$$\mathcal{L}(\mathcal{M}) \quad = \quad \big\{\sigma \in \Sigma^* : \sigma\delta^\omega \in \mathcal{L}_\omega(\hat{\mathcal{M}})\big\}, \quad \mathcal{L}_\omega(\hat{\mathcal{M}}) = \big\{\sigma\delta^\omega : \sigma \in \mathcal{L}(\mathcal{M})\big\}.$$

Hence, $\mathcal{L}(\mathcal{M}_1) = \mathcal{L}(\mathcal{M}_2)$ iff $\mathcal{L}_\omega(\hat{\mathcal{M}}_1) = \mathcal{L}_\omega(\hat{\mathcal{M}}_2)$.

Part (ii). Given two NFAs $\mathcal{M}_1$ and $\mathcal{M}_2$ over the same alphabet Σ, we construct two constraint automata $\mathcal{A}_1$ and $\mathcal{A}_2$ with a single name, say A, and the data domain $Data = \Sigma$ as follows. $\mathcal{A}_i$ arises from $\mathcal{M}_i$ by replacing every edge

$$q \xrightarrow{a} p \text{ in } \mathcal{M}_i \quad \text{by the edge } q \xrightarrow{\{A\}, d_A = a} p \text{ in } \mathcal{A}_i.$$

Then, we have $\mathcal{L}_{TDS}(\mathcal{A}_1) = \mathcal{L}_{TDS}(\mathcal{A}_2)$ iff $\mathcal{L}_\omega(\mathcal{M}_1) = \mathcal{L}_\omega(\mathcal{M}_2)$. ⊠

In the following two subsections, we sketch how standard algorithms for solving the bisimulation/simulation and language equivalence/inclusion problems in ordinary finite-state labeled transition systems can be modified to deal with constraint automata.

6.1. Checking bisimilarity and similarity

Essentially, we can use the well-known partitioning-splitter technique for ordinary labeled transition systems [16, 3,24,13,25,6].

For the comparison of two constraint automaton $\mathcal{A}_1$ and $\mathcal{A}_2$ via bisimulation equivalence or the simulation preorder, we first build the "large" constraint automaton $\mathcal{A} = \mathcal{A}_1 \uplus \mathcal{A}_2$ which arises through the disjoint union of the state

spaces of $\mathcal{A}_1$ and $\mathcal{A}_2$. (The initial states of $\mathcal{A}$ are irrelevant.) Then, we calculate the bisimulation equivalence classes $[q] = \{q' : q \sim q'\}$, or respectively, the simulator sets $q \uparrow = \{q' : q \preceq q'\}$ of $\mathcal{A}$. Finally, we check whether $\mathcal{A}_1 \sim \mathcal{A}_2$ or, respectively, $\mathcal{A}_1 \preceq \mathcal{A}_2$ by investigating the initial states of $\mathcal{A}_1$ and $\mathcal{A}_2$. Note that $\mathcal{A}_1 \sim \mathcal{A}_2$ iff, for any bisimulation equivalence class P in $\mathcal{A}$, we have either $(P \cap Q_{0,1} \neq \emptyset) \wedge (P \cap Q_{0,2} \neq \emptyset)$ or $(P \cap Q_{0,1} = \emptyset) \wedge (P \cap Q_{0,2} = \emptyset)$. Here, $Q_{0,i}$ denotes the set of initial states in $\mathcal{A}_i$. To check whether $\mathcal{A}_1$ is simulated by $\mathcal{A}_2$, we can use the observation that $\mathcal{A}_1 \preceq \mathcal{A}_2$ iff, for any initial state $q \in Q_{0,1}$ of $\mathcal{A}_1$, we have $q \uparrow \cap Q_{0,2} \neq \emptyset$.

6.1.1. Computing the bisimulation quotient

In the following, let $\mathcal{A} = (Q, \mathcal{N}ames, \longrightarrow, Q_0)$ be a constraint automaton. The idea of computing the bisimulation equivalence classes of $\mathcal{A}$ is to generate a sequence $\Pi_0, \Pi_1, \Pi_2, \ldots, \Pi_k$ of partitions of the state-space Q such that Π_i is strictly coarser than Π_{i+1} and finer than the bisimulation quotient $Q/\sim$. As we assume Q to be finite, we get $\Pi_k = Q/\sim$ for some $k \leq |Q|$.

Notation 6.2 (*Partition, (Super-)block, Splitter*). A partition for Q denotes a set $\Pi = \{P_1, \ldots, P_n\}$ of pairwise disjoint, non-empty subsets of Q such that $Q = P_1 \cup \ldots \cup P_n$. The elements of a partition are called blocks. By a super-block of Π, we mean any (non-empty) union of blocks in Π. A splitter for Π denotes a pair (N, P) consisting of a non-empty subset N of $\mathcal{N}ames$ and a super-block P for Π. □

Note that there is a one-to-one correspondence between partitions for Q and equivalences on Q. Given an equivalence $\mathcal{R}$, the quotient space $Q/\mathcal{R}$ is a partition. Vice versa, if Π is a partition, then $\mathcal{R}_\Pi = \{(q_1, q_2) : q_1, q_2 \text{ belong to the same block of } \Pi\}$ is an equivalence with $\Pi = Q/\mathcal{R}_\Pi$.

The initial partition identifies all states (i.e., $\Pi_0 = \{Q\}$). Given the partition Π_i, the next partition Π_{i+1} is obtained by refining Π_i according to a splitter (N, P) of Π_i, which means that we identify exactly those states of each block $B \in \Pi$ where the data constraints $dc(q, N, P)$ coincide up to logical equivalence.

Notation 6.3 (*Refine, Stability*). Let Π be a partition for Q, (N, P) a splitter for Π, and B be a block of Q. Then, we define

$$\mathsf{Refine}(B, N, C) = B/\equiv_{(N,P)}$$

where the equivalence $\equiv_{(N,P)}$ is defined such that $q_1 \equiv_{(N,P)} q_2$ iff $dc(q_1, N, P) \equiv dc(q_2, N, P)$. B is called *stable* with respect to (N, P) if $\mathsf{Refine}(B, N, P) = \{B\}$, i.e., if the data constraints $dc(q, N, P)$, $q \in P$, fall into the same logical equivalence class. We put

$$\mathsf{Refine}(\Pi, N, P) = \bigcup_{B \in \Pi} \mathsf{Refine}(B, N, P).$$

Π is called stable with respect to (N, P) if $\mathsf{Refine}(\Pi, N, P) = \Pi$. Π is called stable if Π is stable w.r.t. any splitter for Π. □

Note that $\mathsf{Refine}(B, N, P)$ is a partition for block B, while $\mathsf{Refine}(\Pi, N, P)$ is a partition for the whole state space Q which is finer than Π (i.e., any block B' of $\mathsf{Refine}(\Pi, N, P)$ can be written as a disjoint union of blocks in Π) and which is stable with respect to (N, P). For instance, refinement of $\Pi_0 = \{Q\}$ according to the splitter (N, Q) yields the partition $\Pi_1 = Q/\equiv_{(N,Q)}$, where $\equiv_{(N,Q)}$ is as in Notation 6.3.

The idea of the bisimulation algorithm (sketched in Algorithm 1) is to stabilize the current partition Π with respect to a splitter (N, P). (In Algorithm 1, we use the notations $dc(q, N)$ and $dc(N, P)$, which stand for $dc(q, N, Q)$ and $\bigvee_{q \in Q} dc(q, N, P)$, respectively.) The correctness of the algorithm follows from the following observations:

- The initial partition $\Pi_0 = \{Q\}$ is coarser than the bisimulation quotient $Q/\sim$.
- Whenever Π is coarser than $Q/\sim$, then $\mathsf{Refine}(\Pi, N, P)$ is coarser than $Q/\sim$ and finer than Π for any splitter (N, P) of Π.
- Π is stable iff the induced equivalence is a bisimulation. Hence, if Π is strictly coarser than $Q/\sim$, then there is a splitter (N, P) such that Π is strictly coarser than $\mathsf{Refine}(\Pi, N, P)$. Moreover, such a pair (N, P) is contained in $\mathsf{Splitters}$.
- Whenever Π is a stable partition that is coarser than $Q/\sim$, then $\Pi = Q/\sim$.

Hence, our algorithm generates a "decreasing" sequence of partitions $\Pi_0, \Pi_1, \Pi_2, \ldots$ that are all coarser than $Q/\sim$. As we assume Q to be finite, we get $\Pi_k = Q/\sim$ for some index k.

Algorithm 1 Partitioning splitter algorithm

$\Pi := Q$; Splitters $:= \big\{(N, Q) : N \subseteq \mathcal{N}ames, \bigvee_{q \in Q} dc(q, N) \not\equiv \mathsf{false}\big\}$;

WHILE Splitters $\neq \emptyset$ **DO**
  choose $(N, P) \in$ Splitters and remove (N, P) from Splitters;
  (* $\Pi :=$ Refine(Π, N, P) *)
  FOR ALL $B \in \Pi$ **DO**
    calculate the logical equivalence classes $D_1, \ldots, D_r$ of the data constraints $dc(q, N, P)$, $q \in B$;

    (* If $r = 1$ then B is stable w.r.t. (N, P) and Refine$(B, N, P) := \{B\}$. *)

    IF there is more than one logical equivalence class (i.e., if $r \geq 2$) **THEN**

      Refine$(B, N, P) := \{B_1, \ldots, B_r\}$, where $B_i = \{q \in Q : dc(q, N, P) \in D_i\}$;
      $\Pi := (\Pi \setminus \{B\}) \cup$ Refine(B, N, C);
      insert all pairs $(\tilde{N}, B_i)$ where $\emptyset \neq \tilde{N} \subseteq \mathcal{N}ames$ and $dc(\tilde{N}, B_i) \not\equiv \mathsf{false}$ into Splitters;
    FI
  OD
END WHILE
return Π (* Π is stable, and hence, $\Pi = Q/\sim$ *)

As for labeled transition systems, with appropriate data structures that support the choice and organization of the splitter candidates (and the blocks in the current partition), the schema sketched in Algorithm 1 can be implemented such that the number of iterations of the WHILE-loop is polynomial bounded in the size (number of states and transitions) of $\mathcal{A}$. More precisely, ignoring the cost to calculate the logical equivalence classes, the time complexity is bounded by $\mathcal{O}(|Q| \cdot | \longrightarrow |)$, as in the Kanellakis/Smolka algorithm [16].[9]

The critical part of Algorithm 1 is the calculation of the logical equivalence classes of data constraints. Recall that the problem of logical equivalence for propositional logical formulae is coNP-complete. A naïve possibility for calcaluting the logical equivalence classes is to consider all data assignments and the truth-values of the data constraints $dc(q, N, P)$, and then to identify exactly those states that yield the same truth-values for all data assignments. Of course, this method is extremely inefficient, because $|\mathcal{N}ames|^{|Data|}$ data assignments have to be considered, which is infeasible for large data domains.

More efficient is a *symbolic approach* based on variants of binary decision diagrams; see e.g. [5,22,12,9,20, 27]. Most appropriate for our purposes seems to be a multi-branching variant such as MDDs [15] that support the representation of functions $f : (V \to D) \to \{0, 1\}$, where V and D are finite. Note that the semantics of a data constraint can be viewed as a function of this type (where we put $V = \mathcal{N}ames$ and $D = Data \cup \{\perp\}$). A detailed description of such an MDD-based implementation of the partitioning splitter algorithm goes beyond the scope of this paper. Instead, we briefly sketch to which extend such decision diagrams support the calculation of the logical equivalence classes for data constraints.

As with other ordered decision diagrams, MDDs enjoy the property to provide *canonical representations*. As is standard for implementations of decision diagram algorithms (see, e.g., [23,9,20]), here we assume an implementation that supports the representation of several functions by nodes in a so-called *shared* decision diagram. Each node v of such a shared decision diagram can be identified with the subdiagram consisting of the nodes that are reachable from v. In this sense, any node v stands for a decision diagram (MDD in our case) and, thus, represents a function

[9] It seems to be hard to meet the bounded $\mathcal{O}(| \longrightarrow | \cdot \log |Q|)$ of the Paige Tarjan algorithm [24]. The reason is that, if $\tilde{P} \subseteq P$ and $dc(q_1, N, P) \equiv dc(q_2, N, P)$ and $dc(q_1, N, \tilde{P}) \equiv dc(q_2, N, \tilde{P})$, then we *cannot* conclude that $dc(q_1, N, P \setminus \tilde{P}) \equiv dc(q_2, N, P \setminus \tilde{P})$. Hence, in our setting, all new sub-blocks (rather than "all but one") must be considered as splitter candidates.

f_v. The canonicity yields that two functions that are represented by nodes v, w of a shared diagram agree iff $v = w$. In our case, this means that two data constraints $dc(q_1, N, P)$ and $dc(q_2, N, P)$ fall into the same logical equivalence class if and only if they are represented by the same name in a shared MDD. Thus, the equivalence-checking problem reduces to the comparison of nodes and, thus, can be performed in constant time. Hence, the calculation of the logical equivalence classes reduces to the construction of the MDD representations for the data constraints $dc(q, N, P)$ as nodes of a shared diagram. As the data constraints $dc(q, N, P)$ can be regarded as propositional formulas with the atoms "$d_A = d$", we may apply standard algorithms for the Boolean operators (such as conjunction, negation, and so on) to generate their MDD representations.

6.1.2. Calculating the simulator preorder

We can use essentially the same schema as for the computation of the simulator sets $q \uparrow = \{q' : q \preceq q\}$ in labeled transition systems. Algorithm 2 shows the main ideas which comprise the computation of a "decreasing" sequence of sets

$$Sim_0(q) \supseteq Sim_1(q) \supseteq \ldots \supseteq Sim_k(q) = q \uparrow .$$

Here, also, several improvements are possible, e.g., following the techniques suggested in [13]. Using appropriate data structures, the number of iterations can be contained within the bound of $O(poly(\mathsf{size}(\mathcal{A})))$. However, the major difficulty is the treatment of logical implication. As for bisimilarity checking, the use of variants of binary decision diagrams seems to be promising.

Algorithm 2 Schema to calculate the simulation preorder

FOR ALL state $q \in Q$ **DO**
  $Sim(q) := \{q' \in Q : dc(q, N) \leq dc(q', N)\}$
OD
Splitters $:= \{(N, p) \in 2^{\mathcal{N}ames} \times Q : \bigvee_{q \in Q} dc(q, N, p) \not\equiv \mathsf{false}\}$;

WHILE Splitters $\neq \emptyset$ **DO**
  choose a pair $(N, p) \in$ Splitters and remove (N, p) from Splitters;
  FOR ALL states $q \in Q$ with $dc(q, N, p) \not\equiv \mathsf{false}$ **DO**
    FOR ALL states $q' \in Sim(q)$ with $dc(q, N, p) \not\leq dc(q', N, Sim(p))$ **DO**
      $Sim(q) := Sim(q) \setminus \{q'\}$;
      Splitters :=
        Splitters $\cup \{(N', q') : N' \subseteq \mathcal{N}ames, q' \in Q, \bigvee_{r \in Q} dc(r, N', q') \not\equiv \mathsf{false}\}$
    OD
  OD
END WHILE
(* $Sim(q) = q \uparrow$ for all states q *)

6.2. Language equivalence checking

Given two bounded constraint automata $\mathcal{A}_1$ and $\mathcal{A}_2$ over a fixed set $\mathcal{N}ames$, the question of whether $\mathcal{A}_1$ and $\mathcal{A}_2$ are language-equivalent can be answered by checking language inclusion in both directions. To check whether $\mathcal{L}_{TDS}(\mathcal{A}_1) \subseteq \mathcal{L}_{TDS}(\mathcal{A}_2)$, we may apply the same techniques as for regular languages (and finite automata) using the observation that

$$\mathcal{L}_{TDS}(\mathcal{A}_1) \subseteq \mathcal{L}_{TDS}(\mathcal{A}_2) \quad \text{iff} \quad \mathcal{L}_{TDS}(\mathcal{A}_1) \cap \overline{\mathcal{L}_{TDS}(\mathcal{A}_2)} = \emptyset.$$

The main steps are as follows. First, we turn $\mathcal{A}_2$ into an equivalent deterministic constraint automaton $\mathsf{det}(\mathcal{A}_2)$ (see Remark 3.9). Then, we construct an automaton $\overline{\mathsf{det}(\mathcal{A}_2)}$ for its complement language, and build the product automaton $\mathcal{A}_1 \bowtie \overline{\mathsf{det}(\mathcal{A}_2)}$ (which represents the intersection language $\mathcal{L}_{TDS}(\mathcal{A}_1) \cap \overline{\mathcal{L}_{TDS}(\mathcal{A}_2)}$; see part (b) of Lemma 4.2). Finally, we check whether $\mathcal{L}_{TDS}(\mathcal{A}_1 \bowtie \overline{\mathsf{det}(\mathcal{A}_2)})$ is empty.

6.2.1. Complementing

For the construction of a constraint automaton for the complement language, we switch to a more general class of constraint automata with accepting states. For constraint automaton $\mathcal{A}$ augmented with a set F of accepting states, let $\mathcal{L}_{TDS}(\mathcal{A}, F)$ denote the language consisting of all TDS-tuples that have infinite runs in $\mathcal{A}$, each involving infinitely often visits to many states in F. (In other words, we use a variant of Büchi automata.) We now assume that we are given a deterministic constraint automaton $\mathcal{A} = (Q, \mathcal{N}ames, \longrightarrow, q_0)$ for which we aim to construct a constraint automaton with Büchi acceptance for the complement language of $\mathcal{A}$. We first extend the state space Q by a new state q_{accept} and add transitions

$$q \xrightarrow{N,g} q_{accept} \quad \text{if } g = \neg dc(q, N),\ q \in Q \text{ and } \emptyset \neq N \subseteq \mathcal{N}ames.$$

Moreover, we add transitions $q_{accept} \xrightarrow{N,\mathsf{true}} q_{accept}$ for each non-empty subset N of $\mathcal{N}ames$. Let $\overline{\mathcal{A}}$ be the resulting constraint automaton. Then,

$$\overline{\mathcal{L}_{TDS}(\mathcal{A})} = \mathcal{L}_{TDS}(\overline{\mathcal{A}}, \{q_{accept}\}).$$

6.2.2. Checking emptiness

For the language inclusion problem, we build the product $\tilde{\mathcal{A}} = \mathcal{A}_1 \bowtie \overline{\mathsf{det}(\mathcal{A}_2)}$ as in Definition 4.1, which we augment with the set $F = \{\langle q, q_{accept}\rangle : q \in Q_1\}$ of accepting states. (Q_1 denotes the state space of $\mathcal{A}_1$.) We need to explain how to check whether $\mathcal{L}_{TDS}(\tilde{\mathcal{A}}, F)$ is empty. For this, we first remove all transitions in $\tilde{\mathcal{A}}$ with an unsatisfiable data constraint. Then, we check, using standard graph algorithms, whether there is an initial state in $\tilde{\mathcal{A}}$ from which a cycle

$$\tilde{p}_0 \xrightarrow{N_1,g_1} \tilde{p}_1 \xrightarrow{N_2,g_2} \cdots \xrightarrow{N_r,g_r} \tilde{p}_r = \tilde{p}_0$$

is reachable such that $\{\tilde{p}_0, \tilde{p}_1, \ldots, \tilde{p}_r\} \cap F \neq \emptyset$ and $N_1 \cup \cdots \cup N_r = \mathcal{N}ames$.

Note that the requirement $\{\tilde{p}_0, \tilde{p}_1, \ldots, \tilde{p}_r\} \cap F \neq \emptyset$ is needed to ensure that the Büchi acceptance condition can be fulfilled. The requirement $N_1 \cup \cdots \cup N_r = \mathcal{N}ames$ is needed for the time-divergence condition for timed data streams.

The complexity of the language inclusion test is dominated by the construction of $\mathsf{det}(\mathcal{A}_2)$ (which is exponential in the size of $\mathcal{A}_2$) and the time needed to solve the satisfiability problem for the data constraints (which is NP-complete). The remaining steps (complementing, construction of the product, and checking emptiness) can be performed in time polynomial in $\mathsf{size}(\mathcal{A}_1)$ and $\mathsf{size}(\mathsf{det}(\mathcal{A}_2))$.

7. Concluding remarks

In this paper, we introduced constraint automata, defined operators for their composition, and presented notions of bisimulation equivalence and language equivalence as well as refinement relations (simulation and language inclusion). Constraint automata allow us to model subtle timing and input/output constraints of Reo connectors, specifically their combined mix of synchronous and asynchronous transitions. This is reflected in our definition of constraint automata and shown in our examples.

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 computer-aided design (CAD) systems. In this context, issues such as whether two Reo connectors R_1 and R_2 have the same observable behavior (in the sense that their induced TDS languages agree) or R_1 can be viewed as a refinement of R_2 (in the sense of TDS language containment) arise naturally and frequently. To treat such questions in an algorithmic way, our compositional semantics can serve as basis for an algorithm that automatically generates a constraint automaton $\mathcal{A}_R$ for a given Reo circuit R. To solve the language problems mentioned above (the questions of whether $\mathcal{L}_{TDS}(\mathcal{A}_{R_1}) = \mathcal{L}_{TDS}(\mathcal{A}_{R_2})$ or $\mathcal{L}_{TDS}(\mathcal{A}_{R_1}) \subseteq \mathcal{L}_{TDS}(\mathcal{A}_{R_2})$ for given Reo circuits R_1 and R_2), we suggest modifications to known methods for finite automata and labeled transition systems to deal with constraint automata.

Given finite, deterministic constraint automata $\mathcal{A}_1$ and $\mathcal{A}_2$, the simplest way to check language equivalence is based on the observation that language equivalence and bisimulation equivalence agree, provided that none of the states in

$\mathcal{A}_i$ accepts the empty TDS language (Theorem 5.3). Thus, we may first remove all states with an empty TDS language (cf. Section 6.2.2) and then check the bisimulation equivalence of the modified automata (cf. Section 6.1). Similarly, language inclusion for two finite, deterministic constraint automata $\mathcal{A}_1$ and $\mathcal{A}_2$ can be checked on the basis of a graph analysis, followed by an algorithm that calculates the simulation preorder.

Although the deterministic version of constraint automata is as expressive as the non-deterministic version, non-deterministic constraint automata offer a useful semantic model for Reo circuits which, e.g., avoids the exponential blowup that may result from applying the powerset construction to an automaton $\exists C[\mathcal{A}]$ (which can be non-deterministic even if $\mathcal{A}$ is deterministic). The algorithms for computing the bisimulation quotient or simulation preorder in non-deterministic constraint automata can be applied here as a sound (but incomplete) verification method to show language equivalence or inclusion.

In contrast to process algebras where notions of weak bisimulation (e.g., Milner's observational equivalence or congruence [21]) are used to abstract from non-observable computations, we use the hiding operator that modifies the given constraint automaton, similar to the deletion of ε-transitions in finite automata. Thus, in our context, there is no need for a notion of weak bisimulation.

In this paper, we restricted ourselves to using constraint automata in the context of the coordination language Reo. However, the use of constraint automata for an operational semantics model is not restricted to Reo. For instance, a recent work demonstrates the usefulness of constraint automata for specifications of software architectures in Alfa [19]. Alfa [18] is a framework for understanding and constructing style-based architectures from a small set of architectural primitives, based on a constructive and compositional framework for software architectures.

In our future activity, we will work out the details of a semantics that models Reo circuits by constraint automata with final states (e.g., to handle deadlocks), fairness to cover the meaning of Reo's fair merge semantics for sink and mixed nodes, and priorities (to deal with synchronous lossy channels) as mentioned in the end of Section 4. Other directions for our future work include the development of temporal logics and model checking algorithms based on constraint automata, optimization algorithms for Reo circuits, and the automated generation of Reo circuits from constraint automata specifications.

References

[1] F. Arbab, Reo: A channel-based coordination model for component composition, Mathematical Structures in Computer Science 14 (3) (2004) 1–38.
[2] F. Arbab, J.J.M.M. Rutten, A coinductive calculus of component connectors, in: D. Pattinson, M. Wirsing, R. Hennicker (Eds.), Recent Trends in Algebraic Development Techniques, Proceedings of 16th International Workshop on Algebraic Development Techniques, WADT 2002, in: Lecture Notes in Computer Science, vol. 2755, Springer-Verlag, 2003, pp. 35–56. http://www.cwi.nl/ftp/CWIreports/SEN/SEN-R0216.pdf.
[3] T. Bolognesi, S.A. Smolka, Fundamental results for the verification of observational equivalence: A survey, in: H. Rudin, C.H. West (Eds.), Protocol Specification, Testing, and Verification, VII, North-Holland, Zürich, Switzerland, 1987, pp. 165–178.
[4] S.D. Brookes, C.A.R. Hoare, A.W. Roscoe, A theory of communicating sequential processes, Journal of the ACM 31 (3) (1984) 560–599.
[5] R. Bryant, Graph-based algorithms for Boolean function manipulation, IEEE Transactions on Computers C-35 (1986).
[6] D. Bustan, O. Grumberg, Simulation-based minimization, ACM Transactions on Computational Logic 4 (2) (2003) 181–206.
[7] B.A. Davey, H.A. Priestley, Introduction to Lattices and Order, Cambridge University Press, 1990.
[8] L. de Alfaro, T.A. Henzinger, Interface automata, in: V. Gruhn (Ed.), Proceedings of the Joint 8th European Software Engeneering Conference and 9th ACM SIGSOFT Symposium on the Foundation of Software Engeneering, ESEC/FSE-01, 10–14 September, in: Software Engineering Notes, vol. 26, 5, ACM Press, New York, 2001, pp. 109–120.
[9] R. Drechsler, B. Becker, Binary Decision Diagrams: Theory and Implementation, Kluwer Academic Press, 1998.
[10] W. Fokkink, Introduction to Process Algebra, in: Texts in Theoretical Computer Science, An EATCS Series, Springer-Verlag, 1999.
[11] R. Grosu, B. Rumpe, Concurrent timed port automata, Technical Report TUM-I9533, Techn. Universität München, 1995. http://www4.informatik.tu-muenchen.de/reports/TUM-I9533.html.
[12] G.D. Hachtel, F. Somenzi, Logic Synthesis and Verification Algorithms, Kluwer Academic Publishers, Boston, 1996.
[13] M. Henzinger, T. Henzinger, P. Kopke, Computing simulations on finite and infinite graphs, in: Proc. FOCS'95, 1995 pp. 453–462.
[14] J.E. Hopcroft, R. Motwani, J.D. Ullman, Introduction to Automata Theory, Language, and Computation, 2nd ed., Addison-Wesley, 2001.
[15] T. Kam, T. Villa, R.K. Brayton, A. Sangiovanni-Vincentelli, Multi-valued decision diagrams: Theory and applications, Multiple-Valued Logic 4 (1–2) (1998) 9–62.
[16] P. Kannelakis, S. Smolka, CCS expressions, finite state processes and three problems of equivalence, in: Proc. 2nd ACM Symposium on the Principles of Distributed Computing 1983, pp. 228–240, Information and Computation 86 (1990) 43–68.
[17] N. Lynch, M.R. Tuttle, An introduction to input/output automata, CWI Quarterly 2 (3) (1989) 219–246.
[18] N.R. Mehta, N. Medvidovic, Composing architectural styles from architectural primitives, in: Proceedings of the 9th European Software Engineering Conference and 11th ACM SIGSOFT Symposium on Foundations of Software Engineering, ESEC/FSE, ACM Press, 2003, pp. 347–350.

[19] N.R. Mehta, M. Sirjani, F. Arbab, Effective modeling of software architectural assemblies using Constraint Automata, Technical Report SEN-R0309, Centrum voor Wiskunde en Informatica, Kruislaan 413, 1098 SJ Amsterdam, The Netherlands, October 2003. http://www.cwi.nl/ftp/CWIreports/SEN/SEN-R0309.pdf.
[20] C. Meinel, T. Theobald, Algorithms and Data Structures in VLSI Design, Springer-Verlag, 1998.
[21] R. Milner, Communication and Concurrency, in: Prentice Hall International Series in Computer Science, Prentice Hall, 1989.
[22] S. Minato, Binary Decision Diagrams and Apllications for VLSI Design, Kluwer Academic Press, 1996.
[23] S. Minato, N. Ishiura, S. Yajima, Shared binary decision diagram with attributed edges for efficient Boolean function manipulation, in: Proceedings of the 27th ACM/IEEE Design Automation Conference, DAC, 1990, pp. 52–57.
[24] R. Paige, R.E. Tarjan, Three partition refinement algorithms, SIAM Journal on Computing 16 (6) (1987) 973–989.
[25] L. Tan, R. Cleaveland, Simulation revisited, Lecture Notes in Computer Science 2031 (2001) 480–495.
[26] W. Thomas, Automata on infinite objects, in: J. van Leeuwen (Ed.), Handbook of Theoretical Computer Science, Elsevier Science Publishers B.V., 1990, pp. 133–191 (Chapter 4).
[27] I. Wegener, Branching Programs and Binary Decision Diagrams. Theory and applications, in: Mongraphs on Disrete Mathematics and Applications, SIAM, 2000.