Synthesis of Reo Circuits For Implementation of Component-Connector Automata Specifications

Farhad Arbab[a,c], Christel Baier[b], Frank de Boer[a,c], Jan Rutten[a,d], Marjan Sirjani[e]

[a] *CWI, Amsterdam, The Netherlands*
{farhad, janr, frb}@cwi.nl
[b] *Universität Bonn, Institut für Informatik I, Germany*
baier@cs.uni-bonn.de
[c] *Universiteit Leiden, The Netherlands*
[d] *Vrije Universiteit, Amsterdam, The Netherlands*
[e] *Sharif University of Technology, Tehran, Iran*

Abstract. Composition of a concurrent system out of components involves coordination of their mutual interactions. In component-based construction, this coordination becomes the responsibility of the glue-code language and its underlying run-time middle-ware. Reo offers an expressive glue-language for construction of coordinating component connectors out of primitive channels. In this paper we consider the problem of synthesizing Reo coordination code from a specification of a behavior as a relation on scheduled-data streams. The specification is given as a constraint automaton that describes the desired input/output behavior at the ports of the components. The main contribution in this paper is an algorithm that generates Reo code from a given constraint automaton.

1 Introduction

Composing components into a concurrent system involves coordination of their mutual interactions. The internals of black-box components cannot be modified to implement such coordinated interactions. Coordination, therefore, becomes the responsibility of the "glue-code" that inter-connects the constituent components of a composite system, and of its underlying run-time middle-ware. Reo [2] offers a powerful glue language for implementation of coordinating component connectors that resemble electronic circuits and are based on a calculus of mobile channels. Reo is being used, for instance, in the context of the Cybernetic Incident Management project [9] for composition of web services, which constitute the black-box components of dynamically configured distributed applications [11]; to model business processes, such as electronic auctions [20]; and for modeling coordination in biological systems [10].

This paper addresses the *synthesis problem of component connectors* with Reo as our target implementation language. The input for this problem is a specification of a coordination protocol and its output is a Reo connector circuit that implements this protocol. Synthesis problems address the issue of the (algorithmic) generation of an implementation from a given specification and have a long tradition in computer science. In the context of switching circuits, the synthesis problem was first raised by Church [8] and is nowadays well-understood. For temporal logical specifications, several synthesis algorithms have been suggested that rely on the close relationship between the synthesis and satisfiability problem or on a game-theoretic view, see e.g. [12, 15, 6, 16, 19, 18, 13]. The output of these synthesis algorithms are some kind of automata or state-transition graphs. Our goal is a step further toward an implementation by generating Reo code from a given automaton specification. Thus, our contribution is more in the spirit of gate-level hardware synthesis from given automata specifications. Our starting point is a specification of a component connector as a relation over *timed data streams* [7, 5], represented by a *constraint automaton* [4]. Constraint automata are variants of labeled transition systems that operationally describe the maximally parallel data-flow activity through the nodes in a Reo circuit. In [4], constraint automata are used to provide an operational semantics for coordination mechanisms formalized by composition of Reo connector graphs. In a constraint automaton, the states of the automaton represent the possible configurations (e.g., the contents of the FIFO-channels of the Reo-connector); transitions going out of a state represent data-flow at that state and its effect on the configuration.

In this paper we are not primarily concerned with the derivation of (constraint) automata representations from higher-level behavior specifications, such as in temporal logic or relations on timed data streams. Similar derivations, for instance, in the field of digital circuit design, are well-known. The main contribution of this paper is an algorithm that takes as input a constraint automaton $\mathcal{A}$ and produces a Reo connector graph that implements the relation on timed data streams specified by $\mathcal{A}$. This is tantamount to compiling an automaton down to actual concurrent executable code for a distributed implementation of the coordination behavior specified by that automaton. Superficially, compiling constraint automata specifications to Reo circuits seems simple. By analogy, derivation of digital circuits from Mealy automata specifications are well understood. However, constraint automata (and Reo circuits) can exhibit far more complex behavior than digital circuits, including combinations of synchrony and asynchrony, and relational, as well as simple (input/output) functional, interdependencies. In the light of this fact, it is far from obvious if synthesis of Reo circuits from constraint automata is possible at all, and if so, whether it can be done efficiently.

The rough idea of our synthesis algorithm is as follows. We first transform the automaton $\mathcal{A}$ into an equivalent *scheduled-data expression* which is a slight variant of an ordinary ω-regular expression. We then construct circuits for the atomic expressions and composition operators on Reo circuits that capture the semantics of concatenation, union, and infinity-closures. The major difficulty is the treatment of the atomic expressions that describe a complex “one-step” coordination scenario with possibly data-dependent synchronous and asynchronous behavior.

The rest of this paper is organized as follows. Section 2 contains a summary of the main features of Reo. Section 3 recalls the definition of constraint automata and their accepted TDS-languages. In Section 4, we show the equivalence of scheduled-data expressions and constraint automata. The construction of a Reo circuit from a given expression is explained in Section 5. Section 6 concludes the paper.

2 A Reo Primer

Reo [2] 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. Components can instantiate, compose, connect to, and perform I/O operations through connectors. Here, as in [5, 4], we do not consider the dynamic creation, composition, and reconfiguration of connectors by components. We restrict our attention to connectors that have a static graphical representation as a *Reo circuit* which coordinates the data-flow through the channels connecting the input/output ports of components.

Reo's notion of *channel* is far more general than its common interpretation and allows for any primitive communication medium with exactly two ends. The channel ends are classified as *source* ends through which data enters and *sink* ends through which data leaves a channel. Although Reo allows for an open-ended set of channel-types with user-defined semantics, for our purposes in this paper, we restrict ourselves to the channel-types shown in Fig. 1.

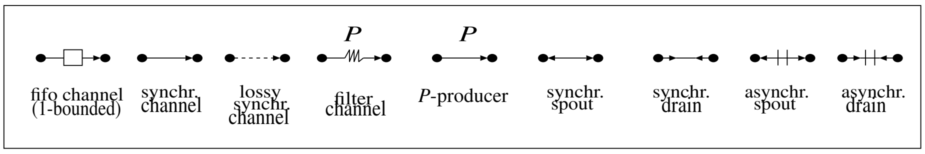


Fig. 1. Basic channel-types in Reo

The simplest form of an asynchronous channel is a *FIFO channel* with one buffer cell (called a 1-bounded FIFO channel or simply a FIFO1 channel). We graphically represent a FIFO1 channel by a small box in the middle of an arrow. In the example in Fig. 1, the left channel-end is a source, and the right end is a sink. The buffer is assumed to be initially empty if no data item is shown in the box (this is the case in Fig. 1). The graphical representation of a FIFO1-channel whose buffer initially contains a data element *d* shows *d* inside the box. FIFO channels with two or more buffer cells can be produced by composing several FIFO1 channels, as for instance, explained in [5, 4].

A *synchronous channel* (depicted as a simple solid arrow) 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 channel* (depicted as a dashed arrow) is similar to a synchronous channel, except that it always accepts all data items through its source end. If it is possible for it to simultaneously dispense the data item through its sink (e.g., there is a take operation pending on its sink) the channel transfers the data item; otherwise the data item is lost. For a *synchronous filter channel*, its "pattern" P (for our purposes here, formalized as a set $P \subseteq Data$) specifies the type of data items that can be transmitted through the channel. Any value $d \in P$ is accepted through its source end iff its sink end can simultaneously dispense d; all data items $d \notin P$ are always accepted through the source end but are immediately lost. The P-producer is a variant of a synchronous channel whose source end accepts any data item $d \in Data$, but the value dispensed through its sink end is always a data element $d \in P$.

More exotic channels permitted in Reo are (a)synchronous *drains* that have two source ends. Because a drain has no sink end, no data value can ever be obtained from these channels. Thus, a synchronous drain accepts a data item through one of its ends iff a data item is also available for it to simultaneously accept through its other end as well. All data accepted by this channel are lost. An asynchronous drain accepts and loses data items through its two source ends, but never simultaneously. Synchronous and asynchronous *spouts* are duals of their corresponding drain channel types, as they have two sink ends.

A complex connector has a graphical representation, called a *Reo circuit*, which can be produced by applying certain composition operators to channels. In our setting, where we do not consider dynamic aspects of the Reo language, a Reo-circuit is a finite graph where the *nodes* are labeled with pair-wise disjoint, non-empty sets of channel ends, and where the edges represent their connecting channels. 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 is called a *source node* if $\mathsf{Src}(A) \neq \emptyset \wedge \mathsf{Snk}(A) = \emptyset$. Analogously, A is called a *sink node* if $\mathsf{Src}(A) = \emptyset \wedge \mathsf{Snk}(A) \neq \emptyset$. Node A is called a *mixed node* if $\mathsf{Src}(A) \neq \emptyset \wedge \mathsf{Snk}(A) \neq \emptyset$. In this paper, it suffices to assume that all mixed nodes are hidden. In other words, we abstract away from their names and formalize the behavior of a Reo circuit by means of the data-flow at its sink and source nodes. Intuitively, source nodes of a circuit are analogous to the input ports, and sink nodes to the output ports of a component, while mixed nodes are its hidden internal details. Components cannot connect to, read from, or write to mixed nodes. Instead, data-flow through mixed nodes is totally specified by the circuits they belong to.

A component can write data items to a source node of a Reo circuit that it is connected to. A 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 of a Reo circuit that it is connected to through input operations.[1] 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 nondeterministically. A sink node, thus, acts as a nondeterministic *merger*. A mixed node is a self-contained "pumping station" that combines the behavior of a sink node (merger) and a source node (replicator) in an atomic iteration of an endless loop: in every iteration a mixed node nondeterministically selects and takes a suitable data item offered by one of its coincident sink channel ends and replicates it into all of its coincident source 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.

Example 1 (Exclusive Router and Shift-Lossy FIFO1 Channel). Fig. 2 a. shows an implementation of an exclusive router built by composing five synchronous channels, two lossy synchronous channels and a synchronous drain. The intuitive behavior of this circuit is that through its source node A, it obtains a data item d from its environment

[1] We consider only the destructive take operation here which, e.g., on a FIFO channel, reads and removes the first data item in its buffer.

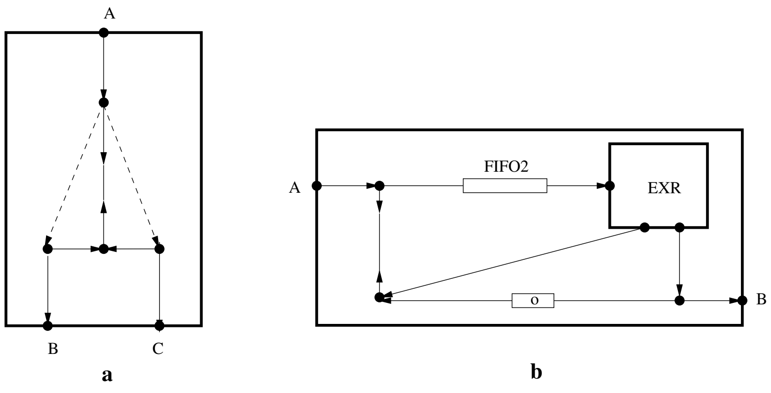


Fig. 2. Exclusive router and shift-lossy FIFO1 channel

and delivers *d* to one of its sink nodes *B* or *C*. If both *B* and *C* are willing to accept *d* then (the merger in the mixed node in the middle of) the exclusive router nondeterministically decides to deliver *d* to either *B* or *C*. No data that passes through *A* can be lost because of the synchronous drain and the two synchronous channels in the middle of the circuit. The synchronous drain ensures that data flow at *A* is synchronized with data flow through the node at its opposite end. The merger inherent in this mixed node guarantees that at most one of its two coincident synchronous channels transfer data, synchronized with the data flow at either *B* or *C*.
The circuit in Fig. 2.b shows an implementation of a shift-lossy FIFO1 channel with source node *A* and sink node *B*. This implementation uses four synchronous channels, a synchronous drain, a FIFO1 channel whose buffer initially contains a token data item, *o*, an empty FIFO2 channel, and an instance of the exclusive router of Fig. 2.a shown as the box labeled EXR. A shift-lossy FIFO1 channel behaves the same as a FIFO1 channel, except that writing to its source end is never blocked. If at the time of a write operation its buffer is full, the stored data item in the buffer is lost and the new data item replaces it in the buffer. The observable behavior of each of these Reo circuits is represented by a constraint automaton in Fig. 3. Derivation of these constraint automata as compositions of the constraint automata representing the behavior of the individual primitives used in their respective Reo circuits appears in [4]. □

In spite of its simplicity, the semantics of Reo is indeed very rich, yielding a surprisingly expressive language [2]. For instance, the relational (as opposed to functional) dependencies that result in "propagation of synchrony" as well as the way in which the local behavior of, e.g., lossy synchronous channels imposes non-local constraints on a circuit, are already evident in the exclusive router of Fig. 2.a. (We use this exclusive router later in this paper in our synthesis of Reo circuits.) Examples of Reo circuits with more interesting behavior can be found elsewhere [1], and the reader is encouraged to see [17] and [5] for the simple, rich, and expressive formal semantics of Reo.
In the remainder of the paper, we discuss the synthesis problem of Reo circuits where the input specification of the desired coordination is given as a *constraint automaton*, as defined in the next section.

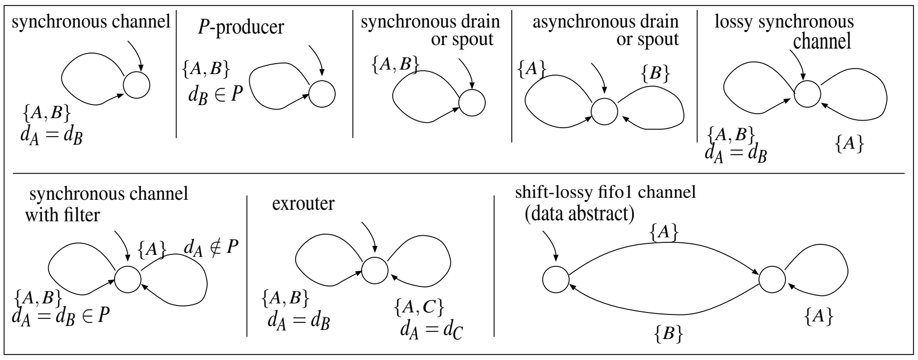
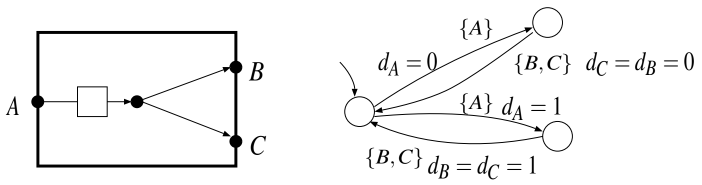


Fig. 3. Constraint automata for some basic channels in Reo

3 Constraint Automata

Constraint automata can serve as an operational model for Reo circuits [4]. The states of an automaton represent the configurations of its corresponding circuit (e.g., the contents of the FIFO channels), while the transitions encode its maximally-parallel stepwise behavior. The transitions are labeled with the maximal sets of nodes on which data-flow occurs simultaneously, and a data constraint (i.e., boolean condition for the observed data values). We start with a simple example for a constraint automaton that models a component with input port A and two output ports B and C which is modeled by a Reo circuit as shown in the left of the picture below.



The picture on the right shows the corresponding constraint automaton where we assume that only bits 0 and 1 can be transmitted through the channels. The initial state stands for the configuration where the buffer is empty, while the two other states represent the configurations where the buffer is filled with one of the data items. The outgoing transitions from the initial state are labeled with the singleton set $\{A\}$ which reflects the fact that in the initial configuration only data-flow at A is possible. If the buffer is filled then data-flow at A is impossible and only B and C can take the value from the buffer.

In the sequel, we specify constraint automata using a nonempty and finite set *Data* consisting of data items that can be sent (and received) via channels and a nonempty and finite set $\mathcal{N} = \{A_1, \ldots, A_n\}$ of names. Intuitively, we may think of the A_i's to be the source or sink nodes of a Reo circuit. We refer to the subsets of $\mathcal{N}$ as node-sets. A data assignment for $\emptyset \neq N \subseteq \mathcal{N}$ is a function $\delta : N \longrightarrow Data$. $DA(N)$ denotes the set of all data assignments for N, and DA the set of all data assignments (on any N). Data *constraints*, which can be viewed as a symbolic representation of *sets* of data assignments, are formally defined as propositional formulas built from the atoms "$d_A \in P$"

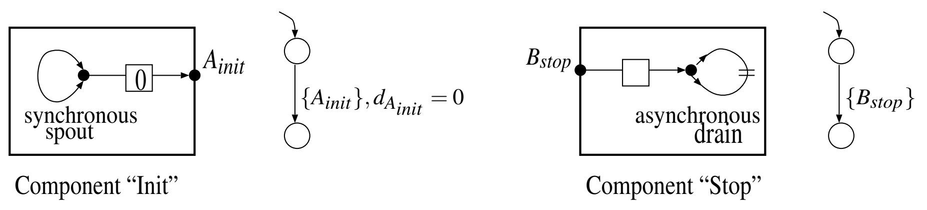


Fig. 4. Reo circuits and automata for an initializer and a terminator

and "$d_A = d_B$", where $A, B \in \mathcal{N}$, $d_A, d_B \in Data$, and $P \subseteq Data$. $DC(N)$ denotes the set of data constraints using only names from N, and DC is a shorthand for $DC(\mathcal{N})$. We simply write "$d_A = d$" rather than "$d_A \in \{d\}$". The symbol $\models$ stands for the obvious satisfaction relation which results from interpreting data constraints over data assignments. Satisfiability and logical equivalence $\equiv$ of data constraints are defined as usual.

Definition 1 (Constraint automata, [4]). A constraint automaton (over *Data*) is a tuple $\mathcal{A} = (Q, \mathcal{N}, \longrightarrow, Q_0)$ where Q is a finite set of states, $\mathcal{N}$ a finite set of nodes, $\longrightarrow$ is a finite subset of $Q \times (2^{\mathcal{N}} \times DC) \times Q$, called the transition relation, and $Q_0 \subseteq Q$ a nonempty set of initial states. We write $q \xrightarrow{N,g} p$ instead of $(q, N, g, p) \in \longrightarrow$ and require that (1) $N \neq \emptyset$ and (2) $g \in DC(N)$ is satisfiable. We call N the node-set and g the guard of the transition. States without any outgoing transition are called terminal. □

The intuitive meaning of a constraint automaton as an operational model for Reo connectors is similar to the interpretation of labeled transition systems as formal models for reactive systems. The sink and source nodes of a Reo connector circuit play the role of the nodes in its corresponding constraint automaton. The states represent the configurations of the connector. The meaning of a transition $q \xrightarrow{N,g} p$ is that in configuration q all the nodes $A_i \in N$ perform (synchronously) I/O-operations that meet the guard g, resulting in a new configuration p, while at the same moment there is no data-flow at the other nodes $A_i \in \mathcal{N} \setminus N$.

Example 2 (Constraint automata). Constraint automata for the various basic channels types, the exclusive router and shift-lossy FIFO1 channel are shown in Figure 3 (where valid guards have been omitted). The automaton for a FIFO1 channel with source A and sink B is the same as the one for the example in the beginning of the section, except that C has to be removed. These automata do not have terminal states as in any configuration data flow at some nodes is possible. The left part of Fig. 4 shows the Reo circuit for an initializer, i.e., a component without input ports (source nodes) and a single output port A_{init} where data-flow at A_{init} happens exactly once.[2] Thus, if we connect A_{init} with an input port A of another component C via a synchronous channel with source A_{init} and sink A then data-flow at A_{init} activates the data-flow at

[2] Data-flow at the node on the left, where the two sink ends of a synchronous spout coincide, is never possible because on the one hand, the sink ends of the spout are obligated to produce their respective data items simultaneously, while on the other hand the merge semantics of sink/mixed nodes does not allows for simultaneous data-flow at both sink ends.

$\mathcal{C}$ but prevents any “restart” of $\mathcal{C}$. The situation is similar for the component “Stop” on the right of the picture where the source node B_{stop} can put a value into the buffer exactly once, because afterward the buffer is filled forever as no data-flow is possible for an asynchronous drain with both source ends coincident on the same node. Thus, if an output port B of a component $\mathcal{C}$ is connected via a synchronous channel with B_{stop} then output at B is possible exactly once. In this sense, component “Stop” can serve to terminate data-flow in other components. □

In [4], we formalized the semantics of a constraint automaton as a relation on timed data streams. For the purposes of this paper, an equivalent, but simpler concept suffices which abstracts away from time and describes the “traces” of a constraint automaton by *scheduled-data streams*: finite or infinite sequences of pairs $\langle N, \delta \rangle$, consisting of a set N of all the nodes that are scheduled to be synchronously (i.e., atomically) active in the next step, together with a data assignment $\delta \in DA(N)$ describing the data values that are input and output.

Definition 2 (Scheduled-data streams, generated language). A scheduled-data stream $\Theta = \Theta(0);\Theta(1);\ldots$ is a finite or infinite sequence of pairs $\Theta(i) \in 2^{\mathcal{N}} \times DC$, denoted by

$$\Theta(i) = \langle \underbrace{\Theta.N(i)}_{\text{node-set}}, \underbrace{\Theta.\delta(i)}_{\text{data assignment}} \rangle,$$

such that $\Theta.N(i)$ is a non-empty node-set and $\Theta.\delta(i)$ a data assignment for $\Theta.N(i)$. We write $|\Theta|$ to denote the length of Θ (which can be ω). The empty scheduled-data stream is denoted by ε. $SDS_{\mathcal{N}}$ or briefly SDS denotes the set of all scheduled-data streams.
Let $\mathcal{A} = (Q, \mathcal{N}, \longrightarrow, Q_0)$ be a constraint automaton, $\Theta \in SDS$ and q a state in $\mathcal{A}$. A q-run for Θ in $\mathcal{A}$ is a path in $\mathcal{A}$

$$\mathbf{q} = q_0 \xrightarrow{N_0, g_0} q_1 \xrightarrow{N_1, g_1} q_2 \xrightarrow{N_2, g_2} \ldots$$

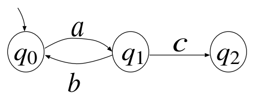
such that (1) $q_0 = q$ and (2) either $\mathbf{q}$ and Θ are infinite or $\mathbf{q}$ consists of $|\Theta|$ transitions and ends in a terminal state and (3) $N_i = \Theta.N(i)$, $\Theta.\delta(i) \models g_i$ for all $0 \leq i < |\Theta|$. The generated language $\mathcal{L}(\mathcal{A})$ of $\mathcal{A}$ is the set of all scheduled-data streams $\Theta \in SDS$ which have a q_0-run in $\mathcal{A}$ for some initial state $q_0 \in Q_0$. □

For instance, the SDS-language generated by the automaton for a synchronous channel consists of all infinite scheduled-data streams Θ with $\Theta.N(i) = \{A, B\}$ and where data assignment $\Theta.\delta(i)$ assigns the same data item to A and B.
Although the formal definition of scheduled-data streams does not impose a relation between the data assignments $\Theta.\delta(i)$, for a given constraint automaton, there can be a link between the data constraints $\Theta.\delta(i)$ and $\Theta.\delta(i+1)$. For instance, the automaton for a FIFO1 channel with source node A and sink node B generates the SDS-language consisting of all infinite scheduled-data streams Θ with $\Theta.N(2i) = \{A\}$, $\Theta.N(2i+1) = \{B\}$, and with $\Theta.\delta(2i) = [A \mapsto d]$ and $\Theta.\delta(2i+1) = [B \mapsto d]$, for some $d \in Data$.

In [4], we explain how an automaton for a Reo circuit can be constructed in a compositional way. (For the purpose of this paper, the details of that construction do not matter. The only thing that we use later, in Section 5, is that by applying the above definition to the automaton for a Reo circuit R, we obtain an SDS-language $\mathcal{L}(R)$ for R.) In what follows we show, conversely, how to construct a Reo circuit from a constraint automaton.

4 Scheduled-Data Expressions

The first step of our construction of a Reo circuit from a given automaton is to transform the automaton into an equivalent ω-regular expression, a so-called *scheduled-data expression*. These are built by ε representing the singleton SDS-language $\{\varepsilon\}$ and the atoms $\langle N, g\rangle$ where $\emptyset \neq N \subseteq \mathcal{N}$ and g is a satisfiable data constraint for N. The SDS-language $\mathcal{L}(\langle N, g\rangle)$ consists of all scheduled-data streams Θ of length 1 such that $\Theta.N(0) = N$ and $\Theta.\delta(0) \models g$. Moreover, we use the standard composition operators ; (concatenation), $\cup$ (union) and the closure operators α^ω (infinitely many repetitions) and α^∞ (finite or infinite repetitions). The formal definition of $\mathcal{L}(\alpha)$ for composite expressions is defined as for ordinary ω-regular expressions and is omitted here.
Similar to the construction of a finite automaton from ordinary regular expressions (see e.g. [14]), we can assign a constraint automaton to any scheduled-data expression that generates the same SDS-language and which is linear in the size of the expression. Since this construction does not play a role in the present paper, its description is omitted. Instead, we use the reverse construction, i.e., of a scheduled-data expression for a constraint automaton. Although to do so, we may apply the standard algorithms for generating (ω-)regular expressions from automata (see e.g. [14]), we suggest here an alternative algorithm. Rather than describing the construction in general, we treat a typical example. Consider the constraint automaton as shown on the left of the following picture where a, b, c are pairs of node-sets with corresponding data constraints.



$$\begin{aligned} \alpha_0 &= a;\alpha_1 \\ \alpha_1 &= (b;\alpha_0) \cup (c;\alpha_2) \\ \alpha_2 &= \varepsilon \end{aligned}$$

Let α_i denote the scheduled-data expression corresponding to (the SDS-language generated by) state q_i, for $i = 0, 1, 2$. The three transitions of this automaton give rise to three equations for the expressions as shown above. Together, they imply the following equation: $\alpha_0 = (a;b;\alpha_0) \cup (a;c)$. This equation can be solved, using the following general laws for scheduled-data expressions: "if $\alpha = (\beta;\alpha) \cup \gamma$ and $\varepsilon \notin \beta$ then $\alpha = \beta^\infty;\gamma$" and "if $\alpha = \beta;\alpha$ and $\varepsilon \notin \beta$ then $\alpha = \beta^\omega$". Applying the first law to the equation above yields the expression $\alpha_0 = (a;b)^\infty;a;c$ for the state q_0.

5 From Scheduled-Data Expressions to Reo

We now address the issue of constructing a Reo circuit for a scheduled-data expression α_0. Because the source and the sink nodes of a Reo circuit play different roles with respect to its environment, and this distinction is abstracted away in scheduled-data expressions (and constraint automata), we first need to identify the "input" and "output" of a circuit by partitioning its node set $\mathcal{N}$. That is, our starting point is a description of a component connector by its input ports $C_1, \ldots, C_n$ and its output ports $D_1, \ldots, D_m$ and by (the scheduled-data expression α_0 of) a given constraint automaton that specifies the observable data flow at the C_i's and D_j's.

In the sequel, let $\mathcal{N} = \{C_1, \ldots, C_n\} \cup \{D_1, \ldots, D_m\}$ contain all nodes occurring in the node-sets N of the atoms $\langle N, g\rangle$ in α_0, where we assume that the C_i's are source

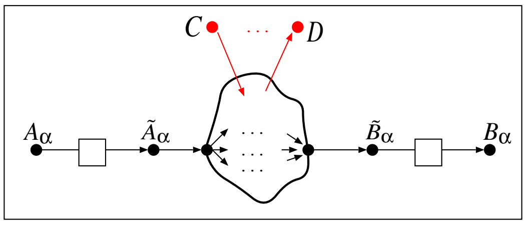


Fig. 5. Structure of the Reo-circuit R_α

nodes and the D_j's are sink nodes. Our goal is the construction of a Reo circuit R with source nodes $C_1,\dots,C_n$ and sink nodes $D_1,\dots,D_m$ such that $\mathcal{L}(\alpha_0) = \mathcal{L}(R)$.

For the construction of R, we use a compositional approach that builds a Reo circuit R_α for each subexpression α of α_0. Fig. 5 shows the general structure of R_α: if the source node A_α is fed from outside with some data element, then it is put into the buffer between A_α and $\tilde{A}_\alpha$. As soon as $\tilde{A}_\alpha$ takes the data element from the buffer, the sub-circuit in the middle is "activated". Similarly, data-flow inside this sub-circuit stops as soon as a data element arrives at $\tilde{B}_\alpha$, which puts it into the buffer between $\tilde{B}_\alpha$ and B_α. Thus, data-flow at the sink node $\tilde{B}_\alpha$ can be viewed as a signal that R_α has "terminated".

The nodes C, D in Fig. 5 are there to indicate that there will be some channels connecting the sub-circuit in the middle of R_α with (some of) the source nodes C and (some of) the sink nodes D in $\mathcal{N}$. The construction of a circuit R for an expression α_0 will be completed by a last step, in which "Init" and "Stop" components, defined in Example 2, are added to begin and end the data-flow of in the circuit R_{α_0}, as shown in Fig. 6. The construction of the circuit will be such that at any moment, *exactly one* of the leftmost and rightmost buffers or buffers inside R_{α_0} will be filled. Thus, we may consider data-flow through R as a *token game*, where the token is passed on from left to right. The reason why we put R_{α_0} in the context of an initializer and a terminator is that

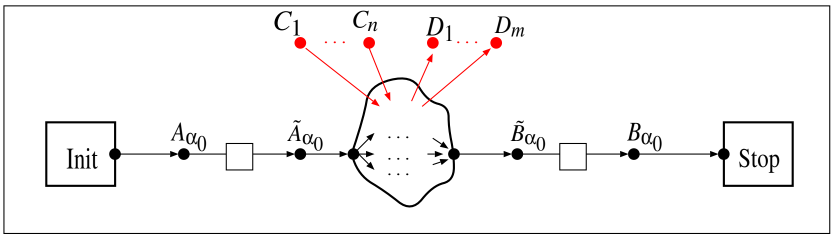
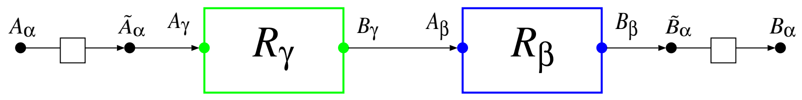


Fig. 6. The final Reo-circuit R

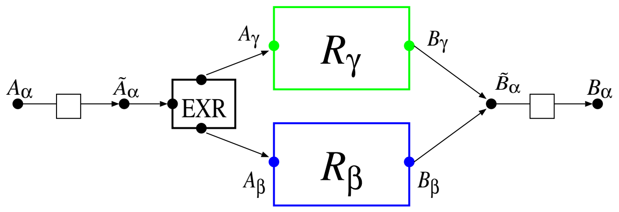
the circuit R_{α_0} allows a "restart" of data flow at node A_{α_0} whenever $\tilde{A}_{\alpha_0}$ has consumed the data item in the buffer between A_{α_0} and $\tilde{A}_{\alpha_0}$. In fact, the initializer ensures that data flow at A_{α_0} occurs exactly once. The reason for using the stop-component is similar.

Concatenation, union and closure. We first explain how to construct a circuit R_α, assuming we have already constructed the circuits for α's subexpressions. (If a subexpression α occurs more than once in α_0, e.g. if $\alpha_0 = \alpha;\alpha$, then we need a copy of the circuits R_α for every syntactic occurrence of α as a subexpression in α_0.) For $\alpha = \gamma;\beta$ the Reo circuit R_α results from combining R_γ and R_β as follows:

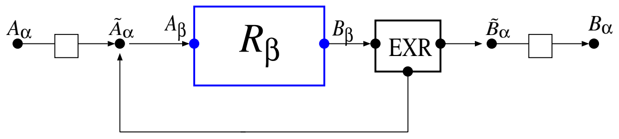


Note that the internal FIFO-channels "at the end" of R_γ and "at the beginning" of R_β (not drawn in the picture) ensure that in the concatenation $\gamma;\beta$ data-flow inside R_β cannot start before data-flow in R_γ has finished.

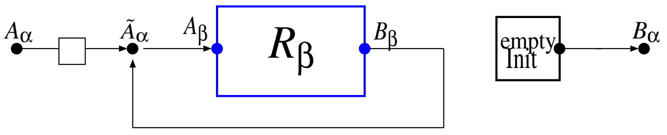
For $\alpha = \gamma \cup \beta$, the Reo circuit R_α is obtained by combining R_γ and R_β with an exclusive router that nondeterministically chooses to "activate" the data-flow in either R_γ or R_β:



The Reo circuit R_α where $\alpha = \beta^\infty$ is obtained from R_β as follows:[3]



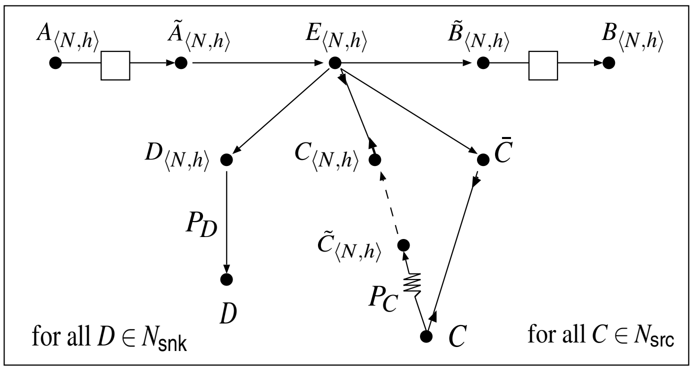
For $\alpha = \beta^\omega$, the Reo circuit has the following structure.



Here, "empty Init" is a variant of the initializer in Ex. 2, where the buffer is initially empty. Thus, data-flow never occurs in "empty Init" or at node B_α. Being non-reachable, it may be omitted; we keep it here so that the circuit retains the general shape of Fig. 5.

The empty expression. For $\alpha = \varepsilon$, we simply use a FIFO1 channel with its source end on node A_ε and its sink end on node B_ε. (Using just a single channel departs from the general schema sketched in Fig. 5, but the nodes $\tilde{A}_\alpha$ and $\tilde{B}_\alpha$ are not needed in our compositional approach.)

[3] The syntax of scheduled-data expressions does not include the Kleene closure $\alpha = \beta^*$. However, it could be treated by simply replacing the exclusive router with a fair exclusive router.



$A_{\langle N,h\rangle}$
$\tilde{A}_{\langle N,h\rangle}$
$E_{\langle N,h\rangle}$
$\tilde{B}_{\langle N,h\rangle}$
$B_{\langle N,h\rangle}$
$D_{\langle N,h\rangle}$
$C_{\langle N,h\rangle}$
$\bar{C}$
P_D
$\tilde{C}_{\langle N,h\rangle}$
P_C
D
C
for all $D \in N_{\mathsf{snk}}$
for all $C \in N_{\mathsf{src}}$

Fig. 7. Reo-circuit $R_{\langle N,h\rangle}$

Atomic expressions. So far the construction of Reo circuits for composite expressions has followed patterns that are familiar from automata theory. Next we come to the most complicated and most interesting step in our construction, namely the construction of a Reo circuit for atomic expressions $\langle N,g\rangle$. The difficulty lies in the fact that such expressions model a computation step of a corresponding Reo circuit, in which certain channel ends are active and others are not. Moreover, we must ensure that at every active channel end, the right data value is input or output.
Let Atoms denote the set of all atomic expressions $\langle N,g\rangle$ of α_0. Recall that N is a nonempty subset of $\mathcal{N} = \{C_1,\ldots,C_n\} \cup \{D_1,\ldots,D_m\}$ and g is a satisfiable data constraint for the nodes in N. We first describe a general technique to design a Reo circuit for the atoms $\langle N,g\rangle \in$ Atoms. (Later we explain how this technique can be made more efficient in various ways.) We first transform g into its canonical disjunctive normal form, which replaces it with an equivalent data constraint $h_1 \vee \ldots \vee h_r$ where each of the h's is a formula of the form

$$h \;=\; \bigwedge_{C\in N_{\mathsf{src}}} (d_C \in P_C) \wedge \bigwedge_{D\in N_{\mathsf{snk}}} (d_D \in P_D)$$

with $N_{\mathsf{src}} = N \cap \{C_1,\ldots,C_n\}$, $N_{\mathsf{snk}} = N \cap \{D_1,\ldots,D_m\}$ and P_C, $P_D \subseteq Data$. E.g., if g is "$d_C = d_D$" then we replace g with $\bigvee_{d\in Data} h_d$ where h_d is $(d_C = d) \wedge (d_D = d)$. Next, we replace $\langle N,g\rangle$ with the equivalent expression $\langle N,h_1\rangle \cup \ldots \cup \langle N,h_r\rangle$, construct the circuits $R_{\langle N,h_k\rangle}$ (see below) and combine them with the union-operator described above. With the formula h as above, a circuit $\mathcal{R}_{\langle N,h\rangle}$ for $\langle N,h\rangle$ is presented in Fig. 7, which we now explain. For the Reo circuit $R_{\langle N,h\rangle}$ of a given $\langle N,h\rangle$, we need a pair of nodes $C_{\langle N,h\rangle}$ and $\tilde{C}_{\langle N,h\rangle}$ for every source node $C \in N_{\mathsf{src}}$, and similarly, one node $D_{\langle N,h\rangle}$ for every sink node $D \in N_{\mathsf{snk}}$, plus one other node $E_{\langle N,h\rangle}$. The same node $\bar{C}$ must be used for all circuits $R_{\langle M,f\rangle}$ where $C \in M$ and $\langle M,f\rangle \in$ Atoms, while the nodes $C_{\langle N,h\rangle}$ and $\tilde{C}_{\langle N,h\rangle}$ are unique for every atomic data expression $\langle N,h\rangle \in$ Atoms where $C \in N$.

We can think of the node $E_{\langle N,h\rangle}$ as a switch that synchronizes the data-flow in the upper sub-circuit with the nodes $D_{\langle N,h\rangle}$, D, and $C_{\langle N,h\rangle}$, $\tilde{C}_{\langle N,h\rangle}$, $\bar{C}$, and C for all source nodes $C \in N_{\mathsf{src}}$ and all sink nodes $D \in N_{\mathsf{snk}}$. The synchronous channel from $E_{\langle N,h\rangle}$ to

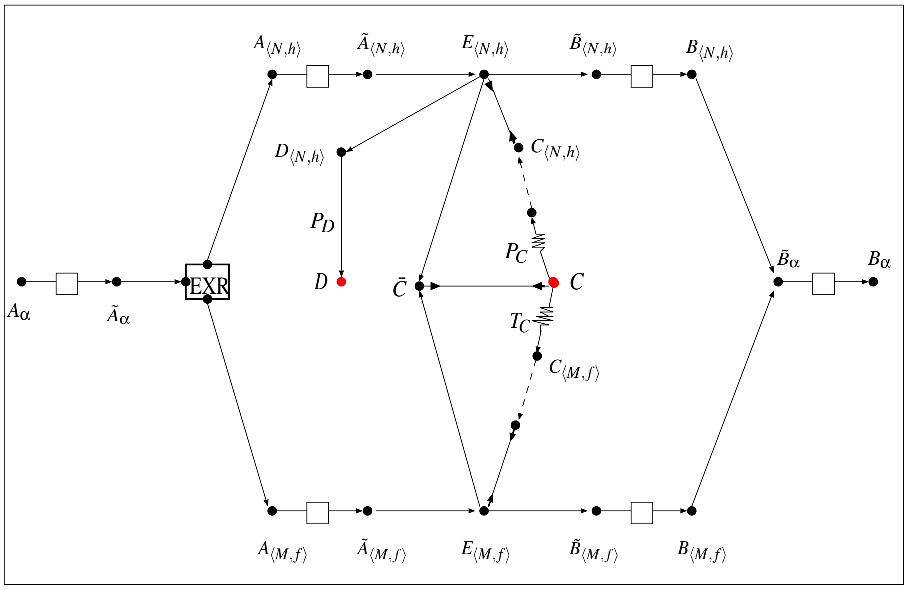


Fig. 8. Reo circuit R_α for $\alpha = \langle N,h\rangle \cup \langle M,f\rangle$ where $N = \{C,D\}$, $M = \{C\}$

$D_{\langle N,h\rangle}$ and the P_D-producer connecting $D_{\langle N,h\rangle}$ with D ensure that any data-flow at $E_{\langle N,h\rangle}$ is synchronized with the receipt of a value $d \in P_D$ at sink node D.

For the source nodes C, the situation is a bit more complicated because we must ensure that C accepts an input value iff C synchronizes with exactly one of the nodes $E_{\langle M,f\rangle}$ where $\langle M,f\rangle$ is a subexpression of α_0 with $C \in M$. The use of perfect synchronous channels is not appropriate because of the replicator semantics of the source nodes. If C were connected with $E_{\langle N,h\rangle}$ via perfect synchronous channels only, then data-flow would block when C appears in two or more atomic subexpressions of α_0. (Note that simultaneous data-flow at different nodes $E_{\langle N,h\rangle}$, $E_{\langle M,f\rangle}$ is not possible.) For this reason, we connect C with $E_{\langle N,h\rangle}$ via a filter channel, a lossy synchronous channel and a synchronous drain through the nodes $C_{\langle N,h\rangle}$ and $\tilde{C}_{\langle N,h\rangle}$. These three channels (1) allow C to pass values even when $E_{\langle N,h\rangle}$ is not available to synchronize with C, and (2) force C to pass a value $d \in P_C$ when it synchronizes with $E_{\langle N,h\rangle}$. To prevent C from passing a value without synchronizing with one of the nodes $E_{\langle M,f\rangle}$ where $C \in M$, we use a synchronous channel connecting $E_{\langle N,h\rangle}$ with $\bar{C}$ and a synchronous drain between $\bar{C}$ and C. These channels ensure that for $C \in M$, C is active exactly when data-flow occurs at $\bar{C}$ and exactly one of the nodes $E_{\langle M,f\rangle}$.

A concrete example for the Reo-circuit which is constructed from the scheduled-data expression $\alpha = \langle N,h\rangle \cup \langle M,f\rangle$ is shown in Fig. 8. Here, we assume that $N = \{C,D\}$, $M = \{C\}$ and h is $(d_C \in P_C) \wedge (d_D \in P_D)$ while f is $d_C \in T_C$. The proof for the correctness of our synthesis algorithm is quite technical and omitted here.

Size of the constructed circuit. In the worst case, the treatment of the atoms $\langle N,g\rangle$ leads to an exponential blow-up (because every disjunctive normal form for g may be exponentially longer than g). However, when we assume that all data constraints in α_0

are given in canonical disjunctive normal form and when we measure the length of α_0 as the total length of all data constraints occurring in (one of the atoms in) α_0 then the total number of channels in the constructed circuit is *linear* in the length of α_0.

Preprocessing. We now explain how a preprocessing phase of the set Atoms can simplify the construction of the circuits for the atomic subexpressions of α_0. We first look for pairs $\langle C, D\rangle$ with $C \in \{C_1, \dots, C_n\}$, $D \in \{D_1, \dots, D_m\}$ such that for all $\langle N, g\rangle \in \mathsf{Atoms}$ either $\{C, D\} \cap N = \emptyset$ or $\{C, D\} \subseteq N$ and $g \leq d_C = d_D$. ($\leq$ denotes logical implication.) Then, we establish a synchronous channel with its source end on node C, its sink end on node D and remove D in the sense that any $\langle N, g\rangle \in \mathsf{Atoms}$ is replaced with $\langle N \setminus \{D\}, g[d_D/d_C]\rangle$ where $g[d_D/d_C]$ means the data constraint resulting from g by the syntactic replacement of any occurrence of d_D with d_C. Second, for any pair (C_i, C_j) of source nodes such that for all $\langle N, g\rangle \in \mathsf{Atoms}$ either $\{C_i, C_j\} \cap N = \emptyset$ or $\{C_i, C_j\} \subseteq N$ and d_{C_j} does not occur in g, we establish a synchronous drain connecting C_i and C_j and remove C_j from Atoms. The same technique can be applied to sink nodes D_i, D_j such that for all $\langle N, g\rangle \in \mathsf{Atoms}$ either $\{D_i, D_j\} \cap N = \emptyset$ or $\{D_i, D_j\} \subseteq N$ and d_{D_j} does not occur in g, where we generate a synchronous spout with its sink ends D_i and D_j and remove D_j. Finally, we look for sink nodes D_i, D_j such that $\langle N, g\rangle \in \mathsf{Atoms}$ implies $\{D_i, D_j\} \cap N = \emptyset$ or $\{D_i, D_j\} \subseteq N$ and $g \leq d_{D_i} = d_{D_j}$ and insert a new sink node D_{ij} with synchronous channels from D_{ij} to D_i and D_j. We then remove D_i, D_j from Atoms and treat D_{ij} as a sink node. A similar transformation $\langle C_i, C_j\rangle \rightsquigarrow C_{ij}$ applies to source nodes such that for all $\langle N, g\rangle \in \mathsf{Atoms}$ either $\{C_i, C_j\} \cap N = \emptyset$ or $\{C_i, C_j\} \subseteq N$ and $g \leq d_{C_i} = d_{C_j}$. However, here we need a Reo connector that checks the equality of two (synchronously) arriving input values.

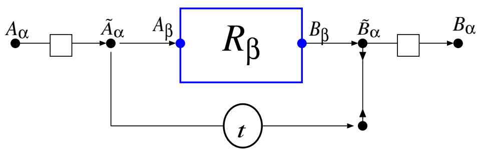
Optimization. As in other algorithmic constructions, our resulting Reo circuits contain certain redundancies which can be optimized away. We can detect and remove them by applying circuit transformation rules that look for recognizable patterns of subcircuits and replace them with their simpler equivalents. For instance, every occurrence of a synchronous channel preceding or following any other channel X can be simplified to only X. In Fig. 8, there are multiple candidates that qualify for the application of various circuit transformation rule. For example, we know that in Fig. 8, data-flow can occur through only one of the top or bottom branches of the circuit (because there is only one token at a time that passes through the entire circuit; the exclusive router; and because the two branches are isolated from one another by drains). This makes the right-hand-side FIFO1 channels on both top and bottom branches redundant.

6 Conclusion

The main contribution of the present paper is a general construction of a Reo circuit from a constraint automaton. Although similar constructions exist in the classical area of automata and digital circuits, the situation here is far more complicated because of two major differences: (1) The behavior specified by constrained automata is generally not functional (from input to output) but relational. (2) In a digital circuit and the Mealy automaton describing it, behavior is always synchronous. In contrast, in Reo, behavior can be synchronous, asynchronous, or (at different steps) a combination of the two. Because of in particular point 2, the classical construction of a circuit from an automaton breaks down, and at forehand, it was by no means obvious how to tackle the problem for Reo. We, therefore, see the algorithm described in the present paper as a major step forward in the automatic synthesis of Reo component connector circuits.

From the theoretical point of view, the results established here and in [4] yield that Reo connector circuits, constraint automata, and scheduled-data streams have the same expressiveness and can be transformed into each other via algorithmic transformations. This result can also be useful in practice as it allows to switch between these three formalisms. For instance, it enables one to use automata-models within the Reo framework to describe (and finally to synthesize) the interfaces of black-box components. On the other hand, our algorithm also illustrates the expressive power of the channel types presented in Fig. 1. (Note that our construction uses all of them, except for the asynchronous spout.)

To some extent, our construction can also be modified to treat real-time constraints, e.g., those formalized by timed scheduled-data expressions of the form $\alpha = \beta^{\leq t}$ stating that data flow described by β must be completed within t time units. (See [3] for a formal treatment of real-time within the Reo framework.). For this, we just connect the node $\tilde{A}_\alpha$ to the node $\tilde{B}_\alpha$ via a synchronous drain and a timer channel with off-option, i.e., a timer channel that allows the timer to be stopped at any point in time before the expiration of its delay. In the picture below, this timer channel is depicted by an arrow with a circle labeled with the delay t in its middle.



A compositional approach similar to the one we suggest here can also be used to provide "Reo-implementations" for processes specified in terms of CCS- or CSP-like process algebras. In fact, some of the typical operators are already included in regular expressions (CCS-like nondeterminism corresponds to union, sequential composition to concatenation, and ε to, e.g., the CCS-process nil). Parallel composition with CCS- or CSP-like synchronization can be realized by establishing appropriate synchronous channels and Reo's join operator. A LOTOS-like disrupt operator $P[>Q$ can be obtained using a Reo component that realizes a switch; this switch is initially "on" and synchronizes with P as long as it is "on" but is turned "off" by Q's first activity (the inhibitor circuit in [2] can be used to construct this switch from our set of primitive channels).

Although the construction presented here is not overly complicated, it can and should still be simplified further and made more efficient. Parts of such considerations have already been sketched in the present paper. In our future work, we will investigate further optimizations and the design of an alternative synthesis algorithm that goes directly from automata to Reo circuits without having the regular expressions as an intermediate step. Furthermore, dynamic reconfiguration of connector circuits is an inherent aspect of Reo that we plan to cover in our future work.

References

1. F. Arbab. Abstract behavior types: A foundation model for components and their composition. In *[?]*, pages 33–70, 2003.
2. F. Arbab. Reo: A channel-based coordination model for component composition. *Mathematical Structures in Computer Science*, 14(3):1–38, 2004.
3. F. Arbab, C. Baier, F. de Boer, and J. Rutten. Models and temporal logics for timed component connectors. In *Proc. SEFM'04*. IEEE CS Press, 2004.
4. F. Arbab, C. Baier, J.J.M.M. Rutten, and M. Sirjani. Modeling component connectors in reo by constraint automata. In *FOCLASA'03*, volume 97 of *ENTCS*, pages 25–41, 2004. Full version see http://web.informatik.uni-bonn.de/I/baier/publikationen.html.
5. F. Arbab and J.J.M.M. Rutten. A coinductive calculus of component connectors. In *Recent Trends in Algebraic Development Techniques, Proc. 16th Int. Workshop on Algebraic Development Techniques (WADT 2002)*, volume 2755 of *LNCS*, pages 35–56, 2003.
6. P.C. Attie and E.A. Emerson. Synthesis of concurrent systems with many similar sequential processes. In *Proc. POPL*, ACM Press, pages 191–201, 1989.
7. M. Broy and K. Stolen. *Specification and Development of Interactive Systems: Focus on Streams, Interfaces and Refinement*. Springer-Verlag, 2000.
8. A. Church. Logic, arithmetic and automata. In *Proc. Int. Congress of Mathematicians*, pages 23–35. Institut Mittag-Leffler, 1962.
9. CIM. http://www.almende.com/cim/.
10. D. Clarke, D. Costa, and F. Arbab. Modeling coordination in biological systems. In *Proc. of the Int. Symposium on Leveraging Applications of Formal Methods (ISoLA 2004)*, 2004.
11. N. Diakov and F. Arbab. Compositional construction of web services using Reo. In *Proc. International Workshop on Web Services: Modeling, Architecture and Infrastructure (ICEIS 2004), Porto, Portugal, April 13-14*, 2004.
12. E.A. Emerson and E.M. Clarke. Using branching time logic to synthesize synchronous skeleton. *Science of Programming*, 2:241–266, 1982.
13. T. Henzinger, R. Jhala, and R. Majumdar. Counterexample-guided control. In *Proc. 30th Int. Colloquium on Automata, Languages, and Programming (ICALP)*, volume 2719 of *LNCS*, pages 886–902, 2003.
14. J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Language, and Computation*. Addison–Wesley, 2nd edition edition, 2001.
15. Z. Manna and P. Wolper. Synthesis of communicating processes from temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 6:68–93, 1984.
16. A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *Proc. 16th Symposium on Principles of Programming Languages*, pages 179–190. ACM Press, 1989.
17. J.J.M.M. Rutten. Component connectors. In *[?]*, chapter 5, pages 73–87. 2004.
18. W. Thomas. On the synthesis of strategies in infinite games. In *Proc. of the 12th Annual Symp. on Theoretical Aspects of Computer Science*, volume 900 of *LNCS*, pages 1–13, 1995.
19. M. Vardi. An automata-theoretic approach to fair realizability and synthesis. In *Proc. CAV*, volume 939 of *LNCS*, pages 267–278, 1995.
20. Z. Zlatev, N. Diakov, and S. Pokraev. Construction of negotiation protocols for E-Commerce applications. *ACM SIGecom Exchanges*, 5(2):11–22, November 2004.