# Synthesis of Reo Connectors for Strategies and Controllers

Christel Baier Joachim Klein
Sascha Klüppelholz

Faculty of Computer Science,
Technische Universität Dresden, Germany

**Abstract**

In controller synthesis, i.e., the question whether there is a controller or strategy to achieve some objective in a given system, the controller is often realized as some kind of automaton. In the context of the exogenous coordination language Reo, where the coordination glue code between the components is realized as a network of channels, it is desirable for such synthesized controllers to also take the form of a Reo connector built from a repertoire of basic channels. In this paper, we address the automatic construction of such Reo connectors directly from a constraint automaton representation.

## 1 Introduction

Synthesis problems have a long tradition in computer science and were first raised by Church [10]. In controller synthesis, the goal is to algorithmically synthesize a controller for a system (often called a *plant*) that ensures a given objective. Several instances of the controller synthesis problem have been studied in the literature, e.g., [1, 13, 14, 5, 6] that differ in the type of system models and objectives, the assumptions on what is visible to the controller and the way how the environment and controller interact with the system.

We are interested in the controller synthesis problem in the context of a component-based approach to software engineering, where it is desirable to provide a clear separation between the aspects of computation inside the components and the coordination of the interactions between the components. The coordination language Reo [3] facilitates such an exogenous approach to coordination, with the components being unaware about the context beyond their interfaces to the outside world. This approach allows the flexible (re)use and exchangeability of heterogeneous components in various situations, with their interactions orchestrated by "glue code" in the form of a network of Reo channels. These channels and the components themselves may be deployed in a flexible, distributed manner. Constraint automata [9] serve as the operational semantics of Reo. They provide a unified framework capturing both the description of the behavior at the component interfaces as well as for the coordination arising from the composition of the Reo primitives in a network.

In this context it is then desirable to automatically derive the necessary coordination behavior for a given set of components by specifying the desired properties of the system. For the Reo and constraint automata framework such algorithms usually produce a controller in the form of a constraint automaton that captures the coordination necessary to achieve the specified objectives. In this paper, we provide a construction to take this a step further and, given such a constraint automaton $\mathcal{A}$, synthesize an equivalent Reo network $\mathcal{C}$. The controller thus ceases to be a monolithic automaton and becomes a Reo connector network itself that integrates with the already existing glue code and is amenable to optimizations and other operations acting on Reo networks.

This construction is for example applicable in the context of alternating-time stream logic (ASL) [12], which provides a formalism to specify a variety of relevant questions in the setting of multiple components and their interactions, e.g., whether a subset of the components are able to cooperate and ensure some property, whether all components can cooperate to ensure that their composition will never deadlock (similar to the notion of interface compatibility in [11]) or whether the interfaces of the components meet each others interface constraints. In all of these cases, if the answer is positive, it is possible to algorithmically derive a constraint automaton that, when composed with the system, ensures that the given desired property holds. Using the construction presented in this paper, this automaton can be realized as a Reo network that orchestrates the interactions of the components to ensure the desired property.

Another area of interest is the application of the results in [7] to the Reo and constraint automaton setting. There, in a partial information and partial control setting, the goal is to synthesize controllers that enforce certain linear time properties in the system. To allow a compositional approach of iteratively treating the individual subformulas of a property given as a conjunction, the constructed controllers are *most-general* in the sense that they capture *all possible* ways in which a given property may be enforced in the system. Applying the presented framework to the setting of Reo and constraint automata, the synthesized controllers can be again regarded as constraint automata (with some additional fairness constraints). These automata can then likewise be transformed into Reo networks using the presented construction.

In [4] an algorithm has been presented that constructs a Reo network from a given constraint automaton. This approach is compositional and relies on a preprocessing step that generates an $\omega$-regular expression from the given constraint automaton. We present here an alternative approach for the generation of a Reo network directly from the constraint automaton $\mathcal{A}$ which reuses some ideas of [4], but avoids the potential exponential blow-up in the construction of an $\omega$-regular expression.

**Outline.** In Sec. 2, we briefly describe the main concepts of constraint automata and Reo. Sec. 3 provides an overview of two use cases for the construction, while Sec. 4 describes the construction of Reo networks for a given constraint automaton.

## 2 Preliminaries

In this section, we provide a brief overview of constraint automata, Reo and their relation.

### 2.1 Constraint automata

Constraint automata (CA) [9] provide a generic operational model to formalize the behavioral interfaces of the components, the network that coordinates the components (i.e., the glue code or connector), and the composite system consisting of the components and the glue code. Constraint automata are variants of labeled transition systems (LTS) where the labels of the transitions represent the (possibly data-dependent) I/O-operations of the components and the network. They support any kind of synchronous and asynchronous peer-to-peer communication. The states of a constraint automaton represent the local states of components and/or configurations of a connector.

To formalize the I/O-activity, constraint automata use a finite set $\mathcal{N}$ of data-flow locations, where each element of $A \in \mathcal{N}$ stands for a data-flow location where I/O can occur, such as the interface ports of components, nodes in the connector network, etc. To simplify the presentation in this paper, we assume that the data items that may occur at each data-flow location are elements of a finite, global data domain $\mathsf{Data}$. Each transition of a constraint automaton is labeled by a pair $(N, g)$, where $N \subseteq \mathcal{N}$ is a set of active data-flow locations and $g$ is a data constraint which restricts the possible data items at the active data-flow locations in $N$. Formally, data constraints are propositional formulas built from the atoms "$d_A = d$", where data item $d \in \mathsf{Data}$ occurs at data-flow location $A \in \mathcal{N}$, and "$d_A = d_B$", where the data items at data-flow locations $A$ and $B$, with $A, B \in \mathcal{N}$, are the same.

**Definition 1** (Data constraints)**.** Data constraints are given by the following grammar:

$$g \quad ::= \quad \text{true} \quad | \quad d_A = d \quad | \quad d_A = d_B \quad | \quad g_1 \vee g_2 \quad | \quad \neg g$$

where $A, B \in \mathcal{N}$ and $d \in \mathsf{Data}$. For a subset $N \subseteq \mathcal{N}$, we denote the set of data constraints using only atoms of the form "$d_A = d$" and "$d_A = d_B$" with $A, B \in N$ by $DC(N)$. Other standard propositional operators such as conjunction ($\wedge$) or implication ($\rightarrow$) can be derived as usual. $d_A \neq d$ stands shortly for $\neg(d_A = d)$. □

**Definition 2** (Constraint automata)**.** A constraint automaton is a tuple $\mathcal{A} = (Q, \mathcal{N}, \mathcal{N}_{\text{in}}, \mathcal{N}_{\text{out}}, \longrightarrow, Q_0)$ where

- $Q$ is a finite set of states,
- $\mathcal{N}$ is a finite set of data-flow locations,
- $\mathcal{N}_{\text{in}}$ and $\mathcal{N}_{\text{out}}$ are disjoint subsets of $\mathcal{N}$,
- $\longrightarrow$ is a subset of $Q \times 2^{\mathcal{N}} \times DC(\mathcal{N}) \times Q$,
- $Q_0 \subseteq Q$ is the non-empty set of initial states.

We write $q \xrightarrow{N,g} p$ instead of $(q, N, g, p) \in \longrightarrow$. For every transition $q \xrightarrow{N,g} p$, we require that $g \in DC(N)$, i.e., that the data constraint only refers to data at the active data-flow locations $A \in N$. $\square$

The subsets $\mathcal{N}_{\text{in}}$ and $\mathcal{N}_{\text{out}}$ of $\mathcal{N}$ provide a characterization of the corresponding data-flow locations as being available for an external connection, which is used during the composition of the constraint automata for the components and the network. Data-flow locations in $\mathcal{N}$ that are not elements of either $\mathcal{N}_{\text{in}}$ or $\mathcal{N}_{\text{out}}$ can be regarded as internal data-flow locations.

Due to the data constraint, each transition label $(N, g)$ stands for a *set of concurrent I/O-operations (CIO)*, which formalizes the assignment of concrete data values to the active data-flow locations $N$, i.e., if the data constraint $g$ is satisfiable each concurrent I/O-operation represents one particular assignment of data values to the active locations $N$ satisfying $g$. Consequently, an *execution* in a constraint automaton is a finite or infinite alternating sequence of states and concurrent I/O-operations representing the steps of the automaton, such that each CIO corresponds to a transition between the corresponding states.

## 2.2 Reo

We provide here a brief overview of the main concepts of Reo, for more details we refer to [3, 9, 8]. Reo is a channel-based, exogenous coordination language. It allows the specification of the coordination glue between components by a network of channels and Reo nodes. Channels in Reo serve as the primitive building blocks for the network. A channel has two distinct channel ends, each being either a source end, through which data enters a channel or a sink end, through which data leaves a channel. The operational semantics of Reo networks can be provided in a compositional way using constraint automata for the channels and an appropriate composition operator (product construction) on constraint automata for the Reo join operation, which joins channel ends together to form Reo nodes in the network.

Reo provides a library of basic channels for a wide variety of common use cases, which can be easily extended by user-defined channels. As an example, Fig. 1 shows three of the basic channels and their constraint automata representation. The synchronous channel – Fig. 1a – synchronizes its source end and its sink end, transferring the data item from the source end $A$ to the sink end $B$. The filter channel – Fig. 1b – behaves like the synchronous channel in the case that the transferred data item matches the filter condition $D \subseteq \mathsf{Data}$ and blocks otherwise. The FIFO1 channel – Fig. 1c – receives a single data item via source end $A$, stores it and subsequently provides that value at sink end $B$.

To combine the various channels to form a network, two or more channel ends are joined together, forming a *Reo node*. The Reo nodes orchestrate the activity of all the connected channel ends by accepting an incoming data item via one of the connected sink ends and synchronously passing it on to one (or more) of the connected source ends. On the incoming side, the Reo node perform a *non-deterministic merge*, choosing exactly one of the channel ends to be active. On the outgoing side, the behavior depends on the Reo node types. For the *standard Reo nodes* (depicted as $\bullet$), the data item is copied and output simultaneously to *all* the connected source ends (replication). *Route nodes* (depicted as $\otimes$) output the data item to *exactly one* of the connected source ends (routing), chosen

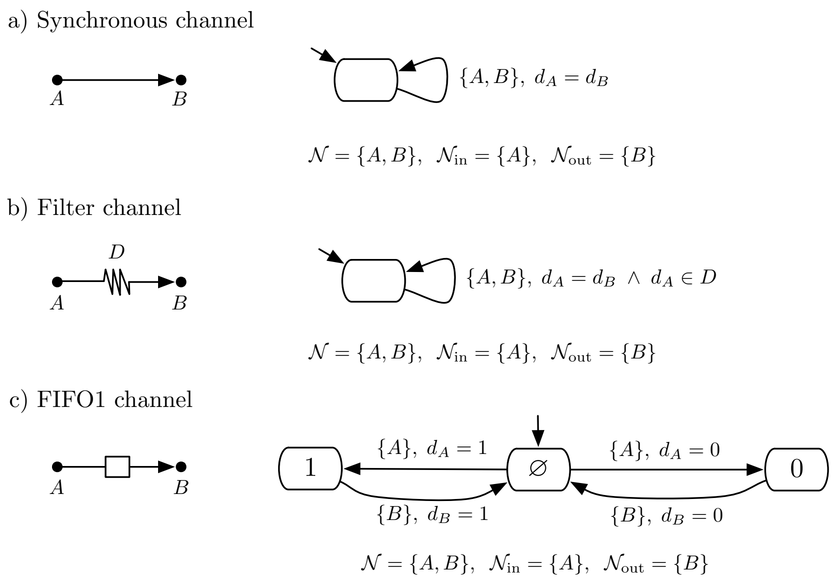

a) Synchronous channel
A
B
{A, B}, d_A = d_B
N = {A, B}, N_in = {A}, N_out = {B}
b) Filter channel
D
A
B
{A, B}, d_A = d_B ∧ d_A ∈ D
N = {A, B}, N_in = {A}, N_out = {B}
c) FIFO1 channel
A
B
1
∅
0
{A}, d_A = 1
{A}, d_A = 0
{B}, d_B = 1
{B}, d_B = 0
N = {A, B}, N_in = {A}, N_out = {B}

Figure 1: Basic Reo channels and the corresponding constraint automata. For the FIFO1 channel, the constraint automaton is shown for $\mathsf{Data} = \{0, 1\}$.

non-deterministically. Similarly to the channels, the operational semantics of the Reo nodes can be captured by constraint automaton.

**Constraint automata product, hiding.** To compositionally obtain the constraint automaton representing a Reo network, a product construction (with appropriate renaming of the data-flow locations) for the individual constraint automata of all the constituent parts of the network can be used, i.e., for the channels, the Reo nodes and the attached components. To facilitate hierarchical modeling and abstract away from implementation details of a given Reo network, a *hiding operator* on constraint automata may be used, which removes certain internal data-flow locations from the constraint automaton by appropriately modifying the transitions and data constraints.

## 3 Scenarios for connector synthesis

We will now present two scenarios where it is possible to algorithmically derive constraint automata for connectors that exogenously realize strategies that enforce a given objective. These constraint automata can then be synthesized as a network of Reo channels that realize the given strategies as connector glue code, as detailed in Sec. 4.

### 3.1 Connectors for game strategies

Game logics such as the alternating-time temporal logic (ATL) [2] allow reasoning about strategies for coalitions of components. In [12] we proposed the alternating-time stream logic (ASL) for reasoning about strategies in the Reo and constraint automata framework. Here, constraint automata are being interpreted as multi-player game structures. Our starting point is a set of components $A = \{\mathcal{A}_1, \ldots, \mathcal{A}_k\}$, a subset $B \subseteq A$ of controllable components and optionally a coordinating network $R$.

We are interested in the existence of a strategy for the controllable components in $B$ to cooperate in such a way that they achieve a common goal. Here, a strategy means that the controllable components can restrict or even refuse performing certain actions or participate to synchronize actions with other components or the environment where they are involved. To reason over $B$-strategies, ASL provides existential and universal quantification of the form $\mathbb{E}_B \varphi$ and $\mathbb{A}_B \varphi$, where $\varphi$ is an ASL path formula that formalizes the common goal of the coalition. The intuitive semantics for formulas of the form $\mathbb{E}_B \varphi$ asserts that there is a $B$-strategy such that all remaining paths of the composite system do satisfy the path formula $\varphi$. The ASL state formula $\mathbb{A}_B \varphi$ asserts that for all $B$-strategies the remaining set of paths in the composite system contains at least one path that satisfies $\varphi$.

In [12] we have seen that finite-memory $B$-strategies are sufficient and can be represented as a constraint automaton $\mathcal{C}$ which we can put in parallel with the components (and the orchestrating network) such that the composite system satisfies the coalitions goal. The technique presented in Section 4 is applicable to synthesize $\mathcal{C}$ as a Reo network which coordinates the components exogenously and guarantees the common goal to be satisfied.

## 3.2 Connectors realizing most-general strategies

In [7], we present a general framework for the compositional synthesis of controllers in a partial information and partial control setting for linear time objectives. Given a linear time objective $\Phi$ and two automata $\mathcal{A}_{\mathrm{ctr}}$, representing the controllable part of the system, and $\mathcal{A}_{\mathrm{env}}$, representing the environment, the goal is to construct a controller $\mathcal{C}$ for the parallel composition $\mathcal{A}_{\mathrm{ctr}} \parallel \mathcal{A}_{\mathrm{env}}$ such that the objective $\Phi$ holds, i.e., such that $\mathcal{C} \parallel \mathcal{A}_{\mathrm{ctr}} \parallel \mathcal{A}_{\mathrm{env}} \models \Phi$. Due to the partial information setting, the controller may not be able to observe or distinguish certain actions occurring in the system. Furthermore, the controller is only able to influence certain "controllable" actions. Given a conjunctive objective $\Phi_1 \wedge \Phi_2 \wedge \ldots \wedge \Phi_k$, the compositional approach consists of constructing a controller $\mathcal{C}_1$ for objective $\Phi_1$ and system $\mathcal{A}_{\mathrm{ctr}} \parallel \mathcal{A}_{\mathrm{env}}$, then constructing a controller $\mathcal{C}_2$ for objective $\Phi_2$ and system $\mathcal{C}_1 \parallel \mathcal{A}_{\mathrm{ctr}} \parallel \mathcal{A}_{\mathrm{env}}$, and so on, such that

$$\mathcal{C}_k \parallel \ldots \parallel \mathcal{C}_1 \;\parallel\; \mathcal{A}_{\mathrm{ctr}} \parallel \mathcal{A}_{\mathrm{env}} \quad \models \quad \Phi_1 \wedge \ldots \wedge \Phi_k.$$

For such a compositional approach, it is necessary that each controller does not employ a single or a few tactics to enforce the objective, but is as permissive as possible in that it preserves all possible ways in which the objective can be enforced, i.e. is *most-general*, so that subsequent controllers are able to enforce their objectives as well if it is possible to enforce the conjunction. In [7], we provide algorithms that construct such most-general controllers for those safety and co-safety objectives that can be enforced in a given system.

As the automata formalism used in [7] is quite general, it can be applied to the Reo and constraint automata framework in a natural fashion. To simplify the presentation, we consider here a scenario without data values, i.e., with a singleton data domain, where only the presence or absence of activity at the various data-flow locations is relevant. The handling of data values can be accomplished either by explicitly treating them in the framework or by introducing additional data-flow locations that encode the data values at the active data-flow locations. We assume that $\mathcal{A}_{\text{ctr}}$ and $\mathcal{A}_{\text{env}}$ are constraint automata, with $\mathcal{A}_{\text{ctr}}$ representing one or more controllable components and the associated glue code, while $\mathcal{A}_{\text{env}}$ represents their environment. Let $\mathcal{N}_{\text{ctr}}$ and $\mathcal{N}_{\text{env}}$ be the set of data-flow locations of $\mathcal{A}_{\text{ctr}}$ and $\mathcal{A}_{\text{env}}$, respectively. We classify the data-flow locations $\mathcal{N} = \mathcal{N}_{\text{ctr}} \cup \mathcal{N}_{\text{env}}$ of the parallel composition $\mathcal{A} = \mathcal{A}_{\text{ctr}} \parallel \mathcal{A}_{\text{env}}$ into the data-flow locations that are *controllable*, *visible* and *invisible* to the controller according to their membership in $\mathcal{N}_{\text{ctr}}$ and $\mathcal{N}_{\text{env}}$. The controllable data-flow locations $\mathcal{N}_{\text{control}} = \mathcal{N}_{\text{ctr}} \setminus \mathcal{N}_{\text{env}}$ are those data-flow locations that are unique to $\mathcal{A}_{\text{ctr}}$. The controller is able to influence the activity at these data-flow locations. The visible data-flow locations $\mathcal{N}_{\text{visible}} = \mathcal{N}_{\text{ctr}} \cup (\mathcal{N}_{\text{ctr}} \cap \mathcal{N}_{\text{env}})$ are those that are either controllable or are shared between $\mathcal{A}_{\text{ctr}}$ and $\mathcal{A}_{\text{env}}$. The visible but uncontrollable data-flow locations represent those data-flow locations in the environment that can be observed but may not be directly influenced by the controller.

The actions of $\mathcal{A} = \mathcal{A}_{\text{ctr}} \parallel \mathcal{A}_{\text{env}}$, i.e., the concurrent I/O-operations $\mathsf{CIO} = 2^{\mathcal{N}}$, can then be straightforwardly classified into the visible and invisible and the controllable and uncontrollable actions. The visible actions $Act_{\text{vis}}$ are those where at least one visible data-flow location is active, while the controllable actions $Act_{\text{ctr}}$ are those where at least one controllable data-flow location is active:

$$Act_{\text{vis}} = \{N \subseteq \mathcal{N} : N \cap \mathcal{N}_{\text{visible}} \neq \varnothing\} \qquad Act_{\text{ctr}} = \{N \subseteq \mathcal{N} : N \cap \mathcal{N}_{\text{control}} \neq \varnothing\}.$$

Due to the partial information setting, the controller is only able to observe those parts of the actions that occur in the system that are visible, i.e., for each action that occurs, the controller only sees its observable. The observables $Obs = 2^{\mathcal{N}_{visible}} \setminus \{\varnothing\}$ are the non-empty subsets of the visible data-flow locations and the observation function $obs : 2^{\mathcal{N}} \to Obs$ maps a visible action $N \subseteq \mathcal{N}$ to the observation $obs(N) = N \cap \mathcal{N}_{visible}$. The framework in [7] furthermore has the concept of *suspend* actions, which suspend further activity of the controllable component until they are woken up by some external observation and are used to allow the controllers to request or deny the possibility of inaction of the controllable component. In the constraint automata context, such suspend actions are modeled by the introduction of a special, controllable suspend data-flow locations in $\mathcal{A}_{\text{ctr}}$.

**Controller** For certain types of objectives and if the objective can be enforced for a given system, it is possible to construct a controller, a finite-state machine that, given the observable history of the system, specifies a subset of actions that the controller is willing to allow. In [7], these controllers are most-general, as they admit all possible ways in which the objective can be enforced. Such a controller $\mathcal{C}$ induces an automaton, which can be regarded as a constraint automaton using the conventions detailed above. The set of data-flow locations of the induced automaton primarily consists of the visible data-flow locations in the system. The controller construction ensures that the controller automaton can not block uncontrollable actions. Additionally, the controller has the ability to annotate actions via a set of annotations. These annotations can be realized in the constraint automaton induced by the controller by introducing fresh, special data-flow locations encoding the set of annotations.

The induced constraint automaton of the controller can then be realized as a Reo network using the construction detailed in Sec. 4, that exogenously controls the system. To properly connect the system, the constructed Reo controller network and potentially further controllers in the compositional approach, a small amount of basic circuitry is needed to ensure that the coordination of the controllers, the controllable components and the environment happens in an appropriate fashion.

**Fairness** A crucial factor in the construction of most-general controllers and thus for the compositional approach is the ability of the controller to require fairness conditions for his local choices. These fairness conditions can be regarded as strong fairness conditions of the general structure "if the controller infinitely often visits a controller state $q$, then it will infinitely often offer certain actions in controller state $q$". The fairness conditions in [7] are slightly more general as they allow the use of sets of states. Importantly, these fairness conditions only constrain the controller choices and do not impose any requirements on the actions of the controllable components or the environment. Furthermore, by design of the controller, they are always realizable.

Naturally, when the controller automaton is synthesized as a Reo network, the controller fairness requirements need also be taken into account for the Reo network to ensure that it properly enforces the desired objective. This can be either handled by allowing the specification of such fairness conditions on the level of a Reo network or by adding appropriate circuitry to the network relying on *fair router nodes*, i.e., nodes that determine how to route the data items in a fair way.

## 4 Realization of a Constraint Automaton by a Reo network

We now describe the construction of an equivalent Reo network for a given constraint automaton $\mathcal{A} = (Q, \mathcal{N}, \mathcal{N}_{\text{in}}, \mathcal{N}_{\text{out}}, \longrightarrow, Q_0)$. We assume that the data-flow locations of $\mathcal{A}$ consist entirely of input and output ports, $\mathcal{N} = \mathcal{N}_{\text{in}} \cup \mathcal{N}_{\text{out}}$, i.e., there are no internal data-flow locations. Furthermore, we assume that the set of initial states $Q_0 = \{q_0\}$ is a singleton set, i.e., there is a single, unique starting state. Non-deterministic choice between multiple starting states can be handled with circuitry to realize the initial non-deterministic choice and first step of the automaton similar to the circuitry used below.

**Basic channels and component connectors.** We first describe the building blocks used in the construction of the Reo network. In addition to the standard basic Reo channels, we require a variant of the FIFO1 channel that can simultaneously output the currently stored data item via its sink end while accepting a new data item to store via its source end. Fig. 2 shows the graphical representation of a such a *simultaneous FIFO1 channel* together with its constraint automaton.

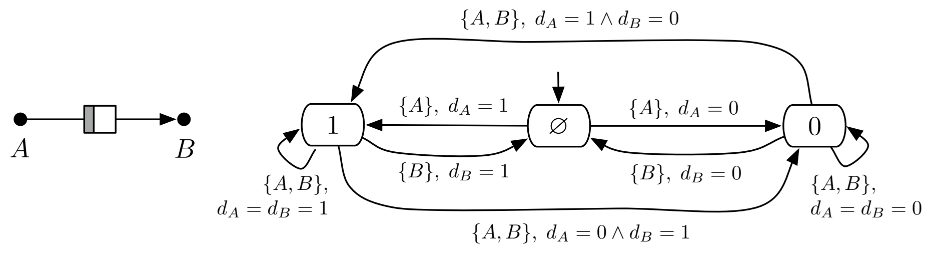


Figure 2: Simultaneous FIFO1 channel for data domain $\mathsf{Data} = \{0,1\}$

In addition to these channels, the construction relies on the use of four component connectors that are used as "primitives":

1. A *merger* has several input ports $A_1, \ldots, A_r$ and one output port $B$. It accepts non-deterministically data from exactly one of the input ports and forwards it synchronously through the output port.

2. An *exclusive router* has one input port $A$ and several output ports $B_1, \ldots, B_r$ synchronously routes an incoming data from port $A$ to exactly one of its output ports.

3. A *replicator* has one input port $A$ and several output ports $B_1, \ldots, B_r$. It sends copies of an incoming data to all of its output ports synchronously.

4. A *data constraint checker* for a data constraint

$$g \in DC(A_1, \ldots, A_r, B_1, \ldots, B_s)$$

has input ports $A_1, \ldots, A_r$ and output ports $B_1, \ldots, B_s$, as well as a special input port $S$. To activate the checker, a token has to be received via port $S$ and synchronously all the input and output ports must be active and the observed data at these ports must fulfill the data constraint $g$.

Fig. 3 shows the graphical representation of these four component connectors together with their constraint automata.

**Remark 1.** The functionality of the *merger*, *exclusive router* and *replicator* component connectors can be implemented in the Reo network by using the standard Reo nodes for the *merger* and *replicator* and route nodes for the *exclusive router*. We nevertheless first treat these as component connectors to provide a clear description of their intended function.

The constraint checkers are treated here as primitives, but using the approach presented in [4] the constraint checkers themselves could also be realized by a Reo network consisting of basic Reo channels. For data constraints in a canonical (disjunctive) normal form the idea is to provide simple Reo networks for the literals $d_A = c$, $d_A = d_B$, etc. and component connectors that realize conjunctions and disjunctions. $\square$

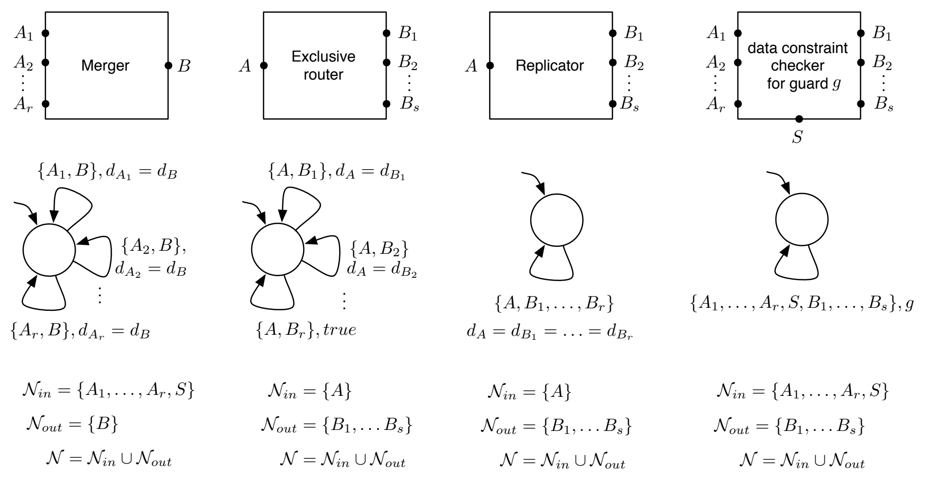

$A_1$
$A_2$
$A_r$
Merger
$B$
$A$
Exclusive router
$B_1$
$B_2$
$B_s$
$A$
Replicator
$B_1$
$B_2$
$B_s$
$A_1$
$A_2$
$A_r$
data constraint checker for guard $g$
$B_1$
$B_2$
$B_s$
$S$
$\{A_1, B\}, d_{A_1} = d_B$
$\{A_2, B\}, d_{A_2} = d_B$
$\{A_r, B\}, d_{A_r} = d_B$
$\{A, B_1\}, d_A = d_{B_1}$
$\{A, B_2\}$ $d_A = d_{B_2}$
$\{A, B_r\}, true$
$\{A, B_1, \ldots, B_r\}$ $d_A = d_{B_1} = \ldots = d_{B_r}$
$\{A_1, \ldots, A_r, S, B_1, \ldots, B_s\}, g$
$\mathcal{N}_{in} = \{A_1, \ldots, A_r, S\}$
$\mathcal{N}_{out} = \{B\}$
$\mathcal{N} = \mathcal{N}_{in} \cup \mathcal{N}_{out}$
$\mathcal{N}_{in} = \{A\}$
$\mathcal{N}_{out} = \{B_1, \ldots B_s\}$
$\mathcal{N} = \mathcal{N}_{in} \cup \mathcal{N}_{out}$
$\mathcal{N}_{in} = \{A\}$
$\mathcal{N}_{out} = \{B_1, \ldots B_s\}$
$\mathcal{N} = \mathcal{N}_{in} \cup \mathcal{N}_{out}$
$\mathcal{N}_{in} = \{A_1, \ldots, A_r, S\}$
$\mathcal{N}_{out} = \{B_1, \ldots B_s\}$
$\mathcal{N} = \mathcal{N}_{in} \cup \mathcal{N}_{out}$

Figure 3: Merger, exclusive router, replicator and data constraint checker

**Idea of the construction:** The idea of creating a Reo network $\mathcal{C}$ that realizes $\mathcal{A}$ is to represent each state $q \in Q$ by a simultaneous FIFO1 channel $f_q$ with a single buffer cell. The Reo network will mimic $\mathcal{A}$'s behavior by a *token game*, where exactly one of these simultaneous FIFO1 channels will be filled at a time. Initially, the token is in the buffer of $f_{q_0}$, corresponding to the initial state $q_0$ in the constraint automaton. Whenever a transition $q \xrightarrow{N,g}_{\mathcal{A}} q'$ fires the token moves from the buffer of $f_q$ to the buffer of $f_{q'}$. This explains the need for using simultaneous FIFO1 channels instead of the standard FIFO1 channels, as a self-loop for an automaton state requires that the corresponding FIFO1 channel be able to simultaneously output and receive the token. The structure of the construction is shown in Fig. 4. For state $q$ in $\mathcal{A}$, we deal with the simultaneous

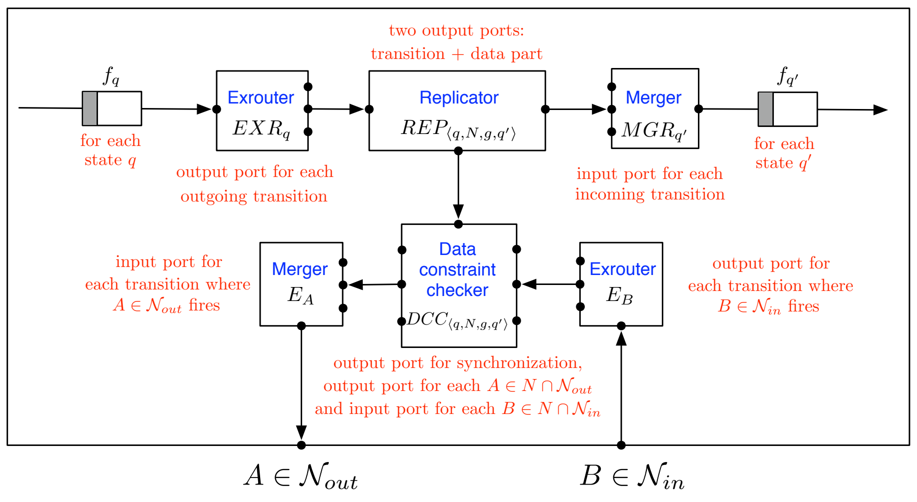

$f_q$
for each state $q$
Exrouter
$EXR_q$
output port for each outgoing transition
two output ports: transition + data part
Replicator
$REP_{\langle q,N,g,q' \rangle}$
Merger
$MGR_{q'}$
input port for each incoming transition
$f_{q'}$
for each state $q'$
input port for each transition where $A \in \mathcal{N}_{out}$ fires
Merger
$E_A$
Data constraint checker
$DCC_{\langle q,N,g,q' \rangle}$
Exrouter
$E_B$
output port for each transition where $B \in \mathcal{N}_{in}$ fires
output port for synchronization, output port for each $A \in N \cap \mathcal{N}_{out}$ and input port for each $B \in N \cap \mathcal{N}_{in}$
$A \in \mathcal{N}_{out}$
$B \in \mathcal{N}_{in}$

Figure 4: Structure of a Reo network $\mathcal{C}$ synthesized from constraint automaton $\mathcal{A}$

FIFO1 channel $f_q$ and an exclusive router $\mathrm{EXR}_q$ and a merger $\mathrm{MGR}_q$. The exclusive router $\mathrm{EXR}_q$ has one output port for each transition emanating in $q$. The merger $\mathrm{MGR}_q$ has one input port for each transition ending in $q$. For each transition $\theta$ in $\mathcal{A}$ there is a replicator $\mathrm{REP}_\theta$ and a data constraint checker $\mathrm{DCC}_\theta$. Furthermore, for each output port $A \in \mathcal{N}_{\text{out}}$ the Reo network contains a merger $\mathrm{MGR}_A$. Dually, for each input port $B \in \mathcal{N}_{\text{in}}$ we deal with an exclusive router $\mathrm{EXR}_B$. Their interface ports are connected with the data constraint checkers of the transitions the ports are involved in.

Each of the simultaneous FIFO1 channels $f_q$ is connected to the corresponding exclusive router $\mathrm{EXR}_q$. The exclusive router "schedules" non-deterministically one of the outgoing transitions $\theta$ and routes the token through the corresponding output port into the replicator $\mathrm{REP}_\theta$. The first task of this replicator is to forward the token towards the simultaneous FIFO $f_{q'}$. The merger $\mathrm{MGR}_{q'}$ ensures that only one of the incoming transitions of state $q'$ can fire at a time. The second task of the replicator $\mathrm{REP}_\theta$ is to synchronize the transition with the data constraint checker $\mathrm{DCC}_\theta$. Thus, the transition $\theta = (q, N, g, q')$ can fire if and only if all ports $A \in N$ fire and the observed data fulfills the data constraint $g \in DC(N)$. For each of the ports $A \in \mathcal{N}$ we have to ensure that they are not involved in more than one transition at a time.

For this purpose, we make use of the merger components $E_A$ in case of the output ports $\mathcal{A} \in \mathcal{N}_{\text{out}}$ and the exclusive routers $E_B$ for the input ports $B \in \mathcal{N}_{\text{in}}$ of $\mathcal{A}$. The synchronous channels from the output port of $E_A$ to $A$ and from the data constraint checkers $\mathrm{DCC}_\theta$ to $E_A$ ensure that data flow at $A$ is always synchronized with data flow at the data constraint checker $\mathrm{DCC}_\theta$ and the replicator $\mathrm{REP}_\theta$ of some transition $\theta = (q, N, g, q')$ if $A \in N$. Similarly, the synchronous channels from $B$ to input port of $E_B$ and from $E_B$ to the data constraint checkers $\mathrm{DCC}_\theta$ ensure that data flow at $B$ is always synchronized with data flow at $\mathrm{DCC}_\theta$ and $\mathrm{REP}_\theta$ of some transition $\theta = (q, N, g, q')$ if $B \in N$. Vice versa, whenever there is some data flow at one of the replicators $\mathrm{REP}_\theta$ then there must be data flow at the corresponding data constraint checker $\mathrm{DCC}_\theta$ and all active ports, i.e., those that appear in the set $N$ of $\theta$.

**Soundness of the construction.** Let $\mathcal{A} = (Q_\mathcal{A}, \mathcal{N}_\mathcal{A}, \mathcal{N}^\mathcal{A}_{\text{in}}, \mathcal{N}^\mathcal{A}_{\text{out}}, \longrightarrow_\mathcal{A}, \{q_0\})$ be a constraint automaton and $\mathcal{C}$ be the Reo network synthesized from $\mathcal{A}$. Using the standard procedure for obtaining a constraint automata for a Reo network, we can compose an automaton representation for $\mathcal{C}$. We denote the constraint automaton resulting from the product of all component connectors, channels and nodes of $\mathcal{C}$ by $\mathcal{A}_\mathcal{C} = (Q_\mathcal{C}, \mathcal{N}_\mathcal{C}, \mathcal{N}^\mathcal{C}_{\text{in}}, \mathcal{N}^\mathcal{C}_{\text{out}}, \longrightarrow_\mathcal{C}, Q_{0,\mathcal{C}})$. The set $\mathcal{N}_\mathcal{C}$ contains all internal nodes and boundary nodes (i.e., ports) of $\mathcal{C}$, while $\mathcal{N}^\mathcal{C}_{\text{in}} = \mathcal{N}^\mathcal{A}_{\text{in}}$ and $\mathcal{N}^\mathcal{C}_{\text{out}} = \mathcal{N}^\mathcal{A}_{\text{out}}$. To allow a comparison of the behavior of $\mathcal{A}$ and $\mathcal{A}_\mathcal{C}$ at the boundary nodes, we abstract away from the internal implementation details, i.e., hide all internal nodes of the network such that the data-flow locations of $\mathcal{A}$ and $\mathcal{A}_\mathcal{C}$ agree. To show that $\mathcal{A}$ and $\mathcal{A}_\mathcal{C}$ exihibit the same behavior, we consider the stronger claim that $\mathcal{A}$ and $\mathcal{A}_\mathcal{C}$ are isomorphic for the relevant, reachable fragment.

**Lemma 1** (Soundness of the construction)**.** *Let $\mathcal{A}$ be a constraint automaton and let $\mathcal{C}$ be the constructed Reo network with constraint automaton product $\mathcal{A}_\mathcal{C}$ as above. Then, the Reo network $\mathcal{C}$ correctly implements $\mathcal{A}$ as the reachable fragments of the constraint automata $\mathcal{A}_\mathcal{C}$ and $\mathcal{A}$ are isomorphic.*

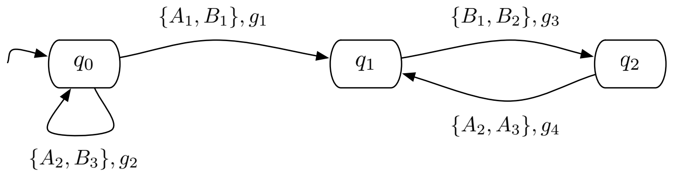


Figure 5: Example of a constraint automaton to be synthesized

**Proof sketch:** The state space of the constraint automaton $\mathcal{A}_\mathcal{C}$ is the Cartesian product of the states of the FIFO1 channels $f_q$ that store the token in $\mathcal{C}$, i.e., $Q_\mathcal{C} = \{\texttt{empty}, \texttt{full}\}^n$, where $n = |Q|$ is the number of states in the original constraint automaton $\mathcal{A}$. The token game starts with a single FIFO1 channel being full and passes the token to exactly one other FIFO1 channel in each step. We denote by $Q'_\mathcal{C} \subseteq Q_\mathcal{C}$ the states of $\mathcal{A}_\mathcal{C}$ where exactly one FIFO1 channel is full, i.e., the states that are relevant for the token game. By construction, the reachable fragment of $\mathcal{A}_\mathcal{C}$, i.e., the states that can be reached via a finite execution from an initial state, is contained in $Q'_\mathcal{C}$. To relate the states of $\mathcal{A}$ and $\mathcal{A}_\mathcal{C}$, let $h : Q_\mathcal{A} \to Q'_\mathcal{C}$ be the bijection that maps each state $q \in Q_\mathcal{A}$ to the corresponding state $h(q)$, i.e., the state of $\mathcal{A}_\mathcal{C}$ where $f_q$ is `full` while all the other buffers $f_{q'}$ with $q' \neq q$ are `empty`.

Furthermore, we have for all $q, q' \in Q_\mathcal{A}$, $N \subseteq \mathcal{N}_\mathcal{C}$ and $g \in DC(N)$:

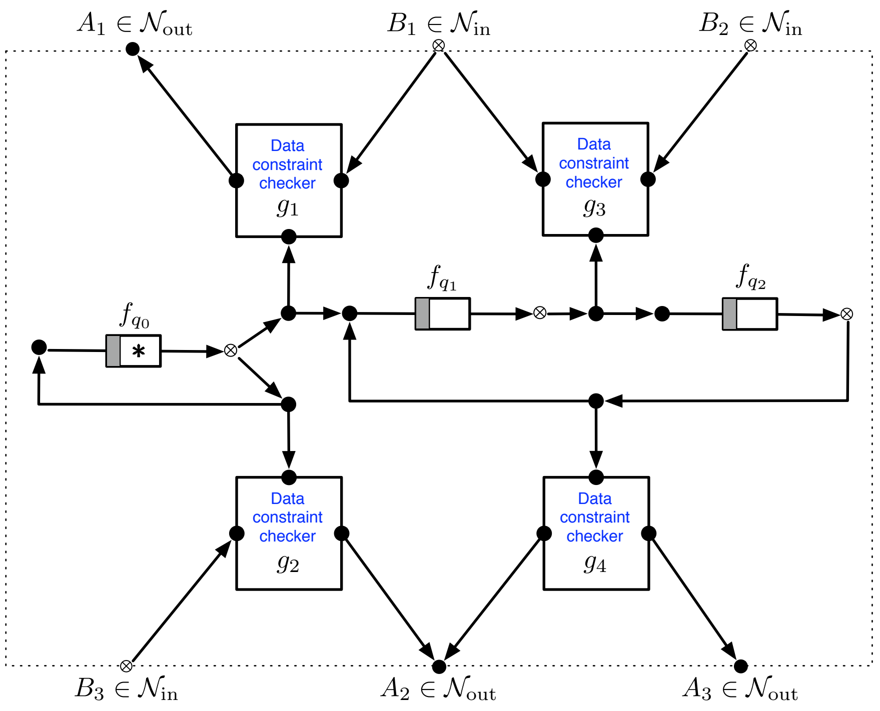
$$h(q) \xrightarrow{N,g}_{\mathcal{A}_\mathcal{C}} h(q') \qquad \text{iff} \qquad q \xrightarrow{N',g'}_{\mathcal{A}} q' \tag{1}$$

where $N' = N \cap \mathcal{N}_\mathcal{A}$ and $g' \equiv \exists[\mathcal{N}_\mathcal{C} \backslash \mathcal{N}_\mathcal{A}]g$, i.e., where $N', g'$ corresponds to $N, g$ after all the internal nodes $A$ in $\mathcal{C}$, $A \in \mathcal{N}_\mathcal{C} \setminus \mathcal{N}_\mathcal{A}$, have been hidden.

To verify equation (1), one has to apply the product construction for constraint automata to all channels and component connectors (exclusive routers, replicators, mergers, and constraint checkers) that appear in the Reo network $\mathcal{C}$. We conclude that, for a state $q$ in $\mathcal{A}$ and the corresponding state $h(q)$ in $\mathcal{A}_\mathcal{C}$, the same concurrent I/O-operations are enabled in $q$ and $h(q)$ after hiding the internals of $\mathcal{A}_\mathcal{C}$. Thus the reachable fragments of the automaton $\mathcal{A}_\mathcal{C}$ for the Reo network $\mathcal{C}$ and the constraint automaton $\mathcal{A}$ are isomorphic after "hiding" all internals of $\mathcal{C}$, i.e., after applying the hide operator for constraint automata to $\mathcal{A}_\mathcal{C}$ to hide the nodes $\mathcal{N}_\mathcal{C} \setminus \mathcal{N}_\mathcal{A}$ only occurring in $\mathcal{A}_\mathcal{C}$. □

**Example 1** (Synthesis)**.** *We close this section on the synthesis of a Reo network from a constraint automaton by an example. Our starting point is the constraint automaton $\mathcal{A} = (Q, \mathcal{N}, \mathcal{N}_{\text{in}}, \mathcal{N}_{\text{out}}, \longrightarrow, \{q_0\})$ with state space $Q = \{q_0, q_1, q_2\}$, the set of data-flow locations $\mathcal{N} = \{A_1, A_2, A_3, B_1, B_2, B_3\}$, and transition relation as depicted in Fig. 5. The data-flow locations are disjointly partitioned into the set of output ports $\mathcal{N}_{\text{out}} = \{A_1, A_2, A_3\}$ and the set of input ports $\mathcal{N}_{\text{in}} = \{B_1, B_2, B_3\}$.*

*The constructed Reo network $\mathcal{C}$ for $\mathcal{A}$ is shown in Fig. 6. As explained in Remark 1 the replicators, mergers and exclusive routers in Fig. 4 have been replaced by the corresponding Reo nodes.* □



$A_1 \in \mathcal{N}_{\text{out}}$
$B_1 \in \mathcal{N}_{\text{in}}$
$B_2 \in \mathcal{N}_{\text{in}}$
Data constraint checker $g_1$
Data constraint checker $g_3$
$f_{q_0}$
$f_{q_1}$
$f_{q_2}$
Data constraint checker $g_2$
Data constraint checker $g_4$
$B_3 \in \mathcal{N}_{\text{in}}$
$A_2 \in \mathcal{N}_{\text{out}}$
$A_3 \in \mathcal{N}_{\text{out}}$

Figure 6: Synthesized Reo network $\mathcal{C}$ for the automaton $\mathcal{A}$ of Fig. 5.

## 5 Conclusion

In this paper, we provide a construction to directly synthesize an equivalent Reo connector from a constraint automaton, facilitating the realization of controllers and strategies in the form of a Reo network. The previous approach in [4] is based on an intermediate translation of the constraint automaton into an $\omega$-regular expression using standard automata to regular expression translation techniques. This expression is then compositionally transformed into a Reo network, by providing networks for the atomic expressions (e.g., data constraints) and compositional operators for the regular expression operators, i.e., concatenation, alternation, the Kleene star (requiring fairness) and the $\omega$-operator. In this construction, each subexpression introduces two FIFO channels to ensure proper synchronization. In contrast, the construction detailed in this paper directly converts the constraint automaton structure into an equivalent network, with only a single FIFO channel representing each state. This avoids the potential exponential blowup in the transformation to $\omega$-regular expressions. Furthermore, the structure of the generated network closely mirrors the structure of the constraint automaton.

## References

[1] M. Abadi, L. Lamport, and P. Wolper. Realizable and unrealizable specifications of reactive systems. In *ICALP'89*, volume 372 of *Lecture Notes in Computer Science*, pages 1–17. Springer, 1989.

[2] R. Alur, T.A. Henzinger, and O. Kupferman. Alternating-time temporal logic. *Journal of the ACM*, 49(5):672–713, 2002.

[3] F. Arbab. Reo: A Channel-Based Coordination Model for Component Composition. *Mathematical Structures in Computer Science*, 14(3):329–366, 2004.

[4] F. Arbab, C. Baier, F. de Boer, J. Rutten, and M. Sirjani. Synthesis of Reo Circuits for Implementation of Component-Connector Automata Specifications. In *Proceedings of the 7th International Conference on Coordination Models and Languages (Coordination '05)*, volume 3454 of *Lecture Notes in Computer Science*, pages 236–251, 2005.

[5] E. Asarin, O. Bournez, T. Dang, O. Maler, and A. Pnueli. Effective Synthesis of Switching Controllers for Linear Systems. *IEEE Special Issue on Hybrid Systems*, 88:1011–1025, 2000.

[6] E. Asarin, O. Maler, and A. Pnueli. Symbolic Controller Synthesis for Discrete and Timed Systems. In *Hybrid Systems II*, volume 131 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 1995.

[7] C. Baier, J. Klein, and S. Klüppelholz. A Compositional Framework for Controller Synthesis. In *CONCUR 2011*, volume 6901 of *Lecture Notes in Computer Science*. Springer, 2011. To appear.

[8] C. Baier, J. Klein, and S. Klüppelholz. Modeling and Verification of Components and Connectors. In *SFM11*, volume 6659 of *Lecture Notes in Computer Science*, pages 114–147. Springer, 2011.

[9] C. Baier, M. Sirjani, F. Arbab, and J. Rutten. Modeling Component Connectors in Reo by Constraint Automata. *Science of Computer Programming*, 61(2):75–113, 2006.

[10] A. Church. Logic, arithmetic, and automata. In *Proc. Int. Congress of Mathematicans*, pages 23–35. Institut Mittag-Leffler, 1962.

[11] L. de Alfaro and T. Henzinger. Interface automata. In *ESEC / SIGSOFT FSE*, pages 109–120, 2001.

[12] S. Klüppelholz and C. Baier. Alternating-time stream logic for multi-agent systems. *Science of Computer Programming*, 75(6):398–425, 2010.

[13] A. Pnueli and R. Rosner. On the Synthesis of a Reactive Module. In *Proceedings of the 16th annual ACM Symposium on Principles of Programming Languages*, pages 179–190. ACM Press, 1989.

[14] W. Wonham. On the control of discrete-event systems. In *Three Decades of Mathematical System Theory*, volume 135 of *Lecture Notes in Control and Information Sciences*, pages 542–562. Springer, 1989.