Project no.: PIRSES-GA-2011-295261

Project full title: Mobility between Europe and Argentina applying Logics to Systems

Project Acronym: MEALS

Deliverable no.: 4.2 / 2

Title of Deliverable: Synthesis of Reo Connectors for Strategies and Controllers

Contractual Date of Delivery to the CEC:	**30-sep-2014**
Actual Date of Delivery to the CEC:	**17-Oct-2014**
Organisation name of lead contractor for this deliverable:	**ULEIC**
Author(s):	**Cristel Baier, Joachim Klein, Sascha Klüppelholz**
Participants(s):	**TUD**
Work package contributing to the deliverable:	**WP4**
Nature:	**R**
Dissemination Level:	**Public**
Total number of pages:	**23**
Start date of project:	1 Oct. 2011 **Duration:** 48 month

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.

Note:

This deliverable is based on material that has been published in Fundamenta Informaticae 130(1):1–20.

This project has received funding from the European Union Seventh Framework Programme (FP7 2007-2013) under Grant Agreement Nr. 295261.

Contents

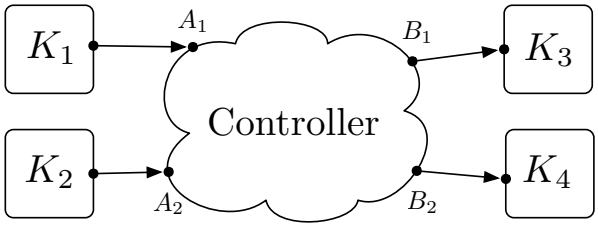


Figure 1: Example scenario with a controller orchestrating four components K_1, K_2, K_3 and K_4

1 Introduction

Synthesis problems have a long tradition in computer science and were first raised by Church [15]. 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, 19, 20, 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.

In this paper, 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. The exogenous approach facilitates flexible (re)use and exchangeability of heterogeneous components in various situations, with their interactions orchestrated by "glue code" in the form of a *component connector*. Such a component connector can be provided by giving an operational description of its behavior, e.g., in the form of an automaton or it can be provided by compositionally building a Reo network out of primitive building blocks, i.e., Reo channels and nodes. The channels and the components themselves may then be deployed in a flexible, distributed manner. Constraint automata [11] serve as an operational semantics for Reo and provide a unified framework capturing both the description of the behavior at the component interfaces as well as the coordination arising from the composition of the Reo primitives in the network.

In this context it is desirable to automatically derive the necessary coordination behavior for a given set of components from a specification of the desired properties of the system. Consider the scenario in Figure 1, where the goal is to find a controller which coordinates four components via their input and output ports. The output of components K_1 and K_2 serves as input for the controller via its input ports A_1 and A_2, while the input ports of components K_3 and K_4 are connected to the output ports B_1 and B_2 of the controller, respectively. For example, the objective for this controller may consist of taking data values produced by the components K_1 and K_2 and passing them to components K_3 and K_4 depending on a set of rules. Given a formalization of the objective, the task is then to automatically synthesize a controller that ensures the given objective. For the Reo and constraint automata framework, the output of algorithms for the synthesis of a controller [18, 9] produce a component connector for the 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 that takes this one step further and, given such a constraint automaton $\mathcal{A}$, synthesize an equivalent Reo network C. The controller thus ceases to be a monolithic automaton and becomes a Reo connector network that integrates with the existing glue code and is then amenable to optimizations and other operations acting on Reo networks, e.g., distributed deployment, graphical visualization, dynamic reconfiguration, etc.

Scenarios. Our construction is for example applicable in the context of the game logic alternating-time stream logic (ASL) for the Reo and constraint automata framework as proposed in [18, 17]. ASL, a variant of the alternating-time temporal logic (ATL) [2], allows reasoning about strategies for coalitions of components. Here, constraint automata are interpreted as multi-player game structures. Our starting point is a set of components, where a subset K of the components are controllable.

We are interested in strategies for the controllable components in K to cooperate in such a way that they achieve a common goal, e.g., to ensure some property, that their composition will never deadlock (similar to the notion of interface compatibility in [16]) or that the components meet each others interface constraints. Here, a strategy means that the controllable components can restrict and even refuse to perform certain or even any actions or participate to synchronize actions with other components or the environment in which they are involved. To reason over K-strategies, ASL provides existential and universal quantification over strategies of the form $\mathbb{E}_K\,\varphi$ and $\mathbb{A}_K\,\varphi$, where φ is an ASL path formula that formalizes the common goal of the coalition. The intuitive semantics for formulas of the form $\mathbb{E}_K\,\varphi$ asserts that there exists a K-strategy such that all remaining paths of the composite system do satisfy the path formula φ. The ASL state formula $\mathbb{A}_K\,\varphi$ asserts that for all K-strategies the remaining set of paths in the composite system contains at least one path that satisfies φ.

In [18, 17] we have seen that finite-memory K-strategies are sufficient and can be represented as a constraint automaton which we can put in parallel with the components (and the orchestrating network) such that the composite system satisfies the coalitions goal. 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 [9] to the Reo and constraint automaton setting. There, in a partial information and partial control setting, we show how to synthesize controllers that enforce certain linear-time properties in the system in a compositional manner. As the automata formalism used in [9] is quite general, it can be applied to the Reo and constraint automata framework in a natural fashion. In particular, the partial information and partial control aspects readily correspond to the exogenous Reo approach. For example, in Figure 1, the controller can only orchestrate the components via the input and output ports A_1, A_2, B_1 and B_2, i.e., by observing the activity at these ports and by exerting control over the activity at those ports that are considered controllable. The controller thus has to infer knowledge about the internal configuration of the components from the available observations of their activity. As the controller is furthermore limited to the manipulation of the port activity and data flow at the controllable ports, it can only influence the behavior of the system in an exogenous manner. In

particular, the controller has no direct ability to manipulate the internals of the components, e.g., by forcing a component to resolve its local non-determinism in a favorable manner.

To allow a compositional approach of iteratively treating the individual parts of a conjunctive objective $\varphi_1 \wedge \varphi_2 \wedge \ldots \wedge \varphi_k$, the constructed controllers are *most-general* in the sense that they capture *all possible* ways in which a given property can be enforced in the system. Thus, given an automaton $\mathcal{A}$ capturing the behavior of the system and its environment as well as a notion of controllability and visibility and multiple objectives $\varphi_1, \ldots, \varphi_k$ that are compatible, i.e., can be enforced in conjunction, it becomes possible to construct a most-general controller C_1 that enforces objective φ_1 for $\mathcal{A}$, a most-general controller C_2 that enforces objective φ_2 for the parallel composition $C_1 \parallel \mathcal{A}$, etc. In the end, the parallel composition of all controllers with the system $C_1 \parallel \ldots \parallel C_k \parallel \mathcal{A}$ enforces the conjunction of the objectives, $\varphi_1 \wedge \ldots \wedge \varphi_k$.

Each of the synthesized controllers can be regarded as a constraint automaton with some additional fairness constraints on the local controller choices, which are crucial for the construction of most-general controllers. These fairness conditions can be regarded as strong fairness conditions of the form "if the controller infinitely often visits a controller state q, then, in state q, it will infinitely often offer certain choices". Importantly, these fairness conditions only constrain the choices of the controller and do not impose any requirements on the actions of the controllable or uncontrollable components or the environment.

Naturally, when the controller automaton is synthesized as a Reo network using the construction presented in this paper, 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 handled in a natural fashion either by specifying such fairness conditions on the level of a Reo network or by adding appropriate, simple circuitry to the network relying on *fair router nodes*, i.e., nodes in the network that choose the routing of the data items in the network in a fair way.

Outline and Contribution. In Section 2, we briefly describe the relevant main concepts of constraint automata and Reo. In Section 3, we then present algorithms for the synthesis of component connectors (Reo networks) from automata-based specifications (constraint automata). We present two different approaches for the synthesis of a Reo network. The first approach consists of a compositional construction, based on the concepts presented by the first author in [4] and used here with slight modifications. This approach relies on a translation of the automaton to an (ω)-regular expression and the subsequent compositional construction of a Reo network from the expression by providing networks for the atoms and the operators used in the expression.

The second approach we present consists of a monolithic construction. Reusing some ideas of the first approach, the construction relies on a direct translation from the automaton to a Reo network, thus avoiding the potential exponential blow-up in the construction of an ω-regular expression. Furthermore, the monolithic construction ensures bisimulation equivalence of the automaton and the constructed network.

2 Preliminaries

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

2.1 Constraint automata

Constraint automata (CA) [11] 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 a 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. Each element $A \in \mathcal{N}$ stands for a data-flow location where I/O can occur, such as the interface ports of components or nodes in the connector network. 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 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, restricting the possible data items at the data-flow locations in N. Formally, data constraints are propositional formulas built from the atoms "$d_A = d$" and "$d_A = d_B$", where $A, B \in \mathcal{N}$ and $d \in$ Data. "$d_A = d$" means that data item $d \in$ Data occurs at data-flow location A, while "$d_A = d_B$" indicates that the data items observed at data-flow locations A and B agree.

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 $d \in$ Data. For a subset $N \subseteq \mathcal{N}$, we denote the set of data constraints over N by $DC(N)$, i.e., the data constraints where all atoms are of the form "$d_A = d$" or "$d_A = d_B$" with $A, B \in N$. Other standard propositional operators such as conjunction ($\wedge$) or implication ($\rightarrow$) can be derived as usual. $d_A \neq d$ stands for $\neg(d_A = d)$ and $d_A \in D$ with $D \subseteq$ Data stands for $\bigvee_{d \in D} d_A = d$. Depending on the concrete instance of Data, other predicates can be derived, e.g., such as $<$ or $\leq$. □

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 $Q \times 2^N \times DC(\mathcal{N}) \times Q$,
- $Q_0 \subseteq Q$ is a (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 $A \in N$. □

The subsets $\mathcal{N}_{\text{in}}$ and $\mathcal{N}_{\text{out}}$ of $\mathcal{N}$ characterize the data-flow locations that are available for an external connection. The data-flow locations $\mathcal{N}_{\text{in}}$ allow data items to be received from the outside, while the data-flow locations $\mathcal{N}_{\text{out}}$ are used to pass data items to the outside. This distinction is important 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.

As there may be a multitude of data values that satisfy a given 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.

Definition 3 (Concurrent I/O-operations (CIO))**.** A concurrent I/O-operation is a partial function assigning data values to the data-flow locations, i.e., a function $c : \mathcal{N} \to \mathsf{Data} \cup \{\perp\}$, where the symbol $\perp$ means "undefined". We write $\mathit{active}(c)$ for the set of active data-flow locations $A \in \mathcal{N}$ with $c(A) \in \mathsf{Data}$. $\mathsf{CIO}_{\mathcal{N}}$, or briefly CIO, denotes the set of all concurrent I/O-operations. The set of concurrent I/O-operations consistent with a transition label (N, g) is then defined as:

$$\mathsf{CIO}(N, g) \stackrel{\text{def}}{=} \{c \in \mathsf{CIO} : \mathit{active}(c) = N \ \wedge \ c \models g\},$$

where $c \models g$ stands for the obvious satisfaction relation which results from interpreting the data constraint g over the data assignments given by c. □

An *execution* in a constraint automaton is then a finite or infinite alternating sequence of states and concurrent I/O-operations representing the steps of the automaton.

Definition 4 (Executions, paths, I/O-streams)**.** Let $\mathcal{A} = (Q, \mathcal{N}, \mathcal{N}_{\text{in}}, \mathcal{N}_{\text{out}}, \longrightarrow, Q_0)$ be a constraint automaton. An *execution* in $\mathcal{A}$ is a finite or infinite sequence

$$\eta \ = \ q_0 \xrightarrow{c_1} q_1 \xrightarrow{c_2} \ldots$$

where $q_0 \in Q_0$ and for all $i \geq 1$, $q_i \in Q$, $c_i \in \mathsf{CIO}$, and $c_i \in \mathsf{CIO}(N, g)$ for some transition $q_i \xrightarrow{N,g} q_{i+1}$ in $\mathcal{A}$, i.e., where each step from state to state is a concurrent I/O-operation in the automaton.

A *path* of $\mathcal{A}$ is then a *maximal* execution, i.e., either an infinite execution or a finite execution $\eta \ = \ q_0 \xrightarrow{c_1} q_1 \xrightarrow{c_2} \ldots \xrightarrow{c_n} q_n$ such that q_n is terminal, i.e., there is no CIO c that can be used to go from q_n to a successor state.

The notion of an *I/O-stream* corresponds to action sequences in labeled transition systems. The I/O-stream $\mathsf{ios}(\eta)$ of a path η is the finite or infinite word over CIO that is obtained by taking the projection to the labels of the transitions. Formally, if $\eta \ = \ q_0 \xrightarrow{c_1} q_1 \xrightarrow{c_2} q_2 \ldots$ is a path in $\mathcal{A}$ then $\mathsf{ios}(\eta) \stackrel{\text{def}}{=} c_1 c_2 \ldots \in \mathsf{CIO}^{\infty}$, where CIO^{∞} denotes the union of the set of finite words CIO^* and the set of infinite words CIO^{ω} over the alphabet CIO, as usual. The language of an automaton $\mathcal{A}$ then consists of the I/O-streams of all paths of $\mathcal{A}$. □

Example 1. As an example, reconsider the scenario presented in Figure 1. Let us assume that the objective for the controller is to accept data values from component K_1 via the data-flow location

A_1 and from K_2 via A_2 when available and to synchronously pass them on alternatingly to K_3 (via B_1) or K_4 (via B_2). Hence, the first data value read via A_1 or A_2 is passed to B_1, the second data value is passed to B_2, the third is passed again to B_1 and so on. If there is data available at both A_1 and A_2 at the same time, the controller non-deterministically chooses to read either from A_1 or A_2. We additionally assume that the data values are integers and that the objective is to ensure that only data with a value less than 4 is read from A_1 and only data with a value less than 3 is read from A_2. Figure 2 shows a constraint automaton $\mathcal{A}$ realizing such a controller. □

2.2 The exogenous coordination language Reo

We provide here a brief overview of the main concepts of Reo, for more details we refer to [3, 11, 10]. Reo is a channel-based, exogenous coordination language. It allows to specify the coordination glue for 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 can enter a channel or a sink end, through which data leaves a channel. Reo nodes are formed when channel ends are joined together. The operational semantics of a Reo network can be provided in a compositional way using constraint automata for the channels and the nodes and an appropriate composition operator (product construction) on constraint automata for the Reo join operation.

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, Figure 3 shows three basic channels and their constraint automata representation. The synchronous channel (cf. Figure 3a) synchronizes activity at its source end and its sink end, transferring the data item from the source end A to the sink end B. The filter channel (cf. Figure 3b) behaves like the synchronous channel if the transferred data item matches the filter condition $D \subseteq \mathsf{Data}$ and blocks otherwise. In this article, we utilize the blocking variant of the Reo filter channel rather than the "lossy" variant, which accepts all data items at A but only passes those data items to its sink end B if they match the filter condition, "losing" the data item otherwise. Both variants can be easily derived from the other using a small Reo network consisting of primitive channels. The FIFO1 channel (cf. Figure 3c) is an example of an asynchronous channel. It receives a single data item via source end A, stores it and subsequently provides the item at sink end B.

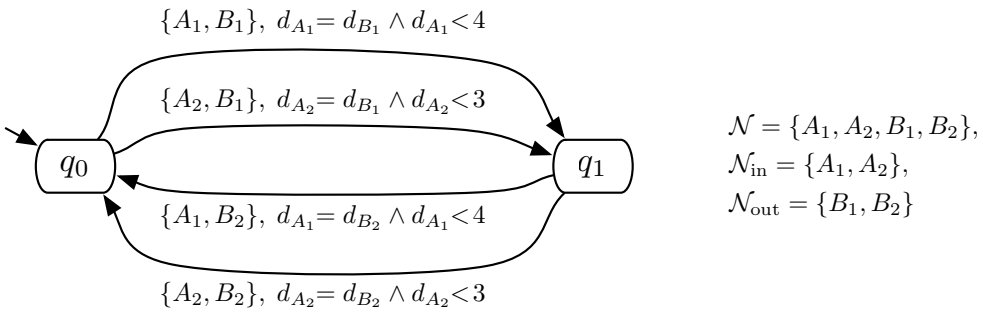
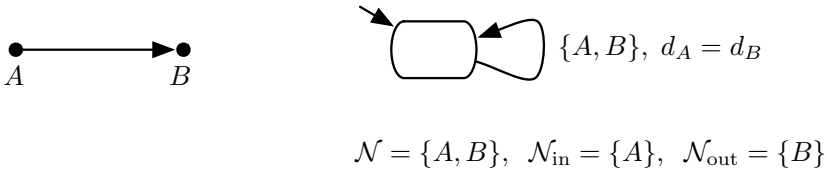
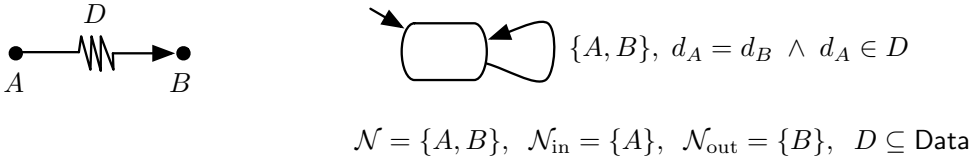


Figure 2: Example constraint automaton $\mathcal{A}$ for coordinating four components (see Figure 1)

a) Synchronous channel



b) Filter channel



c) FIFO1 channel

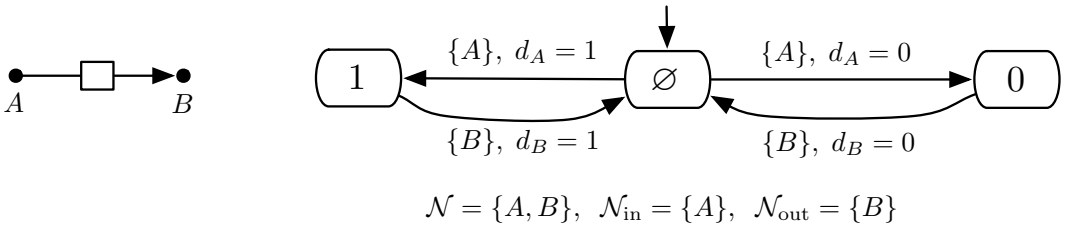


Figure 3: Basic Reo channels and the corresponding constraint automata. For the FIFO1 channel, the constraint automaton is shown for Data = {0, 1}.

To combine various channels into 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, a Reo node performs 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 type. For *standard Reo nodes* (depicted as ●), the data item is copied and output simultaneously to *all* the connected source ends (replication). *Route nodes* (depicted as ⊗) forward the data item to *exactly one* of the connected source ends (routing), chosen non-deterministically. While we treat route nodes here as primitives, they can as well be derived from a *n*-ary router connector using only Reo nodes with the standard replication semantics and a small set of primitive channels [3]. Similarly to the channels, the operational semantics of the Reo nodes can be captured by constraint automata.

As an example, consider Figure 4 where the controller in Figure 1 has been replaced by a Reo network, orchestrating the four components. The Reo node N_1 non-deterministically chooses between data items provided by the components K_1 and K_2 via A_1 and A_2. The data item is then buffered in the FIFO1 channel. The route node N_2 then routes the data item to exactly one of the components K_3 or K_4. Which of the components receives the data item is determined by the condition of the filter channels. If the data item matches condition D then the route node can

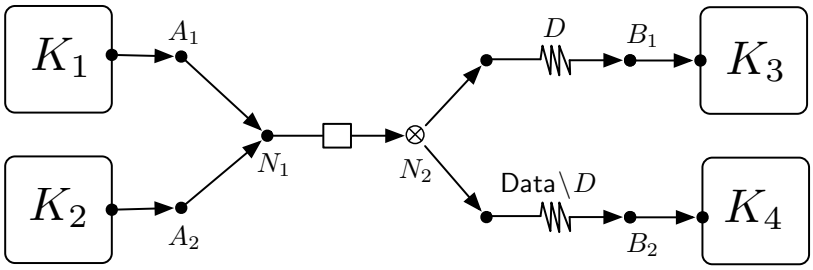


Figure 4: Example of a Reo network orchestrating four components K_1, K_2, K_3 and K_4

only pass the data item to component K_3. In the other case, if the data item does not match D then the route node can only pass the data item to K_4. With different, overlapping conditions on the filter channels, the route node would decide non-deterministically between routing to K_3 and K_4 for the data items that satisfy both filter channel conditions.

Constraint automata product and hiding. To compositionally obtain the constraint automaton representing a Reo network, a product construction [11, 10] for the individual constraint automata of all constituent parts of the network can be used, i.e., the constraint automaton for the composed system arises from the product of the constraint automata of the channels, the constraint automata of the Reo nodes and the constraint automata of the components in the system. The product synchronizes over the activity at the shared data-flow locations of the constraint automata.

To facilitate hierarchical modeling and to abstract away from implementation details of a given Reo network, a *hiding operator* on constraint automata can be used, which removes certain internal data-flow locations from the constraint automaton. In the example depicted in Figure 4, the data-flow locations N_1 and N_2 correspond to the internal nodes of the Reo network. Hiding these locations in the constraint automaton for the composed system then abstracts from the concrete structure of the Reo network orchestrating the components.

In particular, one is often interested in the behavior at the data-flow locations corresponding to the interface ports, i.e., the data-flow locations $A \in \mathcal{N}_{\text{in}} \cup \mathcal{N}_{\text{out}}$. Hiding all data-flow locations $A \in \mathcal{N} \setminus (\mathcal{N}_{\text{in}} \cup \mathcal{N}_{\text{out}})$ thus abstracts from internal data-flow locations. After such hiding, a transition involving only internal data-flow locations, i.e., an "internal" step, then corresponds to a transition with a transition label N, g where $N = \emptyset$.

Definition 5 (Hiding for CA)**.** Let $\mathcal{A} = (Q, \mathcal{N}, \mathcal{N}_{\text{in}}, \mathcal{N}_{\text{out}}, \longrightarrow_{\mathcal{A}}, Q_0)$ be a constraint automaton. The result of hiding a data-flow location $A \in \mathcal{N}$ from $\mathcal{A}$ is the constraint automaton

$$\mathcal{A}' = (Q,\ \mathcal{N} \setminus \{A\},\ \mathcal{N}_{\text{in}} \setminus \{A\},\ \mathcal{N}_{\text{out}} \setminus \{A\},\ \longrightarrow_{\mathcal{A}'}, Q_0).$$

The transition relation $\longrightarrow_{\mathcal{A}'}$ is given by:

$$\frac{q \xrightarrow{N,g}_{\mathcal{A}} p}{q \xrightarrow{N\setminus\{A\},\exists[A]g}_{\mathcal{A}'} p} \qquad \text{where } \exists[A]g = \bigvee_{d \in \mathsf{Data}} g[d_A/d].$$

Here, we write $g[d_A/d]$ to denote the data constraint obtained from g by syntactically replacing all occurrences of atoms of the form "$d_A = d'$" by "true" if $d = d'$ and by "$\neg$true" if $d \neq d'$, as well as replacing the atoms of the form "$d_A = d_B$" for some $B \in \mathcal{N}$ with "$d_B = d$". □

If one is only interested in the streams of concurrent I/O-operations that can be observed at the data-flow locations $A \in \mathcal{N}_{\text{in}} \cup \mathcal{N}_{\text{out}}$, it is possible to further abstract from such "internal" steps by considering only the sequences of CIOs observable at the boundary locations. A (finite or infinite) I/O-stream $\nu \in \mathsf{CIO}^\infty$ is *observable at the boundary locations* of $\mathcal{A}$ if there exists a path $\eta = q_0 \xrightarrow{c_1} q_1 \xrightarrow{c_2} \ldots$ in $\mathcal{A}$ such that ν is obtained from the CIOs $c_1, c_2, \ldots$ of η by (1) hiding the data-flow locations $A \notin (\mathcal{N}_{\text{in}} \cup \mathcal{N}_{\text{out}})$ and (2) removing those CIOs where no data-flow location is active, i.e., $N = \varnothing$. We will use hiding in the comparison of the behavior of the constraint automaton and the Reo network generated from it.

3 Realization of a constraint automaton by a Reo network

In this section, we 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. This is a natural restriction, as controllers generally do not have such internal locations.

We first describe the building blocks used in the construction of the Reo network. Subsequently, we present two approaches for the synthesis of a Reo network, a compositional approach relying on an intermediate translation of a constraint automaton into an ω-regular expression and a monolithic approach, providing a direct translation from a constraint automaton to a Reo network.

Basic channels and component connectors. In addition to the standard Reo channels as detailed in Figure 3, the monolithic construction requires a variant of the standard FIFO1 channel that can output the currently stored data item via its sink end while simultaneously accepting a new data item to store via its source end. Figure 5 shows the graphical representation of a such a *simultaneous FIFO1 channel* together with its constraint automaton.

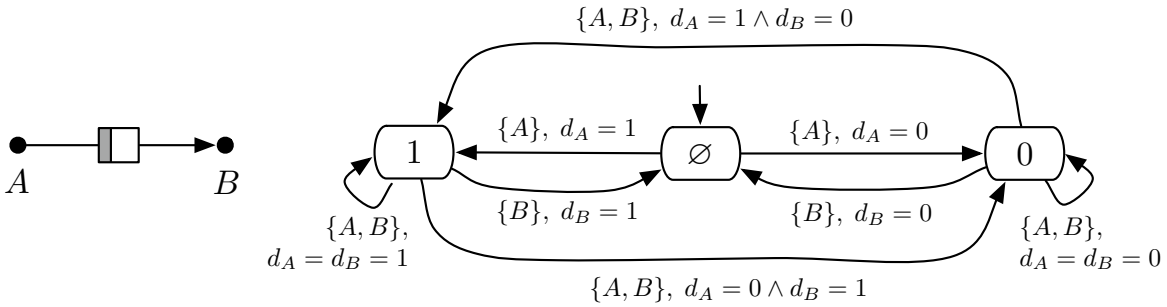


Figure 5: Simultaneous FIFO1 channel for data domain $\mathsf{Data} = \{0, 1\}$

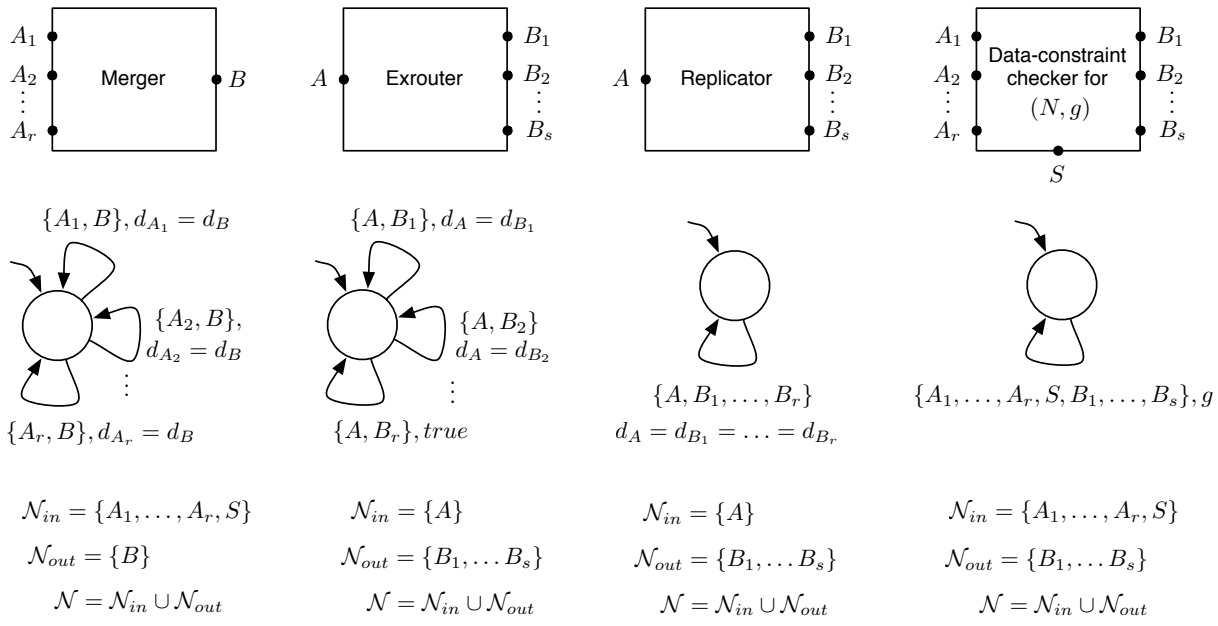


Figure 6: Merger, exclusive router, replicator and data-constraint checker

In addition to the channels, both constructions rely on the use of four component connectors that are used as primitive building blocks:

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 (N, g), with $g \in DC(N)$ and having input ports $A_1, \ldots, A_r \in N$ and output ports $B_1, \ldots, B_s \in N$, as well as a special input port S for triggering. 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.

Figure 6 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 a Reo network by using the standard Reo nodes (●) for the *merger* and *replicator* and route nodes (⊗) for the *exclusive router*. We nevertheless first treat them as

component connectors to provide a clear description of their intended function. In the special case that there are zero input ports for a merger or zero output ports for a replicator or exclusive router, the merger/replicator/exclusive router component connector blocks, as there is no possibility to receive or transmit a data item.

In this paper, we treat the constraint checkers for a given (N, g) as primitives as well. As shown 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, with a size linear in the size of N and g. Likewise, the simultaneous FIFO1 channel could be derived by a Reo network consisting only of primitive channels. □

Connecting the data-constraint checkers. The data-constraint checkers for a data constraint N, g serve as the primitive building blocks in both approaches to ensure data flow at ports N with the data matching the guard g. They are connected to the corresponding output ports $N \cap \mathcal{N}_{\text{out}}$ and input ports $N \cap \mathcal{N}_{\text{in}}$ in such a way that if a given data-constraint checker N, g is triggered via its port S then exactly the ports N are active, the ports $\mathcal{N} \setminus N$ are inactive and that furthermore at any given point of time at most one of the connected data-constraint checkers is active. Figure 7 shows the schema how the data-constraint checkers are connected to the ports in $\mathcal{N}_{\text{in}} \cup \mathcal{N}_{\text{out}}$. For

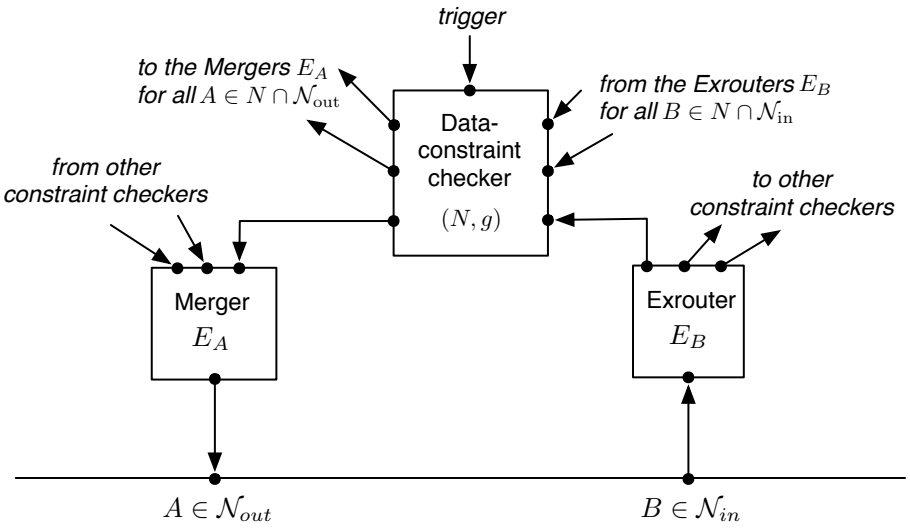


Figure 7: Connecting the data-constraint checkers to the input ports $\mathcal{N}_{\text{in}}$ and output ports $\mathcal{N}_{\text{out}}$.

each output port $A \in \mathcal{N}_{\text{out}}$ the Reo network contains a merger MGR_A. Dually, for each input port $B \in \mathcal{N}_{\text{in}}$ we deal with an exclusive router EXR_B, which are connected to the various data-constraint checkers where the set of active data-flow locations N contains A or B, respectively.

The semantics of the merger and exclusive router then ensure that exactly one connected data-constraint checker is active at the same time, while the synchronous channel to the actual input (from the merger) or output port (to the exclusive router) ensures that the port can only be

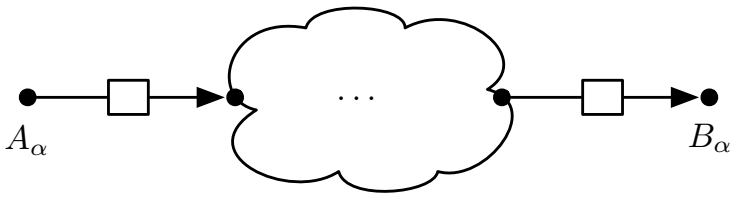


Figure 8: General structure of the Reo network used in the compositional construction.

active if the data item can be transferred to some matching data-constraint checker.

3.1 Compositional synthesis

We now present a compositional approach for the construction of a Reo network from an automaton, based on the concepts in [4]. The construction preserves the language, i.e., the set of I/O-streams observable at the boundary data-flow locations and relies on a two step approach. In a first step, the constraint automaton $\mathcal{A}$ is translated into an I/O-stream expression, an ω-regular expression over the port activity and data flow at the data-flow locations of $\mathcal{A}$. This can be seen as an expression defining the I/O-streams observable at the boundary data-flow locations $\mathcal{N}_{\text{in}} \cup \mathcal{N}_{\text{out}}$. In a second step, the expression is compositionally translated into a Reo network by providing Reo networks for the atoms (activity at data-flow locations and data constraints) as well as Reo networks that compose given component connectors using the (ω-)regular operators, e.g., concatenation, union and repetition. These compositional operations are rather general and can also be used in other contexts to compose Reo networks by applying these operations to component connectors.

As a first step, the automaton is translated into an I/O-stream expression. An I/O-stream expression α is an ω-regular expression, built from ε for the language consisting of the empty I/O-stream, the atoms $\langle N, g\rangle$ with $\emptyset \neq N \subseteq \mathcal{N}$ and $g \in DC(N)$ a satisfiable data constraint for N representing a single step where the data flow locations N are active and the data satisfies g, as well as the standard operators ; (concatenation), $\cup$ (union), α^ω (infinite repetition) and α^∞ (finite or infinite repetition).

Applying standard techniques for the generation of (ω-)regular expressions from automata, it is then possible to construct an I/O-stream expression α from the constraint automaton $\mathcal{A}$ such that the language of α corresponds to the set of I/O-streams observable at the boundary data-flow locations $\mathcal{N}_{\text{in}} \cup \mathcal{N}_{\text{out}}$ of $\mathcal{A}$. Given such an I/O-stream expression $\alpha_{\mathcal{A}}$, the construction then consists of compositionally building a Reo network for $\alpha_{\mathcal{A}}$ by providing networks R_α for all the subexpressions α of $\alpha_{\mathcal{A}}$.

Figure 8 shows the general structure of the Reo networks used in the compositional approach, with the part in the middle representing a given compositional description of a component connector. The basic idea is to play a token game, where a token is passed around to signify the activation and completion of the subconnectors of the Reo network, in our case the Reo networks representing the various subexpressions. Reception of a token at A_α thus corresponds to the "activation" of subexpression α, with the completion of the handling for the subexpression

signaled by passing the token back via B_α. The two FIFO1 buffers serve to isolate the subexpression network from the rest of the network, as well as to ensure (together with the specific circuitry of the various operators) that there is only a single token passed around. The network $R_{\alpha_\mathcal{A}}$ corresponding to the whole I/O-stream expression $\alpha_\mathcal{A}$ generated from $\mathcal{A}$ is then augmented with a simple component that dispenses a single token to $R_{\alpha_\mathcal{A}}$ to activate the whole network and a component to accept the token once handling of $R_{\alpha_\mathcal{A}}$ is finished.

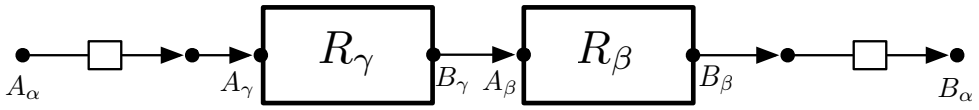


Figure 9: Reo network R_α for $\alpha = \gamma;\beta$ (concatenation)

The *concatenation* of two subexpressions, $\alpha = \gamma;\beta$, is realized as shown in Figure 9 by connecting B_γ of the first subexpression to A_β of the second subexpression, passing the token from R_γ to R_β once R_γ is finished.

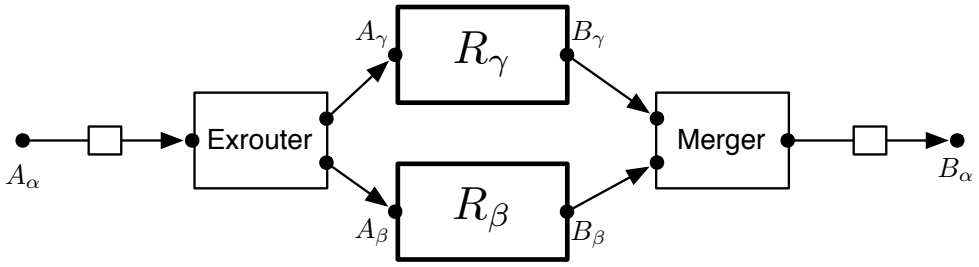


Figure 10: Reo network R_α for $\alpha = \gamma \cup \beta$ (union)

The *union* of two subexpressions, $\alpha = \gamma \cup \beta$, is realized as shown in Figure 10 by an exclusive router which non-deterministically chooses whether to pass the token to either R_γ or R_β. A merger then accepts the token once the chosen subexpression network is finished.

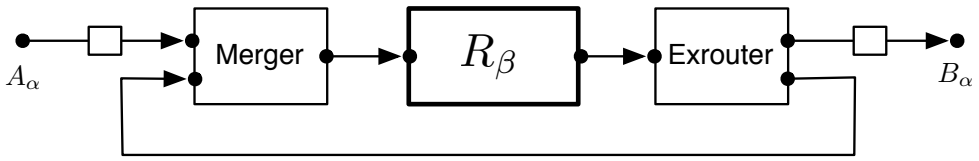


Figure 11: Reo network R_α for $\alpha = \beta^\infty$ (finite or infinite repetition)

Finite or infinite repetition of a subexpression, $\alpha = \beta^\infty$, is realized as shown in Figure 11 by first passing the token to R_β. Once the token is passed back by R_β, an exclusive router non-deterministically chooses whether the repetition is finished or not. If the repetition is finished, the token is passed to the FIFO1 buffer, where it can subsequently be passed via B_α. In the other case, the token is passed back to R_β for another repetition.

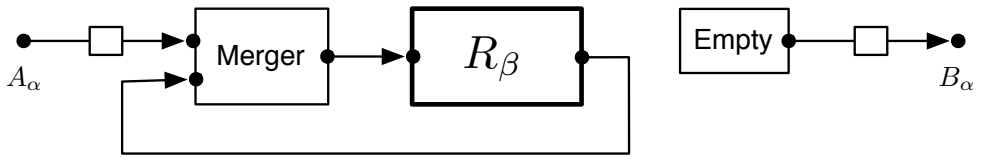


Figure 12: Reo network R_α for $\alpha = \beta^\omega$ (infinite repetition)

Infinite repetition, $\alpha = \beta^\omega$, is then a simple variant of the previous operator, with the token always passed back to R_β as shown in Figure 12. A simple component connector prohibiting any data flow is attached to the source end of the FIFO1 on the right to ensure that no token can ever be passed via B_α. This "Empty" component connector can be straightforwardly realized by a constraint automaton with a single state and no outgoing transitions or by a single primitive Reo channel (a *synchronous spout channel* [3]) where both ends of the channel are attached to the same Reo node, inhibiting any data-flow.

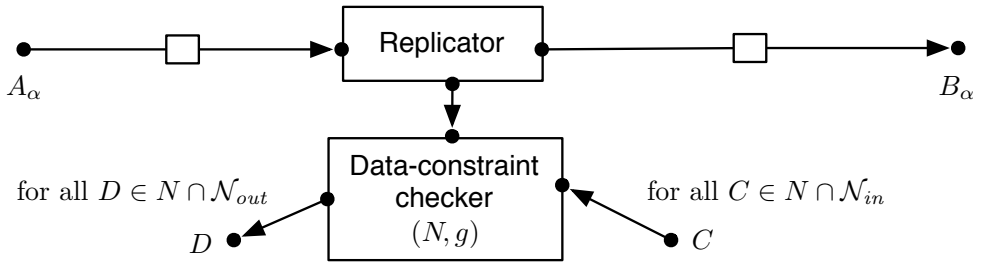


Figure 13: Reo network R_α for $\alpha = \langle N, g \rangle$ (data flow at ports N, satisfying data constraint g)

The atoms $\alpha = \langle N, g \rangle$ are realized by Reo networks of the form depicted in Figure 13. The token is passed from the FIFO1 on the left to the FIFO1 on the right via a replicator, which passes a copy of the token to the data-constraint checker for (N, g), which ensures that exactly the data-flow locations in N are active and that the data flow at these locations satisfies the data constraint g. Note that the copy of the token is solely used to trigger the constraint checker and then immediately destroyed, such that only a single token can leave the network.

The remaining atom, $\alpha = \varepsilon$, the language consisting of the empty I/O-stream, is realized in a straightforward manner by directly connecting the sink end of the FIFO1 channel on the left responsible for the reception of the token (see Figure 8) to the source end of FIFO1 channel on the right, allowing the token to pass through the network without triggering any subexpression network or data-constraint checker.

Size of the Reo network. The translation from $\mathcal{A}$ to an I/O-stream expression α may lead to an exponential blow-up from the size of $\mathcal{A}$. In the second step, the number of elements in the Reo network obtained for α then is linear in the size of α.

Example 2 (I/O-stream expression and Reo network). Consider again the constraint automaton

$\mathcal{A}$ from Figure 2. For $\mathcal{A}$, we obtain the I/O-stream expression $\alpha = (\alpha_1; \alpha_2)^\omega$ with

$$\begin{aligned} \alpha_1 &= \langle \{A_1, B_1\}, d_{A_1} = d_{B_1} \ \wedge \ d_{A_1} < 4 \rangle \quad \cup \quad \langle \{A_2, B_1\}, d_{A_2} = d_{B_1} \ \wedge \ d_{A_2} < 3 \rangle \\ \alpha_2 &= \langle \{A_1, B_2\}, d_{A_1} = d_{B_2} \ \wedge \ d_{A_1} < 4 \rangle \quad \cup \quad \langle \{A_2, B_2\}, d_{A_2} = d_{B_2} \ \wedge \ d_{A_2} < 3 \rangle \end{aligned}$$

Figure 14 shows the Reo network for the subexpression α_1, where the merger, replicator and

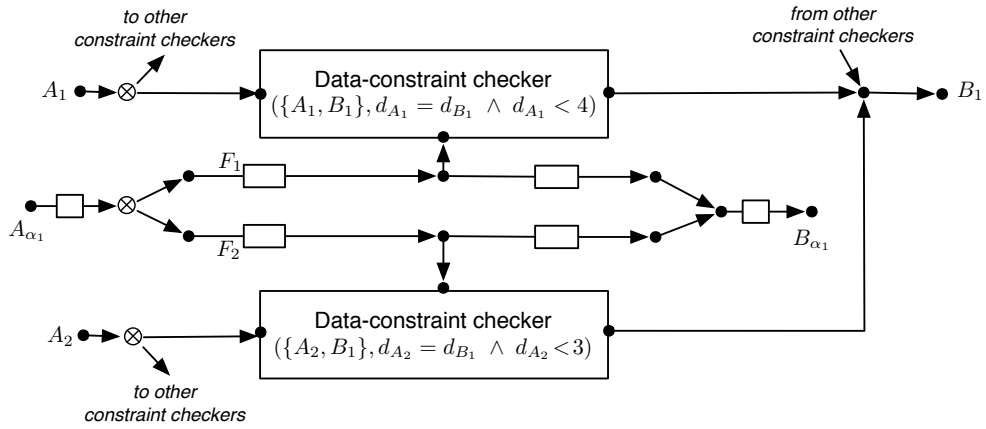


Figure 14: Reo network for $\alpha_1 = \langle \{A_1, B_1\}, d_{A_1} = d_{B_1} \ \wedge \ d_{A_1} < 4 \rangle \cup \langle \{A_2, B_1\}, d_{A_2} = d_{B_1} \ \wedge \ d_{A_2} < 3 \rangle$

router component connectors have been replaced with the corresponding Reo nodes. Once the token enters at A_{α_1}, the route node on the left non-deterministically chooses whether the first or the second subexpression should be activated and passes the token to the corresponding FIFO1 channel F_1 or F_2. The token then moves to the corresponding FIFO1 channel on the right, triggering the data-constraint checker and subsequently leaving the sub network via B_{α_1}. □

While the Reo network obtained by the translation via I/O-stream expressions does provide the same I/O-streams observable at the boundary data-flow locations as the automaton $\mathcal{A}$, the internal steps introduced by the token flowing through the various FIFO1 channels for the subexpressions can lead to differences when the automaton $\mathcal{A}$ is replaced by the Reo network. This can be seen, e.g., when considering the network from Figure 14. In the case that F_1 is full, the exclusive router has chosen to activate the network of the first subexpression. The network is thus committed to the first subexpression and the token can only move if a data item with value less than 4 is available at data-flow location A_1 and can be passed to B_1. If there is never such a data item available at A_1, then the network will block forever, even if appropriate data items at A_2 are available. This differs from the behavior of the constraint automaton $\mathcal{A}$ for the controller, as there data items can be passed as long as either A_1 or A_2 have appropriate items available.

The compositional translation via I/O-stream expressions thus only preserves the *linear-time* behavior of $\mathcal{A}$, i.e., it preserves the language, the possible I/O-streams observable at the boundary data-flow locations, but not necessarily the *branching* behavior. Depending on the objective and structure of the controller, the preservation of the linear-time behavior might be sufficient. In other cases, such as [9] where the controller simultaneously offers multiple choices, the preservation of the branching behavior is crucial. In such cases, the monolithic approach which ensures

bisimulation equivalence of the automaton and the synthesized Reo network presented in the next section can be used. Furthermore, the monolithic approach avoids the potential exponential blow-up that can occur due to the intermediate translation to an I/O-stream expression. Nevertheless, the presented construction provides a general approach for the compositional construction of Reo networks from component connectors using the standard (ω-)regular composition operators.

3.2 Monolithic synthesis

We now present a second, monolithic construction for the synthesis of a Reo network from a constraint automaton $\mathcal{A} = (Q, \mathcal{N}, \mathcal{N}_{\text{in}}, \mathcal{N}_{\text{out}}, \longrightarrow, Q_0)$, which produces a Reo network linear in the size of $\mathcal{A}$ and ensures bisimulation equivalence of the constructed network and the automaton $\mathcal{A}$.

Again, 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. We furthermore assume that there is a single, unique starting state, i.e, $Q_0 = \{q_0\}$. A non-deterministic choice between multiple starting states can be handled with circuitry to realize the initial non-deterministic choice and the first step of the automaton similar to the circuitry used below.

Similar to the approach described in Section 3.1, we construct a Reo network implementing a *token game*, where a single token flows through the network, with the goal of mimicking the behavior of $\mathcal{A}$. But instead of playing the token game on the I/O-stream expressions, the Reo network C constructed in this section directly mimics the constraint automaton $\mathcal{A}$. For this, the Reo network C that realizes $\mathcal{A}$ represents each state $q \in Q$ by a simultaneous FIFO1 channel f_q with a single buffer cell, with the token residing in the buffer of f_q representing that $\mathcal{A}$ is in state q. The token game consists of the token flowing from one buffer to the next and triggering a data-constraint checker on the way according to the transitions of $\mathcal{A}$. 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 $f_{q'}$. As there may be self-loops on states in $\mathcal{A}$, i.e., $q' = q$, we employ simultaneous FIFO1 channels instead of the standard FIFO1 channels, as the simultaneous FIFO1 channels are capable of sending the token and receiving the token in a single step. The structure of the construction is shown in Figure 15. For state q in $\mathcal{A}$, we deal with the simultaneous FIFO1 channel f_q and an exclusive router EXR_q and a merger MGR_q. The exclusive router EXR_q has one output port for each transition emanating in q. The merger MGR_q has one input port for each transition ending in q. For each transition θ in $\mathcal{A}$ there is a replicator REP_θ and a data-constraint checker DCC_θ.

Each of the simultaneous FIFO1 channels f_q is connected to the corresponding exclusive router EXR_q. The exclusive router "schedules" non-deterministically one of the outgoing transitions θ and routes the token through the corresponding output port into the replicator REP_θ. The first task of this replicator is to forward the token towards the simultaneous FIFO1 $f_{q'}$. The merger $\text{MGR}_{q'}$ ensures that only one of the incoming transitions of state q' can fire at a time. The second task of the replicator REP_θ is to synchronize the transition with the data-constraint checker DCC_θ. Thus, the transition $\theta = (q, N, g, q')$ can fire if and only if all ports in N fire

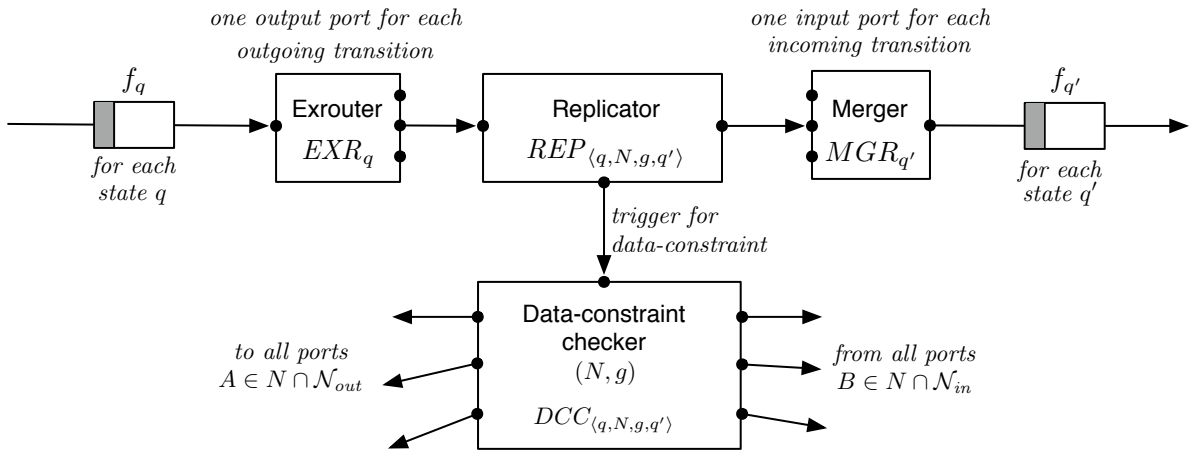


Figure 15: Structure of a Reo network C synthesized from constraint automaton $\mathcal{A}$

synchronously and the observed data fulfills the data constraint $g \in DC(N)$.

Size of the construction. The size of the constructed Reo network C is linear in the size of $\mathcal{A}$, as there is a simultaneous FIFO1 channel for each state of $\mathcal{A}$, one data-constraint checker for each transition in $\mathcal{A}$ and the number of the other channels and nodes connecting the FIFO1 channels and data-constraint checkers is likewise linear in the number of transitions in $\mathcal{A}$.

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 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 C. We denote the constraint automaton resulting from the product of all component connectors, channels and nodes of C by $\mathcal{A}_C = (Q_C, \mathcal{N}_C, \mathcal{N}^C_{\text{in}}, \mathcal{N}^C_{\text{out}}, \longrightarrow_C, Q_{0,C})$. The set $\mathcal{N}_C$ contains all internal nodes and boundary nodes (i.e., ports) of C, while $\mathcal{N}^C_{\text{in}} = \mathcal{N}^{\mathcal{A}}_{\text{in}}$ and $\mathcal{N}^C_{\text{out}} = \mathcal{N}^{\mathcal{A}}_{\text{out}}$. To allow a comparison of the behavior of $\mathcal{A}$ and $\mathcal{A}_C$ 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 $\mathcal{A}_C$ agree. To show that $\mathcal{A}$ and $\mathcal{A}_C$ exhibit the same behavior, we consider the stronger claim that $\mathcal{A}$ and $\mathcal{A}_C$ are isomorphic for their relevant, reachable fragments.

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

Proof. The state space of the constraint automaton $\mathcal{A}_C$ is the Cartesian product of the states of the FIFO1 channels f_q that store the token in C, i.e., $Q_C = \{\texttt{empty}, \texttt{full}\}^n$, where $n = |Q|$ is the number of states in the original constraint automaton $\mathcal{A}$. The token game starts with a single FIFO1 channel (f_{q_0}) being full and in each step, the token is passed to exactly one other FIFO1 channel.

We denote by $Q'_C \subseteq Q_C$ the states of $\mathcal{A}_C$ where exactly one FIFO1 channel is full, i.e., the states that are relevant for the token game. By construction, the reachable fragment of $\mathcal{A}_C$, i.e., the states that can be reached via a finite execution from an initial state, is contained in Q'_C. To relate the states of $\mathcal{A}$ and $\mathcal{A}_C$, let $h : Q_{\mathcal{A}} \to Q'_C$ be the bijection that maps each state $q \in Q_{\mathcal{A}}$ to the corresponding state $h(q)$, i.e., the state of $\mathcal{A}_C$ where f_q is `full` while all the other buffers $f_{q'}$ with $q' \neq q$ are `empty`.

Furthermore, we have a one-to-one correspondence between transitions in $\mathcal{A}$ and transitions in A_C, i.e., for all $q, q' \in Q_{\mathcal{A}}$, $N \subseteq \mathcal{N}_C$ and $g \in DC(N)$:

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

where $N' = N \cap \mathcal{N}_{\mathcal{A}}$ and $g' \equiv \exists[\mathcal{N}_C \setminus \mathcal{N}_{\mathcal{A}}]g$, i.e., where (N', g') corresponds to (N, g) after all the internal nodes A in C, $A \in \mathcal{N}_C \setminus \mathcal{N}_{\mathcal{A}}$, have been hidden. This follows from the construction and can be shown by applying the product construction for constraint automata to the various channels and component connectors (exclusive routers, replicators, mergers and data-constraint checkers) appearing in the Reo network C.

We conclude that, for a state q in $\mathcal{A}$ and the corresponding state $h(q)$ in $\mathcal{A}_C$, the same concurrent I/O-operations are enabled in q and $h(q)$ after hiding the internals of $\mathcal{A}_C$. Thus, the reachable fragments of the automaton $\mathcal{A}_C$ for the Reo network C and the constraint automaton $\mathcal{A}$ are isomorphic after "hiding" all internals of C, i.e., after applying the hide operator for constraint automata to $\mathcal{A}_C$ to hide the nodes $\mathcal{N}_C \setminus \mathcal{N}_{\mathcal{A}}$ only occurring in $\mathcal{A}_C$. □

Example 3 (Reo network, monolithic translation). Reconsider the controller given by the constraint automaton $\mathcal{A}$ in Figure 2. The corresponding Reo network C obtained by the monolithic translation is then shown in Figure 16. The replicator, route and merger component connectors have been replaced by their Reo node counterparts. As $\mathcal{A}$ has two states, the Reo network has two FIFO1 buffers, with the one corresponding to the initial state q_0 of $\mathcal{A}$ initially filled with the token. The route nodes provide the choice between the two alternative transitions per state that can actually be taken. Note that here, in contrast to Figure 14 with the compositional approach, there is no premature internal choice for one of the transitions. Either one of them may be taken as long as there is an appropriate data item available at A_1 or A_2, respectively, and the data item can be written via the corresponding B_1 or B_2 data-flow location.

4 Conclusion

In this article, we provide algorithms to synthesize an equivalent Reo component connector from a constraint automaton, facilitating the realization of controllers and strategies in the form of a Reo network.

We presented two approaches, a compositional approach relying on an intermediate translation into an I/O-stream expression and the compositional construction of a Reo network for the expression, and a monolithic approach providing a direct translation from a constraint automaton to a Reo network. Both approaches synthesize Reo networks where each part of the network either is a primitive Reo channel or node or can be easily derived from a basic set of primitive

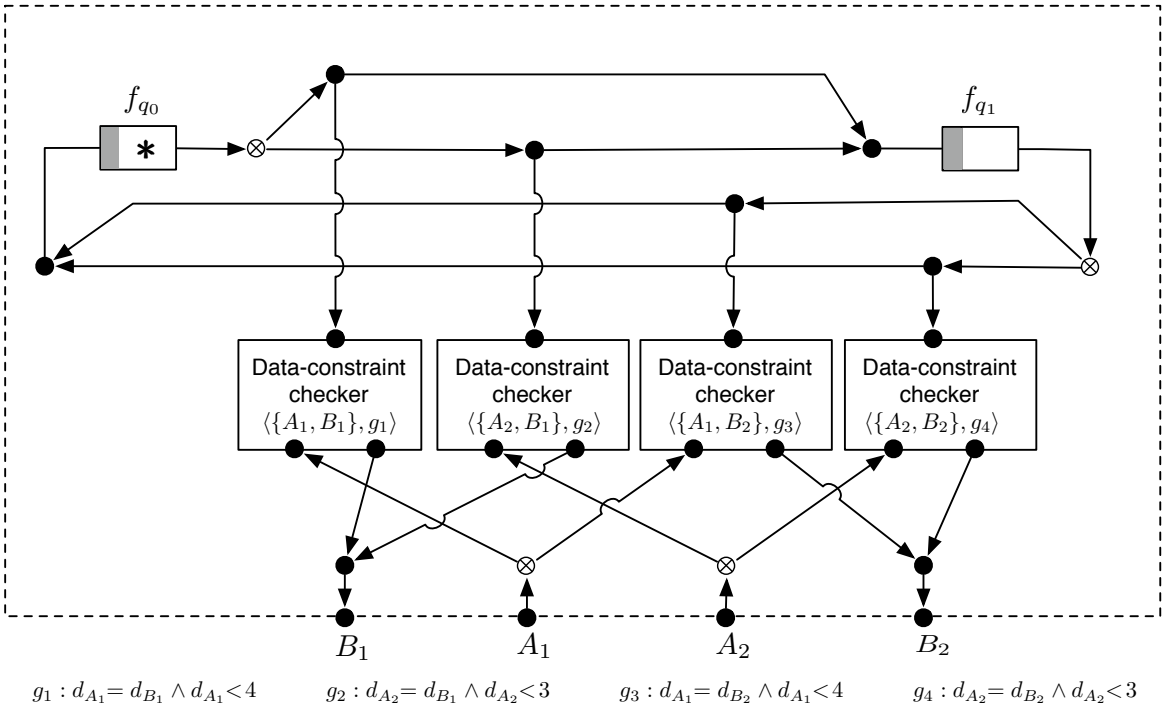


Figure 16: Synthesized Reo network C for the controller given by constraint automaton $\mathcal{A}$ of Figure 2.

Reo channels. While the compositional approach preserves the language of the automaton, i.e., the set of observable I/O-streams, the monolithic approach furthermore ensures the bisimulation equivalence of the automaton and the constructed network. Additionally, the Reo networks resulting from the monolithic construction are linear in the size of the automaton and thus avoid the potential exponential blow-up due to the intermediate translation into an I/O-stream expression in the compositional approach.

We have implemented the monolithic translation procedure in our modeling and verification tool Vereofy [7, 8, 13] for the constraint automaton and Reo framework. The generated Reo networks can then as well be used in the Extensible Coordination Tools (ECT) [14], an Eclipse-based graphical user interface for the manipulation and verification of Reo connectors. Using the bisimulation checker of Vereofy [12, 8], it is further possible to ascertain that the automaton $\mathcal{A}$ and the constructed Reo connector C are indeed equivalent.

Bibliography

[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. 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 *Coordination'05*, volume 3454 of *Lecture Notes in Computer Science*, pages 236–251. Springer, 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, T. Blechmann, J. Klein, and S. Klüppelholz. A Uniform Framework for Modeling and Verifying Components and Connectors. In *Coordination'09*, volume 5521 of *Lecture Notes in Computer Science*, pages 247–267. Springer, 2009.

[8] C. Baier, T. Blechmann, J. Klein, and S. Klüppelholz. Formal Verification for Components and Connectors. In *FMCO'08*, volume 5751 of *Lecture Notes in Computer Science*, pages 82–101. Springer, 2009.

[9] C. Baier, J. Klein, and S. Klüppelholz. A Compositional Framework for Controller Synthesis. In *CONCUR'11*, volume 6901 of *Lecture Notes in Computer Science*, pages 512–527. Springer, 2011.

[10] C. Baier, J. Klein, and S. Klüppelholz. Modeling and Verification of Components and Connectors. In *SFM'11*, volume 6659 of *Lecture Notes in Computer Science*, pages 114–147. Springer, 2011.

[11] 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.

[12] T. Blechmann and C. Baier. Checking equivalence for Reo networks. In *FACS'07*, volume 215 of *Electronic Notes in Theoretical Computer Science*, pages 209–226. Elsevier, 2008.

[13] T. Blechmann, J. Klein, and S. Klüppelholz. Vereofy, Technische Universität Dresden. `http://www.vereofy.de/`.

[14] Centrum Wiskunde & Informatica (CWI) Amsterdam. Extensible Coordination Tools. `http://reo.project.cwi.nl/`.

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

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

[17] S. Klüppelholz. *Verification of Branching-Time and Alternating-Time Properties for Exogenous Coordination Models*. PhD thesis, Technische Universität Dresden, Germany, 2012.

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

[19] 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.

[20] 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.

MEALS Partner Abbreviations

SAU: Saarland University, D

RWT: RWTH Aachen University, D

TUD: Technische Universität Dresden, D

INR: Institut National de Recherche en Informatique et en Automatique, FR

IMP: Imperial College of Science, Technology and Medicine, UK

ULEIC: University of Leicester, UK

TUE: Technische Universiteit Eindhoven, NL

UNC: Universidad Nacional de Córdoba, AR

UBA: Universidad de Buenos Aires, AR

UNR: Universidad Nacional de Río Cuarto, AR

ITBA: Instituto Técnológico Buenos Aires, AR