Journal of Visual Languages and Computing ▮ (▮▮▮▮) ▮▮▮–▮▮▮

Contents lists available at SciVerse ScienceDirect

Journal of Visual Languages and Computing

journal homepage: www.elsevier.com/locate/jvlc

Compositional and behavior-preserving reconfiguration of component connectors in Reo

Christian Krause [a,1,*], Holger Giese [a], Erik de Vink [b]

[a] Hasso Plattner Institute for Software Systems Engineering, Prof.-Dr.-Helmert-Str. 2–3, D-14482 Potsdam, Germany
[b] Department of Mathematics & Computer Science, Eindhoven University of Technology, P.O. Box 513, 5600 MB Eindhoven, The Netherlands

ARTICLE INFO

This paper has been recommended for acceptance by Shi Kho Chang

Keywords:
Coordination languages
Reconfiguration
Behavior-preservation
Semantics
Category theory

ABSTRACT

It is generally accepted that building software out of loosely coupled components, such as in service-oriented systems or mobile networks, yields applications that are more robust against changes and failure of single components than monolithic systems. In order to accommodate for changes in the environment or in the requirements, and anticipate to a component failure, applications are often dynamically adapted by means of a reconfiguration. In this paper, we target the visual channel-based coordination language Reo and introduce a combined structural and behavioral model for graph-based component connectors in Reo. Exploiting concepts from category theory, we model reconfigurations of connectors as transformations of the underlying connector graphs. We show that our connector model has a compositional semantics and lift structural reconfigurations to the semantical level. As a concrete application of our framework, we introduce a notion of behavior-preserving reconfiguration for Reo and provide a sufficient condition to ensure behavior-preservation statically.

1. Introduction

A common approach in software engineering is based on the idea of dividing a system into two orthogonal aspects: (i) the computation performed by a set of (black-boxed) components or services, and (ii) their coordination using specific 'glue code'. Engineering systems using this principle has the advantage that there is clear separation between the encapsulated functional behavior implemented in the components on the one hand, and the description of their allowed interactions in the form of communication protocols on the other.

In order to be able to reason about such systems or to (semi-) automatically derive implementations of them, models with precise semantics are vital. While the specification of the components or services is usually based on behavioral models, e.g. automata or process algebraic descriptions, the composition and the interaction of these functional building blocks is often described using graphical models, e.g. various kinds of Petri nets or structural component connector models.

In this paper, we specifically target component connectors in the channel-based coordination language Reo [1]. In Reo, the coordination of components and services is realized using arbitrarily complex connectors which are represented by graphs that consist of a set of communication channels connected by nodes. While the structure of component connectors is graph-based, their semantics can be given in terms of automata, e.g. *constraint automata* [2].

In practice, however, the complexity of the modeled systems and changes in their requirements or protocols demand to adapt them at runtime. Adjusting a component connector to accommodate for changes in the protocol or to reflect new goals, can be achieved by means of a

* Corresponding author. Tel.: +49 3315509525.
E-mail address: christian.krause@hpi.uni-potsdam.de (C. Krause).
[1] Supported by a research grant of the Hasso Plattner Institute for Software Systems Engineering.

1045-926X/$ - see front matter © 2012 Elsevier Ltd. All rights reserved.
http://dx.doi.org/10.1016/j.jvlc.2012.09.002

Please cite this article as: C. Krause, et al., Compositional and behavior-preserving reconfiguration of component connectors in Reo, Journal of Visual Languages and Computing (2012), http://dx.doi.org/10.1016/j.jvlc.2012.09.002

reconfiguration. Often, reconfiguration not only involves the changing of the runtime parameters of single channels or components, but also requires structural modifications at the level of the component connectors.

In a promising line of research, the theory of graph transformation is used to model reconfigurations at the structural level of systems, e.g. reconfigurations of component connectors [3], Petri nets [5], mobile networks [6] and software architectures [7]. In all these areas, graphs are used because they provide a both natural and formal representation of the structure of the systems in terms of their topology. The idea of graph transformation is to rewrite substructures using matched transformation rules. Such rules provide a syntax to specify structural preconditions in the form of local contexts of system elements, and moreover allow the engineer to define and execute complex reconfigurations as atomic structural modifications.

When using graphs to model the structure of systems and graph transformation to realize their reconfiguration, one of the major challenges is to predict the impact of a structural modification on the behavioral semantics of the systems at hand. For instance, when dynamically adding new channels or nodes to a component connector in Reo, it is important to understand the effect on its execution semantics. In such situations, it is often crucial to ensure that certain behavioral properties are preserved by the reconfiguration.

1.1. Approach and contributions

In this paper, we consider graph transformation-based reconfiguration of component connectors in Reo, as e.g. employed already in [3,4]. In order to reason about the impact of a structural reconfiguration of a Reo connector on its semantics, we present a combined structural and behavioral model for Reo, called *distributed constraint automata with state memory* (DCASM).

Our model permits to specify the topology of a connector together with the semantics of its constituent channels and components and has moreover enough expressive power to derive implementations of Reo that can be used in practice. Specifically, it permits to specify data constraints over infinite data domains by symbolic representations using memory cells as introduced in [8], and moreover carries relevant structural information, i.e., the topology information of the component connectors.

The framework presented in this paper builds on concepts from category theory which facilitates an abstract and elegant way of defining operations and reasoning about properties. Specifically, we define the composition of connector models in our framework using universal properties and phrase the semantics in terms of a compositional functor, which derives behavioral automata models from graphical connector models.

We facilitate the theory of algebraic graph transformation [9] to model reconfigurations as structural transformations of connector graphs as employed in [3,4]. Due to the compositionality of the semantics functor, we are able to lift a structural reconfiguration of a connector graph to the semantical level, i.e., to its automata semantics. Using this approach, we are able to investigate the impact of a structural change in a connector on its overall behavior. As a concrete application of our framework, we introduce a notion of a behavior-preserving reconfiguration and give a sufficient condition to ensure behavior-preservation statically.

This paper extends the results presented in [10] as follows. We generalize the basic model of distributed port automata in [10] to the more expressive model of distributed constraint automata with state memory. Thereby, data constraints for channels (even over infinite data domains) can be modeled in our framework. We transfer the compositionality result for distributed port automata in [10] to the level of distributed constraint automata with state memory in this paper. Furthermore, we show explicitly how graph transformation can be employed to model structural connector reconfigurations and formally define the induced reconfiguration at the semantical level. To the best of our knowledge, none of the existing models for Reo provides a concept for defining and checking behavior-preserving reconfiguration, as introduced in this paper.

1.2. Overview

In Section 2, we recall relevant notions from category theory. We assume familiarity with the fundamental concepts, such as categories, functors and (co)limits. In Section 3, we discuss the modeling concepts for building component connectors in Reo. Specifically, we recall the notions of channels and nodes and explain how they can be composed to construct complex connector graph models. In Section 4, we define the formal semantics of Reo in terms of constraint automata with state memory (CASM) and introduce a notion of simulation to relate two automata. In Section 5, we present a categorical view of the CASM model where we consider automata as objects and simulations as morphisms between objects. Moreover, we define the parallel composition for CASM in categorical terms, i.e., using a universal property. In Section 6, we recall the basic concepts of distributed graph transformation, which are essential in our categorical framework to model the topology of connectors. In Section 7, we combine the categorical models of Sections 5 and 6 to obtain an integrated structural and behavioral model of component connectors, called *distributed constraint automata with state memory* (DCASM). This model builds on concepts from category theory and constitutes the core of our framework. As one of our central results, we show that the semantics of connectors in this model is compositional, i.e., that a structural gluing of connector graphs corresponds to a parallel composition in the automata semantics. In Section 8, we discuss how the theory of graph transformation can be applied in our framework to model the reconfiguration of component connectors. Reconfigurations are specified at the structural level of component connectors, specifically using graph transformation rules that modify the structure of connectors. The compositional semantics of our DCASM model allows us further to formally define the impact of a structural reconfiguration of a connector on its automata semantics. We exploit this correspondence to define a notion of behavior-preserving reconfiguration. We discuss

Please cite this article as: C. Krause, et al., Compositional and behavior-preserving reconfiguration of component connectors in Reo, Journal of Visual Languages and Computing (2012), http://dx.doi.org/10.1016/j.jvlc.2012.09.002

related work in Section 9 and present our conclusions and possible directions for future work in Section 10.

2. Preliminaries

In the following, we briefly recall notions from category theory which are relevant for the framework presented in this paper. For a comprehensive introduction, we refer to standard category theory textbooks, e.g. [11,12].

Categories. A category $\mathbf{C}$ consists of (i) a class of objects $obj_{\mathbf{C}}$, (ii) a class of morphisms $hom_{\mathbf{C}}(A,B)$ for each pair of objects A and B, and (iii) a binary composition operation $\_\circ\_ : hom_{\mathbf{C}}(A,B) \times hom_{\mathbf{C}}(B,C) \to hom_{\mathbf{C}}(A,C)$. For every object A there must exist a unique morphism $id_A : A \to A$, called the *identity of A*, such that $f \circ id_A = f$ and $id_A \circ g = g$ for all morphisms $f : A \to B$ and $g : B \to A$. Moreover, $\circ$ is required to be associative, i.e., $(f \circ g) \circ h = f \circ (g \circ h)$. The category **Set** has sets as objects, functions as morphisms and function composition as composition operation. The category **Graph** has directed graphs as objects, i.e. tuples $G = (V,E,s,t)$ with set of nodes V, set of edges E, and functions $s,t : E \to V$ assigning a source node $s(e)$ and a target node $t(e)$ to each edge, graph homomorphisms as morphisms, and function composition as composition operator. The category $\mathbf{C}^{\mathrm{op}}$ is obtained from a category $\mathbf{C}$ by reverting all morphisms in $\mathbf{C}$.

Functors & natural transformations. A functor $F : \mathbf{C} \to \mathbf{D}$ (also called a *diagram of shape* $\mathbf{C}$ *in* $\mathbf{D}$) associates to each object $A \in \mathbf{C}$ an object $F(A) \in \mathbf{D}$, and to each morphism $f : A \to B \in \mathbf{C}$ a morphism $F(f) : F(A) \to F(B) \in \mathbf{D}$, such that identity morphisms and composition are preserved. A contravariant functor reverses morphisms, i.e., it maps $f : A \to B$ to $F(f) : F(B) \to F(A)$. For two functors $F,G : \mathbf{C} \to \mathbf{D}$, a natural transformation $\eta : F \to G$ associates to every object $A \in \mathbf{C}$ a morphism $\eta_A : F(A) \to G(A) \in \mathbf{D}$ such that for every $f : A \to B \in \mathbf{C}$ it holds that $\eta_B \circ F(f) = G(f) \circ \eta_A$. An *adjunction* consists of two functors $F : \mathbf{D} \to \mathbf{C}$ and $G : \mathbf{C} \to \mathbf{D}$ together with a family of bijections $\Phi_{X,Y} : hom_{\mathbf{C}}(F(Y),X) \to hom_{\mathbf{D}}(Y,G(X))$ that is natural in X and Y, i.e. $G(f) \circ \Phi_{X,Y}(h) \circ g = \Phi_{X',Y'}(f \circ h \circ F(g))$ for all $f : X \to X'$ in $\mathbf{C}$ and $g : Y' \to Y$ in $\mathbf{D}$.

Limits & colimits. A *cone* for a diagram $F : \mathbf{C} \to \mathbf{D}$ is an object $N \in \mathbf{D}$ together with a family of $\mathbf{D}$-morphisms $y = (y_A)_{A \in \mathbf{C}}$ where $y_A : N \to F(A)$, for all $A \in \mathbf{C}$, such that $F(f) \circ y_A = y_B$ for every $f : A \to B$ in $\mathbf{C}$. A *limit* for F is a cone (N, y) for F such that for any other cone (N',y') for F there exists a unique morphism $h : N' \to N$ such that $y'_A = y_A \circ h$ for every object $A \in \mathbf{C}$. A *colimit* in $\mathbf{C}$ is a limit in $\mathbf{C}^{\mathrm{op}}$. A *pullback* is the limit over a *cospan* (a diagram of the shape $\bullet \to \bullet \leftarrow \bullet$). A *pushout* is the colimit over a *span* (a diagram of the shape $\bullet \leftarrow \bullet \to \bullet$). A functor is called *(co)continuous* if it preserves all (co)limits.

3. Channel-based coordination with Reo

Reo [1][2] is a channel-based coordination language for component-based and service-oriented software systems. Coordination in Reo is performed using circuit-like connectors which are built out of primitive channels and nodes. These connectors coordinate components or services from outside and without their knowledge, which is also referred to as *exogenous* coordination. Reo connectors define and implement the allowed interactions between the active entities in a network by means of communication protocols. This includes aspects of concurrency, buffering, ordering, data flow and manipulation.

In the following, we recall the basic notions of Reo and give some introductory examples.

3.1. Channels

Channels in Reo are entities that have exactly two ends, which can be either *source* or *sink* ends. Source ends accept data into, and sink ends dispense data out of their channel. Reo allows directed channels as well as so-called *drain* and *spout* channels, which have respectively two source and two sink ends. Channels may impose constraints on the data flow at their ends. For instance, they can synchronize or mutually exclude data flow, provide buffering or apply data transformations. Although channels can be defined by users in Reo, a small set of basic channels suffices to implement rather complex coordination patterns.

A set of commonly used channels is given in Table 1. The *Sync* channel consumes data items at its source end and dispenses them at its sink end. The I/O operations are performed synchronously and without any buffering. Consequently, the channel blocks if the party at the sink end is not ready to receive any data. The *LossySync* channel behaves in the same way, except that it does not block the party at its source end. Instead, the data item is consumed and destroyed by the channel if the receiver is not ready to accept it. The *SyncDrain* channel is also a synchronous channel, but it differs in the fact that it has two source ends through which it consumes and destroys data items synchronously. Complementary, the *AsyncDrain* consumes a data item from only one of its source ends and can therefore be used to realize mutual exclusion. None of the channels considered so far buffer data items. Buffering can be implemented using the *FIFO1* channel, which is a directed, asynchronous channel with a buffer of size one. The *Filter* channel uses a data constraint, e.g. a regular expression, to decide whether a data item should be passed to the sink end or destroyed by the channel. Finally, the *Transform* channel applies a function to all data items passing through and can therefore be used for data conversion.

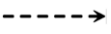
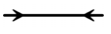
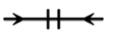
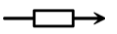
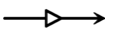
3.2. Nodes

To construct connectors, channels can be joined together using nodes in Reo. A node can be of one out of three types: source, sink or mixed, depending on whether

[2] Reo can also serve as a formal basis for other high-level modeling languages. For example, mappings from business process modeling languages, specifically from BPEL, BPMN, and UML2 sequence diagrams are available [13]. In our earlier work in [14], we have also shown that a simpler variant of the model presented in this paper is already expressive enough to model (reconfigurable) Petri nets with finite capacities.

Please cite this article as: C. Krause, et al., Compositional and behavior-preserving reconfiguration of component connectors in Reo, Journal of Visual Languages and Computing (2012), http://dx.doi.org/10.1016/j.jvlc.2012.09.002

Table 1
Graphical notations of some common Reo channels.

Sync	*LossySync*	*SyncDrain*	*AsyncDrain*	*FIFO1*	*Filter*	*Transform*
						

all coinciding channel ends are source ends, sink ends or a combination of both. Source and sink nodes together form the boundary of a connector, allowing interaction with its environment. A source node acts as a synchronous replicator, i.e., it atomically copies incoming data items to all of its outgoing source ends. On the other hand, a sink node acts as a non-deterministic merger, i.e., it randomly chooses a data item from one of the sink ends for delivery to its connected component. Mixed nodes combine both behaviors by atomically consuming a data item from one sink end and replicating it to all source ends. This can be seen as a $1:n$ synchronization, as opposed to $1:1$ synchronizations (Milner style), or synchronizations of all coinciding channel ends (Hoare style). The choice of the sink end is made non-deterministically. We stress that nodes do not perform any buffering.

3.3. Connectors

In Reo, channels and nodes are joined together to build so-called *connectors* which serve as the glue code between components or services. These connectors coordinate the interactions between the functional building blocks of the system and constrain their behavior to enforce their cooperation.

An important aspect of Reo is that nodes do not buffer data items and, thus, allow synchrony to propagate through a connector. For instance, a connector consisting of an arbitrary long sequence of *Sync* channels joined together using nodes has the same qualitative behavior as a single *Sync*. However, in general Reo allows an arbitrary mixing of synchrony and asynchrony. Moreover, data-aware channels, such as *Filter* and *Transform* can be used to realize connectors whose behavior is influenced by the communicated data and which can apply data conversions as well.

Remark 1 (*Boundary nodes*). For clarity, we depict boundary nodes as open circles and mixed nodes as filled circles in all diagrams.

Example 1. Fig. 1 depicts a component connector in Reo for a simple instant messenger application. Two client components exchange messages via a connector. Messages are sent into *FIFO*1 channels and are thus buffered. When they leave the buffer, they are synchronously replicated by the nodes behind the *FIFO*1 s and sent to both clients. This can succeed only if both clients are ready to accept data. In a nutshell, this connector ensures that clients get – as an acknowledgment – a copy of their own message when the other client has successfully received it.

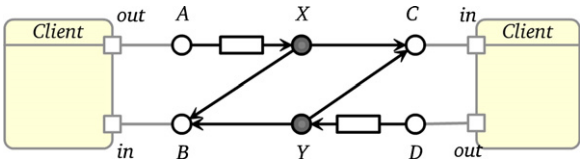


Fig. 1. A simple instant messenger application in Reo.

4. Constraint automata semantics for Reo

Among others, semantics of Reo connectors can be given using *constraint automata with state memory* (CASM) [8,18]. In the following, let **Data** denote a global, possibly infinite data domain. In CASM, states can be enriched with local memory cells. These memory cells can be seen as symbolic and thus finite representation for the data elements in **Data**. Communication and synchronization in CASM is realized using port names. Intuitively, a component can write or read data items to the public ports of a connector, formally represented by its constraint automaton with state memory. Such data items can be stored in memory cells and later updated or output via another port of the connector. Also, data stored in the memory cells of a CASM can influence, via the data constraints in which it can be used, the transitional behavior of the modeled connector. Note also that these automata are expressive enough to derive executable coordinator models from connectors, e.g. in the form of generated Java code in the Reo implementation in the Extensible Coordination Tools (ECT) [15,14].

4.1. Data constraints

We first introduce the language for data constraints, used later as guards on transitions in constraint automata with state memory.

Definition 1 (*Data value*). Let a finite set of port names P and a finite set of memory cells $M = M_S \cup M_T$ be given. The set of *data values over P and M*, written as $\mathbf{DV}(P,M)$, is defined by the grammar

$$v ::= d \mid \mathbf{d}(p) \mid \mathbf{s}.x \mid \mathbf{t}.y$$

where d ranges over **Data**, p over P, x over M_S and y over M_T.

A data value is either (i) a constant $d \in \mathbf{Data}$, (ii) a data item $\mathbf{d}(p)$ currently observed at the port $p \in P$, (iii) the content $\mathbf{s}.x$ of the memory cell $x \in M_S$ in the source state of a transition, or (iv) the content $\mathbf{t}.y$ of the memory cell $y \in M_T$ in the target state of a transition.

Please cite this article as: C. Krause, et al., Compositional and behavior-preserving reconfiguration of component connectors in Reo, Journal of Visual Languages and Computing (2012), http://dx.doi.org/10.1016/j.jvlc.2012.09.002

Definition 2 (*Data constraint*). The set of *data constraints* over the set of port names P and set of memory cells M, written as $\mathbf{DC}(P,M)$, is defined by the grammar

$$g ::= \bot \mid \top \mid \neg g \mid g \wedge g \mid g \vee g \mid (v = v)$$

where v ranges over $\mathbf{DV}(P,M)$.

Equality of data values, together with *falsum* $\bot$ and *verum* $\top$, are the primitive ingredients of data constraints. Furthermore, data constraints can be negated and combined by conjunction and disjunction, where we use the standard Boolean laws to define equivalence of data constraints. We define the preorder $\leq$, which we interpret as logical implication, by putting $g_1 \leq g_2 \Leftrightarrow (g_1 \wedge g_2) \equiv g_1$.

Intuitively, a data constraint is a (composite) expression which allows to compare data values. We use data constraints as guards of transitions—the data constraint should evaluate to true for the transition to be enabled. Note that using the data values $\mathbf{s}.x$ and $\mathbf{t}.y$, the memory cell x of the source, and the memory cell y of the target state of the transition can be inspected. Typically, we interpret a constraint $\mathbf{t}.y = \mathbf{d}(p)$ as a write operation where the data item at p is stored in y. Analogously, we interpret a constraint $\mathbf{d}(p) = \mathbf{s}.x$ as a read operation where a component receives at port p the data value currently stored in x. For brevity, we also write $\mathbf{d}_p$ for $\mathbf{d}(p)$.

4.2. Constraint automata with state memory

Now, we formally define the notion of a constraint automaton with state memory. Our definition is in compliance with the ones given in [8,18].

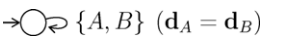
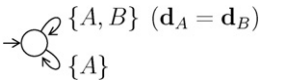
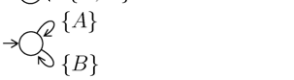
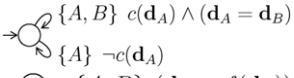
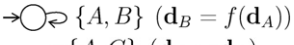
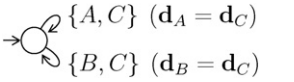
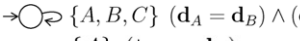
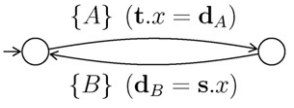
Definition 3 (*Constraint automaton with state memory*). A *constraint automaton with state memory* $\mathcal{A} = (Q,P,M,T,\mu,\theta,Q^0)$ consists of

- a finite set of states Q with $Q^0 \subseteq Q$ a non-empty set of initial states,
- a finite set of port names P,
- a finite set of memory cells M,
- a finite set of transitions T,
- a memory cell location function $\mu : M \to Q$,
- a transition function $\theta : T \to Q \times 2^P \times \mathbf{DC}(P,M) \times Q$, such that for every $t \in T$ with $\theta(t) = \langle q,F,g,q' \rangle$, it holds that $g \in \mathbf{DC}(F,M')$ where $M' = M_S \cup M_T$ with $M_S = \mu^{-1}(q)$ and $M_T = \mu^{-1}(q')$.

For a transition t with $\theta(t) = \langle q,F,g,q' \rangle$, we usually write $q \xrightarrow{F,g} q'$ with $q,q' \in Q$ the source and target state of the transition, $F \subseteq P$ the set of synchronously firing ports, and g the enabling data constraint or *guard*. Note that this guard can refer only to firing ports, i.e. the ports in F, and the memory cells of the source and target states, i.e. M_S and M_T, respectively. For the operational semantics of CASM we refer the reader to [18].

The constraint automata with state memory for the most common channel types in Reo are summarized in Table 2. Note that for the directed *Sync* channel, the data item observed at its source end and its sink end must be identical, whereas for the *SyncDrain* any two data items can be read from its two source ends. We also include two primitives with three ports each: the *Merger* and the *Replicator*. Having three ports, these primitives are no channels. They can be used to compositionally construct the behavior of nodes (cf. [19]). Replicators essentially synchronize all ends, whereas mergers implement a mutual exclusion of the incoming ends. All modeled primitives except for the *FIFO*1 are stateless, i.e., they have only one state.

Table 2
Constraint automata with state memory for some common Reo primitives.

Sync(A,B)	
LossySync(A,B)	
SyncDrain(A,B)	
AsyncDrain(A,B)	
Filter(A,B)	
Transform(A,B)	
Merger(A,B,C)	
Replicator(A,B,C)	
*FIFO*1*(A,B)*	

The automaton for the *FIFO*1 channel illustrates the use of state memory. The transition via the firing set $\{A\}$ and the guard $(\mathbf{t}.x = \mathbf{d}_A)$ models a data flow at the port A and the fact that data item observed at A is stored in the memory cell x of the state representing the full buffer on the right. Similarly, the transition via the firing set $\{B\}$ and the guard $(\mathbf{d}_B = \mathbf{s}.x)$ represents the data flow out of the buffer to the port B. The intuition is that the data item observed for this transition, captured in the memory cell x, should be the same as the one observed at the port A before.

4.3. Simulations

To relate different constraint automata with state memory, we next introduce a notion of simulation. A simulation between two constraint automata with state memory essentially consists of a structure-preserving mapping of states, transitions, port names and memory cells. However, typically we assume that the port names of the target automaton form a subset of the port names of the source automaton. Therefore, we define the mapping of port names in the opposite direction, i.e. from port names of the target to port names of the source automaton. Moreover, we ensure consistency of firing events using conditions on the transitions of the automata. We first need to introduce a technical definition.

Please cite this article as: C. Krause, et al., Compositional and behavior-preserving reconfiguration of component connectors in Reo, Journal of Visual Languages and Computing (2012), http://dx.doi.org/10.1016/j.jvlc.2012.09.002

Definition 4 (*Substitution*). Given a data value $v \in \mathbf{DV}$ (P_1, M_1) and two functions $f_P : P_2 \to P_1$ and $f_M : M_1 \to M_2$. The set of data values $v[f_P,f_M] \subseteq \mathbf{DV}(P_2,M_2)$ is defined as

$$d[f_P,f_M] = \{d\}$$
$$\mathbf{d}(p_1)[f_P,f_M] = \{\mathbf{d}(p_2) \mid f_P(p_2) = p_1\}$$
$$\mathbf{s}.m_1[f_P,f_M] = \{\mathbf{s}.f_M(m_1)\}$$
$$\mathbf{t}.m_1[f_P,f_M] = \{\mathbf{t}.f_M(m_1)\}$$

Given a data constraint $g \in \mathbf{DC}(P_1,M_1)$, the data constraint $g[f_P,f_M] \in \mathbf{DC}(P_2,M_2)$ is obtained from g by substituting all subconstraints of the form '$(v_1 = v_1')$' by the conjunction

$$\bigwedge\{(v_2 = v_2') \mid v_2 \in v_1[f_P,f_M], v_2' \in v_1'[f_P,f_M]\}$$

Note that a constraint $(v_1 = v_1')$ is replaced by an empty conjunction (equivalent to $\top$) if at least one of the sets $v_1[f_P,f_M]$ or $v_1'[f_P,f_M]$ is empty. This may occur for data values of the form $\mathbf{d}(p)$ where p is not in the image of f_P. Now we are in the position to define our notion of simulation for constraint automata with state memory.

Definition 5 (*Simulation*). Let $\mathcal{A}_i = (Q_i,P_i,M_i,T_i,\mu_i,\theta_i,Q_i^0)$, where $i \in \{1,2\}$, be two constraint automata with state memory. A simulation $f : \mathcal{A}_1 \to \mathcal{A}_2$ is a tuple $f = (f_Q, f_P,f_M,f_T)$ consisting of four functions $f_Q : Q_1 \to Q_2$, $f_P : P_2 \to P_1$, $f_M : M_1 \to M_2$ and $f_T : T_1 \to T_2$, such that

1. $f_Q(Q_1^0) = Q_2^0$
2. $\mu_2 \circ f_M = f_Q \circ \mu_1$
3. if $f_T(t_1) = t_2$, with $\theta_1(t_1) = \langle q_1,F_1,g_1,q_1' \rangle$ and $\theta_2(t_2) = \langle q_2,F_2,g_2,q_2' \rangle$, then the following conditions hold: (i) $f_Q(q_1) = q_2$ and $f_Q(q_1') = q_2'$, (ii) $f_P^{-1}(F_1) = F_2$, and (iii) $g_1[f_P,f_M] \leq g_2$.

Note that ports names, as in Definition 4, are mapped in the opposite direction, i.e. $f_P : P_2 \to P_1$. Requirement 1 ensures that the initial states are preserved. Requirement 2 ensures that the mapping of states and the mapping of memory cells are compliant with the memory locations. Finally, requirement 3 ensures consistency of the mapped transitions, i.e., the simulation (i) connects source and target states, (ii) preserves firing of ports, and (iii) maps a guard g_1 to a guard g_2 only if g_1 implies g_2, modulo substitution of port names and memory cells.

If there exists a simulation between two constraint automata with state memory $\mathcal{A}_1$ and $\mathcal{A}_2$, we also write $\mathcal{A}_1 \preceq \mathcal{A}_2$ and interpret $\mathcal{A}_2$ as an abstraction of $\mathcal{A}_1$. In fact, a simulation can be used to introduce an abstraction by (i) forgetting some of the port names, (ii) weakening data constraints, (iii) making two or more states indistinguishable, or (iv) abstracting from the contents of memory cells.

We denote the fact that there exists a bijective simulation between two automata by $\mathcal{A}_1 \simeq \mathcal{A}_2$. A simulation f is called bijective if all the maps f_Q, f_P, f_M and f_T are bijective. Note that this notion of behavioral equivalence is obviously stronger than trace equivalence and strong bisimilarity [20]. However, isomorphy is the natural notion of equivalence which arises in a categorical setting, as we will have in Section 5.

Remark 2 (τ-transitions *and* ε-memory *cells*). For all primitives listed in Table 2, we adapt their constraint automata by adding for each state q in the automaton (i) a transition $q \xrightarrow{\emptyset,\top} q$, and (ii) a memory cell ε_q, or just ε for short. The additional transitions can be interpreted as silent steps, which do not change the internal state of the primitive. In a simulation $f : \mathcal{A}_1 \to \mathcal{A}_2$, a transition of $\mathcal{A}_1$, which does not involve any of the ports in $\mathcal{A}_2$, can be mapped to such a silent transition in $\mathcal{A}_2$. Similarly, mapping a memory cell y in $\mathcal{A}_1$ to ε in $\mathcal{A}_2$ is used to abstract from the content of y. Thus, we consider only weak simulations for the examples in this paper.

Example 2 (*Simulation*). An example of a simulation is depicted in the upper part of Fig. 2. The mapping of the states is indicated using equal indices of the state names. The mapping of port names from right to left is the obvious inclusion $f_P : \{A,B\} \hookrightarrow \{A,B,C\}$. The memory cell x in the source automaton is mapped to x in the target automaton, whereas y is mapped to ε. The transitions via the port C in the source automaton are mapped to τ-transitions in the target automaton. For example, the LHS transition $\langle q_1',\{C\},\mathbf{d}_C = \mathbf{s}.y,q_1 \rangle$ corresponds to the RHS transition $\langle q_1,\emptyset,\top,q_1 \rangle$. Similarly, the LHS transition $\langle q_1',\{A,C\},\mathbf{t}.x = \mathbf{d}_A \wedge \mathbf{d}_C = \mathbf{s}.y,q_2 \rangle$ corresponds to the RHS transition $\langle q_1,\{A\},\mathbf{t}.x = \mathbf{d}_A,q_2 \rangle$.

The Reo connectors from which the automata are derived are shown in the lower part of Fig. 2. The simulation relates the parallel composition of *FIFO*1(*A*,*B*) and

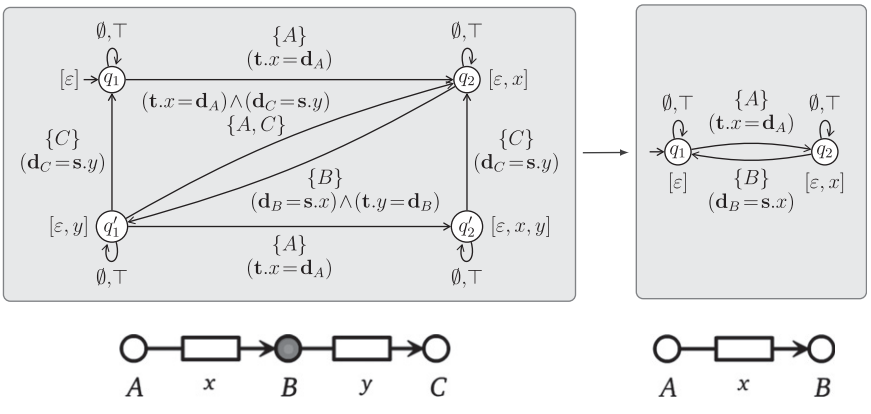


Fig. 2. A simulation of constraint automata with memory (top) and the Reo connectors from which the automata are derived (bottom).

Please cite this article as: C. Krause, et al., Compositional and behavior-preserving reconfiguration of component connectors in Reo, Journal of Visual Languages and Computing (2012), http://dx.doi.org/10.1016/j.jvlc.2012.09.002

FIFO1(*B*,*C*) with a single *FIFO1*(*A*,*B*). Intuitively, the simulation establishes an abstraction in which the details of *FIFO1*(*B*,*C*) are hidden, in line with the absence of the channel *FIFO1*(*B*,*C*) in the RHS connector. Note that, to derive the constraint automata for composite connectors, we introduce a parallel composition operator in Section 5.

5. The category of constraint automata with state memory

To obtain a categorical and, thereby, a more abstract view on constraint automata with state memory, we introduce the category **CASM** in the following. This will enable us to define a parallel composition operator in terms of a universal property, and later, to obtain a formal model of component connectors in Reo which integrates their structural and behavioral aspects.

Definition 6 (*Simulation composition and identity*). Given two simulations $f=(f_Q,f_P,f_M,f_T):\mathcal{A}_1\to\mathcal{A}_2$ and $g=(g_Q,g_P,g_M,g_T):\mathcal{A}_2\to\mathcal{A}_3$. The simulation $g\circ f$ is defined as $g\circ f=(g_Q\circ f_Q,g_P\circ f_P,g_M\circ f_M,g_T\circ f_T)$. For a constraint automaton with state memory $\mathcal{A}=(Q,P,M,T,\mu,\theta,Q^0)$ the simulation $id_{\mathcal{A}}:\mathcal{A}\to\mathcal{A}$ is defined as $id_{\mathcal{A}}=(id_Q,id_P,id_M,id_T)$.

Definition 7 (*Category **CASM***). The category **CASM** is defined as the category with constraint automata with state memory as objects, and simulations as morphisms (cf. Definitions 3, 5 and 6).

It is straightforward to check that identity is neutral and composition of simulations is associative. In the following, we investigate some of the properties of the category **CASM**.

Lemma 1 (*Final object in **CASM***). *Let* $\mathbf{1}\in\mathbf{CASM}$ *be the constraint automaton with state memory that consists of a single state q, an empty set of port names, a single memory cell* ε*, and a single transition* $q\xrightarrow{\emptyset,\top}q$*. Then for any automaton* $\mathcal{A}\in\mathbf{CASM}$ *there exists a unique simulation* $!_{\mathcal{A}}:\mathcal{A}\to\mathbf{1}$.

Lemma 1 states that the CASM depicted in Fig. 3, also denoted as **1**, is the most 'general' constraint automaton with state memory—in the sense that it is a valid abstraction of any possible CASM: $\mathcal{A}\preceq\mathbf{1}$. Note also that the final object is, as always in categorical approaches, defined up to isomorphism. A simulation is an isomorphism if and only if it is bijective.

As we show in the following, the parallel composition of constraint automata with state memory can be defined in terms of a universal property, specifically using pullbacks in **CASM**.

Lemma 2 (*Pullbacks in **CASM***). *Given a cospan of* **CASM***-simulations* $\mathcal{A}_1\xrightarrow{f_1}\mathcal{A}_0\xleftarrow{f_2}\mathcal{A}_2$*. Then the span* $\mathcal{A}_1\xleftarrow{f_1'}\mathcal{A}_3\xrightarrow{f_2'}\mathcal{A}_2$ *with* $\mathcal{A}_3$ *defined below is the corresponding pullback. Using the notation* $\theta_i(t_i)=\langle q_i,F_i,g_i,q_i'\rangle$ *for a transition* $t_i\in T_i$, $i=1..3$, *the automaton* $\mathcal{A}_3$ *is given by*:

- $Q_3=Q_1\times_{Q_0}Q_2$ (*pullback in* **Set**, *analogously for* Q_3^0)
- $P_3=P_1+_{P_0}P_2$ (*pushout in* **Set**)
- $M_3=M_1\times_{M_0}M_2$ (*pullback in* **Set**)
- $T_3=\{\langle t_1,t_2\rangle\in T_1\times T_2\mid(f_{1,T}(t_1)=f_{2,T}(t_2))\wedge(f_{1,P}^{-1}(F_1)=f_{2,P}^{-1}(F_2))\}$
- $\mu_3(\langle x_1,x_2\rangle)=\langle\mu_1(x_1),\mu_2(x_2)\rangle$
- $\theta_3(\langle t_1,t_2\rangle)=\langle\langle q_1,q_2\rangle,F_1+_{F_0}F_2,(g_1[f'_{1,P},f'_{1,M}]\wedge g_2[f'_{2,P},f'_{2,M}]),\langle q_1',q_2'\rangle\rangle$

The simulation $f_1':\mathcal{A}_3\to\mathcal{A}_1$ *consists of the projections* $f'_{1,Q}$, $f'_{1,M}$, $f'_{1,T}$, *and the injection* $f'_{1,P}$. *The simulation* $f_2':\mathcal{A}_3\to\mathcal{A}_2$ *is defined analogously*.

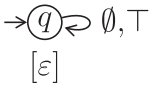


Fig. 3. Final object in **CASM**.

Example 3 (*Pullbacks in **CASM***). The pullback diagram in Fig. 4 illustrates the parallel composition of a *FIFO1*(*A*,*B*) and a *FIFO1*(*B*,*C*) over their shared port *B*, which yields a *FIFO2*(*A*,*C*). The simulations are defined analogously to Example 2. The transitions via {*B*} in $\mathcal{A}_1$ and $\mathcal{A}_2$ are synchronized over the transition via {*B*} in $\mathcal{A}_0$. The transitions via {*A*} in $\mathcal{A}_1$ and via {*C*} in $\mathcal{A}_2$ are both mapped to the τ-transition in $\mathcal{A}_0$. Thus, transitions can be synchronized with a τ-transition, yielding interleaved behavior, or can be synchronized with each other, yielding a concurrent transition via {*A*,*C*} in $\mathcal{A}_3$.

We use the default notation for pullbacks of constraint automata with state memory, i.e., we write $\mathcal{A}_3=\mathcal{A}_1\times_{\mathcal{A}_0}\mathcal{A}_2$. Compared to the join operator for constraint automata with state memory introduced in [18], the parallel composition via pullbacks is more expressive. While the join operator in [18,2] synchronizes only over a shared set of port names, the parallel composition via pullbacks enables the composition of two automata $\mathcal{A}_1$, $\mathcal{A}_2$ over another automaton $\mathcal{A}_0$—the latter can be seen as a shared context of the former. This shared context can be used to synchronize $\mathcal{A}_1$ and $\mathcal{A}_2$ not only over shared port names, but also over corresponding states and memory cells. The join operator in [18,2] is a special case of a pullback in which $\mathcal{A}_0$ is stateless, i.e., has only one state. Moreover, our parallel composition is derived from the simulation notion using a universal property. The categorical construction using pullbacks furthermore includes the morphisms into the original automata and thereby relates them with the resulting constraint automaton with simulations. Using these simulations, the local state of a subconnector, e.g. a channel, can be derived from the global state of the composite connector.

Since the category of constraint automata with state memory has final objects and pullbacks, it follows immediately that it is complete.

Theorem 1 (*Completeness of **CASM***). *The category of constraint automata with state memory is finitely complete, i.e., for every finite diagram in* **CASM** *there exists a limit.*

6. Categorical notions for distributed objects

The examples in the previous sections, particularly Example 2 of a CASM simulation and Example 3 of a

Please cite this article as: C. Krause, et al., Compositional and behavior-preserving reconfiguration of component connectors in Reo, Journal of Visual Languages and Computing (2012), http://dx.doi.org/10.1016/j.jvlc.2012.09.002

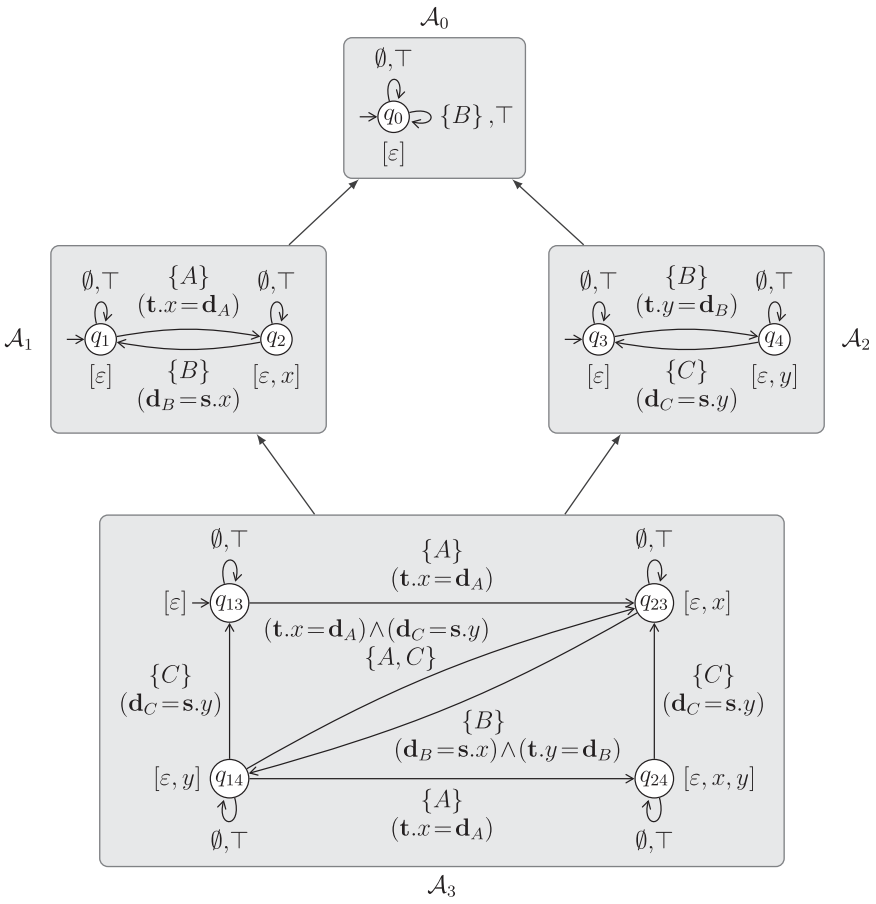


Fig. 4. A pullback of constraint automata with state memory, modeling the parallel composition of a *FIFO*1(*A*,*B*) represented by $\mathcal{A}_1$ and a *FIFO*1(*B*,*C*) represented by $\mathcal{A}_2$ over their shared port *B* as captured by the simulations to $\mathcal{A}_0$.

CASM pullback, indicate that there is a close correspondence between the structure of connectors in terms of graphs on the one hand, and their semantics in terms of automata on the other. To formally describe this correspondence we use the framework of distributed graph transformation, as introduced by Taentzer [21], and later generalized to the notion of transformation of distributed objects by Ehrig et al. [22]. In this section, we recall the basic concepts of this framework and present a new result on compositionality of the flattening functor for distributed graph transformation. In Section 7, we will then show how distributed graph transformation can be used to integrate the structure and semantics of component connectors.

6.1. Distributed objects and morphisms

Distribution of graphs or, more generally, of objects can be described by providing an additional layer of abstraction, namely by modeling the topology of a system using a so-called *network graph*. The nodes in a network graph correspond to objects and the edges to morphisms between objects. Intuitively, a node in a network graph can be used to model a physical or logical location of an object, whereas an edge indicates some kind of inclusion or mapping of the source object into the target object. Multiple outgoing edges from one object indicate that the object is shared among all target objects.

Definition 8 (*Distributed object [22]*). Given a category **C**. A *distributed object* (*N*, *D*) consists of a graph *N*, called the *network graph*, and a commutative diagram $D : N \to \mathbf{C}$, where the graph *N* is interpreted as a finite category.

The network graph *N* describes the topology of the system. The diagram *D* associates to every node $n \in N$ an object $D(n) \in \mathbf{C}$ and to every edge $n \xrightarrow{e} n'$ in *N* a **C**-morphism $D(e) : D(n) \to D(n')$. Following [22], this diagram is required to be commutative, i.e., for any two paths $p_1, p_2 : n \xrightarrow{*} n'$ in *N*, it must hold that $D(p_1) = D(p_2)$. This arises from the assumption that the morphisms associated with edges are used to represent the shared parts among objects.

Example 4 (*Distributed object*). An example of a distributed object is depicted in the top of Fig. 5. Formally, the distributed object is given by a diagram of the shape •←•→•, i.e., a span in the category **Set**. The set in every node can be interpreted as its interface. Specifically, if a node $n \in N$ is used to model a Reo primitive such as a channel, the set $D(n)$ can be identified with its port

Please cite this article as: C. Krause, et al., Compositional and behavior-preserving reconfiguration of component connectors in Reo, Journal of Visual Languages and Computing (2012), http://dx.doi.org/10.1016/j.jvlc.2012.09.002

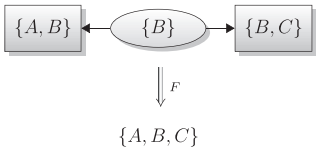


Fig. 5. A distributed object modeling two primitives with a shared interface (top) and its flattening (bottom).

names. In this way, the distributed object in Fig. 5 can be used to model the structure of a component connector in Reo: (i) the node with the set $\{A,B\}$ corresponds to a channel between A and B, (ii) the node with the set $\{B,C\}$ corresponds to a channel between B and C, and (iii) the node with the set $\{B\}$ and its outgoing edges model the sharing of the Reo node B among the two channels. Note that this model is purely structural and describes only the topology of the connector, but not its semantics.

Definition 9 (*Distributed morphism [22]*). Let (N_1,D_1) and (N_2,D_2) be two distributed objects over the category **C**. A morphism $f=(f_N,f_D):(N_1,D_1)\to(N_2,D_2)$ consists of a graph morphism $f_N: N_1\to N_2$ and a natural transformation $f_D: D_1\to D_2\circ f_N$.

$$\begin{array}{ccc} D_1(n) & \xrightarrow{D_1(e)} & D_1(n') \\ {\scriptstyle f_n}\downarrow & & \downarrow{\scriptstyle f_{n'}} \\ D_2(f(n)) & \xrightarrow{D_2(f(e))} & D_2(f(n')) \end{array}$$

The natural transformation f_D assigns to every node n of the network graph N_1 a graph morphism $f_n: D_1(n)\to D_2(f(n))$ which is called the *local* morphism of n. Furthermore, for every edge $n\xrightarrow{e}n'$ in N_1 the above diagram commutes, where for brevity we just write f for the network morphism f_N. Distributed objects and their morphisms form the category **Dis**(**C**) [22].

6.2. Flattening of distributed objects

The flattening of a distributed object can be understood as a way of gluing together the objects of all nodes in the network graph along their shared parts. Formally, the flattening of a distributed object (N, D) can be achieved by considering the colimit of the diagram D [21]. It is well-known that this flattening extends to a functor F : **Dis**(**C**)→**C**, given that **C** is cocomplete [23]. This definition is rather elegant because it defines the flattening in terms of a universal property, and not by means of an algorithm or by referring to an operational semantics.

Example 5 (*Flattening of distributed objects*). Flattening of the distributed object in the top of Fig. 5 yields the set $\{A,B,C\}$ depicted on the bottom. When interpreting the distributed object as a structural model of a component connector in Reo, the flattening yields the set of all port names of this connector.

Considering flattening of distributed objects is particularly interesting when distribution is used to provide logical structure to a flat system. The distributed model can be interpreted as a more high-level view on a flat structure. In this perspective, it is crucial to know whether flattening interacts well with composition. Composing two distributed objects in **Dis**(**C**) and flattening the result should yield the same outcome as first flattening both distributed objects and then composing them in **C**. Since the composition of two structural objects, such as graphs, is commonly modeled by their union, compositionality of the flattening functor F requires that F preserves the categorical equivalent of unions, i.e., pushouts, or more generally, colimits.

Theorem 2 (*Flattening preserves colimits [10]*). *Let* **C** *be a cocomplete category. The flattening functor* F : **Dis**(**C**)→**C** *has a right adjoint and is therefore cocontinuous.*

Distributed objects can be used to describe a logical partitioning of an otherwise flat structure. Due to Theorem 2, composition and transformation of distributed objects can be transparently implemented on the underlying flat structure. In Section 7, this result will be furthermore the key to show compositionality of the CASM semantics for Reo connectors.

7. Distributed constraint automata with state memory

In this section, we present a categorical model of component connectors in Reo, which incorporates (i) the CASM semantics of all primitives, i.e., the channels and components of the connector, and (ii) the structural aspects of the connector, i.e., its topological information. To this end, we combine the categorical model of constraint automata with state memory introduced in Section 5 with the categorical model of distributed objects presented in Section 6.

We model the topology information of a connector using a network graph of a distributed object, as explained in Example 4 above. However, instead of considering only the interfaces of primitives as network nodes, we now also incorporate their semantics. Specifically, we consider distributed objects in which the nodes in the network graph are constraint automata with state memory, and the edges are simulations. However, since simulations map port names in the opposite direction, we need to consider reversed simulations.

Definition 10 (*Category **DCASM***). The category of *distributed constraint automata with state memory* is defined as **DCASM** = **Dis**(**CASM**$^{\mathrm{op}}$).

Reo connectors can be modeled as objects in **DCASM**. We identify every primitive with its corresponding CASM (cf. Table 2 and Remark 2). Additionally, for every node X we consider a stateless CASM with a transition via $\{X\}$. Now we consider all these automata as vertices in a network graph and create for every pair of a primitive and a node an edge between them in the network graph. The edge points toward the CASM of the primitive. However, the **CASM**-simulation goes in the opposite direction and maps all transitions of the primitive that involve the adjacent node X to the self loop transition via $\{X\}$, and all other transitions to the τ-transition. The reason for inverting the edges is the contravariance of the port name mapping in simulations

Please cite this article as: C. Krause, et al., Compositional and behavior-preserving reconfiguration of component connectors in Reo, Journal of Visual Languages and Computing (2012), http://dx.doi.org/10.1016/j.jvlc.2012.09.002

and can be informally motivated by the argument that the edges in the network graph represent primarily structural mappings of the port names, which are mapped in the opposite direction by simulations.

Example 6 (*FIFO2 as a distributed CASM*). We argued in Example 4 that the distributed object in Fig. 5 can be interpreted as a structural model of a connector which consists of two channels connected by a node. In the category **DCASM** we can incorporate also the semantics of the channels. Fig. 6 depicts the Reo connector (bottom) and the corresponding distributed constraint automaton with state memory (top) for two connected *FIFO*1 channels. Note that the edges in the network graph point from the node to the channels, indicating the structural embedding of port names. However, the edges formally correspond to inverse simulations, i.e., simulations from the *FIFO*1 channels to the node, forming a cospan in **CASM**.

7.1. Composing connectors

In categorical approaches where morphisms are used as structural mappings between objects, the composition of objects can be described using the categorical equivalent of unions, i.e., using pushouts. The objects of the category **DCASM** are connectors and their morphisms are structural mappings of network graphs together with simulations between the primitives of the connector. Therefore, pushouts or, more generally, colimits are also the natural choice for composition in **DCASM**.

Theorem 3. *The category* **DCASM** *is finitely cocomplete, i.e., for every finite diagram in* **DCASM** *there exists a colimit.*

This follows from Theorem 1 and the fact that cocompleteness of a category **C** implies cocompleteness of **Dis**(**C**) [22].

Using pushouts, connectors modeled as objects in **DCASM** can be composed by gluing together their respective network graphs. Note that in this view, the semantics of all primitives of the connectors is usually fixed, i.e., it is allowed to identify primitives of the same type only. Formally, the simulations between the primitives are isomorphisms. In this situation, the automata in the network nodes are not changed when composing two connectors using pushouts and, thus, the composition is of purely structural nature. However, the case where the simulations between the primitives are not isomorphisms has also interesting applications. Essentially, it allows us to identify primitives of different types or with parameters. For instance, consider two connectors, both containing a *Filter* channel, one with a filter constraint c and the other one with a different filter constraint c'. Then it is possible to glue these two connectors by identifying the two *Filter* channels, e.g. by mapping them to the same *Sync* channel. The composition result will then contain a single *Filter* channel with the filter constraint $c \wedge c'$.

7.2. Semantics of connectors

We have demonstrated how an object in **DCASM** can be used to model the structure of a connector together with the automata semantics of its primitives. However, we have not shown yet how to derive the semantics of a composite connector in terms of the parallel composition of all its primitives.

Let $(N,D) \in$ **DCASM** be a distributed constraint automaton with state memory, where N is the network graph and D is a diagram in $\mathbf{CASM}^{\mathrm{op}}$. Then the constraint automaton with state memory that represents the semantics of the connector is given by the colimit over D. Since the edges in the network graph are inverse simulations, the colimit of D corresponds to a limit in **CASM**. Moreover, we have shown in Section 6.2 that the colimit of distributed objects extends to a functor, i.e., the flattening functor $F : \mathbf{Dis}(\mathbf{C}) \to \mathbf{C}$ for distributed objects. Therefore, applying F to **DCASM** yields a semantics functor for connectors.

Definition 11 (*Semantics functor*). Let $F : \mathbf{DCASM} \to \mathbf{CASM}^{\mathrm{op}}$ be the flattening functor for distributed constraint automata with state memory. By reverting the arrows, F induces the contravariant functor

$Sem : \mathbf{DCASM} \to \mathbf{CASM}$

which is called the *semantics functor* for distributed constraint automata with state memory.

Example 7 (*Semantics functor*). Fig. 7 shows the application of the semantics functor to the *FIFO*2 connector. The colimit over the network graph corresponds to a limit in **CASM**. Specifically, the limit coincides here with the pullback in Fig. 4.

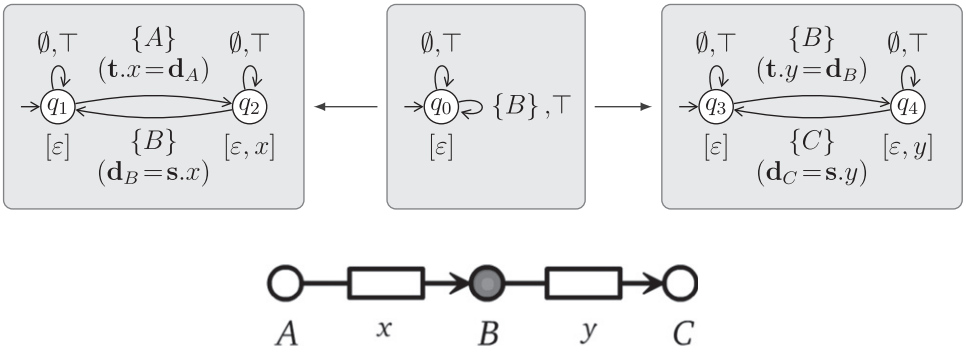


Fig. 6. Reo connector *FIFO*2 (bottom) and the corresponding distributed CASM (top).

Please cite this article as: C. Krause, et al., Compositional and behavior-preserving reconfiguration of component connectors in Reo, Journal of Visual Languages and Computing (2012), http://dx.doi.org/10.1016/j.jvlc.2012.09.002

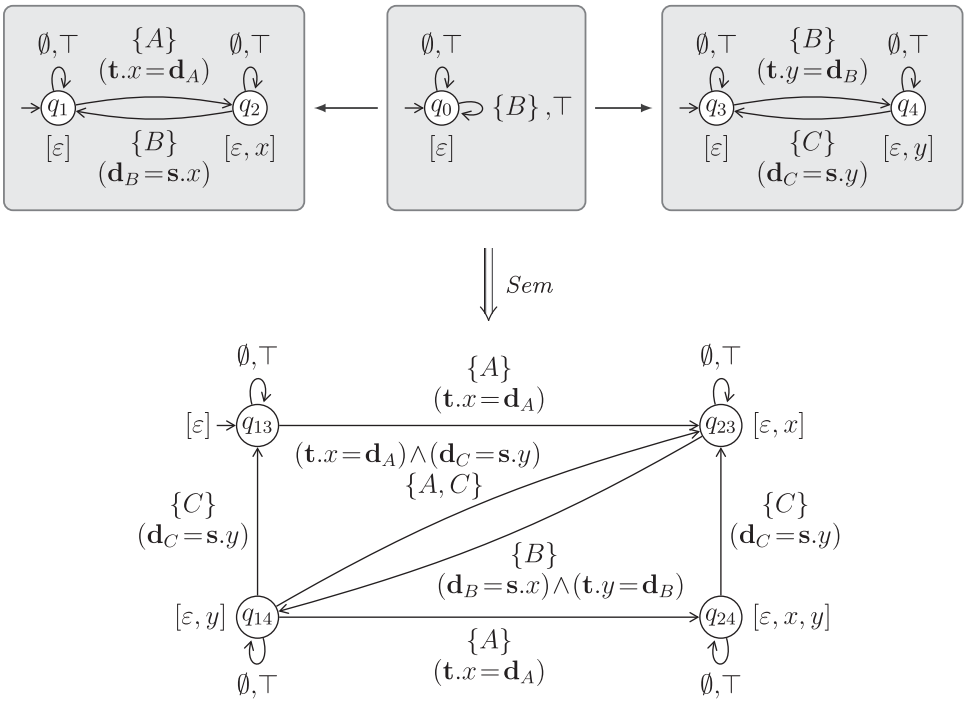


Fig. 7. *FIFO2* connector as a DCASM (top) and its semantics as a CASM (bottom).

Due to the categorical approach, it is straightforward to show the compositionality of the semantics functor.

Theorem 4 (*Compositionality of the semantics functor*). *The semantics functor Sem* : **DCASM**→**CASM** *is compositional, i.e., it maps colimits in* **DCASM** *to limits in* **CASM**.

Thus, we have established that a structural gluing of connectors in **DCASM** which is realized by a pushout of their respective network graphs has a corresponding semantical join operation, i.e., a pullback of their corresponding automata. Note also by Theorem 2 it follows that the structure and the semantics of connectors are tightly connected, i.e., they form a pair of adjoint functors.

8. Reconfiguration by graph transformation

We have shown in Section 7 that the category **DCASM** provides a sound structural and semantical model for component connectors in Reo. The parallel composition of connectors was defined using structural gluings of network graphs, formally given by pushouts in **DCASM**. In this section, we show that reconfigurations, i.e., structural transformations of connectors can be defined and carried out following the *double pushout* (DPO) approach to graph transformation [24,9]. Moreover, since our models also include semantical information, it is possible to statically check whether reconfigurations are behavior-preserving.

In the double pushout approach, a transformation rule is defined as a span of injective morphisms in a category **C**. To apply the DPO approach to component connectors in Reo, we can simply choose **DCASM** for the category **C** in the following definitions.

Definition 12 (*Rule [24]*). Given a category **C**, a *rule* $p = (L \xleftarrow{\ell} K \xrightarrow{r} R)$ consists of three objects L, K and R, called the left-hand side, the gluing object, and the right-hand side, respectively and morphisms $\ell : K \to L$ and $r : K \to R$.

The left-hand side L of a rule p defines the (structural) pattern that must be matched to apply p. The gluing object K contains all elements that are not removed by the rule and R additionally includes those elements that should be added to the object, which – in our case – is a Reo connector modeled by an object in **DCASM**. To reconfigure a given connector M, a transformation rule p can be applied to M with respect to a morphism $m : L \to M$, called the *match*. This match defines in which part of the connector M, the reconfiguration should take place and ensures further that the required structural patterns exist. The actual reconfiguration of the network M using the rule p with the match m is formally defined by a double pushout diagram.

Definition 13 (*Transformation [24]*). Given a rule $p = (L \xleftarrow{\ell} K \xrightarrow{r} R)$, an object M and a morphism $m : L \to M$, a *transformation* $M \overset{p,m}{\Leftrightarrow} N$ is defined as the double pushout diagram

$$\begin{array}{ccccc} L & \xleftarrow{\ell} & K & \xrightarrow{r} & R \\ {\scriptstyle m}\downarrow & (PO) & \downarrow & (PO) & \downarrow \\ M & \longleftarrow & C & \longrightarrow & N \end{array}$$

Operationally, the connector M is reconfigured by (i) removing the occurrence of $L \backslash \ell(K)$ in M, yielding the intermediate connector C, and (ii) adding a copy of $R \backslash r(K)$ to C. The DPO approach has been applied to many

Please cite this article as: C. Krause, et al., Compositional and behavior-preserving reconfiguration of component connectors in Reo, Journal of Visual Languages and Computing (2012), http://dx.doi.org/10.1016/j.jvlc.2012.09.002

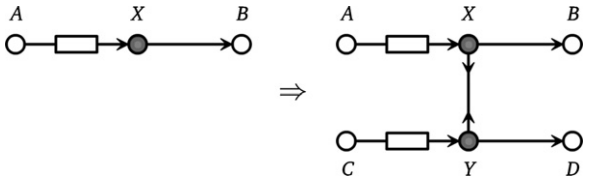


Fig. 8. The structural reconfiguration rule *AddBarrierSync*.

high-level structures, such as typed attributed graphs, hypergraphs and Petri nets. For a detailed discussion on DPO graph transformation we refer to [9]. Here, we focus on the application of graph transformation to model reconfigurations of component connectors.

Example 8 (*Structural reconfiguration*). Fig. 8 depicts the left-hand side and right-hand side of a reconfiguration rule, called AddBarrierSync. The objects are depicted in the standard graph notation for Reo connectors, but are formally given by objects in **DCASM**, analogously to Fig. 6. For brevity, we omit the gluing object K, which coincides with L here. The connector in the right-hand side is called *buffered barrier synchronizer*.[3] Intuitively, this connector synchronizes the two buffered communication channels between A and B, and C and D. Thus, data items can be delivered only together. The structural reconfiguration rule AddBarrierSync allows us to extend the barrier synchronizer to an unbounded number of channels by subsequently adding the lower part of the right-hand side of the rule.

8.1. Semantical reconfiguration

Since the **CASM** semantics is compositional for pushouts, we can lift the concept of structural reconfigurations based on double pushouts to the semantical level, simply by applying the semantics functor to the productions and transformations. Formally, a structural reconfiguration rule given as a span in **DCASM** is mapped to a *semantical reconfiguration rule* given as a cospan in **CASM**. Analogously, a double pushout diagram for a structural reconfiguration yields a double pullback diagram

$$\begin{array}{ccccc} \mathcal{L} & \xrightarrow{\ell} & \mathcal{K} & \xleftarrow{r} & \mathcal{R} \\ \uparrow & (PB) & \uparrow & (PB) & \uparrow \\ \mathcal{M} & \longrightarrow & \mathcal{C} & \longleftarrow & \mathcal{N} \end{array}$$

for the corresponding semantical reconfiguration.

Example 9 (*Semantical reconfiguration*). Fig. 9 depicts the derived semantical reconfiguration rule $\mathcal{A}dd\mathcal{B}arrier\mathcal{S}ync$ for the structural reconfiguration rule AddBarrierSync in Fig. 8. The gluing object $\mathcal{K}$ is identical to the left-hand side $\mathcal{L}$. Note that the two automata are related by a simulation from the right-hand side to the left-hand side of the rule. The state mapping is indicated using equal indices of the state names. The port name map is the obvious inclusion from left to right.

Because of the compositionality of the semantics functor, we obtain a means to analyze the impact of a structural reconfiguration of a connector on its semantics. As a specific example, we discuss a notion of behavior-preservation in the following.

8.2. Behavior-preserving reconfiguration

Since in our approach, we define reconfigurations as structural transformations of component connectors, it is important to be able to analyze the impact of a reconfiguration on the semantics of a connector. In this section, we introduce a notion of behavior-preservation for reconfigurations and show how it can be ensured by means of a static check.

To motivate our formal notion of behavior-preserving reconfiguration, we consider again the reconfiguration rule AddBarrierSync in Fig. 8. For this rule, we would like to ensure the following informal notion of behavior-preservation: If the connector allows a directed communication from a component at the node A to a component at the node B before applying the reconfiguration rule AddBarrierSync, than this communication should also be possible after the reconfiguration. Thus, the ability of sending data through the channels between A and B should be preserved by the reconfiguration. More generally, we would like to ensure that the reconfiguration does not *remove* any behavior.

Formally, we need to show that if $\mathcal{M} \overset{AddBarrierSync,m}{\Rightarrow} \mathcal{N}$ for an arbitrary connector model $\mathcal{M}$ and an arbitrary match m, then also $\mathcal{M} \preceq \mathcal{N}$. Thus, we need to prove that there exists a simulation between the semantics of the connectors before and after the reconfiguration. Such a simulation can be interpreted as a refinement, in the sense that every behavior of $\mathcal{M}$ should be also possible in $\mathcal{N}$. However, the existence of such a simulation turns out to be too strong as a general requirement for behavior-preservation. In particular, a reconfiguration may add new nodes, such as C,Y and D in the rule AddBarrierSync, which should not be relevant for defining the simulation. Therefore, we first introduce a technical notion for restricting the port name sets of constraint automata with state memory.

Definition 14 (*Port name restriction*). Let $\mathcal{A} = (Q,P,M,T,\mu,\theta,Q^0)$ be a constraint automaton with state memory, and $f : \mathcal{A} \to \mathcal{X}$ a simulation. We define the constraint automaton with state memory

$$\mathcal{A}|\mathcal{X} = (Q, f_P(P_{\mathcal{X}}), M, T, \mu, \theta|\mathcal{X}, Q^0)$$

such that $\theta(t) = \langle q,F,g,q' \rangle \Leftrightarrow \theta|\mathcal{X}(t) = \langle q, F \cap f_P(P_{\mathcal{X}}), g|\mathcal{X}, q' \rangle$ where $g|\mathcal{X}$ is the guard obtained from g by removing all subconstraints '$v_1 = v_2$' where $v_1 = \mathbf{d}_p$ or $v_2 = \mathbf{d}_p$ with $p \notin f_P(P_{\mathcal{X}})$. We define the simulations $\alpha_f = (id_Q, id_M, f_P, id_T) : \mathcal{A} \to \mathcal{A}|\mathcal{X}$ and $\beta_f = (f_Q, f_M, id_P, f_T) : \mathcal{A}|\mathcal{X} \to \mathcal{X}$.

Intuitively, for a given simulation $f : \mathcal{A} \to \mathcal{X}$, the automaton $\mathcal{A}|\mathcal{X}$ is derived from $\mathcal{A}$ by restricting its port name set to those port names that have a preimage in $\mathcal{X}$.

[3] The unbuffered variant of the barrier synchronizer in Reo was introduced in [1].

Please cite this article as: C. Krause, et al., Compositional and behavior-preserving reconfiguration of component connectors in Reo, Journal of Visual Languages and Computing (2012), http://dx.doi.org/10.1016/j.jvlc.2012.09.002

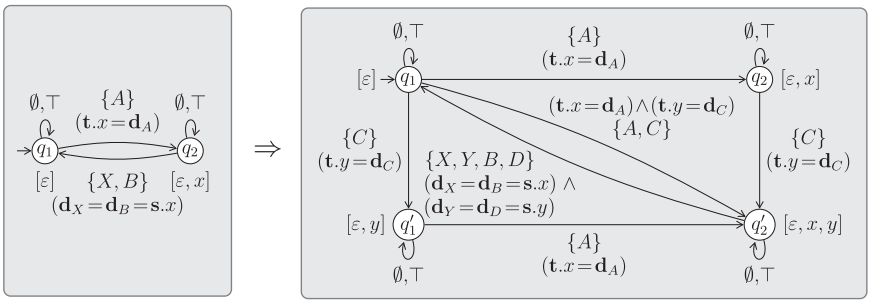
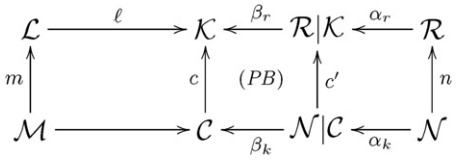


Fig. 9. The derived semantical reconfiguration rule *AddBarrierSync*.

All other port names of $\mathcal{A}$ are hidden. This notion of port name restriction is moreover a factorization of f into α_f and β_f, i.e., the diagram



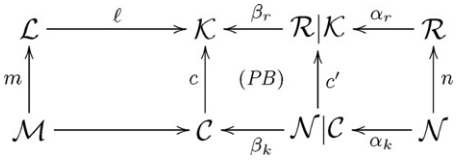
commutes. In the following, we define a notion of behavior-preserving reconfiguration rule.

Definition 15 (*Behavior-preserving reconfiguration rule*). A semantical reconfiguration rule $p=(\mathcal{L}\xrightarrow{\ell}\mathcal{K}\xleftarrow{r}\mathcal{R})$ is called *behavior-preserving* if for any reconfiguration $\mathcal{M}\overset{p,m}{\Leftrightarrow}\mathcal{N}$ given by the cospan $\mathcal{M}\longrightarrow\mathcal{C}\longleftarrow\mathcal{N}$ it holds that $\mathcal{M}\preceq\mathcal{N}|\mathcal{C}$.

Note that $\mathcal{N}|\mathcal{C}$ is the automaton $\mathcal{N}$ restricted to the port names that are preserved by the rule. Therefore, behavior-preserving rules ensure that the input and output of the reconfiguration are related by a simulation, where the output is restricted to the port names preserved by the reconfiguration.

In the following, we provide a sufficient condition for ensuring behavior-preservation of reconfigurations, which can be checked statically, i.e., for a given semantical reconfiguration rule.

Theorem 5 (*Behavior-preserving reconfiguration*). *Given a semantical reconfiguration rule* $p=(\mathcal{L}\xrightarrow{\ell}\mathcal{K}\xleftarrow{r}\mathcal{R})$ *and a simulation* $\mathcal{K}\xrightarrow{f}\mathcal{R}|\mathcal{K}$, *such that* $\beta_r\circ f=id_{\mathcal{K}}$. *Then p is behavior-preserving, i.e.,* $\mathcal{M}\preceq\mathcal{N}|\mathcal{C}$



Example 10 (*Behavior-preserving reconfiguration*). Given the semantical reconfiguration rule $\mathcal{AddBarrierSync}=(\mathcal{L}\xrightarrow{\ell}\mathcal{K}\xleftarrow{r}\mathcal{R})$ in Fig. 9, where $\mathcal{L}=\mathcal{K}$. The automaton $\mathcal{R}|\mathcal{K}$ is derived from $\mathcal{R}$ by restricting the port name sets to {*A*,*X*,*B*}, i.e., by removing the port names *C*,*Y*,*D*. Then we can define a simulation $f:\mathcal{K}\to\mathcal{R}|\mathcal{K}$ from the left-hand side to the right-hand side by mapping state q_1 to q_1 and state q_2 to q_2'. For this simulation, $\beta_r\circ f=id_{\mathcal{K}}$ holds. Thus, the rule $\mathcal{AddBarrierSync}$ is behavior-preserving.

Note that Theorem 5 provides us with a sufficient condition for checking whether a rule is behavior-preserving. In practice, this check can be carried out by constructing a simulation *f*, which yields the identity when composed with β_r. As a first approximation, the simulation *f* can be constructed as a relation $rel\subseteq Q_{\mathcal{K}}\times Q_{\mathcal{R}}$, which is initialized as $rel=\{\langle r_Q(q),q\rangle\mid q\in Q_{\mathcal{R}}\}$ and subsequently reduced by checking locally for all transitions in $\mathcal{K}$ the existence of compatible transitions in $\mathcal{R}|\mathcal{K}$. If after this reduction *rel* is left-total, then there also exists a simulation *f* with the desired properties and we have shown that the reconfiguration rule is behavior-preserving. Thus, the check for behavior-preservation can be done automatically and statically for a given reconfiguration rule.

In a more general setting, behavior-preserving reconfiguration rules can be used to ensure properties which assert the existence of a path in a reconfigurable connector. Specifically, assume that a desired property ϕ holds for the system in its initial configuration $\mathcal{A}$, i.e. $\mathcal{A}\models\phi$, and the property is preserved by simulations, i.e., $\mathcal{A}\models\phi\wedge\mathcal{A}\preceq\mathcal{A}'$ implies $\mathcal{A}'\models\phi$. Then, by showing that all reconfiguration rules of the system are behavior-preserving, we have established that ϕ holds for all possible configurations of the system. Note that there can be infinitely many configurations, and that the size of the connector graphs and their corresponding automata is in general unbounded.

9. Related work

9.1. Reconfiguration in Reo

The reconfiguration approach presented in this paper extends the work in [10] and [14], where port automata are used as underlying semantical framework. Port automata, as introduced in [27], are an abstraction of constraint automata that do not include data. Constraint automata [2] and their extension with state memory [8,18] come equipped with a parallel composition operator, which is a special case of the composition using pullbacks presented in this paper. Since the structure of connectors is not explicitly modeled in these papers, structural reconfigurations in terms of transformations

Please cite this article as: C. Krause, et al., Compositional and behavior-preserving reconfiguration of component connectors in Reo, Journal of Visual Languages and Computing (2012), http://dx.doi.org/10.1016/j.jvlc.2012.09.002

of connector graphs cannot be described in these models. A basic logic for reasoning about connector reconfigurations in Reo, including a model checking algorithm is presented in [28]. As before, connectors are also not formalized as graphs in this work. Moreover, the reconfiguration operations are rather low-level and provide no means for a rule-based definition of reconfigurations. Behavior-preserving reconfiguration is also not considered.

Graph transformation based reconfigurations for Reo are also considered in [3]. Reasoning about dynamic reconfigurations is accomplished by modeling both the execution and the reconfiguration semantics as graph transformations. This enables state space generation and analysis using model checking. However, the approach lacks compositionality and is therefore difficult to apply in general. Reconfiguration for Reo using graph transformation where dataflow events trigger the reconfiguration are considered in [29]. A model for distributed connectors and their reconfiguration is discussed in [4]. However, the semantics of connectors is not taken into consideration.

9.2. Petri nets and workflow nets

Compositionality of semantics and graph-based reconfiguration has been also studied for various kinds of Petri nets. A marking graph semantics of Petri nets is proposed in [30]. Similarly to our approach the authors show compositionality of this semantics using a pair of adjoint functors. A compositional semantics for open Petri nets based on deterministic processes is presented in [31]. Behavior preserving reconfiguration of open Petri nets are studied in [5]. A categorical approach to automata-based semantics for Petri nets is discussed in [32]. However, this approach is more restrictive than our automata model, since concurrent actions imply interleaved semantics. Dynamic changes in workflow nets are treated in [33]. The so-called *dynamic change bug* refers to the problem of ensuring a consistent system state after a reconfiguration by calculating a safe change region and allowing a reconfiguration to take place only in these states. Related to the dynamic change bug, [34] introduces inheritance-preserving transformation rules for workflows which guarantee well-behaved reconfigurations.

9.3. Behavior-preserving model transformation

A comparison of two proof techniques for behavior-preserving model transformation, respectively based on explicit bisimulation construction and borrowed contexts, are discussed in [35]. The authors show behavior-preservation of a model transformation between two artificial languages with graph transformation based operational semantics. Behavior-preservation for refactorings modeled using graph transformations with borrowed contexts is considered in [36], where the authors present a technique to ensure behavior-preservation for complex refactorings consisting of multiple refactoring steps. Correctness of a model-to-code transformation from automata models to an executable, textual language is considered in [37]. The correctness of the transformation is shown by an automated proof for the semantic equivalence of the source and target models using the theorem prover Isabelle/Hol. Both in [35] and [37], the transformations are defined using Triple Graph Grammars (TGGs). Behavior-preservation of a model transformation between graph transformation based simulation and animation models is discussed in [38]. Their main result consists of a condition which ensures behavior-preservation of this class of model transformations. Structure and behavior-preserving refinements of abstract architectural models into platform-specific representations are considered in [39]. In this work, both the operational semantics and the reconfiguration are model using graph transformation rules, and model checking is employed to verify that abstract scenarios can be realized in a platform-specific architecture.

10. Conclusions and future work

We have presented a model for component connectors for the channel-based coordination language Reo, which integrates (i) an automata-based model for the specification of primitives, e.g. channels and components, and (ii) a graph-based model to describe the structure of the system in terms of a connector or network. Our combined structural and semantical model of *distributed constraint automata with state memory* is based on the categorical theory of distributed graph transformation. We defined the semantics of our model using a flattening functor for distributed objects and showed that it is compositional with respect to gluing operations of – in our case – connector graphs. We defined reconfigurations of component connectors in Reo using graph transformations of the underlying connector graphs. Due to the compositionality of our model, we were able to lift the structural transformations of connector graphs to the semantical level, i.e., to the automata semantics of Reo. Thereby, our model allows us to reason about the impact of reconfigurations on the semantics of connectors. As a specific example, we defined a notion of behavior-preserving reconfiguration and presented a sufficient condition to ensure behavior-preservation statically for a given reconfiguration rule.

The intricacy of the problem area, i.e., reasoning about behavioral properties for structural reconfiguration, led us to use a variety of modeling techniques and formal methods in this paper. One of our core observations is that channel-based component connectors in Reo have a graph-based structure and that reconfiguration can be formally modeled using graph transformation. Therefore, we base our approach on the categorical framework of algebraic graph transformation [9]. Our second observation is that the semantics of channels and connectors in Reo can be phrased in terms of an automata model. Thus, as the second formal foundation of our approach, we rely on the semantic model of constraint automata [2,8]. As the first step of integrating these two views on Reo, i.e., the structure and the semantics of connectors, we have developed the categorical constraint automata model **CASM**. We then used categorical notions for describing distributed objects and their transformation, i.e., the framework of distributed graph transformation [21,22] for integrating the structural and the semantical views of

Please cite this article as: C. Krause, et al., Compositional and behavior-preserving reconfiguration of component connectors in Reo, Journal of Visual Languages and Computing (2012), http://dx.doi.org/10.1016/j.jvlc.2012.09.002

connectors. Specifically, we defined the category **DCASM** which adds the topology information of connectors to the automata semantics. In this category, graph transformation can be applied to model reconfiguration of connectors. Moreover, using a general result from distributed graph transformation [10], we were able to show that the automata semantics in this model is compositional. This key property of our model allows us to formally describe the change in the semantics of a connector when it is being reconfigured and to define a notion of behavior-preserving reconfiguration.

To apply our approach in practice, typically the following steps are taken. First, component connectors and their reconfigurations are formally described as graphs and graph transformations, respectively. Second, based on the structure of the modeled connectors and the automata semantics of the used channels, a formal **DCASM**-model of the involved connectors and their reconfigurations is derived. Third, the induced semantics of connectors and their reconfigurations is obtained automatically by applying the semantics functor. Finally, behavior-preservation can be checked by statically analyzing the induced semantical reconfiguration rules.

As future work, we plan to extend our approach to exploit the full theory of algebraic graph transformation. Since our model is based on distributed graph transformation, we expect to be able to apply existing results for this purpose. In the context of behavior-preserving reconfiguration, an open question is whether the sufficient condition for ensuring behavior-preservation can be strengthened also to a necessary condition. Furthermore, we plan to investigate stricter notions of simulations, which can be used to ensure the preservation of safety properties, such as freedom of deadlock.

Finally, we plan to extend the implementation of Reo in the Extensible Coordination Tools (ECTs) [15,14,16,17] to support the modeling and execution of reconfigurable connectors based on the formal framework developed in this paper. Furthermore, we plan to evaluate the applicability of our approach in practice by means of case studies.

References

[1] F. Arbab, Reo: a channel-based coordination model for component composition, Mathematical Structures in Computer Science 14 (2004) 329–366, http://dx.doi.org/10.1017/S0960129504004153.

[2] C. Baier, M. Sirjani, F. Arbab, J. Rutten, Modeling component connectors in Reo by constraint automata, Science of Computer Programming 61 (2006) 75–113, http://dx.doi.org/10.1016/j.scico.2005.10.008.

[3] C. Krause, Z. Maraikar, A. Lazovik, F. Arbab, Modeling dynamic reconfigurations in Reo using high-level replacement systems, Science of Computer Programming 76 (2011) 23–36, http://dx.doi.org/10.1016/j.scico.2009.10.006.

[4] C. Koehler, F. Arbab, E. de Vink, Reconfiguring distributed Reo connectors, in: WADT'09, Lecture Notes in Computer Science, vol. 5486, Springer, 2009, pp. 221–235. http://dx.doi.org.10.1007/978-3-642-03429-9_15.

[5] P. Baldan, A. Corradini, H. Ehrig, R. Heckel, B. König, Bisimilarity and behaviour-preserving reconfigurations of open Petri nets, CoRR abs/0809.4115. http://dx.doi.org.10.2168/LMCS-4(4:3)2008.

[6] A. Khan, R. Heckel, P. Torrini, I. Ráth, Model-based stochastic simulation of P2P VoIP using graph transformation system, in: ASMTA'10, Lecture Notes in Computer Science, Springer, vol. 6148, 2010, pp. 204–217. http://dx.doi.org.10.1007/978-3-642-135682_15.

[7] M. Wermelinger, J.L. Fiadeiro, A graph transformation approach to software architecture reconfiguration, Science of Computer Programming 44 (2) (2002) 133–155, http://dx.doi.org/10.1016/S0167-6423(02)00036-9.

[8] F. Arbab, CASM: constraint automata with state memory, unpublished notes, 2007.

[9] H. Ehrig, K. Ehrig, U. Prange, G. Taentzer, Fundamentals of Algebraic Graph Transformation, EATCS Monographs in Theoretical Computer Science, Springer, 2006.

[10] C. Krause, Distributed port automata, in: GT-VMT'11, vol. 41, Electronic Communications of the EASST, 2011.

[11] S.M. Lane, Categories for the Working Mathematician (Graduate Texts in Mathematics), 2nd edition, Springer, 1998.

[12] J. Adámek, H. Herrlich, G. Strecker, Abstract and Concrete Categories, Wiley-Interscience, New York, NY, USA, 1990.

[13] B. Changizi, N. Kokash, F. Arbab, A unified toolset for business process model formalization, in: FESCA'10, 2010, Tool Demonstration Paper.

[14] C. Krause, Reconfigurable Component Connectors, Ph.D. Thesis, Leiden University, 2011.

[15] ECT - Extensible Coordination Tools, ⟨http://reo.project.cwi.nl⟩.

[16] N. Kokash, C. Krause, E.P. de Vink, Reo+mCRL2: a framework for model-checking dataflow in service compositions, Formal Aspects of Computing 24 (2) (2012) 187–216, http://dx.doi.org/10.1007/s00165-011-0191-6.

[17] C. Verhoef, C. Krause, O. Kanters, R.v.d. Mei, Simulation-based performance analysis of channel-based coordination models, in: COORDINATION'11, Lecture Notes in Computer Science, vol. 6721, Springer, 2011, pp. 187–201. http://dx.doi.org.10.1007/978-3-642-21464-6_13.

[18] B. Pourvatan, M. Sirjani, H. Hojjat, F. Arbab, Symbolic execution of Reo circuits using constraint automata, Science of Computer Programming 77 (7–8) (2012) 848–869, http://dx.doi.org.10.1016/j.scico.2011.04.001.

[19] D. Clarke, D. Costa, F. Arbab, Connector colouring I: synchronisation and context dependency, Science of Computer Programming 66 (3) (2007) 205–225, http://dx.doi.org/10.1016/j.scico.2007.01.009.

[20] R. Milner, Communication and Concurrency, Prentice Hall, 1989.

[21] G. Taentzer, Distributed graphs and graph transformation, Applied Categorical Structures 7 (1999) 431–462, http://dx.doi.org/10.1023/A:1008683005045.

[22] H. Ehrig, F. Orejas, U. Prange, Categorical foundations of distributed graph transformation, in: ICGT'06, Lecture Notes in Computer Science, vol. 4178, Springer, 2006, pp. 215–229. http://dx.doi.org.10.1007/11841883.

[23] J. de Lara, G. Taentzer, Modelling and analysis of distributed simulation protocols with distributed graph transformation, in: ACSD'05, 2005, pp. 144–153. http://dx.doi.org.10.1109/ACSD.2005.27.

[24] A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel, M. Löwe, Algebraic approaches to graph transformation I: Basic concepts and double pushout approach, in: Handbook of Graph Grammars and Computing by Graph Transformation, World Scientific, 1997, pp. 163–245.

[27] C. Koehler, D. Clarke, Decomposing port automata, in: SAC'09, ACM, 2009, pp. 1369–1373. ⟨http://dx.doi.org.10.1145/1529282.1529587⟩.

[28] D. Clarke, A basic logic for reasoning about connector reconfiguration, Fundamenta Informaticae 82 (4) (2008) 361–390.

[29] C. Koehler, D. Costa, J. Proenca, F. Arbab, Reconfiguration of Reo connectors triggered by dataflow, in: GT-VMT'08, Electronic Communications of the EASST, vol. 10, 2008, pp. 1–13.

[30] J. Padberg, H. Ehrig, G. Rozenberg, Behavior and realization construction for Petri nets based on free monoid and power set graphs, in: Unifying Petri Nets, Advances in Petri Nets, Lecture Notes in Computer Science, vol. 2128, Springer, 2001, pp. 230–249.

[31] P. Baldan, A. Corradini, H. Ehrig, R. Heckel, Compositional semantics for open Petri nets based on deterministic processes, Mathematical Structures in Computer Science 15 (2005) 1–35, http://dx.doi.org/10.1017/S0960129504004311.

[32] M. Droste, R.M. Shortt, From Petri nets to automata with concurrency, Applied Categorical Structures 10 (2) (2002) 173–191, http://dx.doi.org/10.1023/A:1014305610452.

[33] W.M.P. van der Aalst, Exterminating the dynamic change bug: a concrete approach to support workflow change, Information Systems Frontiers 3 (3) (2001) 297–317, http://dx.doi.org/10.1023/A:1011409408711.

[34] W.M.P. van der Aalst, T. Basten, Inheritance of workflows: an approach to tackling problems related to change, Theoretical Computer Science 270 (1–2) (2002) 125–203, http://dx.doi.org/10.1016/S0304-3975(00)00321-2.

Please cite this article as: C. Krause, et al., Compositional and behavior-preserving reconfiguration of component connectors in Reo, Journal of Visual Languages and Computing (2012), http://dx.doi.org/10.1016/j.jvlc.2012.09.002

[35] M. Hülsbusch, B. König, A. Rensink, M. Semenyak, C. Soltenborn, H. Wehrheim, Showing full semantics preservation in model transformation—a comparison of techniques, in: Integrated Formal Methods, Lecture Notes in Computer Science, vol. 6396, Springer, 2010, pp. 183–198. http://dx.doi.org.10.1007/978-3-642-16265-7_14.

[36] G. Rangel, L. Lambers, B. König, H. Ehrig, P. Baldan, Behavior preservation in model refactoring using DPO transformations with borrowed contexts, in: ICGT'08, Lecture Notes in Computer Science, vol. 5214, Springer, 2008, pp. 242–256. http://dx.doi.org.10.1007/978-3-540-87405-8_17.

[37] H. Giese, S. Glesner, J. Leitner, W. Schäfer, R. Wagner, Towards verified model to code transformations, in: MoDeV2a'06, ACM/IEEE, 2006.

[38] C. Ermel, H. Ehrig, Behavior-preserving simulation-to-animation model and rule transformation, in: GT-VC'07, Electronic Notes in Theoretical Computer Science, vol. 213, Elsevier Science, 2008, pp. 55–74.

[39] R. Heckel, S. Thöne, Behavior-preserving refinement relations between dynamic software architectures, in: WADT'04, Lecture Notes in Computer Science, vol. 3423, Springer, 2005, pp. 1–27. http://dx.doi.org.10.1007/978-3-540-31959-7_1.

Please cite this article as: C. Krause, et al., Compositional and behavior-preserving reconfiguration of component connectors in Reo, Journal of Visual Languages and Computing (2012), http://dx.doi.org/10.1016/j.jvlc.2012.09.002