

# Reconfiguration of Reo Connectors Triggered by Dataflow

Christian Koehler, David Costa, José Proença, Farhad Arbab

CWI, Amsterdam

**Abstract:** Reo is a language for coordinating autonomous components in distributed environments. Coordination in Reo is performed by circuit-like connectors, which are constructed from primitive, mobile channels with well-defined behaviour. While the structure of a connector can be modeled as a graph, its behaviour is compositionally defined using that of its primitive constituents. In previous work, we showed that graph transformation techniques are well-suited to model reconfigurations of connectors. In this paper, we investigate how the connector colouring semantics can be used to perform dynamic reconfigurations. Dynamic reconfigurations are triggered by dataflow in the connector at runtime, when certain structural patterns enriched with dataflow annotations occur. For instance we are able to elegantly model dynamic Reo circuits, such as just-in-time augmentation of single-buffered channels to a circuit that models a channel with an unbounded buffer. Additionally we extend Reo's visual notation and the Reo animation language to describe and animate dynamically reconfiguring connectors.

**Keywords:** Coordination, reconfiguration, graph transformation, animation.

## 1 Introduction

The coordination paradigm provides models for describing the communication among the components in a composed system. Coordination languages, such as Reo [Arb04], describe the 'gluing' of loosely coupled components, such that a desired system behaviour emerges. The achieved separation of business logic and coordination of the active entities leads to a much cleaner design and helps to handle the greater complexity of large applications. Reo can be applied in various distributed scenarios—from service-oriented to grid computing—as the coordination model is exogenous and independent from the actual component implementation and infrastructure.

In a Reo network, software components are autonomous, self-contained entities that communicate with the outside world via a published interface. To avoid dependencies and to achieve a truly modular, distributed system, Reo proposes the notion of connectors which are used to coordinate the components without their knowledge. By this, the system is divided into two orthogonal aspects: 1) the computation, performed by the components and 2) the coordination of these independent components, performed by the connectors. A major advantage of this design is the ability of changing the topology of the connector, and thereby the behaviour of the system.

The configuration of a Reo connector consists on the interconnection between the structural elements of the connector, together with their states. Communication with the components may change the state of the connector, but not its topology. In this paper, we consider *reconfigurations* of a connector as high-level transformations of its underlying graph structure. Using the theory

of typed, attributed graph transformation, we can directly apply many useful results, such as termination and confluence theorems [KLA07].

In this paper, we explore the interplay between the data flow in a connector and its reconfigurations. For this, we include the connector colouring semantics into the patterns of transformation rules. Transformations are automatically applied depending on the structure, the state and the context of a connector. Connectors are reconfigured at run-time based on this information. This leads to a powerful notion of dynamic connectors. We illustrate the principles through the example of a dynamically growing buffer.

**Related Work** The logic *ReCTL\** was introduced in [Cla08] to reason about connector reconfiguration in Reo. The reconfigurations are performed using the basic primitive operations of the Reo API, e.g. channel creation or node splitting. Up to now we have not provided a technique to reason about our dynamic high-level reconfigurations. However, we plan to look at model checking techniques for graph transformation as proposed in [Ren03a].

*Architectural Design Rewriting* (ADR) [BLMT07] is a framework for general reconfigurable software architectures. As in our approach, reconfigurations are modeled using graph transformation rules. Reconfigurations occur at run-time whenever the system evolves to a configuration that violates the architectural style of the system. In our case we do not fix the use of the reconfiguration to any particular purpose. The dynamic *FIFO* introduced in this paper can be seen as an example of the use of reconfiguration to guarantee dataflow. If a write operation is performed while the buffer is full, the buffer is reconfigured to allow data to be stored and dataflow to occur. ADR is not tied to any architectural style (e.g. client-server, peer-to-peer) while in our case, Reo determines the architecture.

A systematic introduction to animations based on graph transformation concepts was given in [Erm06]. The animation language for Reo that we use in this paper was introduced in [CP07]. It is important to note that the animation language for Reo is not based on graph transformation. Instead the authors introduce an abstract animation language that can be used to compute animation descriptions for a connector compositionally out of the descriptions of its constituent primitives.

**Structure of the paper** This paper is organised as follows. Section 2 gives a general introduction to Reo by introducing the notions of channels, nodes and connectors. An overview of the colouring semantics for Reo is given in Section 3. We recall the concepts of graph-based reconfigurations and provide our contributions to dynamic reconfigurations in Section 4. We discuss the proposed model in Section 5. The status of the current implementation and plans for future work are given in Sections 6 and 7, respectively.

## 2 Reo Overview

Reo is an exogenous coordination language where complex connectors are compositionally built out of simpler ones. The simplest (atomic) connectors in Reo consist of a user defined set of channels, each of which with its particular constraint policy.

A channel is a medium of communication with exactly two directed ends. There are two types of channel ends: *source* and *sink*. A source channel end accepts data into its channel. A sink channel end dispenses data out of its channel.

A channel can connect two components or be composed with other channels using Reo nodes to build more complex connectors. Reo nodes are logical places where channel ends coincide. A node with only source channel ends is a source node; a node with only sink channel ends is a sink node; and finally a node with both source and sink channel ends is a mixed node. We use the term *boundary nodes* to refer indistinguishably to source and sink nodes. Boundary nodes define the interface of a connector. Components connect to and interact anonymously with each other through the interface of the connector by performing I/O operations on its boundary nodes: *take* and *read* operations on sink nodes, and *write* operations on source nodes.

Reo fixes the constraint policy for the dataflow in Reo nodes. Data flows through a node only if at least one sink channel end is pushing data and all the source channel ends can *accept* a copy of the data. In case more than one sink channel end is pushing data, one is picked non-deterministically and all the others are excluded. Data cannot be stored in a node, hence its constraints on dataflow and exclusion must propagate through the connector.

Resolving the composition of the constraint policies of a connector consisting of several channels and nodes is a non-trivial task. In Figure 1 we present two examples of Reo connectors that illustrate how non-trivial dataflow behaviour emerges from composing channels using Reo nodes. The constraints propagate through the (synchronous regions of the) connector to the boundary nodes. The propagation enables a certain context-awareness in connectors. A detailed discussion of this can be found in [CCA07].

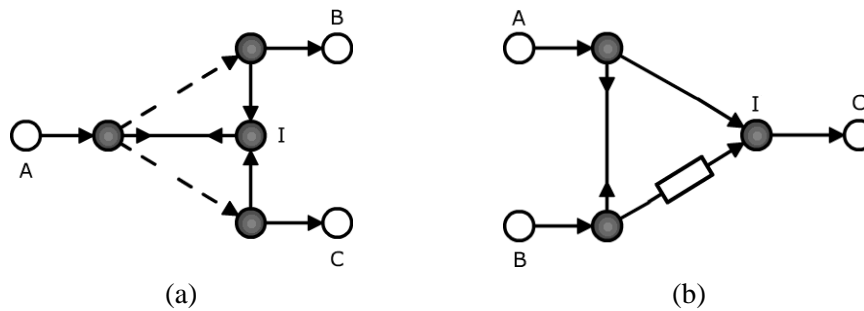


Figure 1: (a) exclusive router, (b) ordering connector.

The two connectors in Figure 1 involve, in total, four different types of channels. We represent mixed nodes as filled circles (●), and boundary nodes as empty circles (○). The *Sync* channel (—→) synchronously takes a data item from its source end and makes it available at its sink end. This transfer can succeed only if both ends are ready to communicate. The *LossySync* (----→) has the same behavior, except that it does not block if the receiver cannot accept data. In this case, the written data item is accepted and destroyed by the channel. The *FIFO<sub>1</sub>* (—□→) is an asynchronous channel that has a buffer of size one. Unlike the prior channels, *FIFO<sub>1</sub>* is a stateful channel. The *SyncDrain* channel (←→) has two source ends through which it can only consume data, and no sink ends. Its behavior can be described as follows: if there are data

items available at both ends, it consumes (and loses) both of them atomically.

The exclusive router, shown in Figure 1a, routes data from  $A$  to either  $B$  or  $C$ . The connector can accept data only if there is a write operation at the source node  $A$ , and there is at least one component attached to the sink nodes  $B$  or  $C$ , which is performing a take operation. If both  $B$  and  $C$  have a take operation, the choice of whether data is routed to  $B$  or  $C$  is made non-deterministically by the mixed node  $I$ . Node  $I$  can accept data only from one of its sink ends. To the other end it gives an exclusion reason for data not to flow, which forces the *LossySync* to lose the data.

The second connector, shown in Figure 1b, imposes an ordering on the flow of the data from the input nodes  $A$  and  $B$  to the output node  $C$ . The *SyncDrain* enforces that data flows through  $A$  and  $B$  synchronously. The empty buffer together with the *SyncDrain* guarantee that the data item obtained from  $A$  is delivered to  $C$  whereas the data item obtained from  $B$  is stored in the  $FIFO_1$  buffer. At this moment the buffer of the  $FIFO_1$  is full and data cannot flow in through either  $A$  or  $B$ , but  $C$  can obtain the data stored in the buffer. The buffer is then empty again.

These informal descriptions of the behavior of connectors can be formalised using the connector colouring semantics, introduced in [CCA07]. The colouring semantics is used to generate animations and to implement Reo, and we discuss it in Section 3.

Reo offers a number of operations to reconfigure and change the topology of a connector at run-time. Operations that enable the dynamic creation of channels, splitting and joining of nodes, hiding internal nodes and more. The hiding of internal nodes is important concerning reconfiguration, because it allows to fix permanently the topology of a connector, such that only its boundary nodes are visible and available. The resulting connector can be viewed as a new primitive connector, or primitive for short, since its internal structure is hidden and its behaviour is fixed. Reconfiguration is impossible on a primitive. We have the basic primitives that include the user defined channels, the Reo nodes, and the I/O operations. Plus the non-basic primitives constructed through the use of the hiding operation.

### 3 Connector Colouring Semantics

Connector Colouring semantics is based on the idea of colouring a connector using a set of colours  $\mathit{Colour}$ . We consider a set  $\mathit{Colour}$  with three colours as in Clarke et al. [CCA07]. One dataflow colour ( $\text{---}$ ) to mark places in the connector where data flows and two colours for no-dataflow ( $\text{---}\rightarrow$ ,  $\text{---}\leftarrow$ ) to mark the absence of dataflow. The reason for having two distinct no-dataflow colours is to be able to trace the exclusion constraints responsible for the no-flow back to their origins. Graphically, the arrow indicates the direction of exclusion, i.e. it points away from the exclusion reason and in the direction that the exclusion propagates.

Colouring a Reo connector in a specific state with given boundary conditions (I/O operations) provides a means to determine the route alternatives for dataflow. Each colouring of a connector is a solution to the synchronization and exclusion constraints imposed by its channels and nodes.

The dataflow allowed by a connector is collected in a *colouring table* whose elements—*colourings*—are functions mapping each node of the connector to a *colour*. The different colourings present in a colouring table of a connector correspond to the alternative ways that the connector can behave in the different contexts where it can be used.

We recall some essential definitions from [CCA07] that formalise the notion of a colouring and of a colouring table. Let  $\mathcal{Node}$  be a finite set of node names.

**Definition 1 (colouring)** A colouring  $c : N \rightarrow \mathcal{Colour}$  for  $N \subseteq \mathcal{Node}$  is a function that assigns a colour to every node of a connector.

**Definition 2 (colouring table)** A colouring table  $T$  over nodes  $N \subseteq \mathcal{Node}$  is a set of colourings with domain  $N$ .

To give semantics to a Reo connector using connector colouring one must provide the colouring tables for the user defined primitives, the channels, used in the construction of that connector. Table 1 shows the channels we use in this paper and their respective colouring tables. We highlight a few points of interest in this table, focusing only on reasons to exclude dataflow. No dataflow at one end of a *Sync* or *SyncDrain*, is enough to prevent dataflow in the all channel. The reason is propagated to the other end. An empty  $FIFO_1$  buffer does not enable data flow on its output end, giving a reason for no dataflow. Dually, a full  $FIFO_1$  buffer gives a reason for having no dataflow on its input end. The second entry of the table for a *LossySync* states that it will lose the data only when a reason for no dataflow is propagated into its output end, which amounts to saying that the channel is unable to transfer the data.

Reo fixes the colouring tables for the other primitives: nodes and the I/O operations. The Table 2 gives a brief account of the connector colouring semantics for these primitives. To comply with the page limit we omit the general colouring table of Reo mixed nodes. We give an example of one possible colouring for a Reo mixed node with 3 source ends and 2 sink ends. For the purpose of this paper that should suffice without compromising the understanding of what follows. For a full description we refer to [CCA07].

**Definition 3 (primitive)** A labelled tuple  $(n_1^{i_1}, \dots, n_k^{j_k})_c$  represents a primitive connector,  $c$ , where for  $0 < \ell \leq k$ ,  $n_\ell \in \mathcal{Node}$ ,  $j_\ell \in \{i, o\}$ ,  $k \geq 1$  is the arity of the primitive, and the labels  $i$  and  $o$  indicate a source node or a sink node respectively, such that a node  $n$  appears at most as  $n^i$  and/or  $n^o$  in  $(n_1^{i_1}, \dots, n_k^{j_k})_c$ . A primitive with colouring is a pair of a primitive with a colouring table  $T$  whose domain ranges over the nodes of the primitive.









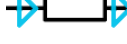

$(n_1^i, n_2^o)_{Sync}$	$(n_1^i, n_2^i)_{SyncDrain}$	$(n_1^i, n_2^o)_{LossySync}$	$(n_1^i, n_2^o)_{FIFO_1}$	$(n_1^i, n_2^o)_{FIFO_1[x]}$
				
				

Table 1: User defined channels, and their colouring tables.

A connector is a collection of primitives composed together, satisfying some well-formedness conditions. As such, the colouring table of a connector is computed from the colouring tables of its constituents.

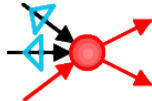




$(n_1^i, n_2^i, n_3^i, n_4^o, n_5^o)_{Node}$	$(n^i)_{Write}$	$(n^o)_{Take}$
	 	 

Table 2: Reo primitives, and their colouring tables.

**Definition 4 (connector)** A connector  $C$  is a tuple  $\langle N, B, E, T \rangle$  where,  $N$  is the set of nodes that appear in  $E$ ;  $B \subseteq N$  is the set of boundary nodes;  $E$  is a set of primitives;  $T$  is a colouring table over  $N$ ; such that (1)  $n \in B$  if and only if  $n$  appears exactly once in  $E$ , and (2)  $n \in N \setminus B$  if and only if  $n$  occurs exactly once as  $n^o$  and as  $n^i$  in  $E$ .

A primitive with a colouring table can straightforwardly be considered as a connector. A connector's semantics is computed by joining the tables of its constituents. Two colourings can only be composed if the common nodes in their domains are coloured with the same colour.

**Definition 5 (join)** Let  $C_k = \langle N_k, B_k, E_k, T_k \rangle$  with  $k \in \{1, 2\}$  be connectors such that  $(N_1 \setminus B_1) \cap (N_2 \setminus B_2) = \emptyset$ , and for each  $n \in B_1 \cap B_2$ ,  $n^i$  appears in  $E_1$  and  $n^o$  appears in  $E_2$ , or vice versa. The join of  $C_1$  and  $C_2$ , is given by:  $C_1 \odot C_2 \doteq \langle N_1 \cup N_2, (B_1 \cup B_2) \setminus (B_1 \cap B_2), E_1 \cup E_2, T_1 \cdot T_2 \rangle$ , where  $\cdot$  is the join operator for two colouring tables defined as:

$$T_1 \cdot T_2 \doteq \{c_1 \cup c_2 \mid c_1 \in T_1, c_2 \in T_2, n \in (dom(c_1) \cap dom(c_2)) \Rightarrow c_1(n) = c_2(n)\}.$$

Figure 2 depicts two colourings of the ordering connector of Figure 1(b). In both colourings, a component is connected to each boundary node and performs an I/O operation: *Write* on both source nodes and *Take* on the sink node. The colouring in (a) describes the dataflow behaviour

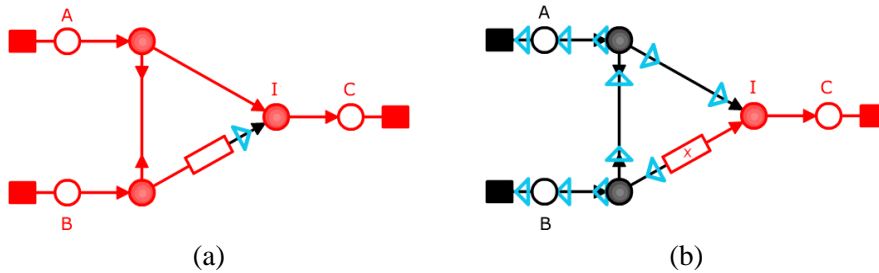


Figure 2: (a) and (b) are two possible colourings of the ordering connector.

of the connector when the buffer is empty, indicating that data flows through the entire connector except for the sink end of the  $FIFO_1$  channel, data is stored in the buffer, and all three I/O operations succeed. This dataflow changes the state of the  $FIFO_1$  channel, changing also its colouring table. The colouring in (b) describes the dataflow behaviour of the connector when the buffer is full. This colouring states that data flows in the connector only at the sink end of the

$FIFO_1[x]$  channel, the buffer is emptied, the take operation on node  $C$  succeeds, and the write operations on nodes  $A$  and  $B$  are delayed.

Figure 3 depicts the two colourings of the exclusive router that are the valid behaviour alternatives when a component is connected to each boundary node and performs an I/O operation: *Write* on the source node and *Take* on both the sink nodes. The colouring in (a) describes the

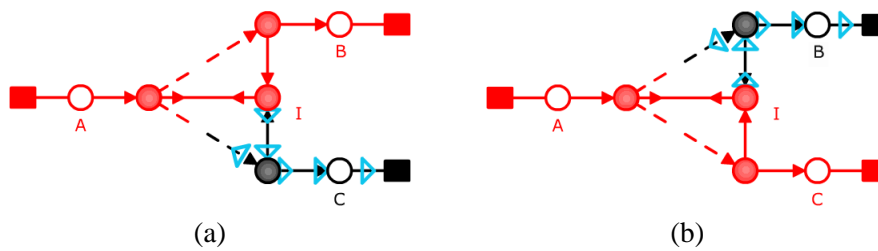


Figure 3: (a) and (b) are two possible colourings of the exclusive router.

dataflow behaviour of the connector when the mixed node  $I$  picks, non-deterministically, node  $B$  to route the data to. Alternatively the colouring in (b) describes the dataflow behaviour of the connector when the mixed node  $I$  picks, non-deterministically, node  $A$  to route the data to.

## 4 Connector Reconfiguration

It has been shown in [KLA07] that the theory of graph transformation can be applied to model connector reconfigurations as high-level transformations of their underlying graph structures. The approach allows to define reconfigurations at a high level of abstraction and therefore can be used to model complex reconfigurations, e.g. refactorings. While in the previous work, we considered these techniques in the context of business process customisation, we combine now these transformations with the connector colouring semantics, described in Section 3.

### 4.1 Reconfiguration Triggered by Dataflow

Dynamic reconfigurations are transformations of connectors at run-time. In the following, we present a framework that allows to define such dynamic reconfigurations by annotating transformation rules with colourings, which leads to a notion of dynamic connectors.

To use graph transformation for connector reconfiguration we make the following assumptions. Connectors are considered as typed, attributed graphs in the following way: i) Reo nodes are vertices of the graph and ii) channels are its edges. The typegraph in this scenario consists of a single node and one edge for each channel type. Edge attributes are used to model channel properties, e.g. the content of a full  $FIFO_1$ . Since channels in Reo are not necessarily directed (cf. the *SyncDrain* channel) we simply assert an underlying direction of the channel to fit the formal model of directed graphs that is usually assumed. Note also that we have given a formal definition of connectors in [KLA07] and showed that it indeed forms an adhesive High-Level Replacement category [EEPT06].

We use the Double-Pushout (DPO) approach [EEPT06] for our connector reconfigurations. Our transformation rules are an extended version of the usual spans of morphisms in the DPO approach. We write a connector reconfiguration rule as

$$p = (\mathit{Colour} \xleftarrow{c} L \xleftarrow{l} K \xrightarrow{r} R)$$

where  $L, K$  and  $R$  are connectors (typed, attributed graphs),  $l, r$  are connector (typed, attributed graph) homomorphisms and  $c$  is a (potentially partial) colouring for the left-hand side of the rule. The rationale behind this extension is that we do not just want to match the structure of a particular connector part, but also its state and the current execution strategy. Note that this introduces a certain asymmetry to the rules, caused by the fact that only the left-hand side is coloured. This is also the reason why we do not model the colouring as attributes of the graphs. Such an extended rule can be applied with respect to a given match  $L \xrightarrow{m} M$  and a current colouring  $M \xrightarrow{k} \mathit{Colour}$  iff

1. the gluing condition holds (see [EEPT06] for more details); and
2. the rule colouring matches the current colouring:  $c = k \circ m$ .

With the latter constraint we extend the pattern of a rule, in the way that a specific colouring has to be matched as well. A transformation rule can and will be applied only if the structure can be matched and a specific behaviour occurs. The extended version of the DPO approach can be summarised as shown in the diagram

$$\begin{array}{ccccc}
 \mathit{Colour} & \xleftarrow{c} & L & \xleftarrow{l} & K & \xrightarrow{r} & R \\
 & \swarrow k & \downarrow m & & \downarrow & & \downarrow \\
 & & M & \xleftarrow{\quad} & C & \xrightarrow{\quad} & N
 \end{array}$$

(PO)                      (PO)

where  $\mathit{Colour}$  is a fixed set of possible flow-colours and  $c$  and  $k$  are colourings. Including the colourings of a connector part makes the pattern matching much more restrictive. In fact, it is so restrictive now that the reconfiguration rules can be invoked by the Reo engine without ‘supervision’. The transformation system becomes in some respect autonomous. The connector is transformed when necessary, without the need for an external party to trigger the reconfiguration.

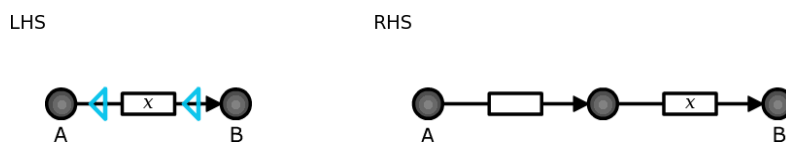


Figure 4: reconfiguration rule for a dynamic  $FIFO$ .

Figure 4 shows a reconfiguration rule for a dynamic (unbounded)  $FIFO$ . The matches from the LHS to the RHS are indicated by using the same node labels. The reconfiguration rule gives rise to a dynamic connector, which we call  $FIFO_{\infty}$ . The  $FIFO_{\infty}$  consists of a sequence of  $FIFO_1$  channels and a reconfiguration rule. The reconfiguration adds a new  $FIFO_1$  in the



beginning whenever the  $FIFO_{\infty}$  is full and someone tries to write to it assuring that way that the write can succeed and the data can flow and be stored in the buffer. The left-hand side matches an arbitrary full  $FIFO_1$  with content  $x$  where someone tries to write to. The reason for the no-flow is the fact that the  $FIFO_1$  is full already, not that there is no data available. If this pattern can be matched, the rule states that the original channel is destroyed and two new  $FIFO_1$  channels are created in its place. The second  $FIFO_1$  is filled with the original content  $x$ .

However, the rule as it is now does not reflect our initial requirement that only the first  $FIFO_1$  in the sequence is replaced by two new ones. This can be achieved by adding negative application conditions [EEPT06] as shown in Figure 5. We need two extra NACs for the dynamic  $FIFO_{\infty}$ .

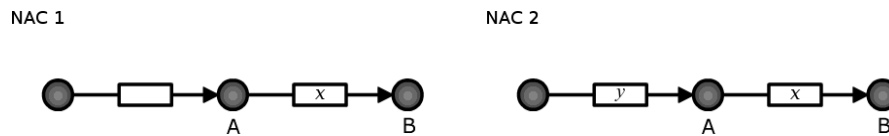


Figure 5: negative application conditions for the  $FIFO_{\infty}$ .

that restrict *where* the rule can be applied. An empty  $FIFO_1$  should always be added at the very beginning of the sequence. Expressed as negative application conditions, this means that there must not be an empty or a full  $FIFO_1$  in front of the one where the original rule (Figure 4) applies.

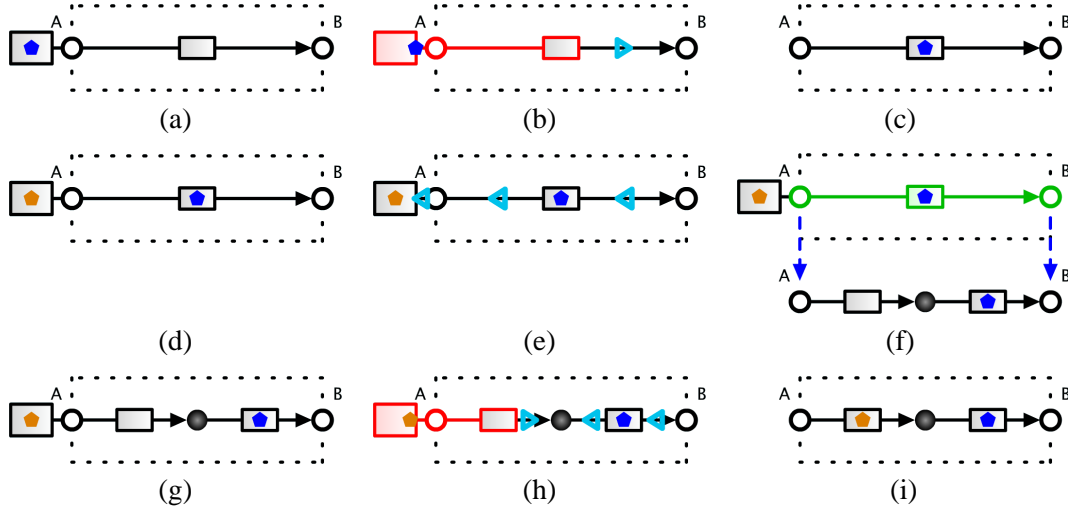
These additional restrictions allow us to apply the transformation rule automatically at runtime when the colouring occurs. For completeness we would have to define also an inverse rule that shrinks the connector again. We omit this here.

## 4.2 Run of a connector

Connectors are executed in an abstract Reo engine. The Reo engine includes two independent components, one to compute colouring tables and to perform the dataflow, and one for computing reconfiguration matches and executing the transformations. We refer to these components as **Df** and **Tr** respectively. Reconfiguration rules are applied locally only in specific regions of the connector. These regions can be formally viewed as disjoint sub-graphs that restrict the domain of the transformations. These regions are the reconfigurable parts of the connector. Each of these regions has a number of reconfiguration rules attached to it, such as the one in Figure 4.

The Reo engine utilises **Df** and **Tr** to execute dynamic connectors. In this scenario a run of the engine consists of performing the following actions:

1. Invoke **Df** to compute the colouring table  $CT$  of the connector for the actual boundary conditions.
2. Choose non-deterministically a colouring  $k$  from the colouring table  $CT$ .
3. For each reconfiguration region, invoke **Tr** to find pattern matches  $m_1, \dots, m_n$  for the colouring  $k$ .
4. Invoke **Df** to execute the dataflow according to  $k$ . The state of the connector is updated.


 Figure 6: run of the  $FIFO_{\infty}$ .

5. For each pattern match in  $m_1, \dots, m_n$ , invoke  $\mathbf{Tr}$  to perform the transformations.

Since the steps 3 and 4 are independent from each other, they can be performed in parallel. Note also that in general, multiple transformation policies could be supported, e.g. apply at most once or until it is not applicable anymore.

Figure 6 depicts a run according to the descriptions above, for the  $FIFO_{\infty}$ . Part (a) shows the basic connector that we want to reconfigure. It consists of a single empty  $FIFO_1$  with a *Write* operation. The region where the transformation rule from Figure 4 should be applied is delimited by the dashed box. Part (b) shows the colouring  $k_1$  and the corresponding dataflow. The transformation engine is invoked with a snapshot of the reconfiguration region and the colouring  $k_1$ . Since the rule does not match colouring  $k_1$  no transformation is performed. In part (c) we see how the dataflow has changed the state of the  $FIFO_1$  and that the write operation disappeared after succeeding. In part (d) a new write operation is attached to the connector. At this point a new run of the engine starts. Figure 6e shows that the new write operation cannot succeed because the  $FIFO_1$  is already full. This event is formally observed by the no-flow colouring  $k_2$ . The transformation engine is invoked again and returns a match  $m$  this time, because  $k_2$  matches the colouring of the reconfiguration rule. Since no dataflow has to be performed, the transformation can immediately be applied, as shown in Figure 6f. The reconfigured connector is shown in part (g). At this point, the write operation succeeds, and the corresponding colouring  $k_3$  is depicted in part (h). There is no rule that matches  $k_3$ , since the only possible match with the full  $FIFO_1$  is restricted by NAC 1, defined in Figure 5. In Figure 6i we show the state of the connector after the dataflow, whereas the write operation is removed and the left  $FIFO_1$  becomes full.

The visual representation we use is based on an extension of the animation language introduced in [CP07]. We use the colour green to highlight the part of the connector that matches the left-hand side of the rule, as depicted in the region inside the dashed box in Figure 6f, and

the connector that appears below corresponds to the right-hand side of the rule. The dashed blue arrows in Figure 6f, pointing from the match to the substitute connector, indicate that the nodes  $A$  and  $B$  are preserved by the transformation. Furthermore, the colour of the new data token matches the colour of the data token in the matched part of the connector, expressing that the argument of the full  $FIFO_1$  channel is the same as the argument of the new full  $FIFO_1$ .

## 5 Discussion

Up to now we have shown that by applying graph transformation techniques to Reo, we are able to describe complex connector reconfigurations using a very compact notation. Transformation rules are extended with colouring information so that they can be applied when a certain behaviour occurs. External invocation of the transformation is not required.

In the current setup, the transformation engine is invoked in each run of the execution of the connector. Even if the required colouring is not present, the engine first tries to match the structural pattern of the current connector and then validates its colouring. If the reconfiguration regions become larger, the pattern matching can cause a significant degradation of performance. Note that general pattern matching for graphs is a problem that is known to be NP-complete.

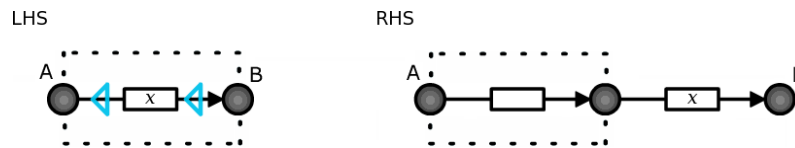


Figure 7: rule for a  $FIFO_\infty$  with an evolving reconfiguration region.

To improve the performance of dynamically reconfiguring connectors, we suggest the following optimisation. We augment the transformation rules with information about the reconfiguration regions. For our example of the dynamic  $FIFO$  we extend the rule as shown in Figure 7. The dashed boxes mark the reconfiguration regions. With this extension we can state that the rule is always applied to the first  $FIFO_1$ . While before the reconfiguration region was growing with each application of the rule, it will now remain as consisting of exactly two nodes and one  $FIFO_1$  only, all the time. This way, the pattern matching is much faster and (in the ideal case) always uniquely determined.

## 6 Implementation

**Reo as a modelling framework** The Reo modelling framework [ECT] consists of a set of plug-ins for the Eclipse<sup>1</sup> development environment such as: a graphical editor, a model checker and a Flash-based animation tool.

Executable instances of connectors can be derived by generating code from Reo specifications or by interpreting these specifications. We generate code from abstract animation descriptions

<sup>1</sup> <http://www.eclipse.org>

when producing the Flash animations [CP07]. We have implemented an interpreter<sup>2</sup>, based on the colouring semantics, for the application domain of web services. Currently we are integrating the dynamic reconfiguration scheme into the interpreter.

**Reo as a runtime architecture** We are also developing a distributed Reo engine [Pro07], where each primitive is deployed and executed as an independent block running in parallel with the other primitives. There is no centralized entity computing the colouring table of the composite connector at each step, but instead the colouring table is obtained by distributed agreement. The engine has a mechanism to pause regions of the connector, making them suitable to be re-configured. However, since the development is still in an early stage we are not able to integrate the dynamic reconfiguration yet.

## 7 Future work

The example of the dynamic *FIFO* is certainly very basic. In general, there may be more than one reconfiguration rule (a graph grammar) attached to a region of a connector. The role that the colouring extension in our rules plays in these more complex scenarios must be investigated. Furthermore, we need to make the notion of the evolving reconfiguration regions, as suggested in Section 5, more specific. Additional partial mappings from the LHS to RHS may be a possible approach to model this.

We are interested in preservation of behavioural properties by transformations. Two different classes of transformations are interesting in this setting: transformations that preserve the behaviour and transformations that change it. On the one hand, reconfigurations that do not change the behaviour are interesting in the area of automated refactoring and optimisation. On the other, reconfigurations that change the behaviour can be used to implement new sub-circuits that adapt the behaviour of a circuit based on dataflow. In this context, it will be interesting to do static analysis of reconfiguration rules to reason about behaviour preservation.

Finally, we want to integrate the dynamic reconfiguration triggered by dataflow into the Reo tools. For this purpose, we will extend both the Eclipse based development tools and the Reo runtime engine for web services. Since we model reconfiguration through graph transformation, and due to the fact that our implementation of the Reo development tools is based on the Eclipse Modeling Framework (EMF)<sup>3</sup>, we plan to implement the reconfiguration extensions using the Tiger EMF Transformation tools [EMT].

## Bibliography

- [Arb04] F. Arbab. Reo: a Channel-based Coordination Model for Component Composition. *Mathematical Structures in Computer Science* 14:329–366, 2004.

---

<sup>2</sup> <http://www.cwi.nl/~koehler/services>

<sup>3</sup> <http://www.eclipse.org/emf>

- [BEK<sup>+</sup>06] E. Biermann, K. Ehrig, C. Koehler, G. Kuhns, G. Taentzer, E. Weiss. Graphical Definition of In-Place Transformations in the Eclipse Modeling Framework. In *Model Driven Engineering Languages and Systems (MoDELS'06)*. 2006.
- [BLMT07] R. Bruni, A. Lluch-Lafuente, U. Montanari, E. Tuosto. Style-based architectural reconfigurations. Technical report TR-07-17, Computer Science Department, University of Pisa, 2007.
- [CCA07] D. Clarke, D. Costa, F. Arbab. Connector Colouring I: Synchronization and Context Dependency. *Sci. Comput. Program.* 66(3):205–22, 2007.  
doi:<http://dx.doi.org/10.1016/j.scico.2007.01.009>
- [Cla08] D. Clarke. A Basic Logic for Reasoning about Connector Reconfiguration. *Fundamenta Informaticæ* 82:1–30, 2008.
- [CP07] D. Costa, J. Proença. Connector Animation: A Compositional Framework to Analyse Reo Connectors. December 2007. Submitted. Available online: <http://www.cwi.nl/~costa/publications/publications.htm>.
- [ECT] Eclipse Coordination Tools. <http://homepages.cwi.nl/~koehler/ect>.
- [EEPT06] H. Ehrig, K. Ehrig, U. Prange, G. Taentzer. *Fundamentals of Algebraic Graph Transformation*. EATCS Monographs in Theoretical Computer Science. Springer, 2006.
- [EMT] Tiger EMF Transformation Project. <http://tfs.cs.tu-berlin.de/emftrans>.
- [Erm06] C. Ermel. *Simulation and Animation of Visual Languages based on Typed Algebraic Graph Transformation*. PhD thesis, 2006.
- [KLA07] C. Koehler, A. Lazovik, F. Arbab. Connector Rewriting with High-Level Replacement Systems. In *Proceedings of FOCLASA 2007, to be published in Electronic Notes in Theoretical Computer Science*. 2007.
- [Pro07] J. Proença. Towards Distributed Reo. Talk presented at CIC workshop, 2007.
- [Ren03a] A. Rensink. GROOVE: A Graph Transformation Tool Set for the Simulation and Analysis of Graph Grammars. <http://www.cs.utwente.nl/~groove>, 2003.
- [Ren03b] A. Rensink. Towards Model Checking Graph Grammars. In Leuschel et al. (eds.), *Workshop on Automated Verification of Critical Systems (AVoCS)*. Technical Report DSSE-TR-2003-2, pp. 150–160. University of Southampton, 2003.