Reconfiguring Distributed Reo Connectors

C. Koehler[1,⋆,⋆⋆], F. Arbab[1], and E.P. de Vink[2]

[1] CWI, P.O. Box 94079, 1090 GB Amsterdam, The Netherlands
[2] Technische Universiteit Eindhoven, Den Dolech 2, Eindhoven, The Netherlands

Abstract. The coordination language Reo defines circuit-like connectors to steer the collaboration of independent components. In this paper, we present a framework for the modeling of distributed, self-reconfigurable connectors based on algebraic graph transformations. Reconfiguring a connector that is composed with others, may involve a change of shared interfaces and may therefore require a reconfiguration of the surrounding connectors as well. We present a method of synchronized local reconfigurations in this setting and discuss a bottom-up strategy for coordinating synchronized reconfigurations in a connector network. We exploit the double-pushout approach for the modeling of reconfigurations, and propose an adaptation of the concept of amalgamation for synchronizing reconfigurations. We use a nondeterministic scheduler as our running example.

1 Introduction

Building software systems using an exogenous coordination language, such as Reo [1], is done by (i) implementing a set of (wrappers for existing) components or services and (ii) composing these entities using a kind of glue code. In the case of Reo, circuit-like *connectors* constitute the glue code. Connectors may consist of other connectors, but are elementarily composed from channels and nodes. Every connector implements a protocol defined by the semantics of its constituents, and the topology of the connector. Reconfiguring a connector means to change its topology and thereby the coordination protocol that it implements. Reconfiguration arises from the need to dynamically adapt the behavior of a system, e.g., in response to a change in its environment, to cope with altered resource availability or to retrofit it for a modified mission. The need for considering *distributed* connectors arises from two concerns. On the one hand, connectors are decomposed into logically separate parts, each of which defines a specific subprotocol. On the other hand, connectors can be deployed on different physical locations in a network. In both cases, the concept of distribution facilitates and promotes the use of black-boxed subconnectors in a larger context.

In this paper, we propose a framework for modeling reconfigurable, distributed Reo connectors. We consider connectors that are distributed over a network and are encapsulated, i.e. their internals are hidden from the outside

⋆ Supported by NWO GLANCE project WoMaLaPaDiA and SYANCO.
⋆⋆ Corresponding author, e-mail `christian.koehler@cwi.nl`.

world and communicate only via a published interface. Connectors are linked together via the interfaces that they share. Reconfiguration of a distributed connector is achieved by reconfiguring its subconnectors. Ultimately, reconfiguration is defined and performed locally, that is to say, in the scope of a single connector, but it can be either triggered from the inside or invoked from the outside. Reconfiguring a connector may involve a change in its interfaces and may require connectors in its neighborhood to reconfigure as well. This implies a need for synchronizing such local reconfigurations into a consistent reconfiguration of the connector as a whole. It goes without saying that in a distributed setting, we cannot assume the existence of a (centralized) third party that monitors and coordinates local reconfigurations. Therefore, other mechanisms should be in place to assure the consistency of a reconfigured network.

In short, this paper contributes the following: We propose a model for reconfigurable, distributed connectors. We utilize the well-studied framework of distributed graph transformation [3, 4] for this purpose. We show, furthermore, how reconfigurations can be defined and performed locally using a synchronization mechanism based on the notion of amalgamation [5–7]. Finally, we propose a distributed strategy to organize the stepwise reconfiguration of large networks.

Related work. Modeling the distribution of systems via embedding interfaces represented as morphisms in a suitable category is also used for open Petri-nets [9]. The explicit modeling of glue code and the exploitation of pushout constructions to deal with composition in this work is similar to ours. However, they do not consider the application of double pushouts and their concept of amalgamation is different. In [9], amalgamation serves the composition of deterministic processes of open Petri-nets, whereas in our approach it is used as a synchronization mechanism for the superposition of local reconfiguration rules.

In [10], Architectural Design Rewriting is proposed as a framework for modeling reconfigurable software architectures. This work deals with hierarchical, non-distributed architectures and uses hyperedge replacement, as opposed to algebraic graph transformations in our work. A general introduction to system modeling and system evolution using graph transformation techniques, including hierarchical and distributed approaches, can be found in [11].

Our approach to coordination achieved by Reo connectors and their dynamic reconfiguration fits in the framework of runtime software adaptation [12, 13] for component-based software engineering. Process algebraic treatments include [14, 15]. To accommodate dynamic reconfiguration, predicted behavioral changes combined with revision of message translation are captured by so-called contextual mappings. However, the focus in this work is not on distribution, which is a key aspect of our paper. A workflow language extension with the so-called configurable elements, e.g. for YAWL, is proposed in [16], with a semantics based on a variant of Petri-nets, called extended workflow nets (EWF-nets).

Structure of the paper. The rest of this paper is organized as follows. Section 2 contains an overview of the coordination language Reo. We describe basic graph transformation techniques using the double pushout approach and an application to Reo in Section 3. In Section 4, we formally introduce distributed

connectors as typed distributed graphs. In Section 5, we introduce reconfigurations for distributed connectors and discuss amalgamation as a synchronization mechanism. Section 6 contains conclusions and future work.

2 Reo Connectors

A Reo connector [1, 2] acts as glue code connecting a number of components together. A connector orchestrates the behavior of the components it connects, enforcing a specific interaction pattern. By influencing the timing of I/O operations of the components, it achieves coordination through constraining their behavior. The computational internals of the components are oblivious to the connector. As such, Reo falls in the class of exogenous coordination languages.

Taking various flavors of channels as primitive building blocks, more complex connectors can be composed in Reo from simpler ones. Channels are point-to-point means to communicate that meet at nodes. Channels have two ends, a source and a sink, or two sources or two sinks.

The set of primitive Reo channels is user-defined, but typically includes the channel types in Table 1. The synchronous channel *Sync*, simultaneously takes a data item from its source end and makes it available at its sink end. The synchronous drain *SyncDrain* has two source ends, but no sink end. If there are data items available at both ends, it consumes and loses both of them simultaneously. The lossy synchronous channel *LossySync* behaves like the *Sync* channel, except that it does not block its source when its sink end cannot accept data. Instead it accepts and loses the data item taken from the source. The $FIFO_1$ is an asynchronous and stateful channel, having a buffer of size one. If its buffer is empty and a data item is available at its source end, the I/O operation succeeds and the item is stored in the buffer. The $FIFO_1$ blocks any further write requests until the data item is delivered through its sink end. It then returns back to its empty state. Other channels are allowed as well, e.g. with filtering capability. The only requirement for a channel is that it has exactly two ends.

Reo distinguishes three kinds of nodes: source nodes, sink nodes and mixed nodes. From a connector's point of view, source and sink nodes are also called input and output nodes, respectively. Collectively, they form the boundary nodes of a connector, which interact with its environment. Mixed nodes, on the other hand, are internal and not accessible from the outside. A mixed node has both incoming and outgoing channels, i.e. channels meeting at the node with their sink and source ends. For a mixed node to fire, it needs at least one of its incoming channels be willing to deliver a data item, while simultaneously all its outgoing

Sync	*SyncDrain*	*LossySync*	$FIFO_1$

Table 1. Some primitive channels.

channels are willing to consume it. The node then nondeterministically selects one of its enabled sink ends and passes its data item to all its source ends. In all of our examples we will represent mixed nodes as filled circles and boundary nodes as empty circles.

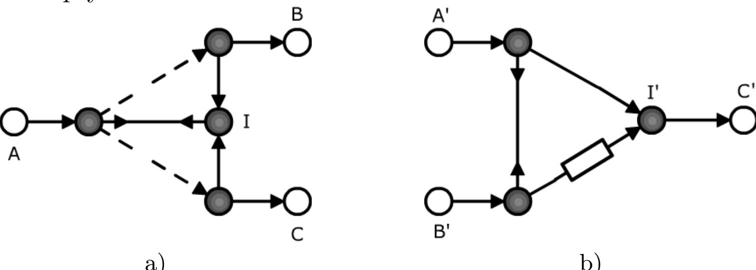


Fig. 1. a) Exclusive router, b) ordering.

Example 1. 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, that performs a take operation. If both B and C allow an output, the choice between B and C is made nondeterministically by the mixed node I. Note that data flows synchronously through this connector, i.e. there is no buffering involved. The exclusive router proves to be useful in a number of situations. The binary exclusive router in Figure 1a) can be generalized straightforwardly to an n-ary one. The symbol $\otimes$ is used as its shorthand in the sequel.

Example 2. The second connector, shown in Figure 1b), imposes an ordering on the data flow from the input nodes A' and B' to the output node C'. The *SyncDrain* enforces synchronous flows through A' and B'. 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 buffer. At this moment, the buffer of the $FIFO_1$ is full and data cannot flow in, neither through A' nor B', as they are coupled by the synchronous drain. However, C' can now obtain the data stored in the buffer.

Reo connectors come equipped with a formal data flow semantics based on so-called connector colorings [17]. The basic idea is to assign to each channel an admissible communication behavior that is compatible with all nodes. Conceptually, the execution cycle for a connector allows for system reconfiguration after a consistent coloring has been established and the corresponding data flow has been accomplished. See [17] for more detail on the coloring semantics, and [18] and [19] for the semantics of Reo based on constraint automata and on tiles, respectively. Currently, the software suite for Reo includes a number of development tools, integrated within Eclipse [20], and runtime engines for various platforms. In particular, a distributed implementation of Reo is available, built on Scala [21], allowing individual nodes to be distributed over the network.[3]

[3] See `http://homepages.cwi.nl/~proenca/distributedreo/`.

3 Reconfiguration by Graph Transformation

First, we recall the basic definitions for typed graphs and introduce our running example. Next, we show how to model basic reconfigurations of Reo connectors using algebraic graph transformation. We refer to [6, 8] for more details on the algebraic approach to graph transformation.

3.1 Typed Graphs

We use a graph model where edges are directed and have identity, i.e. a *graph* is a structure $G = \langle V, E, s, t\rangle$ with V a set of nodes, E a set of edges and $s, t : E \to V$ source and target functions. A *graph morphism* is a pair of functions $h = \langle h_V, h_E\rangle$ that preserve the source and target functions. Graphs and graph morphisms form the category **Graph**.

When modeling Reo connectors as graphs, nodes in a graph represent Reo nodes and edges represent channels. Intuitively, the set of nodes and channels allowed in a connector is given via a *type graph*. For a fixed type graph T, an *instance graph* over T is a pair $\langle G, \mathit{type}\rangle$ where G is a graph and $\mathit{type} : G \to T$ a morphism into the type graph. A *typed graph morphism* $h : \langle G_1, \mathit{type}_1\rangle \to \langle G_2, \mathit{type}_2\rangle$ is a graph morphism $h : G_1 \to G_2$ that preserves the type information, i.e. $\mathit{type}_1 = \mathit{type}_2 \circ h$. Fixing a type graph T, the category of typed graphs over T is denoted with $\mathbf{Graph_T}$.

For the Reo connectors in our running example, we consider a type graph of four different node types, two types of internal nodes: ordinary Reo nodes ● and exclusive routers ⊗, and two types of interface nodes ○, which are either *start* or *finish* nodes. The type graph further includes an edge for every channel type. Note that, for undirected channels, such as the *SyncDrain*, the source and the target of the edge model both a source or a sink end of the channel. We do not restrict the use of channels, i.e. a channel of any type can be connected to a node of any type.[4] We further include additional edge types for primitive components with two ends. The category of Reo graphs is denoted by $\mathbf{Graph_{Reo}}$.

Example 3. Figure 2 depicts two connectors, which constitute our running example. The first connector in Figure 2a) is a *nondeterministic scheduler* in its initial state. It consists of four interface nodes: two *start* nodes and two *finish* nodes, and it is capable of scheduling two tasks. In the first step of execution, the scheduler enables –if possible– one of the *start* nodes by moving the token from the FIFO_1 in the middle to one of the other two FIFO_1s and replicating the token on the selected *start* node. If more than one *start* node can be enabled, the choice is made nondeterministically. At this stage, the scheduler waits until it can enable the corresponding *finish* node. When this step is performed, the connector goes back to its original configuration by moving the token back to the middle.

[4] A more natural way of modeling Reo connectors would use attributed typed graphs with node inheritance (cf. [22]). For simplicity, we restrict to typed graphs here.

The second connector in our example, depicted in Figure 2b), models a *task repository*. A task is a primitive component that can be either in the idle or the processing state. When it is idle and ready to switch to the processing state, it produces a token on its right-hand end. It signals that it can be stopped again using a token on its left-hand end. The repository wraps every task using two *SyncDrain*s and exposes interface nodes for starting and finishing each of them.

In our application scenario, we will connect these two connectors along their interface nodes. However, instead of gluing the nodes and thereby hardwiring the connectors, we will consider them as distributed in a network having an exposed interface to be shared. Before introducing distribution, we first illustrate how to reconfigure connectors using graph rewriting techniques.

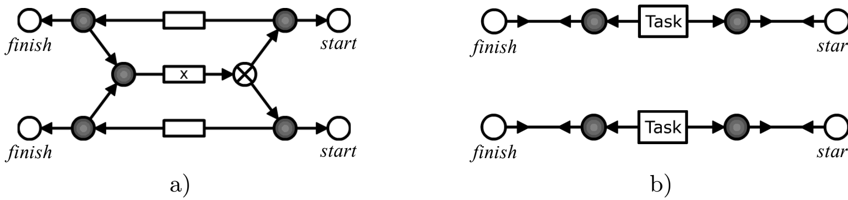


Fig. 2. a) Nondeterministic scheduler for two tasks, b) task repository with two tasks.

3.2 Algebraic Graph Transformation

We follow the double-pushout (DPO) approach to graph transformation. Graphs are transformed by applying graph productions, which we will also refer to as *reconfiguration rules* in our application. In categorical terms, a *production* is a span of injective morphisms $p = L \xleftarrow{\ell} K \xrightarrow{r} R$ in the category **Graph**. The left-hand side L defines the pattern that must be matched to apply the production. K contains all elements that are not removed by the rule and R additionally has those elements of the graph that are created by the rule. Given a production p, a *match* is a morphism $m : L \to M$, where M is the graph to be transformed. A *derivation* $M \overset{p,m}{\Longrightarrow} N$ is an application of p with the match m, formally defined as the following diagram where (1) and (2) are pushouts.

$$\begin{array}{ccccc} L & \xleftarrow{\ell} & K & \xrightarrow{r} & R \\ {\scriptstyle m}\downarrow & (1) & \downarrow & (2) & \downarrow \\ M & \longleftarrow & C & \longrightarrow & N \end{array}$$

Operationally, the graph M is transformed to the graph N by (i) removing the occurrence of $L \backslash \ell(K)$ in M, yielding the intermediate graph C, and (ii) adding a copy of $R \backslash r(K)$ to C. Due to the categorical formulation of the transformation concepts, the approach can be directly transferred to typed graphs, which we use in the sequel to model reconfigurations of Reo connectors.

Example 4. To reconfigure the nondeterministic scheduler and the task repository introduced above, we define a number of reconfiguration rules. Figure 3 depicts an example rule that extends the scheduler with a slot for an additional task. The gluing graph K is not drawn here, as in all of our rules. It is defined as the intersection of the left and the right-hand side. The mappings ℓ and r are indicated by the relative positions of the nodes and channels. In the same way, we define a rule for adding a task to a repository. By reversing these rules, we can realize a removal of tasks or slots in the scheduler, respectively.

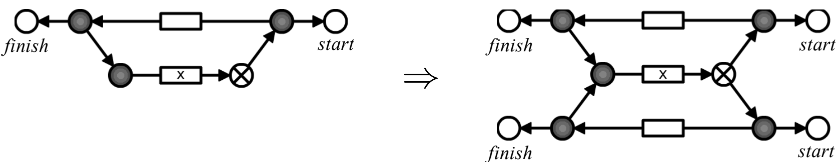


Fig. 3. Reconfiguration rule for extending a nondeterministic scheduler.

4 Distributed Connectors

To model reconfigurations of distributed Reo connectors, we use the framework of *Distributed Graph Transformation*, as introduced by Taentzer [3] for graphs and generalized by Ehrig [4] to transformations of distributed objects. In the following, we recall the notions of the framework that are relevant to the present setting and apply them to Reo.

4.1 Distributed Typed Graphs

Distribution of graphs can be described by adding a second level of abstraction, namely by modeling the topology of a system using a so-called *network graph*. The nodes in a network graph consist of *local graphs* and the edges are morphisms of these local graphs. The idea is that a node models a physical or logical location of a local graph, whereas an edge indicates an occurrence of the source graph in the target graph. In particular, multiple outgoing edges from one local graph model the fact that the source graph is shared among the target graphs.

Formally, a distributed graph is a pair (N, D) where N is an ordinary graph, the network graph, and D is a mapping that associates to every node n in N a local graph $D(n)$ and to every edge $n \xrightarrow{e} n'$ in N a graph morphism $D(e) : D(n) \to D(n')$. In categorical terms, this mapping corresponds to a functor $D : N \to \mathbf{Graph}$, also called a *diagram*, where the graph N is interpreted as a category. Following [4], this functor 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 represent the sharing of the local graphs. Note that due to the categorical formalization, the

concept of distribution can be applied to typed graphs as well by considering functors $D : N \to \mathbf{Graph_T}$. Hence, as a first approximation, distributed Reo connectors can be modeled as distributed typed graphs.

Example 5. An example of a distributed connector is depicted in Figure 4. The network graph is drawn using dashed lines. The nondeterministic scheduler and the task repository appear as nodes in the network graph. The two other nodes of the network graph are interfaces, each containing two interface nodes, viz. a *start* and a *finish*. Further, there are four embeddings of the interfaces into the scheduler and the repository. Type preservation by the embedding guarantees that start nodes map to start nodes and similarly for finish nodes. Obviously, although details are suppressed here, the two embeddings are supposed to have disjoint ranges. However, since the interfaces are embedded into both the scheduler and the repository, these connectors are considered to be connected along their interfaces.

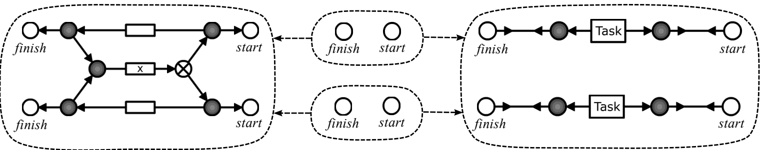


Fig. 4. Nondeterministic scheduler connected to a task repository.

Given two distributed typed graphs (N_1, D_1) and (N_2, D_2), 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 $f_D : D_1 \to D_2 \circ f_N$. We will just write f for the network morphism f_N. By definition, the natural transformation f_D assigns to every node n of N_1 a graph morphism $f_n : D_1(n) \to D_2(f(n))$; f_n is called the *local* graph morphism of n. Furthermore, for every edge $n \xrightarrow{e} n'$ in N_1 the following diagram commutes.

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

Example 6. An example morphism of distributed connectors is given in Figure 5. The unary scheduler in node n' is mapped into a binary scheduler in node $f(n')$. The target network has an interface node and an embedding into the scheduler that is not in the image of the morphism.

The categories of distributed graphs and of distributed typed graphs are denoted by $\mathbf{Dis}(\mathbf{Graph})$ and $\mathbf{Dis}(\mathbf{Graph_T})$. However, for a proper modeling of distributed Reo connectors the typing mechanism of $\mathbf{Dis}(\mathbf{Graph_T})$ is not sufficient, as we will argue in the following.

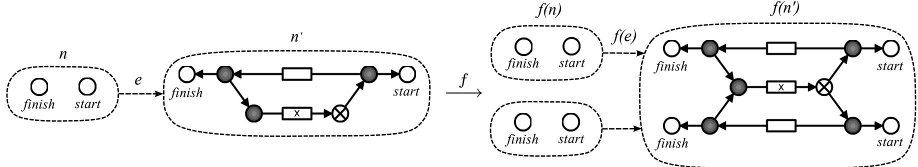


Fig. 5. Example morphism of distributed typed graphs.

4.2 Typed Distributed Graphs

When considering Reo connectors as distributed typed graphs, i.e. objects (N, D) with $D : N \to \mathbf{Graph_T}$, only the local graphs are typed. For our application, we need types at the network level as well, because of the following constraints:

(i) Nodes in a Reo network graph are either *Connectors* or *Interfaces*.
(ii) Edges in a network graph are allowed only from *Interfaces* to *Connectors*.
(iii) An *Interface* is linked to at most two *Connectors*.

A way of dealing with these constraints is to use a typed network graph with multiplicity constraints (cf. [23]). Multiplicity constraints for edges are useful in this context, since they allow to restrict the number of links between interfaces and connectors. Figure 6 depicts a type graph for network graphs that enforces the above constraints. The edge multiplicities make sure that interfaces always connect at most two connectors. Interfaces connecting more than two connectors are avoided, since the merger-replicator semantics of Reo nodes would give the interfaces a non-trivial and potentially unexpected behavior. Hence, using the

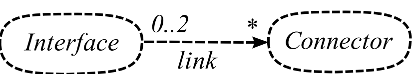


Fig. 6. Network type graph for Reo.

network type graph in Figure 6, we can impose the constraints (i)–(iii) above. However, type constraints that relate the local and the network levels cannot be expressed by this formalism. Therefore, we require the following additional constraint for distributed connectors:

(iv) Local graphs assigned to *Interfaces* consist only of *start* and *finish* nodes.

We can model this constraint using *typed distributed graphs*, as opposed to distributed typed graphs. The latter approach, as alluded to above, involves considering objects (N, D) in the category $\mathbf{Dis}(\mathbf{Graph_T})$, i.e. where D maps the nodes of N to typed graphs and the edges to typed graph morphisms. On the other hand, a *typed distributed graph* is a tuple $(N, D, type)$ where (N, D) is an object

in $\mathbf{Dis}(\mathbf{Graph})$ and *type* : $(N, D) \rightarrow (N_T, D_T)$ a morphism in $\mathbf{Dis}(\mathbf{Graph})$ with $T = (N_T, D_T)$ a fixed distributed graph, called the *distributed type graph*. This means, instead of considering the category $\mathbf{Dis}(\mathbf{Graph_T})$, we use the slice category $\mathbf{Dis}(\mathbf{Graph})\backslash T = \mathbf{Dis}(\mathbf{Graph})_\mathbf{T}$. This typing mechanism is more expressive, since the type graph and the type morphisms are distributed already. Accordingly, we model distributed connectors as typed distributed graphs in the rest of this paper. The distributed type graph for Reo consists on the network level of the type graph in Figure 6. On the local level, the node type *Interface* is mapped to a graph that consists only of the two interface nodes *start* and *finish*. The node type *Connector* is mapped to the default type graph for Reo, as presented in Section 3.1. The edge type *link* is mapped to a graph morphism, that maps the node *start* in *Interface* to the node *start* in *Connector*, and analogously for *finish*. We denote the category of distributed Reo connectors by $\mathbf{Dis}(\mathbf{Graph})_\mathbf{Reo}$.

5 Reconfiguration of Distributed Connectors

In order for the double-pushout approach to apply, we must make sure that $\mathbf{Dis}(\mathbf{Graph})_\mathbf{Reo}$ has pushouts. Since the category $\mathbf{Dis}(\mathbf{Graph})$ is cocomplete (cf. [4]), it follows that $\mathbf{Dis}(\mathbf{Graph})_\mathbf{Reo}$ is cocomplete too, as it is a slice category of $\mathbf{Dis}(\mathbf{Graph})$. Consequently, pushouts exist and we can use DPO-rewriting for modeling reconfigurations of distributed Reo connectors. Next, we show that reconfigurations can be defined locally, i.e. in the scope of a single connector, and we discuss how these local reconfigurations can be synchronized.

5.1 Local Reconfigurations

The need for local reconfigurations arises from the distributed setting, where no global knowledge of the system is available. In the following example, we give distributed versions of the reconfiguration rules of our scheduler application.

Example 7. Two distributed reconfiguration rules p_1 and p_2 are depicted in Figure 7. Rule p_1 extends the nondeterministic scheduler by a slot for an additional task and creates a new interface for this slot. Rule p_2 adds another task to the repository and creates an interface for this task. Note that the reconfiguration of the scheduler on the one hand, and the task repository on the other, are modeled by two separate rules, because we assume that connectors are reconfigured locally. In principle, the connectors can see each other as black boxes that publish only their interfaces and reconfigure themselves on demand.

To reconfigure networks using local rules, we need a way to synchronize reconfigurations. As can be seen from the example above, applying the two reconfiguration rules p_1 and p_2 naively to the network of Figure 4, does not give the desired result, since each rule creates a new interface whereas we need just one that is shared by the two connectors.

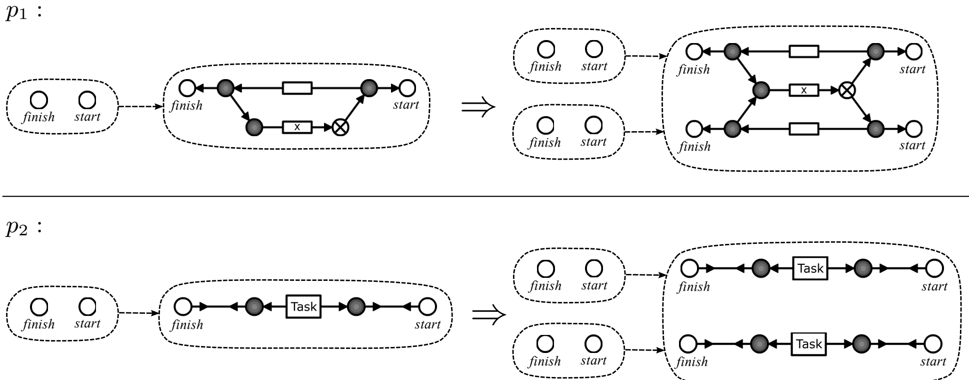


Fig. 7. Rules for extending a distributed scheduler / task repository.

5.2 Synchronizing Local Reconfigurations

Following [5], we use the concept of amalgamation for synchronizing local reconfigurations. We first recall the basic definitions for amalgamated graph transformations.

The synchronization of two productions is achieved by identifying a common subproduction and gluing the productions along this subproduction. Let

$$p_i = L_i \xleftarrow{\ell_i} K_i \xrightarrow{r_i} R_i$$

be two productions with $i \in \{0, 1\}$. The production p_0, together with graph morphisms $in_L^1 : L_0 \to L_1$, $in_K^1 : K_0 \to K_1$, $in_R^1 : R_0 \to R_1$, are called a *subproduction* of p_1, if in the following diagram (1) and (2) commute. Putting $in^1 = \langle in_L^1, in_K^1, in_R^1 \rangle$, we write $in^1 : p_0 \to p_1$ for the embedding of p_0 into p_1.

$$\begin{array}{ccccc} L_0 & \xleftarrow{\ell_0} & K_0 & \xrightarrow{r_0} & R_0 \\ \downarrow in_L^1 & (1) & \downarrow in_K^1 & (2) & \downarrow in_R^1 \\ L_1 & \xleftarrow{\ell_1} & K_1 & \xrightarrow{r_1} & R_1 \end{array}$$

The productions p_1 and p_2 are called *synchronized* with respect to p_0, if p_0 is a subproduction of both p_1 and p_2, denoted by $p_1 \xleftarrow{in_1} p_0 \xrightarrow{in_2} p_2$.

Example 8. A non-trivial, i.e. non-empty, subproduction p_0 of the reconfiguration rules p_1 and p_2 is depicted in Figure 8. The rule creates a new interface node in a network graph. As a general property of our reconfiguration approach, the common subproduction of two synchronized rules always describes an interface change of the involved connectors.

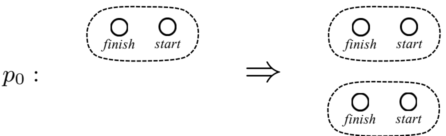
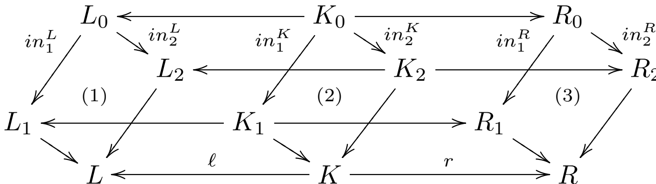


Fig. 8. Common subproduction of p_1 and p_2 modeling the interface evolution.

By making explicit the change of interface due to an update of connectors, synchronized productions can properly describe reconfigurations in a network. Execution of synchronized productions can be achieved using amalgamations. Given two synchronized productions $p_1 \xleftarrow{in_1} p_0 \xrightarrow{in_2} p_2$, the *amalgamated production*

$$p_1 \oplus_{p_0} p_2 : L \xleftarrow{\ell} K \xrightarrow{r} R$$

is constructed by gluing p_1 and p_2 along p_0 using the pushouts (1), (2) and (3) in the diagram below, such that all squares commute. The morphisms ℓ and r are induced by the universal property of the pushout (2). Applying $p_1 \oplus_{p_0} p_2$ to a graph G yields an *amalgamated derivation* $G \Rightarrow X$.



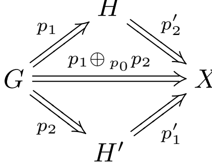
Example 9. Amalgamation of the productions p_1 and p_2 in Figure 7 using the subproduction p_0 in Figure 8, generates the intended rewrite rule for the reconfiguration of the original distributed system in Figure 4. Note that the complete network is reconfigured using local rules in one atomic step.

Even though it provides a proper means of synchronizing local reconfigurations, in general, amalgamation is less-suited for distributed systems. This is because it requires (i) knowledge of all connectors and their reconfiguration rules, and (ii) a centralized entity that is aware of the whole network and that performs the reconfiguration in a non-local fashion. To overcome these problems we show now a different way of applying synchronized rules.

5.3 Local Execution of Synchronized Rules

The reconfiguration mechanism for distributed connectors that we introduce in the following, avoids the problem of amalgamations by performing the reconfiguration asynchronously, where locally only a single connector and its interface are updated at a time. Without repeating the actual constructions, we first recall (the analysis part of) the so-called amalgamation theorem, which provides the means to locally execute synchronized reconfiguration rules. For more details and a proof of the theorem we refer to [5].

Amalgamation Theorem. Let $p_1 \xleftarrow{in_1} p_0 \xrightarrow{in_2} p_2$ be synchronized productions and $G \Rightarrow X$ an amalgamated derivation via $p_1 \oplus_{p_0} p_2$. Then there exist productions p_1' and p_2', called the *remainders* of p_1 and p_2 with respect to p_0, such that the following derivations exist:



Intuitively, the remainders p_1' and p_2' simulate the effect of p_1 and p_2 without performing the action of the subproduction p_0. For our application to distributed Reo connectors, this means that the remainders do the reconfiguration of the connectors without updating the interfaces, or more precisely, assuming that they have been updated already.

Example 10. The remainders of the productions p_1 and p_2 in Figure 7 are the same as the original rules, except that their left-hand sides contain the newly created interface already. The rules merely establish a new connection to the already existing interface, instead of creating it. Hence, the reconfiguration of the distributed connector in Figure 4 can be done by first applying p_1 to update the scheduler including the interfaces and then p_2' to update the task repository accordingly. Analogously, it can be also the case that first the repository together with the interfaces are updated and then the scheduler.

Using the above approach, a network can be reconfigured by a stepwise updating of its constituent connectors. In particular, the connectors can also be black boxes that reconfigure themselves. On the other hand, these local reconfigurations must be coordinated somehow, since the order of local reconfigurations and the choice of which connector updates the common interface is not clear. For this purpose, we discuss a strategy in the next subsection.

5.4 Coordinating Local Reconfigurations

We informally describe a strategy for organizing local reconfigurations in a network. The central idea is that a reconfiguration is triggered locally at one of the subconnectors and that this creates a cascade of follow-up reconfigurations across the network.

Connectors may define synchronized reconfiguration rules, i.e. rules that describe how the connector itself is changed, and further, how its interfaces are updated. We also assume that a connector reconfigures itself triggered by an external request. For this purposes it may publish the names of its reconfiguration rules. Connectors in the neighborhood can invoke these reconfiguration rules via their shared interface (through a communication channel that is not explicitly modeled here). When a rule is invoked, a connector performs the reconfiguration in three steps:

1. Determine the interface where the request came from and the interfaces of those connectors in the neighborhood that also need to be updated.
2. Send reconfiguration requests to those connectors in the neighborhood that must be updated and block until they are reconfigured.
3. Do the local reconfiguration and reconfigure the interface, if necessary, only where the request came from.

We assume that there is an 'active' party in the network that initiates the reconfiguration by invoking a rule on some connector. Every connector can handle only one reconfiguration request at a time. Hence, the request builds up a reconfiguration dependency tree in the network. The root of the tree is where the reconfiguration was initially invoked. The reconfiguration is then executed bottom-up, starting at the leaves until the root is also reconfigured.

Connectors may also respond to a reconfiguration request with a failure. In that case, the failure is forwarded in the network and all reconfigurations performed so far are rolled-back. This ensures the atomicity of the reconfigurations.

6 Concluding Remarks

We presented a framework, exploiting algebraic graph transformations, for the reconfiguration of distributed Reo connectors. This approach allows a black-boxed view on subconnectors for which reconfigurations can be defined and executed locally. We showed how to synchronize local reconfigurations in the absence of a centralized entity, which is a prime assumption in distributed environments.

Future work includes the formal modeling of the distributed strategy for coordinating local reconfigurations. The typing mechanism used for Reo can be further extended. In particular, constraints that ensure disjointness of multiple interfaces are not modeled at present. Our distribution model can also serve as a basis for describing deployment operations, e.g. transparently moving a connector to another network location. Finally, the dynamic reconfiguration approach presented here, needs to be incorporated in the existing distributed Reo implementation. For this purpose, we have already implemented a reconfiguration engine based on algebraic graph transformation as a part of the Eclipse Coordination Tools [20].

References

1. Arbab, F.: Reo: A Channel-based Coordination Model for Component Composition. Mathematical Structures in Computer Science **14** (2004) 329–366
2. Arbab, F.: Abstract Behavior Types: A Foundation Model for Components and Their Composition. Science of Computer Programming **55** (2005) 3–52
3. Taentzer, G.: Distributed Graphs and Graph Transformation. Applied Categorical Structures **7** (1999) 431–462
4. Ehrig, H., Orejas, F., Prange, U.: Categorical Foundations of Distributed Graph Transformation. In Corradini, A., Ehrig, H., Montanari, U., Ribeiro, L., Rozenberg, G., eds.: Proc. ICGT 2006, LNCS 4178 (2006) 215–229

5. Boehm, P., Fonio, H.R., Habel, A.: Amalgamation of Graph Transformations: A Synchronization Mechanism. Journal of Computer and System Sciences **34** (1987) 377–408
6. Corradini, A., Montanari, U., Rossi, F., Ehrig, H., Heckel, R., Löwe, M.: 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) 163–245
7. Taentzer, G., Beyer, M.: Amalgamated Graph Transformations and Their Use for Specifying AGG. In Schneider, H., Ehrig, H., eds.: Dagstuhl Seminar on Graph Transformations in Computer Science, LNCS 776 (1994) 380–394
8. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. EATCS Monographs in Theoretical Computer Science. Springer (2006)
9. Baldan, P., Corradini, A., Ehrig, H., Heckel, R.: Compositional Semantics for Open Petri Nets based on Deterministic Processes. Mathematical Structures in Computer Science **15** (2005) 1–35
10. Bruni, R., Lafuente, A.L., Montanari, U., Tuosto, E.: Style-based Architectural Reconfigurations. Bulletin of the EATCS **94** (2008) 181–180
11. Engels, G., Heckel, R.: Graph Transformation as a Conceptual and Formal Framework for System Modeling and Model Evolution. In Montanari, U., Rolim, J., Welzl, E., eds.: Proc. ICALP 2000, LNCS 1853 (2000) 127–150
12. Yellin, D., Strom, R.: Protocol specifications and component adaptors. ACM Transactions on Programming Languages and Systems **19** (1990) 292–333
13. Canal, C., Murillo, J., Poizat, P.: Software adaptation. L'Object **12** (2006) 9–31
14. Brogi, A., Cámera, J., Canal, C., Cubo, J., Pimentel, E.: Dynamic contextual adaptation. Electronics Notes in Theoretical Computer Science **175** (2007) 81–95
15. Cubo, J., Salaün, G., Cámara, J., C.Canal, Pimentel, E.: Context-based adaptation of component behavioural interfaces. In Murphy, A., Vitek, J., eds.: Proc. COORDINATION 2007, LNCS 4467 (2007) 305–323
16. Gottschalk, F., Aalst, W.v.d., Jansen-Vullers, M., La Rosa, M.: Configurable workflow models. Journal of Cooperative Information Systems **17** (2008) 177–221
17. Clarke, D., Costa, D., Arbab, F.: Connector Colouring I: Synchronisation and Context Dependency. Science of Computer Programming **66** (2007) 205–225
18. Baier, C., Sirjani, M., Arbab, F., Rutten, J.: Modeling component connectors in Reo by constraint automata. Science of Computer Programming **61** (2006) 75–113
19. Arbab, F., Bruni, R., Clarke, D., Lanese, I., Montanari, U.: Tiles for Reo (extended abstract). In Corradini, A., Gadduci, F., eds.: WADT08, preliminary proceedings, Technical Report TR–08–15, Dipartemento di Informatica, Università di Pisa (2008) 21–24
20. Arbab, F., Koehler, C., Maraikar, Z., Moon, Y.J., Proenca, J.: Modeling, Testing and Executing Reo Connectors with the Eclipse Coordination Tools. In Canal, C., Pasareanu, C., eds.: Proc. FACS 2008. (2008) To appear.
21. Odersky, M.: The Scala Experiment: Can We Provide Better Language Support for Component Systems? In Morrisett, J., Peyton Jones, S., eds.: Proc. POPL, ACM (2006) 166–167
22. de Lara, J., Bardohl, R., Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Attributed Graph Transformation with Node Type Inheritance. Theoretical Computer Science **376** (2007) 139–163
23. Taentzer, G., Rensink, A.: Ensuring structural constraints in graph-based models with type inheritance. In: FASE, LNCS 3442 (2005) 64–79